Universidade Federal de Juiz de Fora

Programa de Pós-Graduação em Modelagem Computacional

Tiago Marques do Nascimento

Implementação e Avaliação de Desempenho de Dois Algoritmos de Balanceamento de Carga para *Clusters* Híbridos

Juiz de Fora

2018

Tiago Marques do Nascimento

Implementação e Avaliação de Desempenho de Dois Algoritmos de Balanceamento de Carga para *Clusters* Híbridos

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Modelagem Computacional.

Orientador: Marcelo Lobosco

Coorientador: Rodrigo Weber dos Santos

Juiz de Fora

2018

Ficha catalográfica elaborada através do programa de geração
automática da Biblioteca Universitária da UFJF,
com os dados fornecidos pelo(a) autor(a)

Nascimento, Tiago Marques do.
Implementação e Avaliação de Desempenho de Dois Algoritmos de Balanceamento de Carga para Clusters Híbridos / Tiago Marques do Nascimento. -- 2018.
87 f. : il.

Orientador: Marcelo Lobosco
Coorientador: Rodrigo Weber dos Santos
Dissertação (mestrado acadêmico) - Universidade Federal de Juiz de Fora, ICE/Engenharia. Programa de Pós-Graduação em Modelagem Computacional, 2018.

1. Balanceamento de Carga. 2. Ambientes Híbridos. 3. Modelagem Matemática. 4. Modelagem Computacional. 5. Imunologia Computacional. I. Lobosco, Marcelo, orient. II. Santos, Rodrigo Weber dos, coorient. III. Título.

Tiago Marques do Nascimento

Implementação e Avaliação de Desempenho de Dois Algoritmos de Balanceamento de Carga para *Clusters* Híbridos

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Modelagem Computacional.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Marcelo Lobosco - Orientador
Universidade Federal de Juiz de Fora

Professor Dr. Rodrigo Weber dos Santos - Coorientador
Universidade Federal de Juiz de Fora

Professor Dr. Ruy Freitas Reis
Universidade Federal de Juiz de Fora

Professor Dr. Raphael Yokoingawa de Camargo
Universidade Federal do ABC

AGRADECIMENTOS

Não é fácil atingir este ponto de maturidade acadêmica, e com certeza não se chega aqui sem a ajuda de pessoas que se importam com você e o ajudam nas horas que mais precisa. Gostaria de agradecer primeiramente à Deus por me dar forças para passar pelos momentos mais difíceis, à minha familia, principalmente à minha mãe Sandra Helena e meu pai Edmir Monteiro, que sempre me apoiou em minhas decisões e sempre esteve do meu lado nos momentos bons e ruins, à minha namorada Taynara Nankassa, que cruzou meu caminho no meio desta jornada, mas que foi importante para me manter motivado a buscar e nunca desistir dos meus objetivos e, finalmente, ao meu orientador Marcelo Lobosco e ao meu coorientador Rodrigo Weber, por todos os ensinamentos e por me guiarem durante boa parte da minha vida acadêmica, não poderia ter escolhido profissionais melhores para desempenhar este papel.

Também gostaria de agradecer a todos os amigos e companheiros que fiz ao longo do caminho, os quais pude compartilhar conhecimentos e experiências e crescer tanto como profissional quanto como pessoa. Agradeço também aos meus colegas de trabalho e por fim à Stefanini, por ter me dado a oportunidade de vivenciar o mercado de trabalho e de crescer profissionalmente, podendo seguir com minha carreira acadêmica ao mesmo tempo.

“O trabalho vai preencher uma grande parte da sua vida, e a única maneira de ficar completamente satisfeito é fazer o que você acredita ser um bom trabalho. E a única forma de fazer um bom trabalho é amar aquilo que você faz. Se você ainda não descobriu o que é, continue procurando. Não se acomode. Da mesma forma que acontece com as coisas do coração, você vai saber quando encontrar.”

Steve Jobs

RESUMO

Este trabalho trata da implementação e análise de desempenho de dois algoritmos de balanceamento de carga para aplicações baseadas em paralelismo de dados, quando estas são executadas em um ambiente híbrido de memória distribuída. Neste trabalho, ambiente híbrido é definido como um ambiente computacional composto por dispositivos que contêm um ou mais elementos de processamento com distintas capacidades computacionais, sendo estes CPUs, APUs, GPUs, entre outros. O objetivo dos algoritmos de balanceamento de carga é equalizar o tempo de computação, ou seja, fazer com que os elementos de processamento recebam uma carga de trabalho proporcional a sua capacidade de processamento, de modo que finalizem suas execuções aproximadamente ao mesmo tempo. Para testar e validar os algoritmos de balanceamento de carga, utilizou-se um modelo computacional do Sistema Imune Humano (SIH) que descreve a resposta espaço-temporal de algumas das células e moléculas do SIH na presença de um antígeno, que neste trabalho é representado pelo Lipopolissacarídeo (LPS). Duas versões do balanceador de carga foram desenvolvidas, o balanceador de carga estático, que mantêm a alocação de carga nos dispositivos até o final do processamento, e o balanceador de carga dinâmico, que pode alterar a alocação de carga nos dispositivos ao longo da execução. Após os testes realizados com as duas versões dos balanceadores de carga, pode-se concluir que, para a aplicação avaliada, a versão dinâmica foi mais eficiente que a versão estática.

Palavras-chave: Balanceamento de carga. Ambientes híbridos. Modelagem Matemática. Modelagem Computacional. Imunologia Computacional.

ABSTRACT

This work presents an implementation and performance analysis of two load balancing algorithms for applications based on data parallelism, when these applications are executed in hybrid distributed memory environments. In this work, hybrid environment is defined as a computational environment composed by a set of processors and accelerators, in other words, processing elements that have different processing capabilities, such as CPUs, APUs, GPUs or other types of accelerators. The purpose of the load balancing algorithm is to equalize the computation time, *i.e.* the load balancing algorithm has to send to the processing elements a workload proportional to their processing power, so they can finish their executions at about the same time. In order to test and validate the load-balancing algorithms, a computational model of the Human Immune System (HIS) was used to describe the space-temporal response of some cells and molecules of the HIS in the presence of an antigen, represented in this work by the Lipopolysaccharide (LPS). Two versions of the load balancer have been developed. The first one, the static load balancer, maintains the load allocation in the devices until the end of the execution. The second one, the dynamic load balancer, may change the load allocation on devices during the execution. After the tests performed with the two versions of the load balancer, it can be concluded that, for the evaluated application, the dynamic version was more efficient than the static one.

Key-words: Load balancing. Heterogeneous Computing. Mathematical Modelling. Computational Modelling. Computational Immunology.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

APU	*Accelerating Processing Unit*
BCD	Balanceamento de Carga Dinâmico
BCE	Balanceamento de Carga Estático
CA	Citocina Anti-Inflamatória
CH	Citocina Pró-Inflamatória
CPU	*Computing Processing Unit*
CUDA	*Compute Unified Device Architecture*
EDP	Equação Diferencial Parcial
FWT	*Fast Wavelet Transform*
G	Grânulo Protéico
GPU	*Graphics Processing Unit*
GRN	*Gene Regulatory Networks*
HSP	*Hitting Set Problem*
HUMA	*Heterogeneous Unified Memory Architecture*
LPS	Lipopolissacarídeo
MA	Macrófago Ativo
MPI	*Message Passing Interface*
MR	Macrófago em Repouso
N	Neutrófilo
NA	Neutrófilos Apoptóticos
OpenMP	*Open Multi-Processing*
PAMP	*Pathogen-associated Molecular Pattern*
PRR	*Pattern Recognition Receptors*
PThreads	*POSIX Threads*
PU	*Processing Unit*

RRTM	*Rapid Radiation Transfer Model*
SIH	Sistema Imune Humano
SIMD	*Single Instruction Multiple Data*
SSE	*Streaming SIMD Extensions*
WI	*Work Item*
WG	*Work Group*

LISTA DE SÍMBOLOS

μ_{LPS}	Taxa de decaimento do LPS
λ_{LPS_N}	Taxa de fagocitose do LPS pelos Neutrófilos
λ_{LPS_MA}	Taxa de fagocitose do LPS pelos Macrófagos Ativos
σ_{LPS_MR}	Taxa de ativação dos Macrófagos em Repouso, seguida pela fagocitose por estes do LPS
γ_{CA}	Taxa de saturação da Citocina Pró-Inflamatória no tecido
d_{LPS}	Taxa de difusão do LPS
μ_{MR}	Taxa de decaimento dos Macrófagos em Repouso
M^{max}	Concentração máxima de macrófagos no tecido
$P^{max}_{MR\_CH}$	Permeabilidade máxima do endotélio aos Macrófagos em Repouso, causada pelas Citocinas Pró-Inflamatórias
$P^{min}_{MR\_CH}$	Permeabilidade mínima do endotélio aos Macrófagos em Repouso, causada pelas Citocinas Pró-Inflamatórias
η_{MR_CH}	Concentração das Citocinas Pró-Inflamatórias que exercem 50% do efeito máximo no aumento de permeabilidade do endotélio aos Macrófagos em Repouso
$P^{max}_{MR\_G}$	Permeabilidade máxima do endotélio aos Macrófagos em Repouso, causada pelos Grânulos Protéicos
$P^{min}_{MR\_G}$	Permeabilidade mínima do endotélio aos Macrófagos em Repouso, causada pelos Grânulos Protéicos
η_{MR_G}	Concentração dos Grânulos Protéicos que exerce 50% do efeito máximo no aumento de permeabilidade do endotélio aos Macrófagos em Repouso
d_{MR}	Taxa de difusão dos Macrófagos em Repouso
q_{CH_MR}	Taxa de quimiotaxia dos Macrófagos em Repouso pelas Citocinas Pró-Inflamatórias
μ_{MA}	Taxa de decaimento dos Macrófagos Ativos
d_{MA}	Taxa de difusão dos Macrófagos Ativos
q_{CH_MA}	Taxa de quimiotaxia dos Macrófagos Ativos pelas Citocinas Pró-Inflamatórias

μ_N	Taxa de decaimento dos Neutrófilos
N^{max}	Concentração máxima de neutrófilos no tecido
$P^{max}_{N\_CH}$	Permeabilidade máxima do endotélio aos Neutrófilos, causada pelas Citocinas Pró-Inflamatórias
$P^{min}_{N\_CH}$	Permeabilidade mínima do endotélio aos Neutrófilos, causada pelas Citocinas Pró-Inflamatórias
η_{N_CH}	Concentração das Citocinas Pró-Inflamatórias que exercem 50% do efeito máximo no aumento de permeabilidade do endotélio aos Neutrófilos
d_N	Taxa de difusão dos Neutrófilos
q_{CH_N}	Taxa de quimiotaxia dos Neutrófilos pelas Citocinas Pró-Inflamatórias
μ_{CH}	Taxa de decaimento das Citocinas Pró-Inflamatórias
β_{LPS_N}	Taxa de produção das Citocinas Pró-Inflamatórias por Neutrófilos, quando na presença de LPS
ω_{CH}	Limite de concentração das Citocinas Pró-Inflamatórias suportado no tecido
κ_{CA}	Taxa de inibição da produção da Citocina Pró-Inflamatória, tanto por Neutrófilos quanto por Macrófagos Ativos, quando na presença das Citocinas Anti-Inflamatórias
β_{LPS_MA}	Taxa de produção das Citocinas Pró-Inflamatórias por Macrófagos Ativos, quando na presença de LPS
d_{CH}	Taxa de difusão das Citocinas Pró-Inflamatórias
λ_{ND_MA}	Taxa de fagocitose dos Neutrófilos Apoptóticos pelos Macrófagos Ativos
d_{ND}	Taxa de difusão dos Neutrófilos Apoptóticos
μ_G	Taxa de decaimento dos Grânulos Protéicos
α_{N_G}	Taxa de produção dos Grânulos Protéicos por Neutrófilos
d_G	Taxa de difusão dos Grânulos Protéicos
μ_{CA}	Taxa de decaimento das Citocinas Anti-Inflamatórias
β_{MR_ND}	Taxa de produção das Citocinas Anti-Inflamatórias por Macrófagos em Repouso, quando na presença de Neutrófilos Apoptóticos

ω_{CA}	Limite de concentração das Citocinas Anti-Inflamatórias suportado no tecido
β_{MA}	Taxa de produção das Citocinas Anti-Inflamatórias por Macrófagos Ativos
d_{CA}	Taxa de difusão das Citocinas Anti-Inflamatórias

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

A tecnologia computacional está em constante evolução. A cada dia vemos novas arquiteturas computacionais que superam as anteriores em termos de poder de processamento. Uma forma de aumentar esse poder de processamento está em aumentar o consumo de energia por unidade de tempo (potência) dos computadores.

Entretanto, a potência dos computadores alcançou um ponto de estagnação no início do século XXI. Uma alternativa para aumentar o poder computacional dos dispositivos foi então aumentar o número de unidades de processamento, o que permite que uma quantidade maior de dados possa ser processada simultaneamente. Atualmente, existem várias arquiteturas computacionais, como as CPUs *multicore*, GPUs e APUs, contendo dezenas ou até milhares de núcleos de processamento, que utilizam o conceito de processamento paralelo. Em meio a essas novas arquiteturas computacionais, foi dada mais força a programação paralela: para obter o melhor desempenho possível, os programas de computador devem utilizar o máximo de núcleos de processamento disponíveis simultaneamente [16].

A programação paralela é ainda mais importante no desenvolvimento de aplicações científicas. Muitas destas necessitam realizar o processamento de uma grande quantidade de dados. Por vezes, esse processamento pode ser feito de forma independente, *i.e.* os dados podem ser processados paralelamente. Este paralelismo é conhecido como paralelismo de dados. Para estas aplicações, o uso de aceleradores, como GPUs, pode levar a uma grande melhoria no desempenho pela redução no tempo de computação dos dados [16]. Como consequência, simulações com maior demanda de processamento podem ser realizadas em menos tempo, o que pode contribuir para que respostas a questões científicas relacionadas às simulações sejam obtidas mais rapidamente.

A fim de melhorar ainda mais o desempenho, é possível usar simultaneamente vários dispositivos para executar um programa paralelo, aproveitando ao máximo todos os recursos computacionais disponíveis. Por exemplo, é comum vermos computadores contendo CPUs e GPUs. No contexto de alguns *frameworks* para computação paralela, a única função da CPU é enviar o trabalho para que a GPU faça a sua execução em paralelo, ou seja, a CPU fica em um estado ocioso durante toda a execução pela GPU. É possível utilizar as CPUs para executar este trabalho, distribuindo parte dele entre seus múltiplos núcleos de processamento, melhorando o desempenho da aplicação.

Entretanto, existe uma grande dificuldade encontrada nesta abordagem: como os dispositivos executam em velocidades distintas, é possível que alguns fiquem em um estado ocioso enquanto outros, mais lentos, finalizam sua execução. Portanto, é importante

determinar a quantidade de trabalho que cada dispositivo receberá para processar através de esquemas de balanceamento de carga de trabalho.

1.2 Objetivo

O objetivo principal deste trabalho é demonstrar que é possível obter ganhos de desempenho ao executar uma aplicação paralela em um ambiente híbrido de memória distribuída, utilizando todos os recursos computacionais disponíveis para a computação. A hipótese deste trabalho é que o uso simultâneo de todos os recursos computacionais para resolver um problema baseado em paralelismo de dados reduz o tempo de computação, quando comparado a outras combinações de uso dos mesmos recursos, como o uso apenas das CPUs na computação ou o uso de CPUs e de GPUs de modo intercalado.

Além disso, pretende-se demonstrar que os eventuais ganhos de desempenho são uma consequência dos esquemas de balanceamento de carga que serão utilizados, ou seja, o simples uso simultâneo de todos os recursos computacionais disponíveis não garante uma redução no tempo de computação em relação à abordagem que usa CPUs e GPUs de modo intercalado. O esquema de balanceamento de carga permite que todos os recursos computacionais do sistema de memória distribuída sejam aproveitados de forma que o tempo que cada dispositivo leva para executar a sua carga seja aproximadamente igual.

1.3 Método

Este trabalho implementa uma versão paralela de um modelo computacional do Sistema Imune Humano (SIH), utilizando para isso um esquema de balanceamento de carga onde a quantidade de trabalho de cada dispositivo é determinado em tempo de execução. O objetivo do balanceamento de carga é equalizar o tempo que cada dispositivo leva para computar os seus dados, evitando assim que fiquem ociosos enquanto ainda existem dados a serem processados por outros dispositivos. O desenvolvimento do código utilizou o *framework* OpenCL [16], pois este possibilita que um mesmo código seja compilado e executado em diversas arquiteturas computacionais, *i.e.* um mesmo código pode ser executado em dispositivos heterogêneos. Foram feitos testes em um *cluster* heterogêneo, *i.e.* dispositivos de características distintas (CPUs e GPUs) estão presentes em um mesmo computador, e vários destes computadores são conectados por meio de uma rede rápida de transmissão de dados. Para a troca de mensagens nesta rede, é utilizada a biblioteca OpenMPI.

O SIH é responsável pela defesa imunológica do corpo humano, ou seja, o seu objetivo é detectar a presença de organismos invasores e recrutar células e moléculas para o local onde os mesmos estão presentes para que possam ser neutralizados. Pigozzo [23] implementou inicialmente o modelo computacional $1D$, que descreve toda a dinâmica do

processo de reação imunológica do organismo à presença de um antígeno, representado por uma molécula presente na membrana de bactérias, o lipopolissacarídeo(LPS). A versão inicial do SIH foi desenvolvida para execução sequencial, *i.e.* apenas um dos núcleos de processamento da CPU era usado para executar o modelo computacional, resultando em horas ou até mesmo dias de computação, dependendo dos parâmetros utilizados na simulação. O simulador do SIH usado neste trabalho é baseado em uma versão $3D$ paralela, desenvolvida inicialmente em um segundo trabalho [21], mas que utiliza CPUs e GPUs de modo intercalado na computação.

Dois esquemas de balanceamento de carga foram implementados neste trabalho. O primeiro usa um intervalo inicial de passos de tempo para calcular as cargas que cada dispositivo irá receber, e essa distribuição de cargas é mantida até o final da execução. Este esquema é chamado de balanceamento de carga estático (BCE). O segundo esquema realiza atualizações na distribuição de carga ao longo da simulação, ou seja, de tempo em tempo as cargas de tabalho de cada dispositivo são recalculadas e, eventualmente, uma redistribuição das cargas é realizada. O objetivo desta abordagem é corrigir eventuais desbalanceamentos que venham a ocorrer durante a simulação. Este esquema é chamado de balanceamento de carga dinâmico (BCD).

Foram desenvolvidas versões distintas do simulador do SIH, sem o uso de balanceamento de carga, com uso do BCE e com o uso de BCD. As versões foram executadas em um ambiente híbrido de memória distribuída. Foram coletados os tempos de execução de cada uma das versões quando executadas em diversas combinações de uso dos recursos computacionais disponíveis, como o uso de CPUs e de GPUs de modo intercalado e simultâneo. Os tempos são então analisados para verificar se a hipótese de trabalho é verdadeira.

1.4 Terminologia

Existem muito trabalhos que abordam computação paralela em ambientes heterogêneos. É importante estabelecer uma definição para os termos que serão utilizados nesta dissertação, visto que um mesmo conceito pode ser definido por termos diferentes na literatura.

O termo computação heterogênea, usado neste trabalho para se referir a uma computação realizada simultaneamente em distintos dispositivos como CPUs e GPUs, é definido de diferentes formas na literatura[15], entres eles: computação colaborativa, híbrida, cooperativa ou sinergética, coprocessamento, método de divisão e conquista, *etc.*

GPUs são frequentemente denominadas na literatura como aceleradores, mas também podem ser chamadas de unidades de processamento ou dispositivos[15]. Quanto às CPUs, elas são frequentemente chamadas como *hosts*, onde seu único propósito é enviar

tarefas para as GPUs, entretanto, elas também podem ser utilizadas na computação. Neste caso, as CPUs também podem ser chamadas de aceleradores[15]. Neste trabalho o termo dispositivo será utilizado para referenciar GPUs e CPUs.

Como o poder de processamento das CPUs é muitas vezes inferior ao das GPUs em aplicações que utilizam o paralelismo de dados, ao utilizá-los em conjunto é necessário estabelecer uma quantidade de dados inferior para ser calculada pelas CPUs em relação às GPUs. Neste trabalho, o esquema usado para determinar a quantidade de dados que cada dispositivo recebe para processar é denominado balanceamento de carga, sendo realizado em tempo de execução e que pode ser de dois tipos: estático, onde as cargas são calculadas em um estágio inicial da computação e mantidas até o final da mesma; ou dinâmico, onde as cargas são calculadas iterativamente durante toda a computação.

1.5 Organização

O restante deste trabalho é organizado da seguinte forma. O capítulo 2 apresenta alguns trabalhos relacionados ao tópico desta pesquisa, assim como trata das diferenças e semelhanças a este trabalho. O capítulo 3 apresenta a fundamentação teórica necessária para o entendimento deste trabalho, descrevendo o modelo biológico do SIH, assim como o modelo matemático usado para descrever o fenômeno de resposta imunológica. Em seguida, são apresentados os métodos numéricos usados para resolver o modelo matemático. Também são descritas as bibliotecas de programação usadas para a implementação paralela do SIH, assim como as dificuldades em adaptar essas bibliotecas para resolver o problema proposto. O capítulo 4 apresenta a implementação do SIH para dispositivos homogêneos, onde as cargas são igualmente distribuídas entre os mesmos, em seguida, todos os ajustes envolvidos na implementação deste modelo para possibilitar a execução em dispositivos heterogêneos é apresentada, bem como os esquemas de balanceamento de carga estático e dinâmico, assim como a estrutura de dados e as particularidades envolvidas na implementação da mesma. O capítulo 5 apresenta os resultados obtidos usando o simulador do SIH como *benchmark*, assim como a análise destes resultados. Finalmente, o capítulo 6 apresenta as conclusões sobre os esquemas de balanceamento de carga propostos e planos para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma breve revisão bibliográfica de trabalhos encontrados na literatura que versam sobre temas correlatos ao desta dissertação.

Esquemas de balanceamento de carga foram amplamente abordados na literatura, entretanto, algumas distinções podem ser destacadas em relação a este trabalho, entre elas, o tipo de paralelismo adotado, assim como particularidades nos esquemas de balanceamento propostos. Os trabalhos são então apresentados em seções que buscam destacar aspectos que os distinguem das escolhas realizadas nesta dissertação.

2.1 Paralelismo de Tarefa e Dados

No contexto da programação paralela, dois tipos distintos de paralelismo são amplamente utilizados no desenvolvimento de aplicações. No primeiro, denominado paralelismo de tarefas, tarefas constituídas por diferentes conjuntos de instruções são executadas simultaneamente. Um exemplo de paralelismo de tarefas é o mecanismo chamado de *pipeline*. Neste uma aplicação é dividida em tarefas ou estágios, e cada estágio produz um resultado parcial que é repassado para o estágio seguinte. O resultado de um processamento está disponível após passar por todos os estágios. Como os estágios são independentes entre si e executam em paralelo, podem computar vários dados simultaneamente, um em cada estágio. No segundo tipo de paralelismo, conhecido como paralelismo de dados, um mesmo conjunto de instruções é executado para dados diferentes. Os dados são divididos entre as unidades de processamento, que executam um mesmo conjunto de instruções para seu subconjunto de dados [16].

Uma heurística de balanceamento de carga em sistemas heterogêneos baseada em paralelismo de tarefa é descrita no trabalho de Piyush *et al.* [13]. Os autores apresentam um modelo no qual processos constituem um programa, sendo que tais processos podem ou não ser executados independentemente uns dos outros. Por exemplo, quando um processo B depende do processo A, a execução do processo B é bloqueada até que uma resposta do processo A seja enviada ao processo B, para que o mesmo retome sua execução. O algoritmo balanceador proposto [13] constrói um grafo de dependências em tempo de execução, onde os processos que não possuem dependências são colocados no topo da lista de prioridade de execução. Dois *benchmarks* são utilizados para testar o método proposto de balanceamento de carga.

Nesta dissertação dois algoritmos distintos para balanceamento de carga foram desenvolvidos para aplicações com paralelismo de dados executadas em ambientes heterogêneos e avaliadas com uma aplicação que simula o funcionamento do SIH.

2.2 Atribuição de Diferentes Funções para Dispositivos Heterogêneos

Alguns esquemas de balanceamento de carga para dispositivos heterogêneos podem também atribuir diferentes funções para cada dispositivo na computação. Podemos observar esta diferenciação no trabalho de Panetta *et al.* [20], que atribui papéis diferentes para as CPUs e GPUs na computação.

Os autores abordam o problema da migração de *Kirchhoff*, que envolve o processamento de dados espaciais adquiridos no leito do oceano. Uma fonte emite sinais de onda, que refletem em diferentes níveis do subterrâneo marinho e são captados por receptores. O conjunto de sinais, denominado de amostras, deve ser posteriormente processado para mapear de forma consistente o subterrâneo, gerando imagens. Tal processo é denominado Migração Sísmica[20].

De acordo com os autores, o melhor desempenho em GPUs ocorre quando há um conjunto igual de operações sendo executadas para distintos dados por cada *thread*, como em uma máquina vetorial. Portanto, para obter uma boa eficiência na computação, é necessário atribuir *threads* idênticas de uma instância do *loop* que implementa a migração de *Kirchhoff* em cada GPU, enquanto que as CPUs se encarregam das operações de filtrar os dados de entrada, passo anterior à execução do *loop* de migração. Os resultados mostraram que dividir a computação entre CPUs e GPUs resulta em um aumento de desempenho, quando comparado à execução apenas na CPU do sistema. Também é observado que o processo de filtragem das amostras, que não pode ser paralelizado, passou a representar uma porção muito maior do tempo de execução.

Dois tipos distintos de computação são portanto atribuídos à CPUs e GPUs. O processo de filtragem das amostras, realizado pela CPU, é menos intenso e não é paralelizável. Já o processo de computação dos *loops* de migração é altamente paralelizável e executada nas GPUs. Diferente desse trabalho, nesta dissertação as mesmas operações são realizadas simultaneamente pelas CPUs e GPUs.

2.3 Computação Homogênea e Heterogênea

Outro fator a se considerar na programação paralela de múltiplos dispositivos é a arquitetura dos mesmos. É trivial distribuir a carga entre dispositivos com as mesmas capacidades computacionais. Entretanto, o uso de dispositivos com diferentes capacidades computacionais impõe dificuldades para alcançar bons desempenhos na execução da aplicação, já que dispositivos com maior capacidade de computação levam menos tempo para processar, entrando em um estado ocioso enquanto aguardam os dispositivos com menor capacidade de computação terminarem seu processamento. É portanto necessário balancear a carga computada quando os dispositivos presentes são heterogêneos.

Borelli *et al.* [5] apresentam uma implementação paralela em GPUs do método de

busca exaustiva para o problema de inferência de Redes Regulatórias de Genes (GRN). O método de busca exaustiva é executado nas GPUs e sua função é retornar um valor de *fitness* para cada instância do problema, onde cada instância é representada por um conjunto de genes cujos valores de expressão influenciam o gene alvo escolhido. O valor de expressão de cada gene pode ser binário (0 ou 1) ou ternário (valores negativos, nulos ou positivos). No trabalho, foi usado o *Mean Conditional Entropy* (MCE) como função de critério da busca exaustiva. Como existe um conjunto grande de genes alvo, foi determinada pelos autores a separação destes genes em grupos que, em seguida, são designados para diferentes blocos de *threads* das GPUs, onde cada *thread* de um bloco é responsável por processar um gene alvo de seu bloco. O objetivo desta abordagem é fazer uso da memória compartilhada dos dispositivos, já que a latência é menor do que a memória global.

Os resultados mostraram que:

- Uma quantidade maior de genes alvo processada por bloco de GPU resulta em tempos melhores de execução. Isto ocorre já que é processada uma quantidade maior de dados por vez, o que minimiza o acesso à memória global e maximiza o uso da memória compartilhada;
- O algoritmo que executa em uma única GPU teve desempenho melhor do que a versão que executou nos núcleos de processamento da CPU;
- O ganho de desempenho da versão que executa em múltiplas GPUs, quando comparado com a versão que executa nos núcleos de processamento da CPU, é alto, principalmente quando o número de genes do sistema é grande, pois os recursos computacionais de cada dispositivo são melhor aproveitados;
- Ao comparar a versão de valores de expressão binária com a versão de valores de expressão ternária, a versão binária tem um desempenho melhor, o que se deve ao fato de que os registradores da GPU são usados mais intensamente, limitando a quantidade de *threads* que executam simultaneamente em cada bloco do dispositivo.

O problema tratado por Borelli *et al.* não possui qualquer dependência temporal. Consequentemente, o conjunto de dados pode ser distribuído entre os dispositivos e ao término da execução, o resultado final é obtido, não sendo necessário portanto qualquer tipo de sincronização ou troca de dados entre os dispositivos envolvidos na computação.

Outro trabalho que trata da obtenção de soluções exatas para o problema de GNR, que não apresenta dependências tanto espaciais quanto temporais, é proposto por Danilo *et al.* [3], utilizando o método combinatório *Hitting Set Problem*(HSP). O autor propõe um algoritmo que implementa o HSP em um problema que contêm milhares de variáveis (genes) em paralelo, eliminando problemas que impactavam a escalabilidade do

mesmo e, portanto, os ganhos de desempenho obtidos quando o número de dispositivos que participam da computação aumenta. O HSP é um problema combinatório. Nele, se deseja encontrar um subconjunto solução $S \subset P$, não nulo e de cardinalidade mínima onde, para cada subconjunto $C_i \subset P$, ao menos um elemento $s_i \in S$ pertença à C_i, onde P é o conjunto de elementos do problema. No contexto do GRN, a solução representa o conjunto mínimo de genes cujo valor de expressão afete um gene alvo específico. Para obter o conjunto solução, deve-se realizar testes combinatórios de várias instâncias, onde cada uma representa uma lista de genes. Como as instâncias são independentes umas das outras e existem milhões delas, é possível realizar a computação em paralelo para obter um bom desempenho da aplicação.

Nos experimentos realizados pelos autores, é utilizada a computação heterogênea em um ambiente com GPUs de arquiteturas computacionais distintas. Para realizar a distribuição do trabalho entre os dispositivos, é usado neste trabalho um esquema mestre-escravo, onde o processo mestre, representado por um núcleo de processamento da CPU, é responsável por enviar tarefas aos processos escravos, representados pelas GPUs. As instâncias usadas para os testes combinatórios são geradas pelos próprios dispositivos, dessa forma, a comunicação entre *host* e dispositivo é reduzida, o que aumenta os ganhos de desempenho. Como não existem dependências no cálculo combinatório, a comunicação entre dispositivos é negligível. Finalmente, foi possível obter ganhos de desempenho próximos aos lineares, conforme o número de dispositivos usados na computação cresce.

Podemos também citar outro trabalho[26] que apresenta um balanceador de carga dinâmico capaz de ajustar as cargas, usando para isto um modelo composto de um sistema de equações não lineares. Para validar o modelo proposto, os autores usaram *benchmarks* de aplicações de álgebra linear (multiplicação de matrizes), mercado de ações (método *Black-Scholes*) e bioinformática (inferência de redes regulatórias de genes GRN), em um ambiente híbrido de memória distribuída, composto por GPUs e CPUs. Os autores desenvolveram o algoritmo na liguagem C, usando o *framework* StarPU [1] para realizar a distribuição de cargas entre os dispositivos. Nos testes realizados pelos autores, foi concluído que o algoritmo de balanceamento de carga proposto diminui o tempo de computação das aplicações, e quando comparado com outros algoritmos dinâmicos, os maiores ganhos foram observados com uma quantidade grande de dispositivos envolvidos na computação, assim como uma quantidade superior de carga computada pelos mesmos.

2.4 Balanceamento Estático e Dinâmico

Não é trivial a tarefa de implementar um modelo computacional paralelo para um sistema composto por dispositivos heterogêneos, visto que com capacidades computacionais distintas, cada dispositivo realiza a computação em tempos diferentes. A decisão sobre o uso do esquema de balanceamento de carga estático ou dinâmico é, portanto, muito

importante para obter o máximo possível de desempenho. Apesar de ajustar as diferenças no tempo que cada dispositivo leva para computar suas cargas durante toda a simulação, o esquema de balanceamento de carga dinâmico (BCD) requer uma frequente troca de informações entre dispositivos. Analogamente, o esquema de balanceamento de carga estático (BCE) realiza o cálculo e distribuição das cargas uma única vez, o que diminui o gargalo de comunicação.

Grande parte das aplicações científicas realizam a simulação de um fenômeno da natureza, utilizando equações matemáticas para descrevê-las. Tais equações nos permitem descrever a resposta imunológica do nosso corpo, a propagação elétrica pelas células do coração ou até mesmo estudar como doenças infecciosas se propagam em uma determinada população.

Alguns modelos computacionais, como o usado neste trabalho, utilizam equações parabólicas, e o algoritmo do cálculo numérico de tais equações exigem que sejam implementadas dois tipos de *loops*, o interno e o externo. O *loop* externo se refere ao tempo de simulação, onde o número de iterações representa a quantidade de passos de tempo concluídos. Já o *loop* interno se refere a toda a dinâmica entre as populações do modelo no domínio onde a simulação é realizada. Esta dinâmica é representada computacionalmente por instruções aritméticas.

Para algumas aplicações com características semelhantes a do SIH, o tempo necessário para executar as instruções do *loop* interno pode variar; aplicações que apresentam esta característica são chamadas neste trabalho de irregulares. Já em outras aplicações o tempo para executar as instruções no *loop* interno é sempre o mesmo, motivo pelo qual estas aplicações são chamadas neste trabalho de regulares. Esta característica impacta diretamente o balanceamento de carga, já que aplicações irregulares apresentam tempos de execução que variam durante a simulação, ou seja, um esquema de BCE não será capaz de determinar uma carga ideal para cada dispositivo, sendo necessários ajustes ao longo da simulação. Tais ajustes só são possíveis utilizando um esquema de BCD. Consequentemente, o esquema de BCE é mais apropriado para aplicações regulares.

O trabalho de Agulleiro *et al.* [8] trata de uma técnica de BCD proposta para reconstrução tomográfica 3D, onde um volume tridimensional é particionado em várias fatias bidimensionais. Estas fatias podem então ser computadas em paralelo tanto em GPUs quanto em CPUs utilizando um método de reconstrução. A técnica proposta consiste em iniciar *threads* para computar em paralelo as fatias bidimensionais usando para isso os núcleos de processamento da CPU, enquanto outras *threads* da CPU se encarregam de enviar tarefas, que implementam o algoritmo de reconstrução, para execução nas GPUs do sistema. O código paralelo executado nas CPUs foi desenvolvido em POSIX Threads (PThreads) [11], enquanto o código paralelo executado nas GPUs foi desenvolvido em CUDA[25]. As tarefas consistem no envio de dados bidimensionais da memória principal

para memória da GPU, sua computação, e posterior cópia dos dados processados da memória da GPU para a memória principal. Quando uma *thread* termina de processar uma fatia, ela se torna ociosa, e mais trabalho é enviado para a mesma, caraterizando uma técnica de balanceamento dinâmico, onde trabalhos são enviados na medida em que dispositivos se tornam ociosos. Para minimizar o custo adicional de comunicação entre a CPU e as GPUs ao enviar tarefas e também para aproveitar os recursos computacionais dos dispositivos ao máximo, as fatias bidimensionais são enviadas em pacotes, denominados *slats*. Como as GPUs geralmente processam os *slats* mais rapidamente que as CPUs, é enviada uma carga maior de *slats* para as primeiras. Segundo os autores, o tamanho dos *slats* enviados por vez para GPUs e CPUs não muda durante a execução do programa, diferentemente do que ocorre na proposta que será apresentada ao longo desta dissertação.

Lu *et al.* [12] abordam o uso de um esquema estático para computação heterogênea do Modelo de Transferência de Radiação Rápida (RRTM), utilizado no campo de pesquisa de radiação física, um dos pincipais processos atmosféricos físicos. No Modelo RRTM, são realizadas iterações em uma malha bidimensional que representa as informações atmosféricas. Cada ponto da malha depende apenas de informações dos vizinhos, tornando possível dividi-la em torno dos eixos x e y em várias sub-malhas e designá-las para serem processadas em paralelo em dispositivos distintos. O uso de dispositivos distintos motiva a utilização da técnica de balanceamento de carga. Os autores listam três possíveis formas para executar a aplicação com o uso dos diferentes dispositivos. Na primeira, CPUs preparam os dados para serem executados paralelamente pelas GPUs. Na segunda, CPUs e GPUs realizam a mesma tarefa de preparar e computar diferentes dados. Na terceira, CPUs e GPUs trabalham de forma cooperativa para processar o mesmo conjunto de dados. A terceira opção foi escolhida pelos autores, e a divisão de cargas é determinada uma única vez a partir da análise do poder de processamento de cada dispositivo, *i.e.* o balanceamento é estático. Como se faz necessário obter a informação dos pontos da malha vizinhos aos computados por cada núcleo de processamento de um determinado dispositivo, é necessário realizar a comunicação entre os dispositivos. Isso é feito utilizando o *Message Passing Interface* (MPI) [19]. A paralelização do código é feita utilizando OpenMP [19] para CPUs e CUDA para GPUs. Durante a execução, é gerado um processo para cada CPU, responsável por designar tarefas para processar uma determinada porção da malha utilizando OpenMP, e também enviar um *kernel* CUDA para computar a porção da malha designada às GPUs.

Outro exemplo de uma abordagem que emprega BCE é o trabalho de Bernabe *et al.* [2], que paraleliza uma implementação do algoritmo *3D Fast Wavelet Transform* (3D-FWT) em dispositivos heterogêneos. O 3D-FWT é um algoritmo que consiste na obtenção de uma soma de funções, também chamadas de *wavelets*, que representam um conjunto de pontos discretos. A implementação consiste em alocar na memória do *host* os *frames* de um vídeo armazenado em arquivo, que em seguida são encaminhados para

os dispositivos para que se possa processar o *1D-FWT* na dimensão do tempo. Cada dispositivo processa quatro *pixels* de cada *frame*, onde os *frames* do vídeo são processados em passos de quatro *frames*. Em seguida, é aplicado o 2D-FWT nas dimensões de largura e altura de cada *pixel*. Para calcular a proporção de *pixels* que cada dispositivo irá processar, é primeiramente executada uma amostra idêntica de *frames* em todos os dispositivos. O resultado desta amostra de desempenho irá determinar a carga que cada dispositivo receberá para calcular o 3D-FWT. A amostra de desempenho é feita através de um *profile* em cada dispositivo que mede as capacidades computacionais destes. A partir do *profile*, são determinadas as cargas que serão designadas para processamento em cada um. Para CPUs, os *kernels* utilizados para processamento são implementados em PThreads, já os *kernels* das GPUs podem ser implementados utilizando CUDA para GPUs da NVIDIA e OpenCL[16] para GPUs Radeon.

Os trabalhos abordados nesta seção utilizam apenas um dos esquemas de balanceamento de carga, enquanto que nesta dissertação ambos os esquemas são estudados a fim de analisar os impactos de cada um no desempenho da aplicação estudada.

2.5 Resumo do Capítulo

Neste capítulo, foram abordados alguns trabalhos que utilizam computação heterogênea. Procurou-se destacar aspectos que os distinguem das escolhas realizadas nesta dissertação. Os trabalhos mencionados neste capítulo se diferenciam pelo paralelismo que usam, que pode ser de tarefa ou de dados; pela forma com que os dispositivos heterogêneos cooperam para executar a tarefa; pelo uso de arquiteturas homogêneas ou heterogêneas na computação; o esquema de balanceamento usado, que pode ser estático ou dinâmico. Os trabalhos são resumidos na Tabela 1.

Tabela 1 – Principais características dos trabalhos apresentados neste capítulo.

	Paralelismo de dados	Dispositivos realizam mesma função	Arquiteturas heterogêneas	Balanceamento
Piyush *et al.* [13]	Não	Sim	Sim	Dinâmico
Panetta *et al.* [20]	Sim	Não	Sim	Estático
Borelli *et al.* [5]	Sim	Sim	Não	Não implementa
Danilo *et al.* [3]	Sim	Sim	Sim	Dinâmico
Luis *et al.* [3]	Sim	Sim	Sim	Dinâmico
Agulleiro *et al.* [8]	Sim	Sim	Sim	Dinâmico
Lu *et al.* [12]	Sim	Sim	Sim	Estático
Bernabe *et al.* [2]	Sim	Sim	Sim	Estático

3 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os fundamentos teóricos necessários para o entendimento do restante deste trabalho. São destacados o funcionamento do sistema imunológico, o modelo matemático utilizado para reproduzi-lo, os métodos numéricos utilizados na implementação do modelo matemático e as ferramentas utilizadas na sua paralelização.

3.1 Sistema Imune

O sistema imune é composto por uma grande gama de moléculas, células, tecidos e órgãos responsáveis pela defesa do organismo. Existem dois tipos de resposta, inata e adaptativa. O sistema imune inato é a primeira linha de defesa do organismo, composta de barreiras físicas e químicas, além de moléculas e células de defesa. Um exemplo de mecanismo de barreira física é o tecido epitelial, que forma uma barreira capaz de impedir a entrada de invasores que possam causar danos ao nosso organismo. A saliva e o ácido clorídrico do estômago funcionam como barreiras químicas. Entretanto, é comum que organismos como vírus, bactérias, fungos e outros consigam passar por estas barreiras. Células de defesa como macrófagos e neutrófilos são então acionados para realizar a eliminação dos agentes patogênicos. Trata-se de uma resposta genérica, que independe do patógeno. Já o mecanismo adaptativo produz uma defesa especifica para combater um patógeno em particular. Este sistema de defesa, entretanto, não é ativado de imediato. O início de sua atuação no combate à doenças pode levar semanas para se iniciar, dependendo do organismo. Apesar da demora na sua ativação, o sistema imune adaptativo é capaz de combater e neutralizar os agentes patogênicos com eficácia. Após a neutralização, o sistema adaptativo é capaz de aperfeiçoar seus mecanismos de defesa contra o agente combatido de forma que uma posterior infecção pelo mesmo organismo seja rapidamente reconhecida e neutralizada. Essa habilidade que o sistema imune adaptativo tem de reconhecer agentes patogênicos é denominada memória imunológica [21].

O foco desta seção é representar alguns dos mecanismos de defesa do sistema imune inato que atuam no combate a um patógeno genérico. Neste trabalho o patógeno é representado pela endotoxina lipopolissacarídeo.

3.1.1 Lipopolissacarídeos

O lipopolissacarídeo, ou LPS, é uma endotoxina imuno-estimulante encontrada na membrana que envolve alguns tipos de bactérias. No processo de combate destas bactérias pelo sistema imune inato, o LPS é liberado no organismo, provocando uma reação inflamatória intensa. Esta reação inflamatória é resultante do processo de recrutamento de outras células e microorganismos do sistema imune inato para o local de infecção,

que precisam ser ativadas para neutralizar a endotoxina. Os mesmos serão descritos nas próximas seções [21].

3.1.2 Leucócitos

Também chamados de glóbulos brancos, os leucócitos são um grupo de células que têm origem na medula óssea e circulam na corrente sanguínea. Alguns desses tipos de células podem extravasar da corrente sanguínea e, em seguida, migrar para o local de infecção, sendo este último processo denominado quimiotaxia. Em especial, neutrófilos, macrófagos e monócitos são os leucócitos que atuam na defesa do organismo, realizando a fagocitose de organismos invasores, substâncias tóxicas e células mortas. Após sua atuação no combate a organismos invasores, essas células morrem, em um processo denominado apoptose [21].

3.1.3 Neutrófilos

Os neutrófilos são os leucócitos mais abundantes na corrente sanguínea. Sua função é localizar organismos invasores e realizar a sua fagocitose. Após a fagocitose, os neutrófilos iniciam o processo de apoptose, tornando-se neutrófilos apoptóticos, processo esse que ocorre em um período de alguns dias. Uma outra função dos neutrófilos é a produção dos grânulos protéicos, de grande importância para o processo de defesa imunológica do organismo. [21]

3.1.4 Grânulos Protéicos

Como mencionado, os grânulos protéicos são substâncias produzidas pelos neutrófilos. Esta substância é liberada enquanto os neutrófilos percorrem a corrente sanguínea até o local de infecção. Sua principal função é a adesão monocítica, ou seja, são responsáveis por atrair os monócitos, que circulam nos vasos sanguíneos, e também de retê-los na região de infecção, onde podem contribuir para a neutralização dos antígenos. [21]

3.1.5 Citocinas

As citocinas são proteínas secretadas por algumas células do sistema imune inato que possuem a função de alterar a concentração de células que atuam nos locais de infecção. Elas podem ser classificadas em dois tipos: citocinas pró-inflamatórias e anti-inflamatórias. As citocinas pró-inflamatórias, produzidas pelos macrófagos, possuem a função de aumentar a permeabilidade dos vasos sanguíneos, o que consequentemente permite o recrutamento de mais macrófagos e também neutrófilos ao local de infecção.

As citocinas também estão ligadas indiretamente à produção de quimiocinas, pois elas contribuem para a ativação dos neutrófilos, capazes de produzir a quimiocina IL-8,

contribuindo finalmente para a ativação de novas células importantes no processo de resposta imunológica, já que possuem propriedades pró-inflamatórias.

Os neutrófilos apoptóticos, na presença de macrófagos, são capazes de produzir citocinas anti-inflamatórias. Estas citocinas tem o papel de inibir a produção de citocinas pró-inflamatórias, com o intuito de regular o fluxo de células do sistema imune para o local de infecção. [21]

3.1.6 Quimiocinas

As quimiocinas são pequenas citocinas, ou proteínas de ativação, que contribuem para o processo de recrutamento de células do sistema imune pelo processo de quimiotaxia. Células como neutrófilos e monócitos possuem receptores de quimiocinas, o que permitem a elas detectar a presença de quimiocinas, consequentemente as direcionando ao local de infecção.

Durante uma resposta imunológica, a produção de quimiocinas contribui para o recrutamento de novas células no combate à infecções, possuindo portanto propriedades pró-inflamatórias, ou, durante processos naturais de desenvolvimento ou produção de tecidos, elas podem regular a migração de células a estes locais, possuindo portanto propriedades homeostáticas. [21]

3.1.7 Macrófagos

Macrófagos são células originárias do processo de diferenciação dos monócitos. Os monócitos presentes na corrente sanguínea, ao serem atraídos para o local de infecção por meio de atração química, se transformam em macrófagos. O processo de quimiotaxia é realizado pela produção de quimiocinas e citocinas liberadas no meio de infecção.

O termo macrófago tem origem grega e significa “grande comedor”, já que são células grandes que realizam o processo de fagocitose de células invasoras. Eles possuem tempo de vida longo quando comparados aos neutrófilos, pois permanecem no organismo por meses ao contrário de dias, sempre em alerta para detectar e neutralizar a invasão de patógenos, sendo portanto as sentinelas do nosso organismo.

Os macrófagos, assim com outras células do SIH, possuem em sua membrana receptores de reconhecimento de padrões (PRR - *Pattern Recognition Receptors*). Por meio destes receptores, é possível o reconhecimento de várias células e moléculas que não pertencem ao nosso organismo e precisam ser eliminadas. O processo de reconhecimento é realizado por meio da identificação de padrões moleculares associados à patógenos (PAMP - *Pathogen Associates Molecular Pattern*) pelos receptores. Por exemplo, moléculas como o LPS possuem um padrão molecular capaz de ser reconhecido pelos PRRs.

Existem dois estados em que podemos encontrar os macrófagos, em repouso, no qual

estas células permanecem imóveis e realizam funções de limpeza do nosso organismo, como a eliminação de restos celulares, ou elas podem se tornar ativas e móveis, estimuladas pela presença de quimiocinas e citocinas, onde desempenham papel fundamental na fagocitose e eliminação de organismos e substâncias invasoras.

A Figura 1 apresenta, de forma esquemática, o relacionamento entre todas as células, moléculas e antígeno apresentados ao longo desta seção. Os macrófagos em repouso, na presença do LPS, fagocitam o mesmo, ao mesmo tempo em que se ativam, tornando-se mais móveis e aumentando a sua atuação no combate. Quando em estado de ativação, os macrófagos produzem as citocinas pró-inflamatórias, que aumentam a permeabilidade dos vasos sanguíneos, permitindo que células do sistema imune entrem no tecido e, em seguida, o processo de quimiotaxia permite que tais células migrem para a região de infecção. As citocinas pró-inflamatórias são também produzidas pelos neutrófilos, cuja função também é fagocitar o LPS. Outra função dos neutrófilos é produzir os grânulos protéicos, que aumentam a adesão monocítica. Após fagocitarem o LPS, os neutrófilos sofrem o processo de apoptose, tornando-se neutrófilos apoptóticos, que são fagocitados pelos macrófagos, quando ativados. As citocinas anti-inflamatórias são produzidas pelos macrófagos. Sua função é a de reverter o processo inflamatório, *i.e.* as células e moléculas que foram recrutadas ao local de infecção para combater o antígeno se dispersam [21].

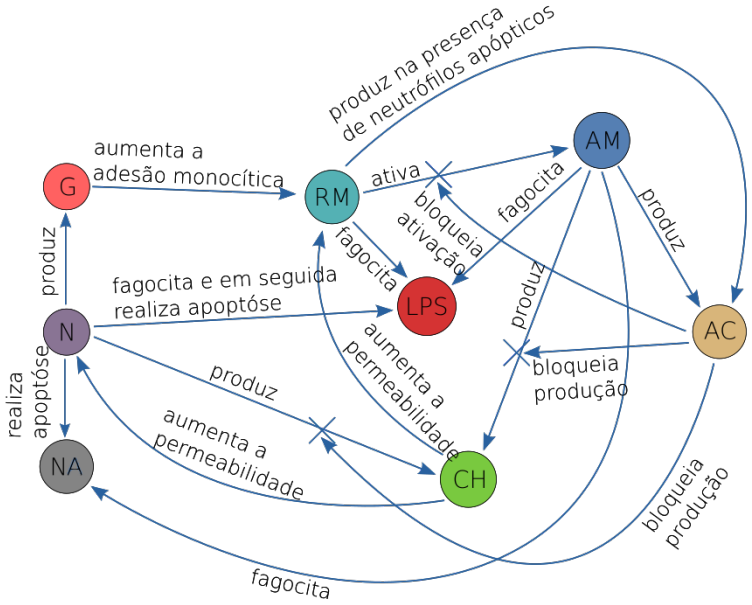

Figura 1 – Diagrama do modelo do Sistema Imune Inato. Adaptado de [23].

3.2 Modelo Matemático

Descrever o comportamento do SIH é uma tarefa complexa, sendo necessários conhecimentos multidisciplinares, principalmente em biologia, física e matemática. Tendo

os detalhes do funcionamento do SIH sido suficientemente compreendidos do ponto de vista biológico, é necessário descrever como as concentrações de cada célula e molécula desse sistema evolui no tecido ao longo do tempo. Nesta dissertação, um modelo matemático de equações diferenciais parciais (EDPs) é usado para descrever as taxas de mudança dessas concentrações ao longo do tempo e do espaço, desde que sejam conhecidas as concentrações iniciais e condições de contorno de cada população do SIH, assim como todos os parâmetros das equações. O conjunto de EDP usado neste trabalho foi desenvolvido originalmente por Pigozzo *et al.* [23] para um domínio 1D e posteriormente estendido por Rocha *et al.* [24] para um domínio 3D.

3.2.1 Lipopolissacarídeo

O lipopolissacarídeo funciona como um antígeno e é representado no modelo matemático pela sigla *LPS*, descrita pela Equação 3.1.

$$
\begin{cases}
\frac{\partial LPS}{\partial t} = dec_{LPS} + fag_{LPS\_N} + fag_{LPS\_MA} + atv\_fag_{LPS\_MR} + dif_{LPS}, t > 0, LPS \in \Omega \\
dec_{LPS} = -(\mu_{LPS} \times LPS), t > 0, LPS \in \Omega \\
fag_{LPS\_N} = -(\lambda_{LPS\_N} N \times LPS), t > 0, LPS \in \Omega \\
fag_{LPS\_MA} = -(\lambda_{LPS\_MA} MA \times LPS), t > 0, LPS \in \Omega \\
atv\_fag_{LPS\_MR} = -(\sigma_{LPS\_MR} MR \times LPS) \left(\frac{1}{1+\gamma_{CA} CA}\right), t > 0, LPS \in \Omega \\
dif_{LPS} = (d_{LPS} \Delta LPS), t > 0, LPS \in \Omega \\
d_{LPS} \nabla LPS \cdot \vec{\eta} = 0, t > 0, LPS \in \partial\Omega \\
LPS(x, y, z, t = 0) = LPS_0(x, y, z), LPS \in \Omega
\end{cases}
\tag{3.1}
$$

O termo dec_{LPS} representa o decaimento do *LPS* no tecido, com respectiva taxa μ_{LPS}. O termo fag_{LPS_N} representa a fagocitose do *LPS* pelos neutrófilos *(N)*, com respectiva taxa λ_{LPS_N}. O termo fag_{LPS_MA} representa a fagocitose do *LPS* pelos macrófagos ativos *(MA)*, com respectiva taxa λ_{LPS_MA}. O termo $atv\_fag_{LPS\_MR}$ representa a ativação dos macrófagos em repouso (MR), seguida pela fagocitose do *LPS*, esse evento ocorre com taxa σ_{LPS_MR}. A presença de citocina anti-inflamatória inibe o recrutamento de macrófagos na região de infecção. O termo γ_{CA} influencia neste fenômeno. Finalmente, o

termo dif_{LPS} representa a difusão de LPS no tecido, com taxa d_{LPS}.

3.2.2 Macrófago em Repouso

Os macrófagos em repouso são representados no modelo matemático pela sigla MR, sendo modelados pela Equação 3.2.

$$
\begin{cases}
\frac{\partial MR}{\partial t} = dec_{MR} + atv_{MR} + flu_{MR} + dif_{MR} + qui_{CH\_MR}, t > 0, MR \in \Omega \\
dec_{MR} = -(\mu_{MR} \times MR), t > 0, MR \in \Omega \\
atv_{MR} = -(\sigma_{LPS\_MR} MR \times LPS) \left(\frac{1}{1+\gamma_{CA} CA}\right), t > 0, MR \in \Omega \\
flu_{MR} = (perm_{MR\_CH} + perm_{MR\_G}) \times (M^{max} - (MR + MA)), t > 0, MR \in \Omega \\
perm_{MR\_CH} = \left((P^{max}_{MR\_CH} - P^{min}_{MR\_CH}) \times \left(\frac{CH}{CH+\eta_{MR\_CH}}\right)\right) + P^{min}_{MR\_CH}, t > 0, MR \in \Omega \\
perm_{MR\_G} = \left((P^{max}_{MR\_G} - P^{min}_{MR\_G}) \times \left(\frac{G}{G+\eta_{MR\_G}}\right)\right) + P^{min}_{MR\_G}, t > 0, MR \in \Omega \\
dif_{MR} = (d_{MR} \Delta MR), t > 0, MR \in \Omega \\
qui_{CH\_MR} = -q_{CH\_MR}(\nabla(MR \nabla CH)), t > 0, MR \in \Omega \\
(d_{MR} \nabla MR - q_{CH\_MR} MR \nabla CH) \cdot \vec{\eta} = 0, t > 0, MR \in \partial\Omega \\
MR(x, y, z, t = 0) = MR_0(x, y, z), MR \in \Omega
\end{cases}
\tag{3.2}
$$

O termo dec_{MR} representa o decaimento do MR no tecido, com respectiva taxa μ_{MR}. O termo atv_{MR} representa a ativação do MR em contato com o LPS, o qual se torna MA. O termo de ativação com encontro ocorre com taxa σ_{LPS_MR}, como na Equação 3.1. O γ_{CA} determina a inibição da ativação do MR. O termo flu_{MR} representa o fluxo de MR para o tecido, onde M^{max} determina o limite de macrófagos suportados no tecido. Os termos $perm_{MR\_CH}$ e $perm_{MR\_G}$ definem a permeabilidade do endotélio, controlado pelas citocinas pró-inflamatórias (CH) e pelos grânulos protéicos (G). As taxas η_{MR_CH} e η_{MR_G} definem o quanto as respectivas concentrações de CH e G contribuem para a variação entre permeabilidade máxima e mínima do endotélio, definidas por $P^{max}_{MR\_CH}$ $P^{min}_{MR\_CH}$ para CH e $P^{max}_{MR\_G}$ $P^{min}_{MR\_G}$ para G. O termo dif_{MR} representa a difusão de MR no tecido, com taxa d_{MR}. Finalmente, o termo qui_{CH_MR} representa a quimiotaxia do

MR na região da infecção, influenciado pela presença de *CH*, com taxa q_{CH_MR}.

3.2.3 Macrófago Ativo

Os macrófagos ativos são representados no modelo matemático pela sigla *MA*, e modelados pela Equação 3.3.

$$\begin{cases} \frac{\partial MA}{\partial t} = dec_{MA} + atv_{MA} + dif_{MA} + qui_{CH\_MA}, t > 0, MA \in \Omega \\ dec_{MA} = -\mu_{MA} \times MA, t > 0, MA \in \Omega \\ atv_{MA} = \sigma_{LPS\_MR} MR \times LPS \left(\frac{1}{1+\gamma_{CA} CA} \right), t > 0, MA \in \Omega \\ dif_{MA} = d_{MA} \Delta MA, t > 0, MA \in \Omega \\ qui_{CH\_MA} = -q_{CH\_MA}(\nabla(MA \nabla CH)), t > 0, MA \in \Omega \\ (d_{MA} \nabla MA - q_{CH\_MA} MA \nabla CH) \cdot \vec{\eta} = 0, t > 0, MA \in \partial\Omega \\ MA(x, y, z, t = 0) = MA_0(x, y, z), MA \in \Omega \end{cases} \tag{3.3}$$

O termo dec_{MA} representa o decaimento do *MA* no tecido, com respectiva taxa μ_{MA}. O termo atv_{MA} representa a ativação do *MR* em contato com o *LPS*, o qual se torna *MA*. O termo de ativação com encontro ocorre com taxa σ_{LPS_MR}, como na Equação 3.1. O γ_{CA} determina a inibição da ativação do *MR*. O termo dif_{MA} representa a difusão de *MA* no tecido, com taxa d_{MA}. Finalmente, o termo qui_{CH_MA} representa a quimiotaxia do *MA* na região da infecção, influenciado pela presença de *CH*, com taxa q_{CH_MA}.

3.2.4 Neutrófilo

A sigla N representa os neutrófilos no modelo matemático, que são modelados pela Equação 3.4.

$$\begin{cases} \frac{\partial N}{\partial t} = dec_N + fag\_apo_{LPS\_N} + flu_N + dif_N + qui_{CH\_N}, t > 0, N \in \Omega \\ dec_N = -(\mu_N \times N), t > 0, N \in \Omega \\ fag\_apo_{LPS\_N} = -(\lambda_{LPS\_N} N \times LPS), t > 0, N \in \Omega \\ flu_N = perm_{N\_CH} \times (N^{max} - N), t > 0, N \in \Omega \\ perm_{N\_CH} = (P^{max}_{N\_CH} - P^{min}_{N\_CH}) \times \left(\frac{CH}{CH+\eta_{N\_CH}}\right) + P^{min}_{N\_CH}, t > 0, N \in \Omega \\ dif_N = (d_N \Delta N), t > 0, N \in \Omega \\ qui_{CH\_N} = -q_{CH\_N}(\nabla(N \nabla CH)), t > 0, N \in \Omega \\ (d_N \nabla N - q_{CH\_N} N \nabla CH) \cdot \vec{\eta} = 0, t > 0, N \in \partial\Omega \\ N(x, y, z, t = 0) = N_0(x, y, z), N \in \Omega \end{cases} \tag{3.4}$$

O termo dec_N representa o decaimento do N no tecido, com respectiva taxa μ_N. O termo $fag\_apo_{LPS\_N}$ representa a apoptose, ou morte celular, decorrente da fagocitose do LPS pelo neutrófilo; esse fenômeno ocorre com taxa λ_{LPS_N}. O termo flu_N representa o fluxo de MR para o tecido, onde N^{max} determina o limite de neutrófilos suportados no tecido. O termo $perm_{N\_CH}$ define a permeabilidade do endotélio, controlado pelas citocinas pró-inflamatórias CH. A taxa η_{MR_CH} define o quanto a concentração de CH contribui para a variação entre permeabilidade mínima e máxima do endotélio, definidas por $P^{max}_{N\_CH}$ e $P^{min}_{N\_CH}$. O termo dif_N representa a difusão de N no tecido, com taxa d_N. Finalmente, o termo qui_{CH_N} representa a quimiotaxia de N na região da infecção, influenciado pela presença de CH, com taxa q_{CH_N}.

3.2.5 Citocina Pró-Inflamatória

A sigla *CH* representa as citocinas pró-inflamatórias no modelo matemático, que são modelados pela Equação 3.5.

$$
\begin{cases}
\frac{\partial CH}{\partial t} = dec_{CH} + prod_{N\_CH} + prod_{MA\_CH} + dif_{CH}, t > 0, CH \in \Omega \\
dec_{CH} = -(\mu_{CH} \times CH), t > 0, CH \in \Omega \\
prod_{N\_CH} = (\beta_{LPS\_N} N \times LPS) \times \left(1 - \frac{CH}{\omega_{CH}}\right) \times \left(\frac{1}{1+\kappa_{CA} CA}\right), t > 0, CH \in \Omega \\
prod_{MA\_CH} = (\beta_{LPS\_MA} MA \times LPS) \times \left(1 - \frac{CH}{\omega_{CH}}\right) \times \left(\frac{1}{1+\kappa_{CA} CA}\right), t > 0, CH \in \Omega \\
dif_{CH} = (d_{CH} \Delta CH), t > 0, CH \in \Omega \\
d_{CH} \nabla CH \cdot \vec{\eta} = 0, t > 0, CH \in \partial\Omega \\
CH(x, y, z, t = 0) = CH_0(x, y, z), CH \in \Omega
\end{cases}
\tag{3.5}
$$

O termo dec_{CH} representa o decaimento do *CH* no tecido, com respectiva taxa μ_{CH}. O termo $prod_{N\_CH}$ representa a produção de *CH* por *N* na presença de *LPS*, com taxa β_{LPS_N}. A taxa ω_{CH} determina a concentração máxima de *CH* suportada na região da infecção, enquanto que a taxa κ_{CA} inibe a produção de *CH* na presença de *CA*. O termo $prod_{MA\_CH}$ representa a produção de *CH* por *MA* na presença de *LPS*, com taxa β_{LPS_MA}. A taxa ω_{CH} determina a concentração máxima de *CH* suportada no tecido, enquanto que a taxa κ_{CA} inibe a produção de *CH* na presença de *CA*. De modo análogo, o termo $prod_{N\_CH}$ representa a produção de *CH* por *N*. O termo dif_{CH} representa a difusão de *CH* no tecido, com taxa d_{CH}.

3.2.6 Neutrófilo Apoptóticos

A sigla *ND* representa neutrófilos apoptóticos no modelo matemático, que são modelados pela Equação 3.6.

$$
\begin{cases}
\frac{\partial ND}{\partial t} = apo_N + fag\_apo_{LPS\_N} + fag_{MA} + dif_{ND}, t > 0, ND \in \Omega \\
apo_N = (\mu_N \times N), t > 0, ND \in \Omega \\
fag\_apo_{LPS\_N} = (\lambda_{LPS\_N} N \times LPS), t > 0, ND \in \Omega \\
fag_{MA} = -(\lambda_{ND\_MA} MA \times ND), t > 0, ND \in \Omega \\
dif_{ND} = (d_{ND} \Delta ND), t > 0, ND \in \Omega \\
d_{ND} \nabla ND \cdot \vec{\eta} = 0, t > 0, ND \in \partial\Omega \\
ND(x, y, z, t = 0) = ND_0(x, y, z), ND \in \Omega
\end{cases}
\tag{3.6}
$$

O termo apo_N representa a apoptose de *N*, controlado pela taxa μ_N. A aptoptose de *N* pode ocorrer naturalmente ou na presença de *LPS*, este fenômeno é representado pelo termo $fag\_apo_{LPS\_N}$ e é afetado pela taxa λ_{LPS_N}. O termo fag_{MA} representa a fagocitose de *ND* por *MA*, controlado pela taxa λ_{ND_MA}. O termo dif_{ND} representa a difusão de *ND* no tecido, com taxa d_{ND}.

3.2.7 Grânulo Protéico

Os grânulos protéicos são representados no modelo matemático pela sigla G, e modelados pela Equação 3.7.

$$\begin{cases} \frac{\partial G}{\partial t} = dec_G + prod_{N\_G} + dif_G, t > 0, G \in \Omega \\ dec_G = -(\mu_G \times G), t > 0, G \in \Omega \\ prod_{N\_G} = \alpha_{N\_G} \times N, t > 0, G \in \Omega \\ dif_G = (d_G \Delta G), t > 0, G \in \Omega \\ d_G \nabla G \cdot \vec{\eta} = 0, t > 0, G \in \partial\Omega \\ G(x, y, z, t = 0) = G_0(x, y, z), G \in \Omega \end{cases} \tag{3.7}$$

O termo dec_G representa o decaimento do G no tecido, com respectiva taxa μ_G. O termo $prod_{N\_G}$ representa a produção de G por N, com taxa α_{N_G}. Finalmente, o termo dif_G representa a difusão de G no tecido, com taxa d_G.

3.2.8 Citocina Anti-Inflamatória

As citocinas anti-inflamatórias são representadas no modelo matemático pela sigla CA, e modeladas pela Equação 3.8.

$$\begin{cases} \frac{\partial CA}{\partial t} = dec_{CA} + prod_{MR\_CA} + prod_{MA\_CA} + dif_{CA}, t > 0, CA \in \Omega \\ dec_{CA} = -(\mu_{CA} \times CA), t > 0, CA \in \Omega \\ prod_{MR\_CA} = (\beta_{MR\_ND} MR \times ND) \times \left(1 - \frac{CA}{\omega_{CA}}\right), t > 0, CA \in \Omega \\ prod_{MA\_CA} = (\beta_{MA} MA) \times \left(1 - \frac{CA}{\omega_{CA}}\right), t > 0, CA \in \Omega \\ dif_{CA} = (d_{CA} \Delta CA), t > 0, CA \in \Omega \\ d_{CA} \nabla CA \cdot \vec{\eta} = 0, t > 0, CA \in \partial\Omega \\ CA(x, y, z, t = 0) = CA_0(x, y, z), CA \in \Omega \end{cases} \tag{3.8}$$

O termo dec_{CA} representa o decaimento do CA no tecido, com respectiva taxa μ_{CA}. O termo $prod_{N\_CA}$ representa a produção de CA por MR na presença de ND, com taxa β_{MR_ND}. O termo $prod_{MA\_CA}$ representa a produção de CA por MA, com taxa β_{MA}. A taxa ω_{CA} determina a concentração máxima de CA suportado no tecido. O termo dif_{CA} representa a difusão de CA no tecido, com taxa d_{CA}.

3.2.9 Condições Inicial e de Contorno

Para que os modelos matemáticos propostos estejam bem postos, é necessário determinar as concentrações das populações no passo de tempo inicial (condições iniciais), assim como o comportamento das variações de concentração das populações do SIH (LPS, MA, MR, N, etc...) na borda do domínio (condições de contorno). As condições iniciais e de contorno foram apresentadas em conjunto com as equações apresentadas ao longo das últimas subseções sob a forma geral dada pelas Equações 3.9, 3.10 e 3.11:

$$d_C \nabla C \cdot \vec{\eta} = 0, t > 0, C \in \partial\Omega, \tag{3.9}$$

$$(d_C \nabla C - q_{A\_C} C \nabla A) \cdot \vec{\eta} = 0, t > 0, C \in \partial\Omega, \tag{3.10}$$

$$C(x, y, z, t = 0) = C_0(x, y, z), C \in \Omega \tag{3.11}$$

onde C representa uma das populações do SIH, d_C representa a taxa de difusão de C no tecido, A representa o agente responsável pelo processo de quimiotaxia com taxa q_{A_C}, Ω representa o domínio e $\partial\Omega$ o contorno do domínio. A equação 3.9 representa a condição de contorno do sistema. Ela especifica que as taxas de variação nas bordas são sempre nulas, portanto uma condição de contorno do tipo *Neumann Homogênea*. Quando C sofre tanto os processos de difusão quanto de quimiotaxia, é necessário usar a Equação 3.10 para estabelecer a condição de contorno. Já a equação 3.11 representa a condição inicial do sistema, cujos valores $C_0(x, y, z)$ são previamente conhecidos.

3.3 Métodos Numéricos

Encontrar soluções analíticas para EDPs não é uma tarefa trivial, com exceções de alguns tipos específicos de equações, como as Equações do Calor e da Onda [27]. Além disso, encontrar soluções analíticas requer conhecimentos matemáticos que vão além do escopo deste trabalho. Entretanto, tais equações podem ser resolvidas por meio de técnicas numéricas, cujo objetivo é passar o problema de um domínio contínuo para um discreto, onde só se conhece a solução em um conjunto finito de pontos, e descrevê-la usando métodos iterativos.

Os métodos usados neste trabalho para descrever essas equações são: a) Diferenças Finitas, empregando um esquema *Upwind* para a discretização espacial da quimiotaxia das células e moléculas do SIH, e b) o Método de Euler, para discretização temporal da evolução das concentrações.

3.3.1 Diferenças Finitas

É possível aproximar o valor de derivadas parciais usando o método das Diferenças Finitas [10], que pode ser de três tipo: *forward*, *backward* e *central*, representados respectivamente pelas Equações 3.12, 3.13 e 3.14.

$$\Delta_h[f](x) = f(x+h) - f(x) \tag{3.12}$$

$$\nabla_h[f](x) = f(x) - f(x-h) \tag{3.13}$$

$$\delta_h[f](x) = f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right) \tag{3.14}$$

A Equação 3.15 exemplifica o método das Diferenças Finitas do tipo *central* no domínio $\mathbb{R}^4$, representado pelas dimensões x, y, z e t. Neste exemplo, o método é aplicado para a derivada parcial de x, onde o espaço é discretizado em intervalos de comprimento h.

$$\begin{cases} \delta_h[f](x,y,z,t) = f(x+\frac{h}{2},y,z,t) - f(x-\frac{h}{2},y,z,t) \\ \\ \frac{\partial f(x,y,z,t)}{\partial x} \approx \frac{\delta_h[f](x,y,z,t)}{h} = \frac{f(x+\frac{h}{2},y,z,t) - f(x-\frac{h}{2},y,z,t)}{h} \\ \\ \frac{\partial^2 f(x,y,z,t)}{\partial x^2} \approx \frac{\delta_h^2[f](x,y,z,t)}{h^2} = \frac{f(x+h,y,z,t) - 2f(x,y,z,t) + f(x-h,y,z,t)}{h^2} \end{cases} \tag{3.15}$$

Este método é usado no SIH para aproximar as derivadas de primeira e segunda ordem das equações descritas anteriormente. Isto pode ser feito pois conhecemos as concentrações das células e moléculas em pontos igualmente espaçados do domínio $\mathbb{R}^3$.

3.3.2 Esquema *Upwind*

O esquema *Upwind* [6] é um abordagem numérica utilizada no modelo computacional do SIH para obter resultados mais estáveis e confiáveis, tendo em vista que a precisão dos resultados é afetada pelo esquema de convecção. Seu objetivo é analisar a direção de propagação da onda em relação à reta característica de EDPs hiperbólicas para realizar ajustes no método das Diferenças Finitas. Dependendo da direção de propagação da onda, os ajustes (*bias*) podem pode ser do tipo *backward*, caso a direção de propagação seja

positiva, ou *forward*, caso a direção de propagação seja negativa. O método é descrito no domínio $\mathbb{R}^4$ (x, y, z e t) pela Equação 3.16 [23].

$$\frac{\partial f(x,y,z,t)}{\partial t} + a(\frac{\partial f(x,y,z,t)}{\partial x} + \frac{\partial f(x,y,z,t)}{\partial y} + \frac{\partial f(x,y,z,t)}{\partial z}) = 0$$

$$\begin{cases} a\frac{\partial f(x,y,z,t)}{\partial x} \approx a\frac{f(x,y,z,t)-f(x-h,y,z,t)}{h} & se\ a > 0 \\ a\frac{\partial f(x,y,z,t)}{\partial x} \approx a\frac{f(x+h,y,z,t)-f(x,y,z,t)}{h} & se\ a < 0 \\ \\ a\frac{\partial f(x,y,z,t)}{\partial y} \approx a\frac{f(x,y,z,t)-f(x,y-h,z,t)}{h} & se\ a > 0 \\ a\frac{\partial f(x,y,z,t)}{\partial y} \approx a\frac{f(x,y+h,z,t)-f(x,y,z,t)}{h} & se\ a < 0 \\ \\ a\frac{\partial f(x,y,z,t)}{\partial z} \approx a\frac{f(x,y,z,t)-f(x,y,z-h,t)}{h} & se\ a > 0 \\ a\frac{\partial f(x,y,z,t)}{\partial z} \approx a\frac{f(x,y,z+h,t)-f(x,y,z,t)}{h} & se\ a < 0 \end{cases} \tag{3.16}$$

3.3.3 Método de Euler

O método de Euler explícito, que emprega uma diferença finita progressiva (*forward*) na derivada do tempo, é usado para a discretização temporal. O método permite calcular a concentração das populações em um tempo posterior ao atual baseando-se apenas nos valores de suas concentrações atuais. A implementação numérica é descrita na Equação 3.17, onde C representa uma das populações do SIH. É importante mencionar que os passos de tempo são discretizados por Δt.

$$C_{atual} = C_{passado} + \frac{dC}{dt} \times \Delta t \tag{3.17}$$

3.4 Programação Paralela

Nesta seção são apresentados, brevemente, as ferramentas computacionais utilizadas para o desenvolvimenta das versões paralelas dos códigos, *Message Passing Interface* (MPI) [19] e OpenCL [16].

3.4.1 MPI

Ao se paralelizar uma aplicação para execução em um *cluster* de computadores, muitas vezes se faz necessária a troca de dados entre dispositivos localizados em computadores diferentes. A fim de prover uma interface de comunicação dos dispositivos neste ambiente híbrido de memória distribuída, o padrão MPI é utilizado neste trabalho por meio da biblioteca *OpenMPI*.

Este padrão define que, para cada processo que constitui o sistema distribuído, um identificador único é associado ao mesmo, sendo este chamado de *rank*. O *rank* permite enviar dados a um processo em particular por meio de rotinas de envio e recebimento de mensagens ponto a ponto (*e.g. MPI_Send* e *MPI_Recv*).

Além das rotinas de comunicação de dados, com MPI ainda é possível sincronizar a execução de todos os processos. A sincronização é um recurso essencial para manter a integridade da computação realizada. Por exemplo, o simulador do SIH apresenta uma dependência temporal, onde um passo de tempo t não pode ser iniciado caso hajam pontos ainda não calculados no passo de tempo anterior $t-1$. Neste caso, é necessário que todos os processos envolvidos na computação dos pontos estejam sincronizados no passo de tempo $t-1$, para que o próximo passo de tempo seja iniciado. Este processo é realizado por meio da rotina bloqueante *MPI_Barrier*, onde todos os processos precisam chamar esta função para que os mesmos possam retomar a computação a partir deste ponto.

Outras funcionalidades essenciais do MPI incluem as primitivas de comunicação coletiva, como a que realiza uma operação de *broadcast*, onde um mesmo conjunto de dados pode ser enviado de um processo com um *rank* específico à todos os outros processos. A rotina *MPI_Broadcast* implementa esta funcionalidade. Outra operação coletiva é a operação de junção, onde os processos envolvidos enviam, cada um, um conjunto de dados à um processo com determinado *rank*, que os agrupa em um único conjunto de dado. Essa funcionalidade é implementada pela rotina *MPI_Gather*. Ambas as rotinas são utilizadas extensivamente neste trabalho.

No contexto do simulador SIH executado em sistemas de memória distribuída, é necessário que haja um processo com um *rank* atribuído em cada computador do sistema, executando em um núcleo de processamento da CPU. Caberá a estes processos estabelecerem a comunicação entre todos os computadores. Como será apresentado no próximo capítulo, uma parte essencial da simulação do SIH é a realização da troca das informações de bordas entre dispositivos do sistema. Quando os dispositivos estão localizados no mesmo computador, a troca de bordas é uma tarefa trivial, envolvendo apenas a cópia de posições de memória. Entretanto, quando esta troca de bordas envolve dispositivos localizados em computadores distintos, faz-se necessário o uso das rotinas de envio e recebimento de mensagens, especificamente *MPI_Send* e *MPI_Recv*. Cada

processo de cada computador é responsável por enviar e receber os dados necessários e, em seguida, decidir para qual dispositivo local (CPU ou GPU) esta informação será encaminhada.

Outra importante etapa da computação na qual o MPI participa está no processo de balanceamento de carga. Como será melhor detalhado no capítulo 4, o tempo que cada dispositivo do sistema leva para computar os pontos da malha para um intervalo de passos de tempo é medido, e este resultado é então usado para balancear as cargas entre os dispositivos. É necessário designar um dos processos do sistema para realizar este cálculo e, para que este obtenha os tempos medidos de todos os dispositivos do sistema, é necessário que todos os outros processos enviem mensagens com informações dos tempos medidos dos seus respectivos dispositivos. A rotina *MPI_Gather* pode ser usada para esta tarefa, já que cada processo pode coletar os tempos medidos para os dispositivos locais e encaminhar esta informação para um processo com determinado *rank*, estabelecido neste trabalho como o processo com o identificador 0, de modo que este último realize o cálculo das novas cargas. Em seguida, essa informação é transmitida à todos os demais processos através da rotina *MPI_Broadcast*. Com essa informação todos os dispositivos podem fazer os devidos ajustes de carga, como ilustrado pela Figura 2.

3.4.2 OpenCL

OpenCL (*Open Computing Language*) [16] é um *framework* padronizado criado pela indústria (Khronos Group) com o objetivo de auxiliar no desenvolvimento de aplicações paralelas que executam em sistemas heterogêneos, compostos por uma combinação de CPUs, GPUs, e outros processadores/aceleradores. A primeira versão do OpenCL foi lançada em dezembro de 2008, e sua versão mais recente é o OpenCL 2.0.

Dispositivos OpenCL, tais como CPUs, GPUs, entre outros, executam as instruções do programa, chamadas de *kernels*. Um dispositivo OpenCL é composto por *working groups*, que são blocos ou unidades de computação, sendo constituídos por *work-items* (WIs), que representam as *threads* que processam os dados paralelamente usando os núcleos de processamento do dispositivo. Os seguintes passos generalizam as ações que uma aplicação OpenCL deve seguir para executar em uma plataforma heterogênea [16]:

1. Encontrar todas as plataformas disponíveis;
2. Encontrar todos os componentes (dispositivos) de uma plataforma heterogênea;
3. Descobrir as características de cada dispositivo que compõe a plataforma;
4. Criar um contexto para cada dispositivo, que representa o ambiente no qual os *kernels* serão executadas;
5. Criar os *kernels* que serão executados nos contextos dos dispositivos;

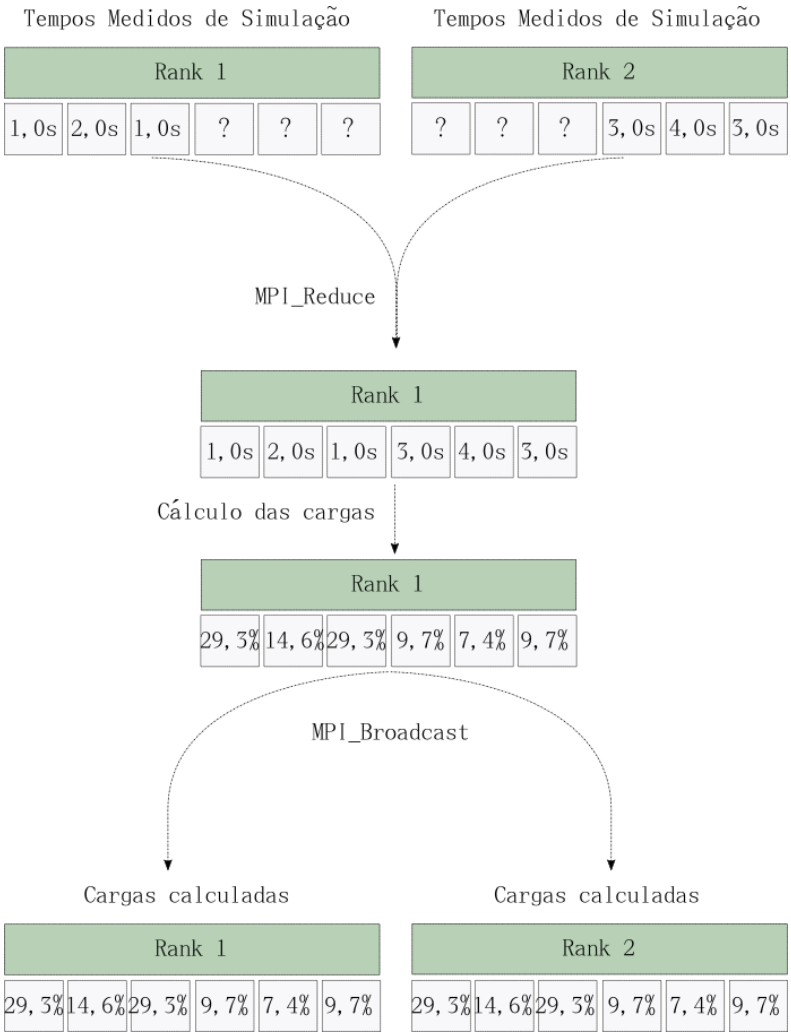

Figura 2 – Comunicação envolvida no balanceamento de carga entre duas máquinas, cada uma com três dispositivos. Os processos, identificados por seus *ranks*, fazem as medições dos tempos de execução e usam a rotina *MPI_Gather* para enviar os mesmos para o processo de *rank* 1. Este realiza o cálculo das cargas e envia os resultados para todos os computadores usando rotina *MPI_Broadcast*.

6. Alocar a memória nos dispositivos;
7. Criar e configurar a fila de comando, usada para transmitir as instruções do *host*, geralmente um processo da CPU, para os dispositivos;
8. Executar os *kernels* em cada dispositivo;
9. Coletar os resultados.

Uma plataforma OpenCL inclui um *host*, que é responsável por interagir com os dipositivos. A interação entre o *host* e os dispositivos é feita através da fila de comando. Comandos são enviados à fila de comando e permanecem lá até serem enviados ao dispositivo para serem executados. Existem três tipos de comandos que podem ser enviados: execução de *kernel*, operações de leitura e escrita na memória e eventos de sincronização. Os

comandos em uma única fila podem executar na ordem em que são enviados para a fila de comando (execução em ordem), ou podem ser executados em qualquer ordem (execução fora de ordem). O programador pode forçar uma ordem específica de execução usando mecanismos de sincronização explícita. Filas de comando, especialmente aquelas que implementam execução fora de ordem, podem ser usadas para implementar um esquema de balanceamento de carga automático baseado no padrão paralelo de mestre-escravo [16, 14], mecanismo usado em outros trabalhos[3, 4]. Entretanto, o padrão paralelo de mestre-escravo é mais apropriado para problemas baseados no paralelismo de tarefas [14]. Neste trabalho, propõe-se uma solução distinta baseada na execução em ordem de problemas com paralelismo de dados, com suporte a dispositivos do tipo CPU e GPU. São também utilizadas duas filas de comando para operações de leitura e escrita da memória, assim como a execução de *kernels* simultâneamente.

Para realizar o balancemento de carga, é necessário medir o tempo que cada dispositivo levou para executar o seu *kernel*. Para isso, são utilizadas rotinas de sincronização, onde eventos são atribuídos a cada comando de execução de *kernel* enviado para a fila. Uma chamada bloqueante *clWaitForEvents* é então realizada. Os tempos de execução gravados nos eventos atribuídos são então usados para o cálculo das cargas.

3.5 Resumo do Capítulo

Este capítulo apresentou algumas células e moléculas que compõem o SIH. Apesar de existirem dois tipos de resposta imune, inata, uma resposta geral, independente do agente patogênico, e adaptativa, específica para combater um determinado patógeno, este trabalho tratará da implementação paralela de um simulador do SIH inato. Para isso foi apresentado o modelo matemático que representa toda a dinâmica do processo de resposta imunológica inata, onde uma série de Equações Diferenciais Parciais (EDPs) descrevem as taxas de variação das células e moléculas do SIH a partir das condições iniciais que representam as concentrações de cada população no início da simulação. A partir do modelo matemático, implementa-se um modelo computacional, usando para isso métodos numéricos. Através da discretização espacial, feita com o uso do método das Diferenças Finitas e do método *Upwind*, e a discretização temporal, feita usando o Método de Euler, o modelo analítico é implementado em sua versão sequencial. Para sua paralelização são usados padrões como MPI e OpenCL, também apresentados ao longo deste capítulo.

4 MÉTODOS

A resolução das EDPs que modelam as células e moléculas do SIH pode ser realizada em paralelo, como observado na seção 3.3, o que torna este tipo de aplicação ideal para ser executada em paralelo em dispositivos que possuem múltiplos núcleos de processamento.

Este capítulo tem como objetivo apresentar a implementação do simulador do SIH em um ambiente computacional heterogêneo, *i.e.* serão abordados todos os mecanismos necessários para realizar o processo de balanceamento de carga. Deste modo, o capítulo descreverá os algoritmos de balanceamento de carga estático e dinâmico, que são o foco deste trabalho. Apesar dos algoritmos serem genéricos, no sentido de que podem ser usados para balancear carga em qualquer aplicação com características semelhantes a do simulador do SIH, este capítulo também abordará alguns aspectos técnicos relacionados a implementação. Para facilitar o entendimento destes aspectos, inicialmente será descrita a versão usada em um ambiente homogêneo, ou seja, em um ambiente que não necessita de nenhum esquema de balanceamento de carga. A execução do SIH em dispositivos homogêneos foi previamente abordada por Xavier [21] e Rocha [24]. Apesar da implementação em dispositivos heterogêneos apresentar semelhanças com a implementação em dispositivos homogêneos (*e.g.*, paralelismo de dados e a comunicação assíncrona entre dispositivos), é importante ressaltar que a heterogeneidade dos dispositivos usados neste trabalho levou a mudanças na implementação. Os trabalhos anteriores não utilizam CPUs e GPUs para realizar cálculos simultaneamente, ficando as GPUs responsáveis pela computação e as CPUs pelo gerenciamento da comunicação. Neste trabalho modificações foram efetuadas para que tanto CPUs quanto GPUs pudessem processar os dados da aplicação de modo simultâneo.

4.1 Dispositivos Homogêneos

Dispositivos homogêneos apresentam a mesma arquitetura de hardware, portanto, possuem as mesmas capacidades computacionais. Por este motivo, qualquer esquema de balanceamento de carga é desnecessário quando a intenção é executar em tais dispositivos, já que o processo de distribuição de cargas requer apenas que a quantidade total de dados computados seja igualmente distribuída entre os dispositivos participantes da computação. O foco desta seção é portanto explicar todos os fundamentos computacionais da implemetação númerica do modelo matemático do SIH em tais ambientes. Foca-se nesta seção a execução em múltiplas GPUs localizadas em máquinas distintas, sem o emprego de CPUs para realizar computação.

4.1.1 Implementação Numérica

A implementação do modelo computacional do SIH se baseia nos métodos numéricos de Euler, para discretização temporal, e o método de Diferenças Finitas com uso do esquema *Upwind*, para discretização espacial.

Para que o espaço seja discretizado, a região do tecido que será simulada precisa ser representada por uma grade ou vetor tridimensional, onde cada ponto é uniformemente separado nos eixos x, y e z e contém as informações da concentração de LPS, Macrófagos em Repouso, Macrófagos Ativos, Neutrófilos, Citocinas Pró-Inflamatórias, Neutrófilos Apoptóticos, Grânulos Protéicos e Citocinas Anti-Inflamatórias. Este vetor é chamado neste trabalho de malha, cuja representação é mostrada na Figura 3.

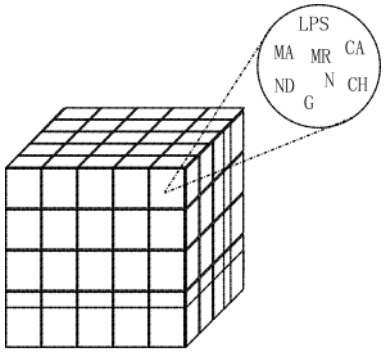

Figura 3 – Representação da malha tridimensional, onde X, Y e Z representa o número de pontos que subdivide cada eixo coordenado x, y e z. V representa o número de variáveis do modelo.

Para realizar o cálculo numérico das derivadas em um determinado ponto da malha, é necessário obter informações da região que contorna este ponto, ou seja, para calcularmos a derivada em relação aos eixos x, y, e z de um ponto localizado em (x, y, z), precisamos conhecer os valores das localizações $(x + \Delta x, y, z)$, $(x - \Delta x, y, z)$, $(x, y + \Delta y, z)$, $(x, y - \Delta y, z)$, $(x, y, z + \Delta z)$ e $(x, y, z - \Delta z)$.

Para a discretização temporal, são criadas duas malhas, uma delas irá armazenar as variáveis do modelo no tempo anterior, que serão então usadas para calcular seus novos valores no passo de tempo atual. Esses novos valores, após calculados, serão armazenados em uma outra malha. Após o cálculo, as malhas trocam sua função no próximo passo de tempo (*e.g.*, se no passo de tempo i a malha A representa as concentrações do passo de tempo anterior $i - 1$ e a malha B é usada para armazenar as concentrações do passo de tempo atual i, na próxima iteração, $i + 1$, a malha B irá representar as concentrações do passo de tempo anterior i, enquanto que a malha A será usada para armazenar as concentrações do passo de tempo atual $i + 1$). Este processo é denominado neste trabalho de troca de malhas. A vantagem desta implementação é que os pontos podem ser calculados sem que sejam necessárias operações de sincronização. O cálculo dos pontos usando apenas

uma malha para armazenar os valores reduz o uso de memória, mas implica na adição de operações de sincronização no acesso a alguns dos pontos da malha, aqueles compartilhados entre dispositivos vizinhos. Esse custo de sincronização pode levar a grande perda de desempenho na execução. Rocha e colaboradores [24] apresentam com mais detalhes a implementação numérica do Método de Euler no modelo computacional do SIH.

A implementação dos métodos numéricos descritos, ao invés do uso de bibliotecas que oferecem tal funcionalidade, nos permitiu maior flexibilidade para paralelizar o código, sendo possível, por exemplo, a implementação de diferentes algoritmos de balanceamento de carga. Se uma biblioteca numérica fosse usada ao invés de uma implementação própria, isto não seria possível. O código do simulador foi implementado originalmente em C no escopo de outros trabalhos[22, 24]. Apesar de Rocha e colaboradores [24] também proporem uma implementação paralela do simulador SIH usando o mesmo conjunto de equações e o mesmo método numérico para implementá-lo, este trabalho usa uma abordagem distinta. Enquanto Rocha e colaboradores [24] usam CUDA [9] para implementar o conjunto de equações, este trabalho usa OpenCL [16]. Além disso, Rocha e colaboradores [24] usam uma plataforma homogênea como sua plataforma de hardware alvo. Este trabalho explora todos os recursos disponíveis em um *cluster* de computadores, assim como propõe um esquema para realizar o balanceamento de carga nesta plataforma heterogênea.

4.1.2 Troca de Bordas

Conforme descrito na última seção, durante a computação realizada por uma *thread* de um dispositivo no passo de tempo atual, faz-se necessário o acesso aos dados produzidos pelas *threads* vizinhas no passo de tempo anterior, *e.g.* a *thread* que realiza computações na posição (x, y, z) no passo de tempo atual da malha precisa acessar as posições $(x+\Delta x, y, z)$, $(x-\Delta x, y, z)$, $(x, y+\Delta y, z)$, $(x, y-\Delta y, z)$, $(x, y, z+\Delta z)$ e $(x, y, z-\Delta z)$ computada pelas *threads* vizinhas no passo de tempo anterior.

Entretanto, algumas destas *threads* vizinhas podem ter produzido dados localizados em dispositivos distintos. É necessario portanto realizar a troca destes dados, chamados de bordas, entre os computadores, de modo a permitir a correta computação pelas *threads*. As primitivas de comunicação *MPI_Send* e *MPI_Recv* são usadas para este fim (quando os dispositivos estão localizados em computadores distintos), assim como as primitivas de comunicação OpenCL *clEnqueueReadBuffer* e *clEnqueueWriteBuffer* (quando os dispositivos estão localizados no mesmo computador), a cada passo de tempo. Não se pode iniciar um novo passo de tempo sem que os dados tenham sido trocados. O custo de comunicação nessa etapa é alto e afeta significativamente o tempo total de simulação.

4.1.3 Sobreposição de Computação e Comunicação

Uma alternativa para minimizar o custo da troca de bordas foi apresentada por Xavier [21]. Apenas a computação dos pontos localizados na região de fronteira entre dispositivos necessita da borda do vizinho para ser realizada. É possível realizar a computação dos pontos internos (todos os pontos que não pertencem à região de fronteira da malha) sem os dados computados por um dispositivo vizinho. Deste modo, é possível realizar a computação dos pontos internos e a troca de bordas simultaneamente. Como o número de pontos internos é significativamente maior que o número de pontos de borda, o tempo de computação para cada passo de tempo é minimizado, visto que a comunicação é sobreposta no tempo com uma operação de computação. O processo de troca de bordas é representado esquematicamente na Figura 4.

A troca de bordas entre dispositivos, assim como sua sobreposição com a computação dos pontos internos, requer a introdução de operações de sincronização, além da já mencionada troca de mensagens com uso de MPI. A sincronização é feita de duas formas: em cada dispositivo, usando a função da biblioteca OpenCL *clFinish*, chamada pelo *host* e cujo objetivo é bloquear a execução do programa até que a fila de comando tenha encerrado a execução de suas tarefas; e entre todos os processos presentes em todos os computadores do *cluster*, usando a função *MPI_Barrier*. Tanto as operações de cópia de dados quanto as de sincronização são muito custosas, já que envolvem o uso da rede para comunicação remota entre dispositivos, devendo portanto ser evitadas.

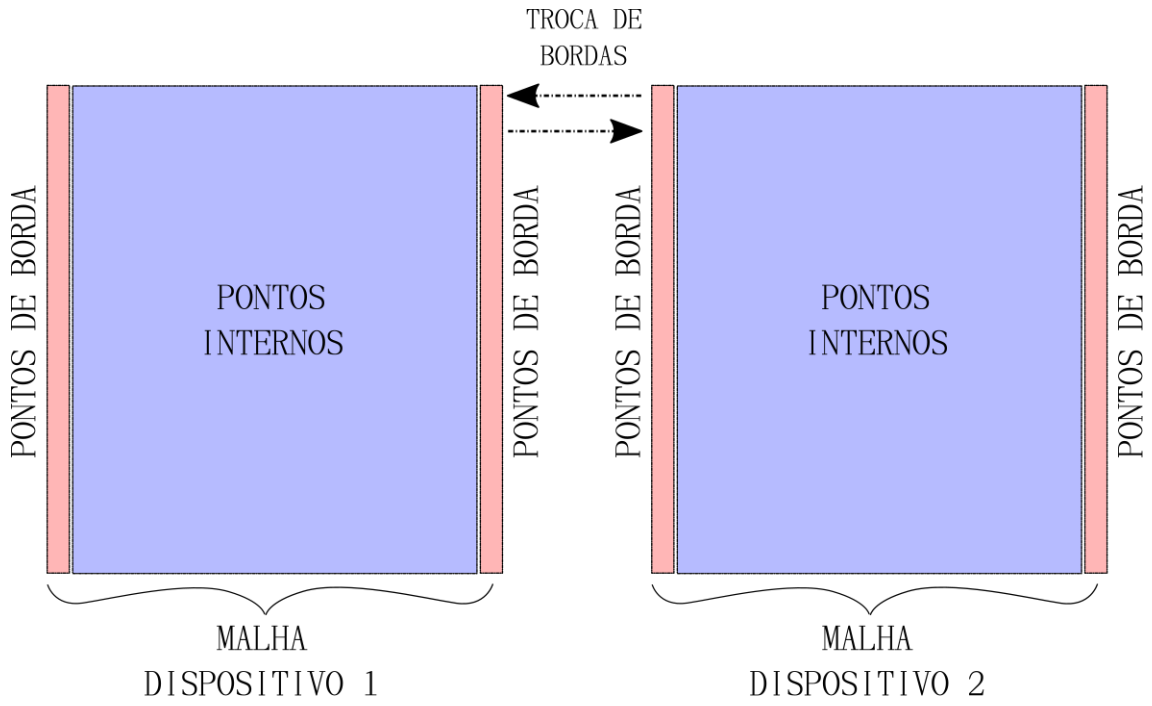

Figura 4 – Representação esquemática do processo de troca de bordas.

4.1.4 Distribuição de Tarefas

Uma melhoria introc lo SIH para dispositivos homogêneos, desenvolvida neste trabalho, foi a divisão dos dados a serem computados por cada dispositivo. Como explicado na seção 3.4.2, os dados que são computados em paralelo são denominados *work-items* (WIs). Eles são enviados para execução nos múltiplos blocos de unidades de processamento de cada dispositivo, denominados *work-groups*(WGs). A quantidade de WIs que compõem um único WG é definida pelo programador, entretanto, existe um limite para este valor, estabelecido pelas capacidades computacionais do dispositivo. Uma tarefa é composta por vários WIs, responsáveis pela execução paralela dos dados.

Um número alto de WIs introduz um custo adicional de comunicação, já que as mesmas são agrupadas em WGs e então enviadas para serem executadas nos dispositivos através da fila de comando.

A fim de limitar a quantidade total de WIs e, consequentemente, os WGs que são enviados aos dispositivos, e reduzir o custo adicional introduzido pela fila de comando, um esquema de distribuição dos dados a serem computados em uma tarefa foi desenvolvido, onde cada WI é responsável por computar múltiplos dados.

Para determinar a quantidade de dados computados por um WI, é necessário estabelecer a quantidade total de WGs que representa uma tarefa, representado por N_{WGs}, assim como a quantidade WIs que é executado em cada WG, representado por N_{WIs_WG}. A quantidade de dados calculados por cada WI, representado por N_{dados_WI}, é calculado usando a Equação 4.1:

$$N_{dados\_WI} = \frac{N_{dados}}{N_{WGs} \times N_{WIs\_WG}}, \tag{4.1}$$

onde N_{dados} é a quantidade total de dados que compõem uma tarefa; *e.g.* caso hajam 100 dados para serem computados em uma tarefa utilizando 2 WGs de tamanho 5, cada WI irá calcular $\frac{100}{2\times5} = 10$ dados da tarefa.

O acesso aos dados por cada WI é organizado seguindo o padrão de acesso coalescente de memória, a fim de minimizar a quantidade de acessos à memória global do dispositivo. Nele, cada WI acessa os dados em intervalos de valor $N_{WGs} \times N_{WIs\_WG}$, *e.g.* se $N_{WGs} = 2$, e $N_{WIs\_WG} = 5$, isto significa que o intervalo de acesso à memória é igual a $N_{WGs} \times N_{WIs\_WG} = 10$. Consequentemente, os endereços acessados por cada WI são $\{E, E+10, E+20, E+30\dots\}$, onde E é o endereço inicial da memória. Esta organização é exemplificada na Figura 5.

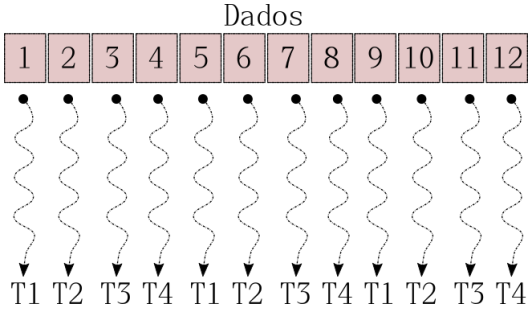

Figura 5 – Exemplo do acesso à memória por 4 *threads* concorrentes, onde cada *thread* é responsável por computar 3 dados, separados em intervalos iguais.

4.2 Dispositivos Heterogêneos

A principal dificuldade em usar dispositivos heterogêneos colaborativamente na computação está ligado à distribuição de carga de trabalho entre os mesmos. Pode-se destacar na literatura duas abordagens para realizar o balanceamento de carga: a estática [2], onde as cargas dos dispositivos são determinadas em um estágio inicial da computação, e a dinâmica [8], onde as cargas são distribuídas no decorrer da computação, já que muitas vezes se desconhece *a priori* o real desempenho de cada dispositivo, o que torna difícil determinar as cargas que deverão ser designadas aos mesmos antes que a computação seja realizada.

Neste trabalho, as duas abordagens de balanceamento foram usadas. O uso de cada abordagem depende das características da aplicação: enquanto o balanceamento estático é indicado para aplicações regulares, ou seja, cujo tempo de computação de uma determinada carga permance o mesmo a cada passo da simulação, o balanceamento dinâmico é indicado para aplicações irregulares, cujo tempo de computação das cargas varia a cada passo da simulação. A variação no tempo de execução de uma mesma quantidade de dados a cada passo da simulação dificulta a tarefa de encontrar uma carga fixa ideal a ser usada até o término da simulação, sendo necessários ajustes ao longo da sua execução.

4.2.1 Distribuição das Cargas nos Dispositivos

Um aspecto chave na implementação paralela em dispositivos heterogêneos é a partição de dados a serem processados pelos dispositivos. Quando empregam-se simultaneamente CPUs e GPUs no processamento, duas questões surgem. A primeira é que a quantidade de núcleos de processamento presentes em uma GPU é muito maior do que os presentes em uma CPU. Além disso, a GPU foi especificamente projetada para executar programas que seguem o modelo *Single Instruction Multiple Data* (SIMD), em que o mesmo conjunto de instruções é usado para computar diferentes dados, diferentemente das CPUs, projetadas para lidar com paralelismo de instruções e *threads*[7]. Portanto, um esquema de balanceamento de carga deve ser usado para assegurar que nenhum dispositivo

fique ocioso enquanto outros estão sobrecarregados.

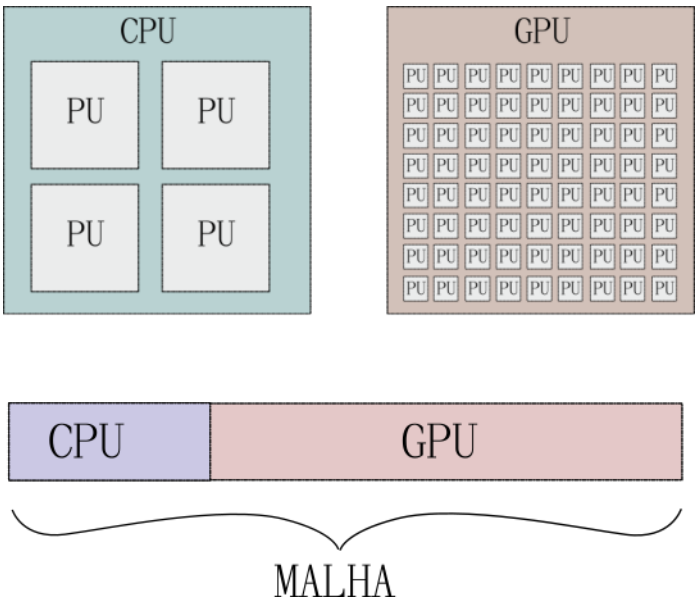

Figura 6 – Divisão de trabalho entre CPUs e GPUs. Os múltiplos núcleos de processamento da CPU e da GPU são representados na figura pela sigla PU (*processing unit*).

A Figura 6 descreve a idéia basica: dados da malha são separados em duas partes, das quais uma será computada pela CPU, enquanto a outra será computada pela GPU. O conjunto de intruções a serem computadas em cada dispositivo, denominado *kernel*, é o mesmo. A quantidade de dados designada para a CPU e GPU depende das suas respectivas capacidades computacionais. Como o número de unidades de processamento das GPUs é geralmente muito maior do que o das CPUs, o seu poder de processamento paralelo é maior, portanto, uma carga maior deve ser designada a elas.

4.2.2 Algoritmo de Balanceamento de Carga Estático

O Algoritmo 1 ilustra a implementação do balanceamento de carga estático. Primeiramente, o MPI e o OpenCL são inicializados. Em seguida, todas as plataformas e dispositivos são identificados e os *ranks* são definidos para cada processo. O contexto e o *kernel* são então criados. Finalmente, as filas de comando, memória e parâmetros são configurados.

Para o primeiro passo de tempo e também durante um número I dos passos de tempo iniciais, os tempos de execução dos *kernels* são coletados. É importante ressaltar que existe uma única implementação dos *kernels* para a CPU e para a GPU, mas o tempo para executá-la varia de acordo com cada dispositivo devido à diferenças de *hardware*. Os tempos de computação coletados são então usado para calcular a quantidade total de dados que cada dispositivo irá receber. O número I dos passos a ser utilizado para o cálculo da carga é determinado pelo usuário.

1: inicializar o MPI e o OpenCL;
2: alocar memória em cada dispositivo;
3: dividir cargas igualmente em cada dispositivo;
4: iniciar medição de tempo;
5: **for** um único passo de tempo (*probing*) **do**
6: computar os dados internos;
7: sincronizar;
8: computar os dados de borda;
9: sincronizar;
10: **end for**
11: finalizar medição de tempo;
12: computar $P_i^{(t)}$ e reajustar as cargas para cada dispositivo;
13: iniciar medição de tempo;
14: **for** um número I de passos de tempo **do**
15: computar os dados internos e realizar troca de bordas simultâneamente;
16: sincronizar;
17: computar os dados de borda;
18: sincronizar;
19: **end for**
20: finalizar medição de tempo;
21: computar $P_i^{(t)}$ e reajustar as cargas para cada dispositivo;
22: **for** o restante dos passos de tempo **do**
23: computar os dados internos e realizar troca de bordas simultaneamente;
24: sincronizar;
25: computar os dados de borda;
26: sincronizar;
27: **end for**
28: Finalizar MPI e OpenCL;

Algoritmo 1: O algoritmo de balanceamento de carga estático.

Conceitualmente, o balanceamento de carga estático é realizado da seguinte forma. Inicialmente, para um único passo de tempo, o trabalho é igualmente dividido entre todos os dispositivos do sistema distribuído. O balanceamento realizado neste passo único de tempo é denominado neste trabalho de *probing*. Após esse único passo de tempo, o tempo de computação gasto por cada dispositivo é coletado. Com base nesses tempos, as cargas são calculadas de acordo com a equação 4.6. Após o *probing*, nos próximos I passos de tempo, o tempo de computação de cada dispositivo é medido, e um novo balanceamento é realizado, novamente utilizando a equação 4.6. Após calculadas e redistribuídas, as novas cargas calculadas para cada dispositivo permaneceram fixas até que a simulação seja concluída. Deve-se ressaltar que, em todos os passos de tempo da simulação, é utilizado o processo de sobreposição da computação dos pontos internos com a troca de bordas e, em seguida, é feita a computação dos pontos de borda.

4.2.3 Algoritmo de Balanceamento de Carga Dinâmico

No balanceamento dinâmico, os tempos de computação dos *kernels* são medidos iterativamente, a cada I passos de tempo decorridos, onde novamente o valor I pode ser definido pelo usuário. A cada I passos de tempo, é calculado um novo balanceamento. Entretanto, novos balanceamentos podem não ser realizados caso a diferença entre a carga atual e a nova carga seja inferior a um determinado limite estabelecido pelo usuário. Este limite é chamado de *threshold*. Visto que a operação de rebalanceamento pode ser custosa por envolver a redistribuição dos dados a serem computados, tenta-se eliminar este custo, em especial quando a diferença entre a carga calculada e a carga em uso é pequena. Com aumento da quantidade de passos computados, espera-se que a diferença entre a nova carga e a carga usada anteriormente tenda a diminuir e, consequentemente, os custos para reajustar as cargas se tornem maiores do que os ganhos em desempenho obtidos com esse ajuste, não sendo necessário reajustar as cargas. O Algoritmo 2 descreve todo o processo.

```
1: inicializar o MPI e o OpenCL;
2: alocar memória em cada dispositivo;
3: dividir cargas igualmente em cada dispositivo;
4: iniciar medição de tempo;
5: for um único passo de tempo do
6:    computar os dados internos
7:    sincronizar;
8:    computar os dados de borda;
9:    sincronizar;
10: end for
11: finalizar medição de tempo;
12: computar P_i^(t) e reajustar as cargas para cada dispositivo;
13: iniciar medição de tempo;
14: for um número de passos de tempo do
15:    if Intervalo I de balanceameno de carga alcançado then
16:       finalizar medição de tempo;
17:       computar P_i^(t);
18:       if |P_i^(t) − P_i^(t−1)| > threshold then
19:          reajustar as cargas para cada dispositivo
20:       end if
21:       iniciar medição de tempo;
22:    else
23:       computar os dados internos e realizar troca de bordas simultaneamente;
24:       sincronizar;
25:       computar os dados de borda;
26:       sincronizar;
27:    end if
28: end for
29: Finalizar MPI e OpenCL;
```

Algoritmo 2: O algoritmo de balanceamento de carga dinâmico.

4.2.4 Equação de Balanceamento de Carga

A equação matemática usada pelos dois algoritmos de balanceamento de carga para calcular o percentual de dados que cada dispositivo receberá para computar é descrita nesta seção.

Para balancear as cargas entre D dispositivos no passo de tempo atual t, deseja-se que os tempos de computação T de cada dispositivo no tempo atual sejam iguais, ou seja:

$$T_0^t = T_1^t = \cdots = T_D^t = \beta, \tag{4.2}$$

onde β é um valor constante, equivalente ao tempo de computação equalizado. Em seguida, se assumirmos que a carga a ser dividida entre os dispositivos deve ser diretamente proporcional aos seus tempos de computação ($Carga/Tempo = Constante$), então:

$$\frac{P_i^t}{T_i^t} = \frac{P_i^{t-1}}{T_i^{t-1}}, \tag{4.3}$$

onde P_i^{t-1} é a carga do dispositivo i no passo de tempo anterior, P_i^t é a carga do dispositivo i no passo de tempo atual, T_i^{t-1} é o tempo de computação do dispositivo i no passo de tempo anterior e T_i^t é o tempo de computação do dispositivo i no passo de tempo atual, onde $i \in \{0, 1, 2...D\}$. Substituindo a Equação 4.2 na Equação 4.3, obtêm-se:

$$P_i^t = \frac{P_i^{t-1} \times \beta}{T_i^{t-1}}. \tag{4.4}$$

As cargas são medidas neste trabalho como porcentagens, portanto, a soma de todas elas deve ser igual à 100%, ou seja:

$$\sum_{i=1}^{N} P_i^t = 1 \tag{4.5}$$

O próximo passo então é normalizar as cargas encontradas na Equação 4.4, dividindo-

as pela Equação 4.5:

$$
\begin{aligned}
P_i^t &= \frac{P_i^{(t-1)} \times \beta}{T_i^{(t-1)} \times \sum_{i=1}^{N} P_i^t} \\
&= \frac{P_i^{(t-1)} \times \beta}{T_i^{(t-1)} \times \sum_{i=1}^{N} \left(\frac{P_i^{(t-1)} \times \beta}{T_i^{(t-1)}} \right)} \\
&= \frac{P_i^{(t-1)} \times \beta}{T_i^{(t-1)} \times \beta \sum_{i=1}^{N} \left(\frac{P_i^{(t-1)}}{T_i^{(t-1)}} \right)} \\
&= \frac{P_i^{(t-1)}}{T_i^{(t-1)} \times \sum_{i=1}^{N} \left(\frac{P_i^{(t-1)}}{T_i^{(t-1)}} \right)}.
\end{aligned}
\tag{4.6}
$$

4.2.5 *Probing* de Cargas

Ambos os esquemas de balanceamento de cargas dependem de um valor I, que determina o intervalo de passos de tempo em que será calculada a nova distribuição de cargas entre dispositivos.

As primeiras versões dos algoritmos de balanceamento de carga, não apresentadas nesse trabalho, distribuíam as cargas igualmente entres todos os dispositivos, e usavam I passos de tempo para calcular uma nova distribuição. Caso o usuário escolhesse um valor alto para I, os dispositivos com menores capacidades computacionais computariam durante muitos passos de tempo usando uma carga maior do que a ideal, enquanto os dispositivos com maior capacidade receberiam uma carga menor do que a ideal, impactando negativamente o desempenho.

Tal desperdício pode ser reduzido estimando-se as cargas após a execução de um único passo de tempo. Com uma estimativa inicial dada por este passo de *probing*, é realizada então uma redistribuição das cargas que já leva em consideração o desempenho de cada dispositivo. Mesmo que as cargas calculadas no único passo de tempo não sejam próximas às ideais para cada dispositivo heterogêneo, dificilmente o desempenho será inferior caso o cálculo dos primeiros I passos de tempo seja realizado com cargas iguais para cada dispositivo do sistema.

O objetivo do *probing* é, portanto, reduzir a possibilidade de desbalanceamento entre os dispositivos durante os passos de tempo anteriores ao primeiro balanceamento de carga, realizado tanto pelo esquema estático quanto dinâmico.

4.2.6 *Threshold* de Computação

Um possível alto custo oriundo do emprego do esquema de balanceamento de carga dinâmico decorre da comunicação envolvida no processo de rebalanceamento de cargas, já que, a cada vez que se detecta um desbalanceamento no tempo de execução, é necessário que haja uma redistribuição dos dados a serem computados por cada dispositivo no passo de tempo seguinte. Este processo é representado pela Figura 7.

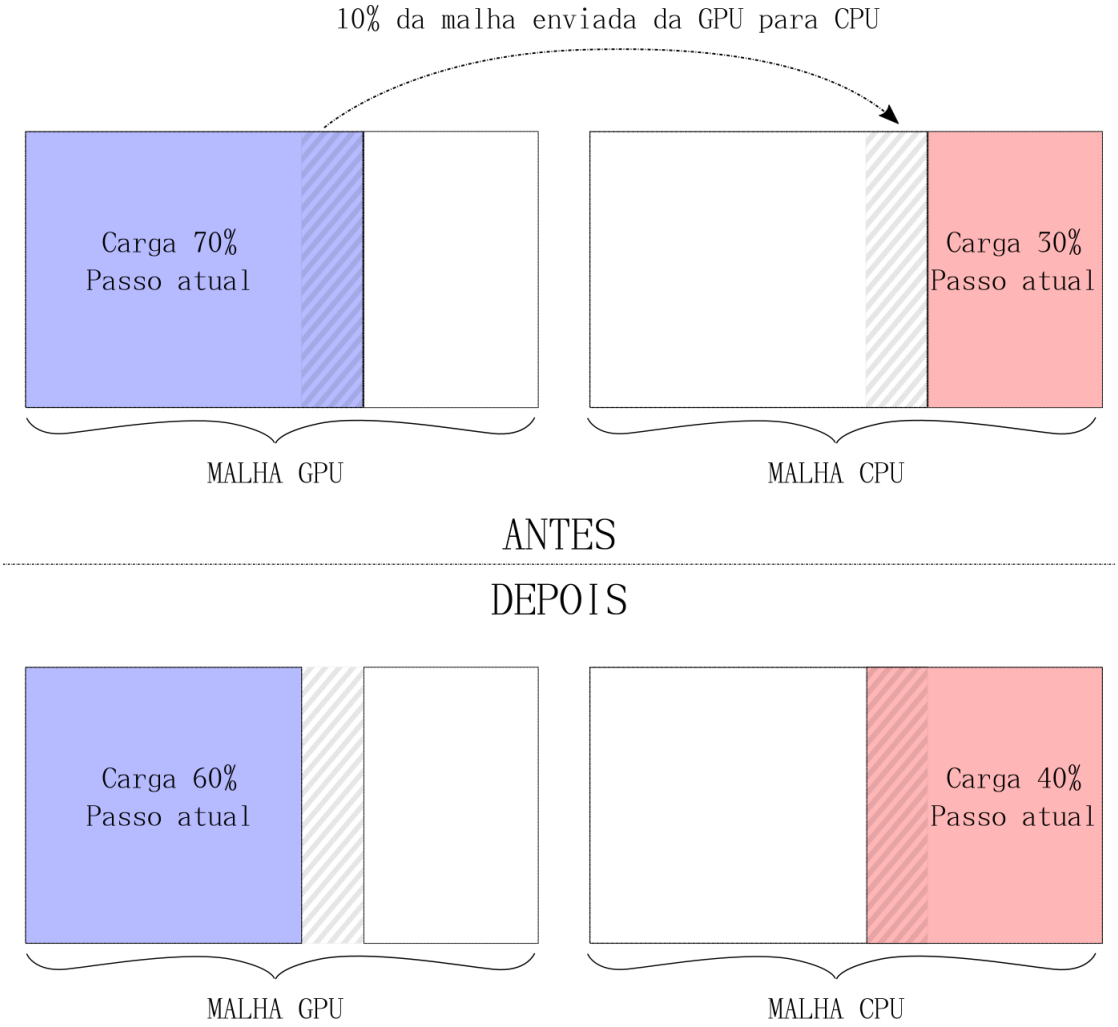

Figura 7 – Processo de distribuição de cargas entre uma CPU e uma GPU. O ajuste de cargas determinou uma diminuição de 10% da carga da GPU, consequentemente, a CPU receberá 10% da carga da GPU. O envio de dados entre os dispositivos, decorrente do rebalanceamento de carga, introduz custos de comunicação adicionais para a simulação.

Entretanto, caso a mudança nas cargas entre as I iterações usadas para o balanceamento seja pequena, é possível ignorar tal ajuste, já que os ganhos em desempenho podem não compensar o tempo necessário para redistribuir as cargas entre os dispositivos. Assim, para minimizar o custo adicional de comunicação, foi implementada a variável *threshold*, que determina a diferença mínima entre as novas cargas calculadas e as atualmente utilizadas para que seja realizado o processo de redistribuição das cargas. Caso a diferença

mínima não seja alcançada, as cargas dos dispositivos são mantidas, sendo reavaliadas após a execução de I iterações.

4.3 Resumo do Capítulo

Este capítulo abordou a implementação do simulador do SIH em dispositivos heterogêneos. Para facilitar o entendimento da implementação, inicialmente discutiu-se a versão que executa em um *cluster* composto por dispositivos homogêneos, ou seja, onde as capacidades computacionais de cada dispositivo são as mesmas. Essa implementação para execução em ambientes homogêneos foi proposta no escopo de trabalhos anteriores [21, 24] e serviu como base para a implementação da versão para execução em dispositivos heterogêneos, tendo em vista que a implementação numérica, as principais estruturas de dados para representação da malha, o mecanismo de troca de bordas e de sobreposição de execução com comunicação foram mantidos, apesar de implementados com *frameworks* distintos (CUDA e OpenCL). Uma importante melhoria feita neste trabalho está na distribuição das tarefas nos núcleos de processamento de cada dispositivo, cujo objetivo é diminuir os custos da fila de mensagem, usada para enviar as tarefas da CPU até os dispositivos. Todas essas características foram apresentadas ao longo desta seção.

Em seguida, abordou-se a implementação em dispositivos heterogêneos, onde as capacidades computacionais diferem entre os dispositivos. Para lidar com a heterogeneidade dos dispositivos, foram propostos dois algoritmos para balanceamento de carga, um estático e outro dinâmico. O algoritmo estático realiza uma divisão dos dados a serem computados por cada dispositivo, mantendo a divisão fixa até o fim da execução. A versão dinâmica altera essa divisão caso a nova divisão se diferencie da atualmente usada por uma quantidade maior que um limiar (*threshold*). Os pseudo-algoritmos para cada esquema de balanceamento foram então apresentados, bem como a equação utilizada para computar a carga destinada a cada dispositivo.

5 RESULTADOS

Este capítulo avalia o desempenho dos algoritmos de balanceamento de carga estático e dinâmico, usando para este propósito o simulador do SIH [24, 23].

5.1 Ambiente Computacional

Todos os testes foram executados em um pequeno *cluster* composto por 6 nós, onde cada nó possui dois processadores AMD 6272 (totalizando 32 núcleos de computação), 32 GB de memória principal, e duas GPUs NVidia Tesla M2075, cada uma contendo 448 núcleos CUDA e 6 GB de memória global e executam a versão 3.10.0 do *kernel* Linux. A versão OpenMPI 3.2, a versão 1.2 do OpenCL e a versão 4.8.5 do compilador *gcc* foram usados para compilar e executar todos os algoritmos. As máquinas são conectadas por uma rede Gigabit Ethernet. Apesar dos processadores AMD terem um total de 32 núcleos, existe uma unidade de cálculo de ponto flutuante (FPU) compartilhada para cada par de núcleos de processamento, *i.e.* de todos os 192 (6×32) núcleos de processamento contidos no *cluster*, apenas 96 estão disponíveis para realizar os cálculos da simulação, tendo em vista que as variáveis do modelo foram declaradas como tipos ponto-flutuante.

5.2 Resultados Numéricos

O objetivo do simulador do SIH é descrever toda a dinâmica do processo de resposta imunológica do nosso corpo à presença de um antígeno, o LPS. O resultado da simulação indica como as concentrações das populações de células e moléculas variaram desde o início da simulação, onde toda esta informação está armazenada na malha tridimensional que representa a discretização espacial da região do tecido infectada. Para demonstrar este processo, foi realizada uma simulação com 1.000.000 de passos de tempo, que representa 12 horas, em uma malha de tamanho $50 \times 50 \times 50$, com as mesmas condições iniciais e parâmetros descritos no apêndice A.

Para ilustrar da simulação, em intervalos de 10.000 passos de tempo, uma seção bidimensional nos eixos y e z da malha é gravada para todas as populações, onde $x = 0$. Ao final da simulação, a evolução temporal das populações pode ser observada nesta seção da malha. O resultado é apresentado nas Figuras 8, 9, 10, 11, 12, 13, 14, 15.

5.3 Experimentos

Seis versões do SIH foram executadas: 1) a sequencial, que utiliza apenas um dos núcleos de processamento da CPU, 2) a versão que usa apenas as CPUs sem balanceamento de carga, 3) a versão que usa apenas GPUs sem balanceamento de carga, 4) a versão que usa ambos os dispositivos, ainda sem nenhum esquema de balanceamento de carga e, por

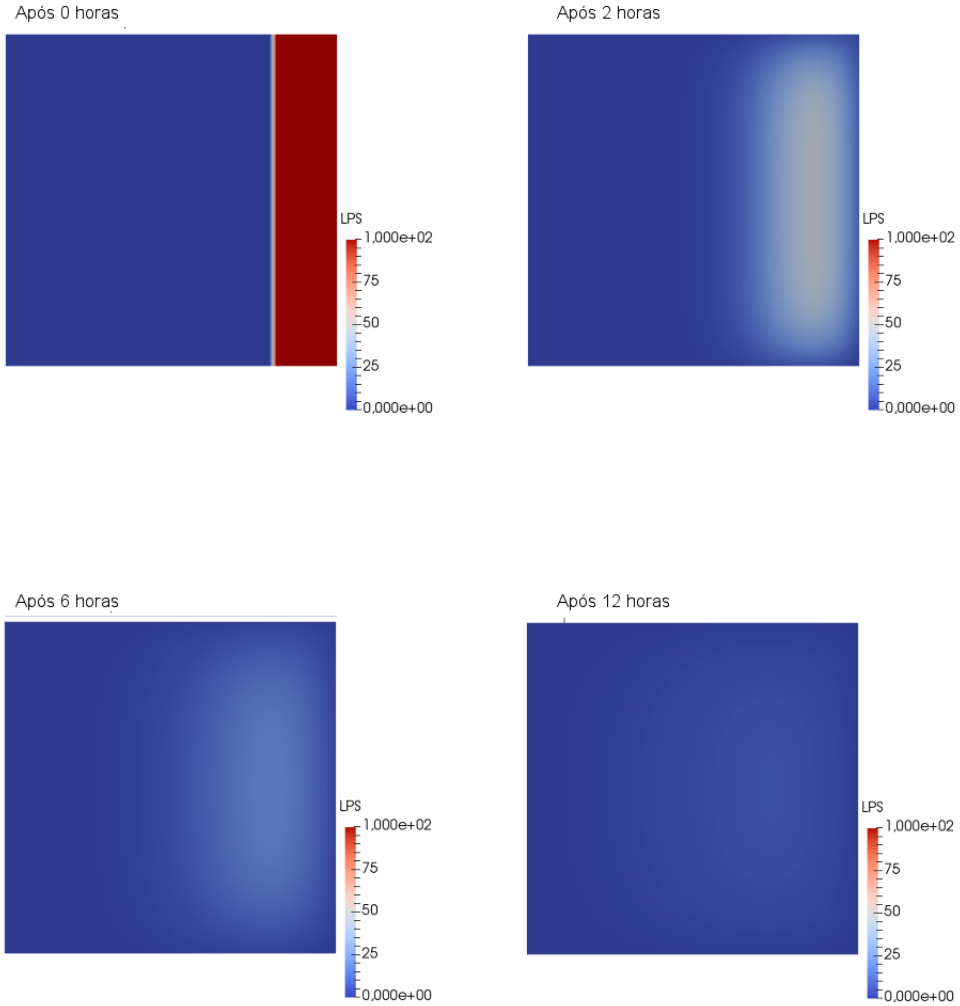

Figura 8 – Evolução do LPS ao longo da simulação nos pontos ($x = 0$, y, z).

fim, duas versões, 5) a que usa o esquema de balanceamento de carga estático e 6) a que usa o esquema dinâmico, onde ambas as versões usam todos os dispositivos disponíveis, *i.e.* CPUs e GPUs, na computação.

Para cada versão do SIH, foram realizadas 3 execuções, em seguida, foram calculados as médias, e os desvios permaneceram abaixo de 3%, onde o desvio é calculado pela equação 5.1, onde t_1, t_2 e t_3 são os tempos medidos para as 3 execuções. O *cluster* usa uma fila de sistema que garante exclusividade no acesso à cada máquina pela aplicação, para que processos de outros usuários não interfiram no tempo total de computação. Os tempos encontrados neste trabalho apresentam uma diferença em relação aos encontrados em [17] devido à alterações na versão dos softwares dos computadores do *cluster*.

$$máx\left(\left|máx(t_1,t_2,t_3) - \left(\frac{t_1+t_2+t_3}{3}\right)\right|, \left|mín(t_1,t_2,t_3) - \left(\frac{t_1+t_2+t_3}{3}\right)\right|\right) \tag{5.1}$$

Essa seção é dedicada a detalhar cada versão e seus respectivos experimentos, assim

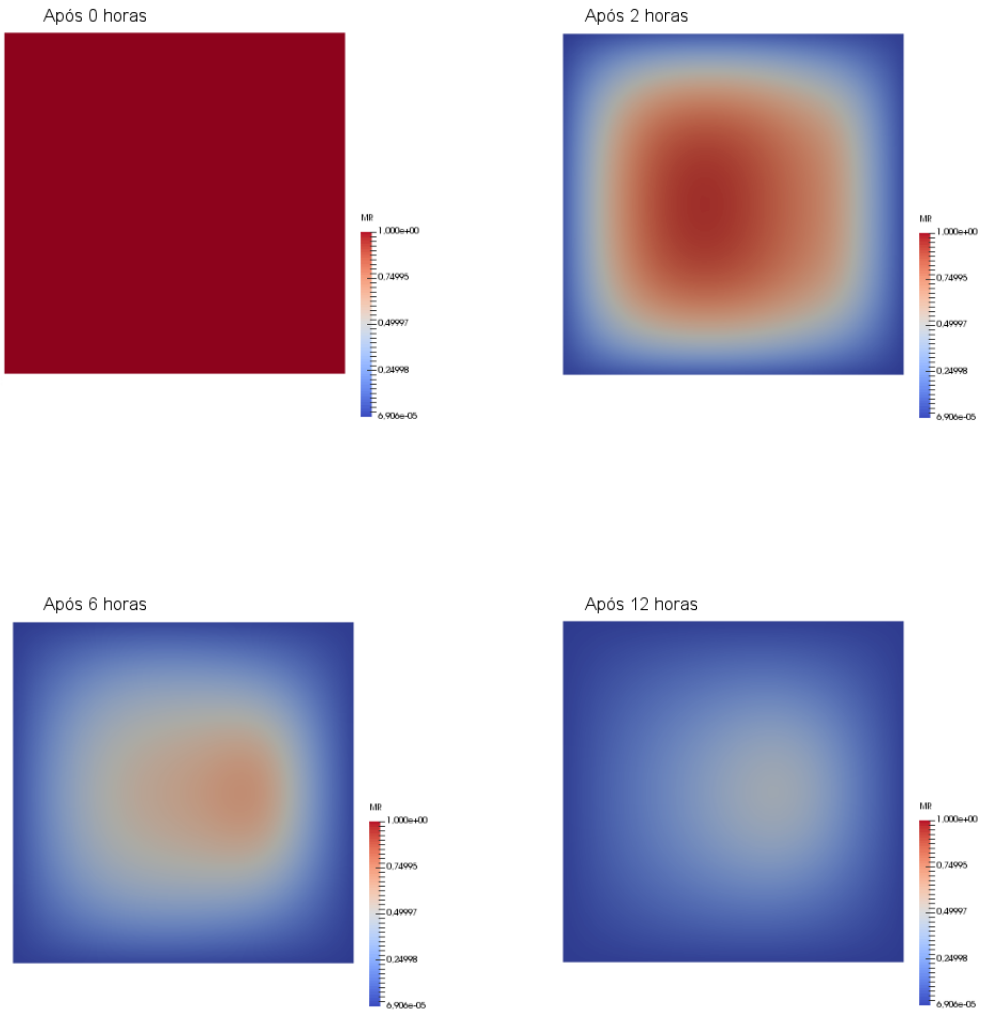

Figura 9 – Evolução do MR ao longo da simulação nos pontos ($x = 0$, y, z).

como os parâmetros utilizados no mesmo.

5.3.1 Parâmetros dos Experimentos

Uma malha de tamanho $50 \times 50 \times 3200$ foi usada para realizar os experimentos. Os valores usados para indicar as condições iniciais e os parâmetros da simulação são descritos no apêndice A. Um total de 100.000 passos de tempo foram executados. Tanto para a versão de balanceamento estática quanto dinâmica do algoritmo, o intervalo para balanceamento de carga é igual à 1% dos passos de tempo (1.000). Para a versão dinâmica, o *threshold* de balanceamento de carga é igual à $0,0025\%$, *i.e.* se a diferença da carga anterior para a atual for menor do que 200 pontos, as cargas não são redistribuidas [18]. Os *kernels* são executados em 16 *work-groups* de tamanho 64 para as CPUs, e 16 *work-groups* de tamanho 512 para as GPUs. O algoritmo de balanceamento de carga dinâmico executa periodicamente, onde o periodo é definido pelo intervalo de balanceamento de carga, a fim de ajustar o montante de dados que cada dispositivo irá receber até o final da computação.

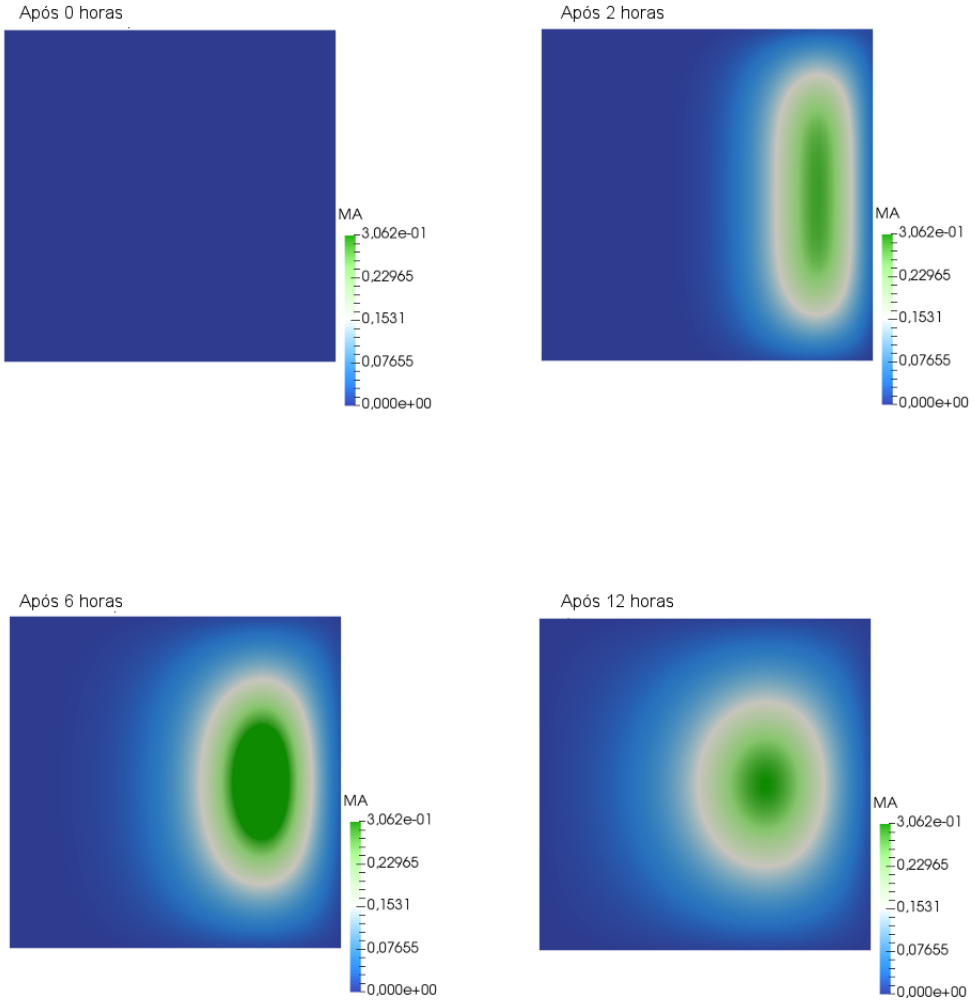

Figura 10 – Evolução do MA ao longo da simulação nos pontos ($x = 0$, y, z).

O uso de uma malha com dimensão maior no eixo z foi escolhida para minimizar o custo adicional da troca de bordas, já que os eixos x e y definem a quantidade de pontos representados por uma borda. Aumentar a quantidade de pontos internos ao mesmo tempo em que o tamanho da borda permanece o mesmo tem impactos diretos na escalabilidade do algoritmo, sendo possível obter ganhos em desempenho mais próximos aos lineares conforme o número de dispositivos aumente. Em condições ideais, o gargalo imposto pelo tempo de comunicação envolvido na troca de bordas não existiria, cada dispositivo teria acesso direto à memória do dispositivo vizinho, dispensando operações de envio de dados. Consequentemente, seria possível obter ganhos de desempenho lineares, pois seria levado em consideração apenas o tempo de computação dos pontos internos no tempo de execução do simulador.

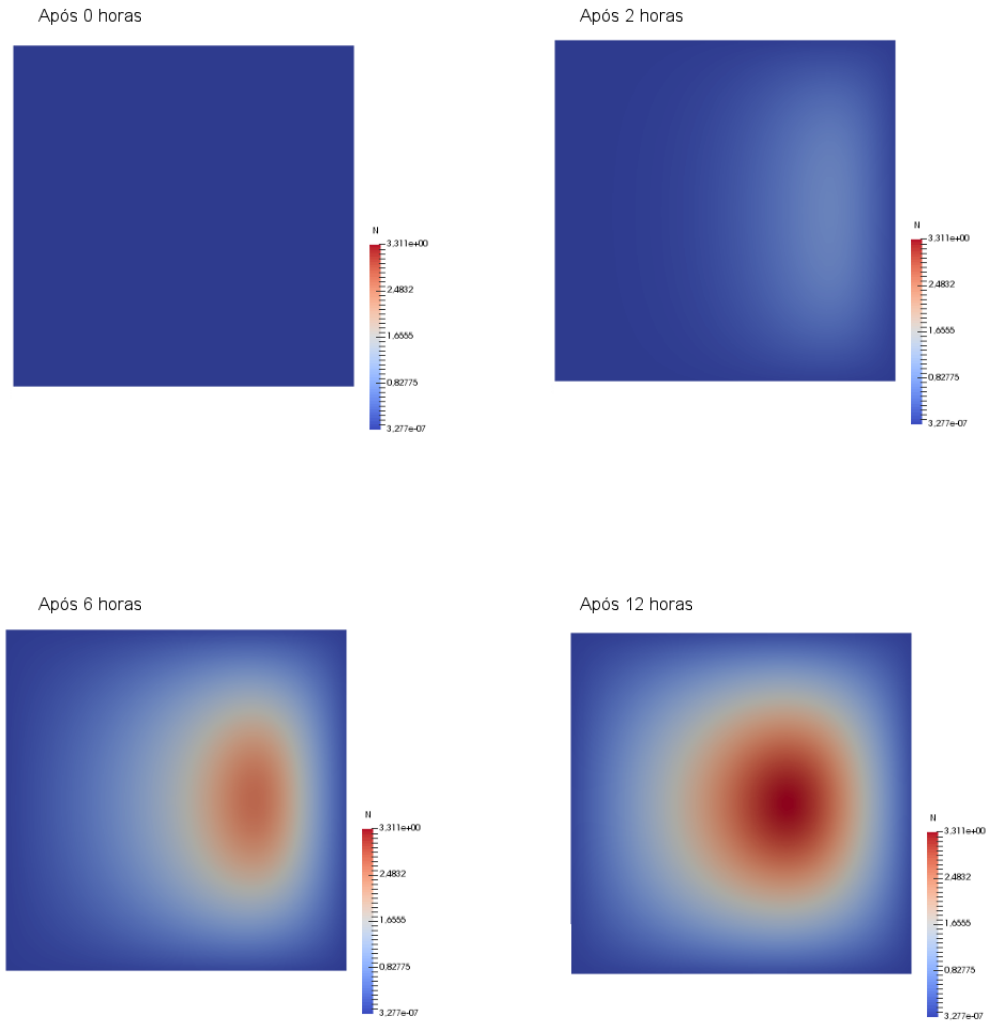

Figura 11 – Evolução do N ao longo da simulação nos pontos ($x = 0$, y, z).

5.3.2 Versão Sequencial

A versão sequencial é executada em apenas um dos núcleos de processamento da CPU. Devido a essa limitação de recursos computacionais, a duração do tempo que a versão sequencial levaria para calcular os 100.000 passos de tempo da simulação é extremamente demorada, portanto, foi decidido estimar o tempo total de execução usando para isso 10.000 passos de tempo, o que leva 45.173 segundos para computar. Estima-se que o tempo de computação necessário para executar 100.000 passos de tempo seja de 451.737 segundos, ou mais de 5 dias. Uma simulação típica de 1.000.000 de passos de tempo, que representa 24 horas de infecção, levaria mais de 50 dias para finalizar.

5.3.3 Versão Sem Balanceamento de Carga entre CPUs

Como a versão sequencial usa apenas um dos núcleos de processamento da CPU. Uma alternativa para melhorar o desempenho é usar todos os núcleos de processamento

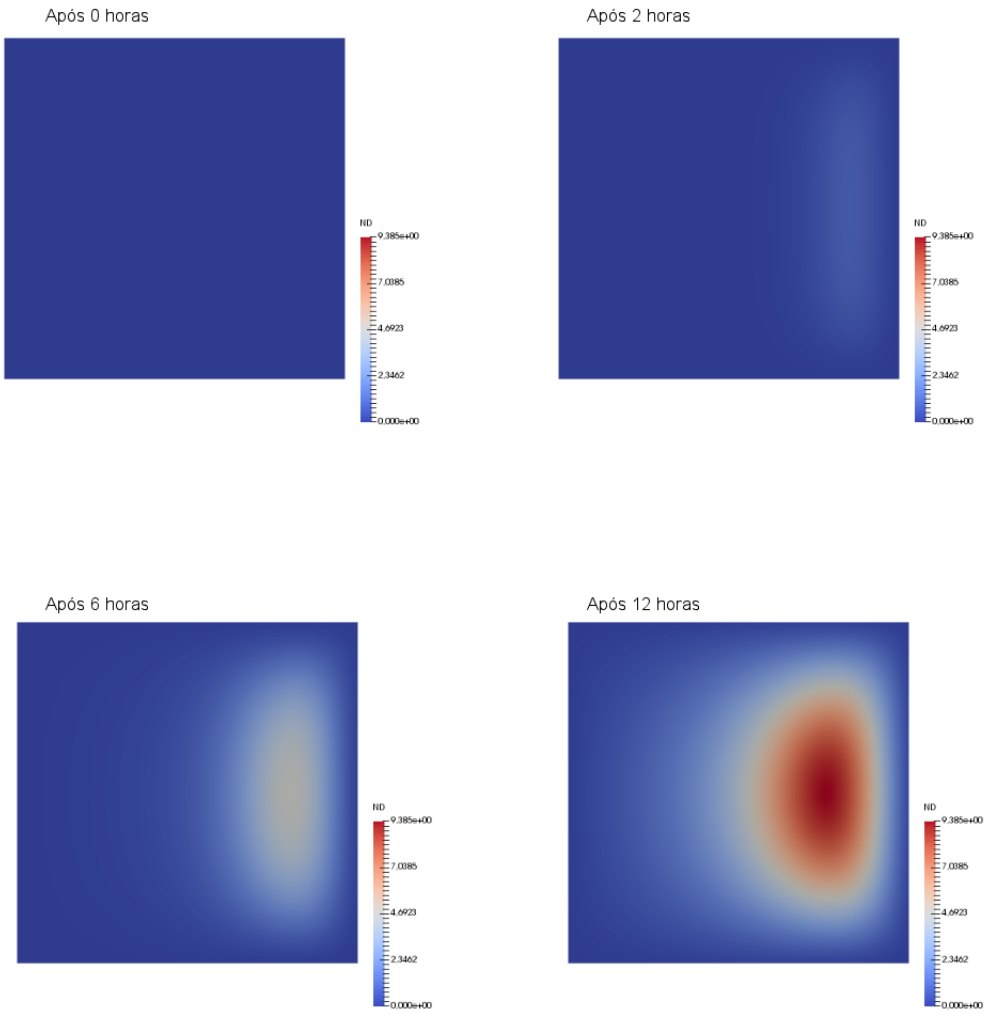

Figura 12 – Evolução do ND ao longo da simulação nos pontos ($x = 0$, y, z).

disponíveis em um nó para realizar a computação. Além disso, é possível aumentar o número de computadores usados na computação, já que o *cluster* usado neste trabalho dispõe de 6 nós. Como as CPUs apresentam uma grande quantidade de núcleos de processamento (32), é possível obter ganhos de desempenho significativos quando esta versão é comparada à versão sequencial apresentada na seção 5.3.2, chegando a até 48 vezes nas execuções com 6 nós do *cluster*. Os resultados são apresentados na Tabela 2.

5.3.4 Versão Sem Balanceamento de Carga entre GPUs

CPUs são dispositivos limitados quanto à sua capacidade de processamento em aplicações que usam o modelo de paralelismo de dados. A razão para isto está no número relativamente pequeno de núcleos de computação. Por outro lado, as GPUs são arquiteturas computacionais desenvolvidas especificamente para acelerar aplicações SIMD, já que possuem centenas ou até mesmo milhares de núcleos de processamento.

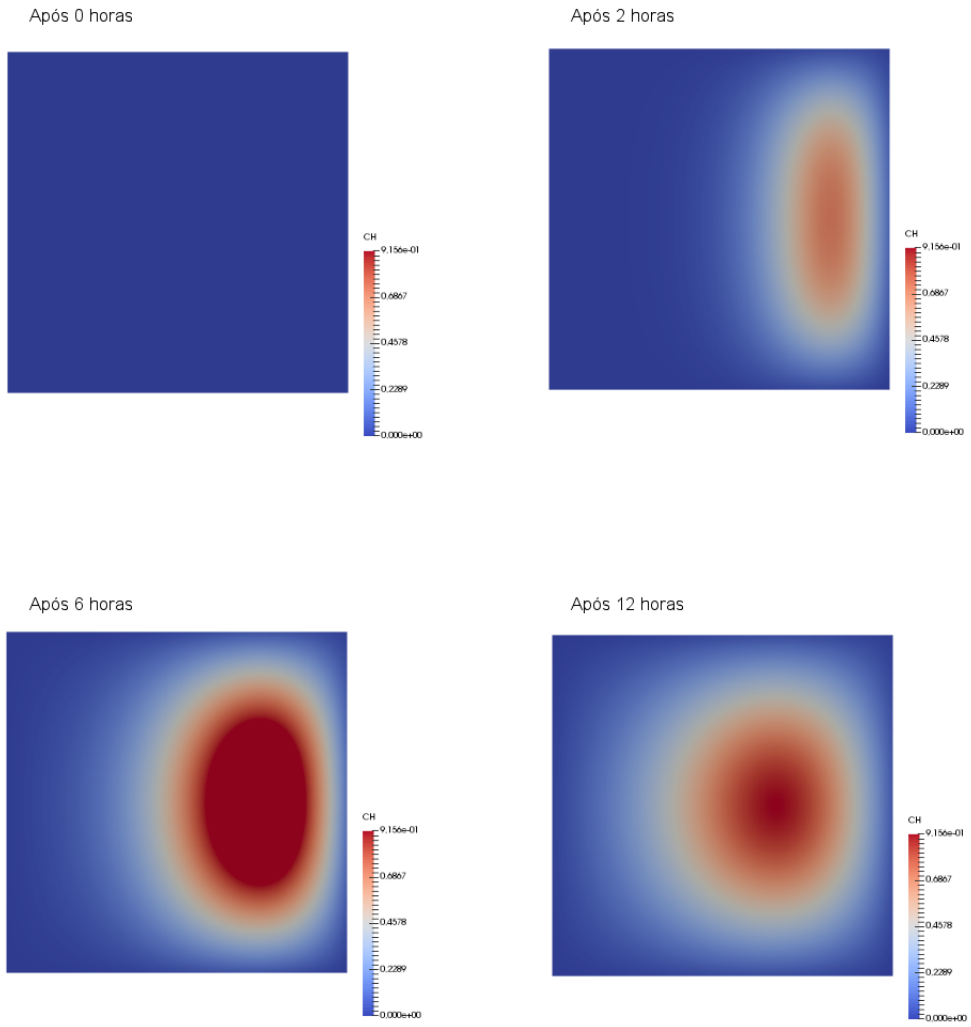

Figura 13 – Evolução do CH ao longo da simulação nos pontos ($x = 0$, y, z).

Tabela 2 – Resultados experimentais da versão paralela, usando CPUs e sem emprego de balanceamento de carga, visto que as CPUs são homogêneas. Apresenta-se para cada configuração o tempo médio de execução, o desvio-padrão (valor percentual) e a aceleração em relação à versão sequencial.

	Tempo(s)	Desvio-padrão(%)	Aceleração
32 núcleos CPU	38.484	0,08	11,7
64 núcleos CPU	20.961	0,08	21,6
96 núcleos CPU	15.431	0,17	29,3
128 núcleos CPU	11.863	0,01	38,1
160 núcleos CPU	10.395	0,07	43,6
192 núcleos CPU	9.367	0,02	48,2

A segunda abordagem para melhorar o desempenho da execução sequencial do SIH é, portanto, substituir as CPUs do sistemas pelas GPUs. Neste contexto, as CPUs tem o papel de *host*, ou seja, são usadas apenas para enviar as tarefas para que as GPUs façam a computação, além de serem responsáveis pela comunicação.

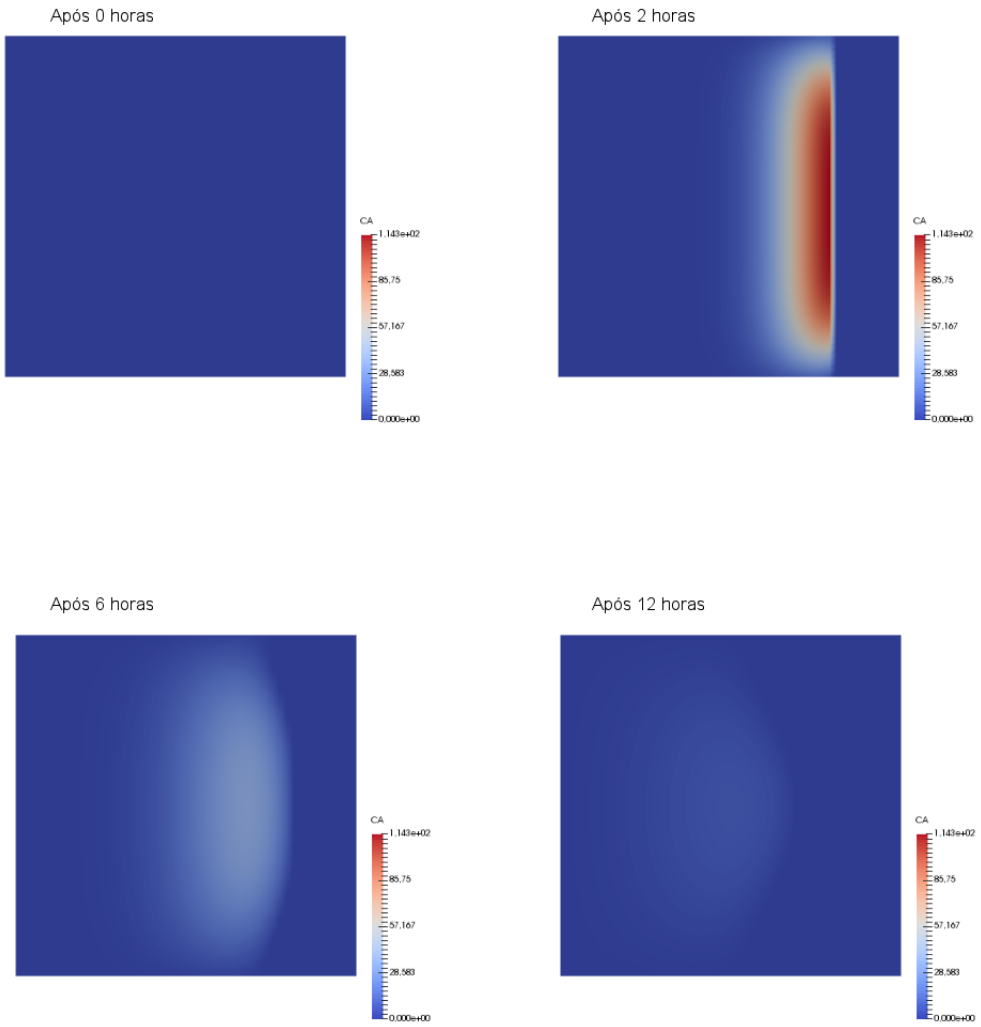

Figura 14 – Evolução do CA ao longo da simulação nos pontos ($x = 0$, y, z).

Usando os 6 nós do *cluster*, é possível observar ganhos de desempenho que chegam à 702 vezes quando comparados com a versão sequencial apresentada na seção 5.3.2. A Tabela 3 apresenta os resultados.

Tabela 3 – Resultados experimentais da versão paralela, usando GPUs e sem emprego de balanceamento de carga, visto que as GPUs são homogêneas. Apresenta-se para cada configuração o tempo médio de execução, o desvio-padrão (valor percentual) e a aceleração em relação à versão sequencial.

	Tempo(s)	Desvio-padrão(%)	Aceleração
896 núcleos GPU	3.494	0,002	129,3
1.792 núcleos GPU	1.760	0,04	256,7
1.344 núcleos GPU	1.206	0,3	374,6
2.688 núcleos GPU	910	0,7	496,4
4.480 núcleos GPU	750	0,5	602,3
5.376 núcleos GPU	643	0,6	702,5

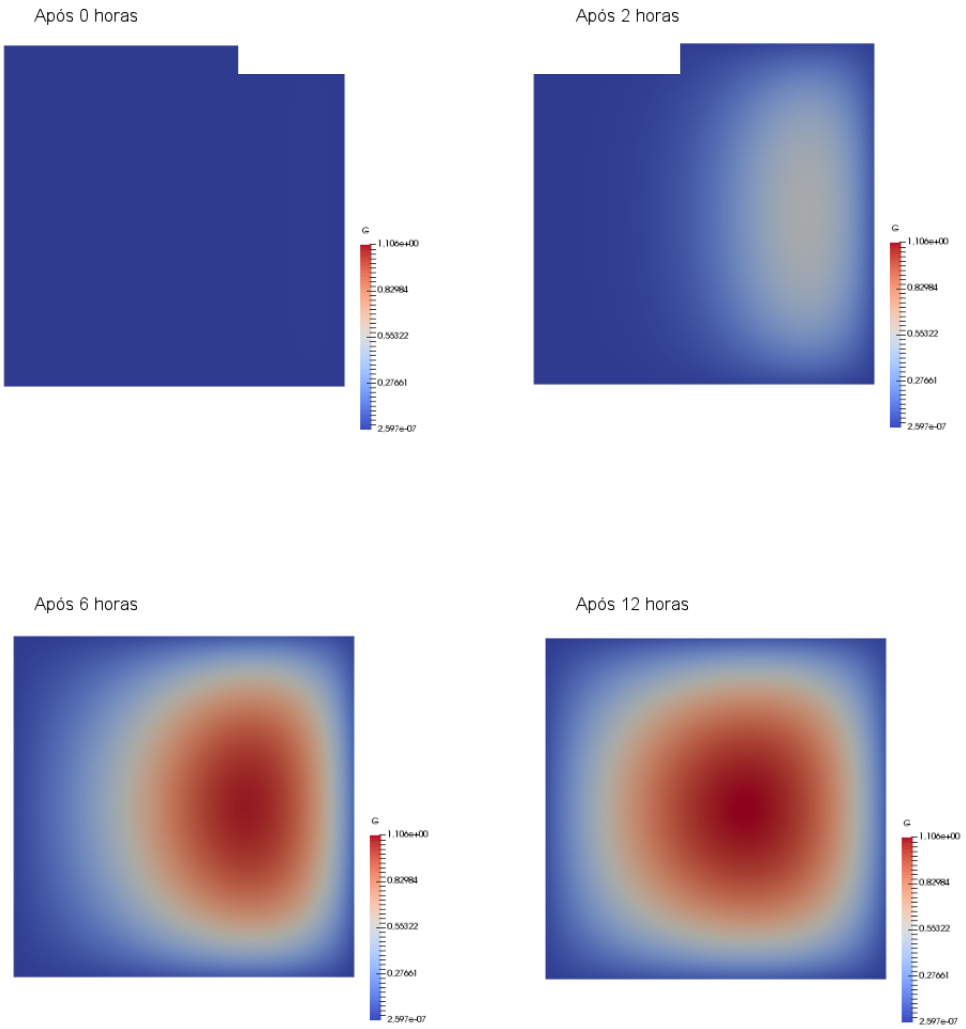

Figura 15 – Evolução do G ao longo da simulação nos pontos ($x = 0$, y, z).

5.3.5 Versão Sem Balanceamento de Carga em CPUs e GPUs

Apesar da versão paralela usando GPUs apresentada na seção 5.3.4 apresentar ótimos ganhos de desempenho, ainda é possível obter melhorias, já que é possível usar as CPUs na computação de forma colaborativa com as GPUs, pois suas únicas funções são enviar as tarefas para computação pelas GPUs e realizar a comunicação, ficando em um estado ocioso durante a maior parte da execução do simulador do SIH.

Uma abordagem inicial para incluir as CPUs no sistema é distribuir as cargas igualmente entre CPUs e GPUs, para que todos os dispositivos possam realizar a computação. Os resultados são mostrados na tabela 4.

É possível observar que tal abordagem não é apropriada para dispositivos heterogêneos. Apesar dos ganhos de desempenho serem melhores quando comparados à versão sequencial apresentada na seção 5.3.2 (ganhos de até 215 vezes), e do que a versão paralela com CPUs apresentada na seção 5.3.3 (até 4 vezes), a versão paralela usando apenas GPUs

Tabela 4 – Resultados experimentais da versão paralela, usand tanto as CPUs quanto as GPUs, mas sem emprego de balanceamento de carga. Apresenta-se para cada configuração o tempo médio de execução, o desvio-padrão (valor percentual) e a aceleração em relação à versão sequencial.

	Tempo(s)	Desvio-padrão(%)	Aceleração
896 núcleos GPU e 32 núcleos CPU	15.637	0,03	28,9
1.792 núcleos GPU e 64 núcleos CPU	6.212	0,04	72,7
1.344 núcleos GPU e 96 núcleos CPU	4.144	0,05	109,0
2.688 núcleos GPU e 128 núcleos CPU	6.738	0,03	67,0
4.480 núcleos GPU e 160 núcleos CPU	2.495	0,19	181,1
5.376 núcleos GPU e 192 núcleos CPU	2.100	0,07	215,1

apresentada na seção 5.3.4 apresenta melhor desempenho, com ganhos de até 3 vezes.

5.3.6 Versão Com Balanceamento de Carga Estático

Os resultados observados na versão sem balanceamento de carga em CPUs e GPUs apresentados na seção 5.3.5 reforçam a necessidade de se conhecer as capacidades computacionais de cada dispositivo que participa da computação, a fim de que se possa designar as cargas de forma que nenhum dispositivo fique em um estado ocioso, sem carga, enquanto outro continua computando.

A versão de balanceamento de carga estático implementa o algoritmo 1, cujo objetivo é calcular as capacidades computacionais dos dispositivos e designar as cargas em um estágio inicial da computação, a fim de equalizar o tempo de computação das cargas em cada dispositivo. A Tabela 5 apresenta os resultados.

Tabela 5 – Resultados experimentais da versão paralela, usand tanto as CPUs quanto as GPUs, com emprego de balanceamento de carga estático. Apresenta-se para cada configuração o tempo médio de execução, o desvio-padrão (valor percentual) e a aceleração em relação à versão sequencial.

	Tempo(s)	Desvio-padrão(%)	Aceleração
896 núcleos GPU e 32 núcleos CPU	3.574	1,07	126,4
1.792 núcleos GPU e 64 núcleos CPU	1.832	0,67	246,6
1.344 núcleos GPU e 96 núcleos CPU	1.243	2,69	363,4
2.688 núcleos GPU e 128 núcleos CPU	937	2,22	482,1
4.480 núcleos GPU e 160 núcleos CPU	753	1,17	600,0
5.376 núcleos GPU e 192 núcleos CPU	644	0,93	701,4

Os ganhos de desempenho são observados quando comparados com a versão sequencial, com a versão paralela em CPUs e com a versão sem balanceamento em CPUs e GPUs, com ganhos de desempenho de 701 vezes, 15 vezes e 3 vezes, respectivamente. Este resultado indica que a decisão de incluir um esquema de balanceamento de carga foi essencial para melhorar o desempenho da computação do simulador do SIH nos dispositivos heterogêneos do *cluster*. Entretanto, a abordagem de balanceamento de carga estática não

é a melhor versão, já que a mesma apresenta uma perda de desempenho de $0,2\%$ quando comparada a melhor versão sem balanceamento em GPUs, ou seja, a inclusão das CPUs na computação não está de fato contribuindo na melhora do desempenho da aplicação.

5.3.7 Versão Com Balanceamento de Carga Dinâmico

A segunda abordagem envolve a versão que implementa o algoritmo de balanceamento dinâmico, apresentado no Algoritmo 2, cujo objetivo é realizar ajustes nas cargas de todos os dispositivos durante toda a execução do simulador do SIH. Os resultados da execução utilizando o algoritmo dinâmico são apresentados na Tabela 6.

Tabela 6 – Resultados experimentais da versão paralela, usand tanto as CPUs quanto as GPUs, com emprego de balanceamento de carga dinâmico. Apresenta-se para cada configuração o tempo médio de execução, o desvio-padrão (valor percentual) e a aceleração em relação à versão sequencial.

	Tempo(s)	Desvio-padrão(%)	Aceleração
896 núcleos GPU e 32 núcleos CPU	3.261	$0,09$	$138,5$
1.792 núcleos GPU e 64 núcleos CPU	1.661	$0,09$	$272,0$
1.344 núcleos GPU e 96 núcleos CPU	1.145	$0,54$	$394,5$
2.688 núcleos GPU e 128 núcleos CPU	979	$1,14$	$461,4$
4.480 núcleos GPU e 160 núcleos CPU	735	$0,10$	$614,6$
5.376 núcleos GPU e 192 núcleos CPU	632	$0,11$	$714,8$

Como a Tabela 6 mostra, a decisão de ajustar as cargas durante toda a simulação melhorou o desempenho desta versão em relação as demais, já que apresenta o menor tempo obtido entre todos os experimentos (632 segundos), sendo possível acelerar a execução em 714 vezes, quando comparado ao tempo da versão sequencial.

5.4 Análise dos Esquemas de Balanceamento de Carga

A Figura 16 apresenta uma comparação entre os ganhos de desempenho obtidos por cada versão, quando comparam-se os melhores tempos destas.

Apesar de se esperar que o uso de todos os recursos computacionais, aliados ao uso de um algoritmo de balanceamento de carga, levasse a uma melhora no desempenho do simulador do SIH, não era esperado que a versão dinâmica superasse a estática em termos de desempenho, já que o simulador do SIH é uma aplicação supostamente regular, onde as mesmas instruções são executadas a cada passo de tempo da simulação, o que consequentemente resultaria em um tempo igual de computação dos passos de tempo. Consequentemente, seria possível obter as cargas ideais para cada dispositivo em um estágio inicial da simulação, o que favoreceria o esquema de balanceamento estático. Posteriormente, descobrimos que o SIH é de fato uma aplicação com características irregulares, onde inúmeros fatores contribuem para tal irregularidade, que serão discutidos a seguir.

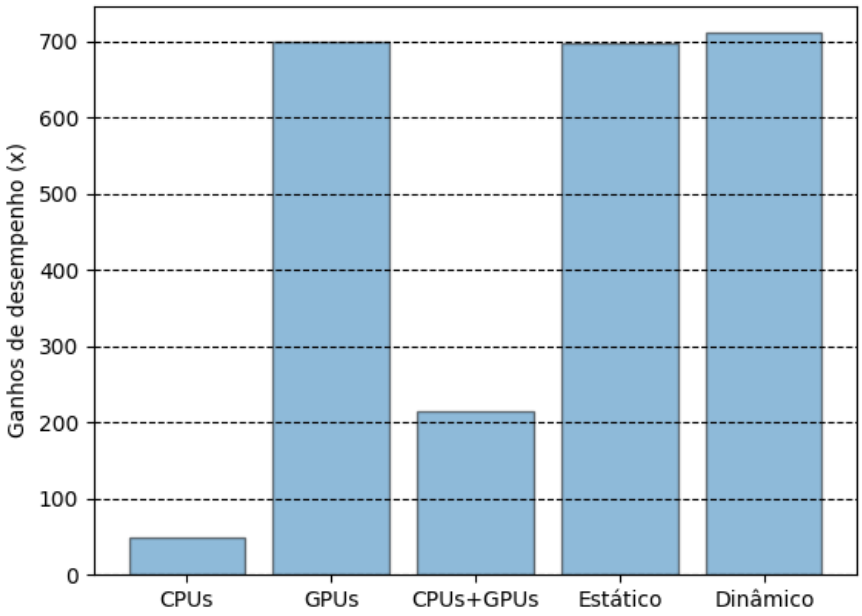

Figura 16 – Ganhos de desempenho de todas as versões, quando o melhor tempo de cada uma é comparado com a versão sequencial.

Para melhor entender porque o algoritmo de balanceamento de carga dinâmico teve um desempenho melhor do que o estático, foi realizado um experimento onde foram medidos para cada algoritmo, dinâmico e estático, os tempos de execução em cada dispositivo, acumulados durante os últimos 1.000 passos de tempo, assim como a quantidade de dados que cada dispositivo recebe para computar. As execuções foram realizadas 3 vezes tanto para a versão estática quanto a dinâmica, com as respectivas médias e desvios calculados. As Tabelas 7 e 8 apresentam o tempo de execução para cada dispositivo para os esquemas de balanceamento de carga estático e dinâmico, respectivamente. Além disso, a Figura 17 mostra a quantidade de dados que cada dispositivo recebe para computar, para ambos os esquemas de balanceamento. Nesta figura, apenas a execução em 6 nós é apresentada para fins de comparação entre os esquemas de balanceamento e a quantidade de dados apresentada para a versão de balanceamento de carga dinâmica é relativa ao último balanceamento, realizado no passo de tempo 99.000, enquanto que para a versão estática, a quantidade de dados apresentada é relativa ao balanceamento de carga realizado no passo de tempo 1.000, que é o único ajuste de carga feito após o *probing*.

Como pode ser observado, a Tabela 7 mostra que a versão de balanceamento de carga estática sofre um desbalanceamento no tempo de execução dos dispositivos. Por exemplo, usando 6 nós, a diferença do tempo de computação entre o dispositivo mais rápido e o mais lento é de aproximadamente 14%; já para o algoritmo de balanceamento de carga dinâmico (Tabela 8), a maior diferença entre os dispositivos é aproximadamente

Tabela 7 – Tempo de computação em cada dispositivo para a versão estática. Os tempos são medidos em segundos. Tempos em negritos indicam desbalanceamentos. Os desvios permaneceram abaixo de 5%.

	1 Nó	2 Nós	3 Nós	4 Nós	5 Nós	6 Nós
Tempo (s) GPU #1	3.157	1.578	1.053	817	646	540
Tempo (s) GPU #2	3.162	1.583	1.051	815	642	541
Tempo (s) CPU #1	**3.507**	**1.429**	**949**	**665**	**537**	**470**
Tempo (s) GPU #3		1.582	1.057	818	646	544
Tempo (s) GPU #4		1.581	1.052	814	644	543
Tempo (s) CPU #2		**1.781**	**1.008**	**668**	**548**	**480**
Tempo (s) GPU #5			1.054	815	646	539
Tempo (s) GPU #6			1.055	811	647	545
Tempo (s) CPU #3			**1.187**	**533**	**562**	**461**
Tempo (s) GPU #7				818	645	545
Tempo (s) GPU #8				815	648	544
Tempo (s) CPU #4				**808**	**517**	**482**
Tempo (s) GPU #9					652	552
Tempo (s) GPU #10					649	542
Tempo (s) CPU #5					**637**	**530**
Tempo (s) GPU #11						547
Tempo (s) GPU #12						539
Tempo (s) CPU #6						**508**

Tabela 8 – Tempo de computação em cada dispositivo para a versão dinâmica. Os tempos são medidos em segundos. Os desvios permaneceram abaixo de 1%.

	1 Nó	2 Nós	3 Nós	4 Nós	5 Nós	6 Nós
Tempo (s) GPU #1	3.187	1.582	1.055	800	638	536
Tempo (s) GPU #2	3.187	1.582	1.055	800	638	536
Tempo (s) CPU #1	3.184	1.578	1.051	798	636	534
Tempo (s) GPU #3		1.582	1.055	800	638	536
Tempo (s) GPU #4		1.582	1.055	800	638	536
Tempo (s) CPU #2		1.580	1.051	798	636	534
Tempo (s) GPU #5			1.055	800	638	536
Tempo (s) GPU #6			1.055	800	638	536
Tempo (s) CPU #3			1.053	833	636	534
Tempo (s) GPU #7				800	638	536
Tempo (s) GPU #8				800	638	536
Tempo (s) CPU #4				799	636	534
Tempo (s) GPU #9					638	536
Tempo (s) GPU #10					638	536
Tempo (s) CPU #5					637	535
Tempo (s) GPU #11						536
Tempo (s) GPU #12						536
Tempo (s) CPU #6						535

$1,2\%$. Podemos, portanto, atribuir este desbalanceamento à má distribuição das cargas, durante toda a simulação, entre os dispositivos presentes na computação na versão estática,

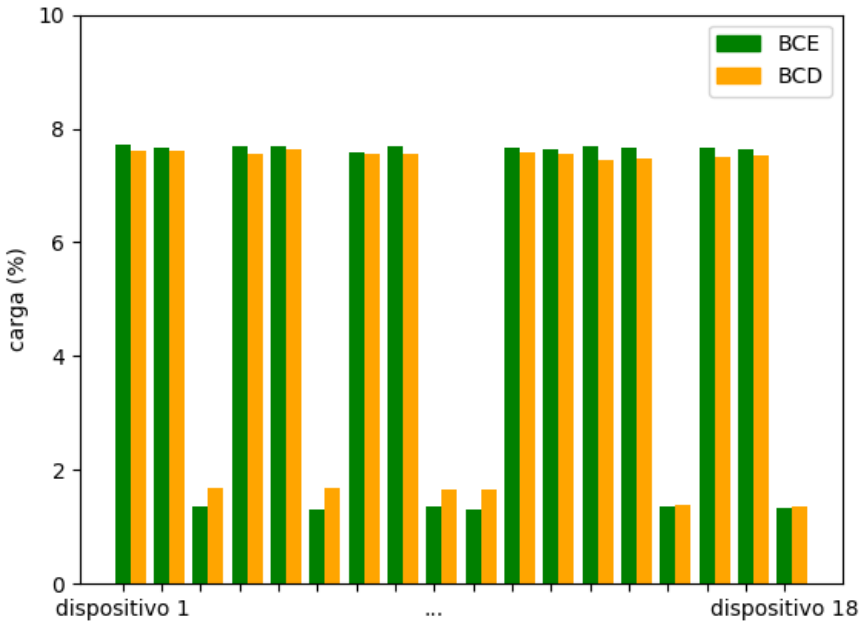

Figura 17 – Quantidade de dados designado para cada dispositivo após o último ajuste de carga, ao executar em 6 nós. BCE é uma abreviação para balanceamento de carga estático e BCD para balanceamento de carga dinâmico.

o que é corrigido para a versão dinâmica, já que as cargas computadas para esta versão apresentam tempos de computação de cada dispositivo muito próximos uns aos outros.

A Equação de balanceamento de carga 4.6 assume que o tempo de computação de um dispositivo é diretamente proporcional a carga que este recebe, *i.e.* se dobrarmos a carga, o tempo de computação consequentemente irá dobrar. Caso a relação entre tempo de computação e carga não seja diretamente proporcional, o esquema de balanceamento de carga terá dificuldades para encontrar as cargas ideias, já que o esquema irá assumir que as cargas calculadas irão equilibrar os tempos dos dispositivos para os próximos passos de tempo.

A fim de verificar se o tempo de computação é diretamente proporcional à carga do dispositivo, foi realizada uma série de pequenos experimentos onde, para 1.000 passos de tempo (o intervalo de balanceamento estabelecido nos experimentos desta seção), foi executado o simulador do SIH para malhas com diferentes tamanhos $\{50 \times 50 \times 320$, $50 \times 50 \times 640$, $50 \times 50 \times 960$, $\ldots$, $50 \times 50 \times 3200\}$. Os tempos de computação para cada experimentos foram medidos 3 vezes, com as respectivas médias e desvios calculados, e são apresentados na Tabela 9, tanto para GPUs quanto para CPUs, assim como o tempo de computação esperado, *i.e.* o tempo que deveria ser medido caso a carga fosse diretamente proporcional ao tempo de computação.

Como pode-se observar nas Tabelas 9 e 10, a medida que a carga aumenta, o tempo

Tabela 9 – Comparação em uma CPU do *cluster* entre o tempo de computação, medido em segundos, e o tempo de computação esperado caso o este tivesse um aumento proporcional ao aumento da malha inicial de $50 \times 50 \times 320$. As execuções foram realizadas em 1.000 passos de tempo. Os desvios permaneceram abaixo de 3%. Todos os núcleos da CPU foram usados neste experimento.

Tamanho da Malha	Tempo CPU (s)	Tempo CPU Esperado (s)
$50 \times 50 \times 320$	44	–
$50 \times 50 \times 640$	78	88
$50 \times 50 \times 960$	111	132
$50 \times 50 \times 1.280$	146	176
$50 \times 50 \times 1.600$	178	220
$50 \times 50 \times 1.920$	212	263
$50 \times 50 \times 2.240$	246	307
$50 \times 50 \times 2.560$	280	351
$50 \times 50 \times 2.880$	314	395
$50 \times 50 \times 3.200$	347	439

Tabela 10 – Comparação em uma GPU do *cluster* entre o tempo de computação, medido em segundos, e o tempo de computação esperado caso o este tivesse um aumento proporcional ao aumento da malha inicial de $50 \times 50 \times 320$. As execuções foram realizadas em 1.000 passos de tempo. Os desvios permaneceram abaixo de 1%. Todos os núcleos da GPU foram usados neste experimento.

Tamanho da Malha	Tempo GPU (s)	Tempo GPU Esperado (s)
$50 \times 50 \times 320$	4	–
$50 \times 50 \times 640$	7	7
$50 \times 50 \times 960$	10	11
$50 \times 50 \times 1.280$	14	15
$50 \times 50 \times 1.600$	17	18
$50 \times 50 \times 1.920$	21	22
$50 \times 50 \times 2.240$	24	26
$50 \times 50 \times 2.560$	28	30
$50 \times 50 \times 2.880$	31	33
$50 \times 50 \times 3.200$	35	37

de computação aumenta em uma proporção menor do que seria aguardada, para ambos os dispositivos. A situação é mais grave para a execução em CPU, onde para muitos dos experimentos os erros ficam na casa dos 25%. A versão estática não tem a oportunidade de corrigir essa situação pelo fato de fazer o cálculo e distribuir as cargas uma única vez, o que explica as diferenças nas distribuições de cargas encontradas nas versões estática e dinâmica.

Outro ponto a se observar está na estrutura da malha. Como o balanceamento de carga para a versão estática ocorre no passo de tempo 1.000, quando os pontos nulos ainda são predominantes na malha, os cálculos são feitos mais rapidamente devido à otimizações aritméticas realizadas pelo compilador. Conforme o número de pontos nulos

Tabela 11 – Tempos individuais (s) de cada etapa da simulação da versão de balanceamento dinâmica. Os desvios permaneceram abaixo de 2%.

	Computação dos pontos internos e troca de bordas	Computação das bordas	Balanceamento de carga	Total
1 nó	3.191, 8	68, 5	0, 1	3.260, 4
2 nós	1.589, 2	71, 4	0, 3	1.660, 9
3 nós	1.069, 3	75, 1	0, 3	1.144, 7
4 nós	881, 6	96, 5	0, 7	978, 8
5 nós	655, 9	78, 0	0, 5	734, 4
6 nós	553, 3	78, 1	0, 6	632, 0

diminui no decorrer da simulação, o tempo de execução para os dispositivos se torna mais lento. Apenas o algoritmo de balanceamento de carga dinâmico é capaz de se ajustar à essa mudança, como indicado pela Tabela 8. Esta observação também ajuda a explicar o desbalanceamento encontrado na versão de balanceamento estática, e porque a versão dinâmica tem um desempenho melhor. Tais otimizações são abordadas com mais detalhes na seção 5.5.

Um fator que poderia introduzir um problema para o esquema de balanceamento de carga dinâmico é o fato de que a simulação é realizada em um ambiente de memória distribuída (*cluster*), o que faz com que os custos de comunicação sejam muitos altos já que os dados trafegam em uma rede de comunicação de alta latência, Gigabit Ethernet no caso deste trabalho. Entretanto, os custos de comunicação não introduzem um gargalo significativo ao algoritmo de balanceamento dinâmico, que já este tem um desempenho melhor do que o estático.

A fim de compreender porque a comunicação realizada na simulação do SIH não afeta negativamente o esquema de balanceamento dinâmico, foi realizado um experimento onde os tempos individuais de cada etapa envolvida na simulação de um passo de tempo são medidos, em 3 execuções de 100.000 passos de tempo com intervalos de balanceamento de carga iguais à 1.000, ou seja, são feitas 100 tentativas de ajustes de carga no total, visto que o ajuste só será aplicado se a quantidade de modificações for maior que o *threashold*, as médias e desvios são então calculadas. As etapas de um passo de tempo consistem em: computar os pontos internos sobreposto ao processo de troca de bordas; computar os pontos de borda e realizar o balanceamento de carga para cada um dos dispositivos. Os resultados são apresentados na Tabela 11.

Como pode ser observado, os custos de balanceamento são negligíveis, portanto, o uso do balanceamento dinâmico introduz custos baixos de comunicação. Por este motivo, o ganho de desempenho obtido pelos ajustes de carga compensam a perda de desempenho causada pelo custos de comunicação relacionados ao ajuste das cargas dos dispositivos, que é mínimo.

5.5 Otimizações do Compilador

O estado da malha nos passos iniciais da simulação dificulta o processo de encontrar uma carga ótima para os dispositivos, já que nos primeiros passos da simulação apenas uma região pequena da malha possui valores não nulos. Processos físicos, como a difusão do LPS pelo tecido ou a quimiotaxia decorrente de sua presença, fazem com que as concentrações das populações do SIH se alterem ao longo do tempo e do espaço. Em termos computacionais, com o progresso da simulação, o número de pontos nulos, que representam a ausência de uma população em um determinado ponto do tecido, diminui, já que a medida que o tempo passa as populações simuladas vão sendo transportadas para pontos da malha que eram nulos.

Essa situação acaba impactando o tempo de computação e, consequentemente, os algoritmos de balanceamento de carga. O tempo de computação na CPU é menor quando os cálculos são realizados com valores nulos devido à otimizações aritméticas realizadas pelo compilador: quando um operando é zero, uma operação envolvendo cálculos ponto-flutuante não precisa ser realizada para que o resultado seja conhecido. Entretanto, quando os valores não são nulos, o cálculo númerico das EDPs requer a execução de todos os estágios do *pipeline* aritmético ponto-flutuante, que pode levar dezenas de ciclos de *clock* para executar, consequentemente aumentando o tempo de computação. Portanto, alguns dispositivos podem ficar ociosos enquanto outros realizam computações mais demoradas, dependendo da quantidade de pontos nulos presentes no conjunto de dados que lhe foi designado.

Se uma porção da malha, composta em sua maioria por valores nulos, for designada para uma CPU durante a fase de medição de desempenho dos dispositivos, por exemplo, o seu tempo de computação será reduzido devido às otimizações aritméticas. Consequentemente, por parecer mais rápida do que realmente é, a CPU irá receber na etapa de redistribuição de dados mais dados do que ela deveria de fato computar. Conforme os valores não nulos se propagam pela malha, a CPU irá demandar mais tempo para computar. Se o algoritmo estático decidir a divisão definitiva de cargas enquanto uma grande parte da malha contiver valores nulos, a divisão será prejudicada. Apenas o algoritmo de balanceamento de carga dinâmico é capaz de ir se ajustando ao novo estado da malha, a medida que as mudanças no tempo de computação forem ocorrendo por conta da quantidade de valores nulos ir se modificando.

Analogamente, se dois dispositivos com as mesmas capacidades computacionais realizarem cálculos em regiões diferentes da malha, uma com uma quantidade pequena de pontos nulos e outra com uma quantidade grande de pontos nulos, o tempo de computação de ambos os dispositivos irá diferir. É errado portanto supor que dispositivos homogêneos levam o mesmo tempo para realizar a mesma computação neste cenário, devendo-se levar em conta não apenas suas capacidades computacionais, mas também o conjunto de dados

nos quais as instruções são realizadas.

Outro ponto importante a ser destacado está no fato de que CPUs sofrem um impato muito maior com as otimizações aritméticas quando comparado à GPUs. Durante a simulação do SIH, o tempo de computação das CPUs cresce a uma taxa maior do que as GPUs. Podemos concluir que, para CPUs, o tempo de computação no passo de tempo i é maior do que o tempo de computação no passo de tempo $i-1$, por conta da distribuição de pontos nulos na malha. Para GPUs, os tempos de computação permanecem equilibrados, independentemente do passo de tempo considerado. Isto pode ser observado na Figura 18.

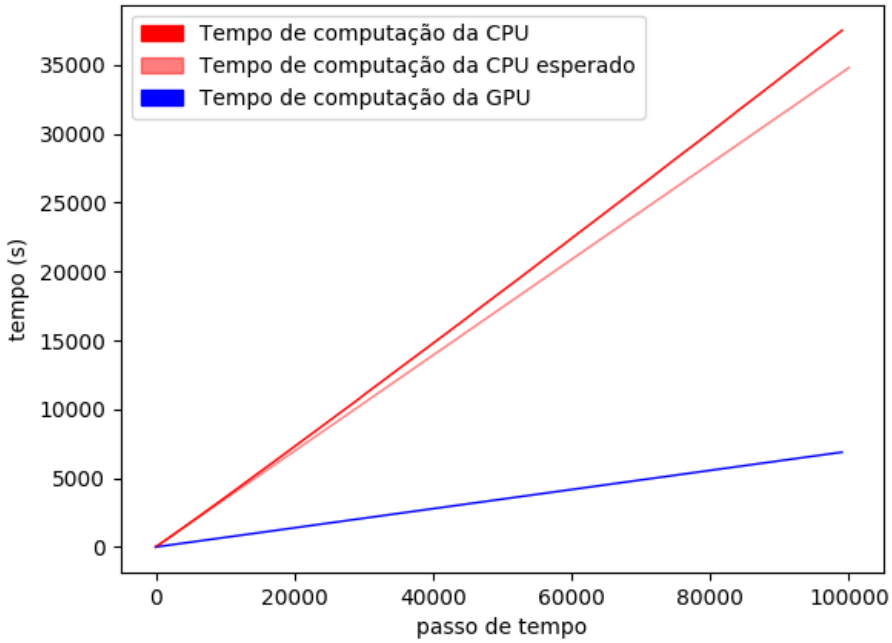

Figura 18 – Aumento do tempo de computação da CPU de cada passo de tempo em uma simulação de 100.000 passos de tempo. O tempo esperado de computaçao é medido caso o tempo de computação para cada passo de tempo fosse o mesmo durante toda a simulação, entretanto, devido ao crescimento de pontos não nulos na malha, o tempo de computação de cada passo de tempo cresce conforme a simulação progride.

Foi também realizado um experimento onde foi coletada a quantidade pontos não nulos na malha em uma simulação de $1,000,000$ de passos de tempo, em intervalos de $10,000$ passos de tempo. A Figura 19 apresenta os resultados. É possível observar que o valor inicial de pontos não nulos é pequeno, mas cresce rapidamente, até se estabilizar, finalmente, o número de pontos não nulos diminui gradativamente durante o restante da simulação. Toda essa dinâmica entre valores nulos e não nulos afeta o tempo de computação dos dispositivos.

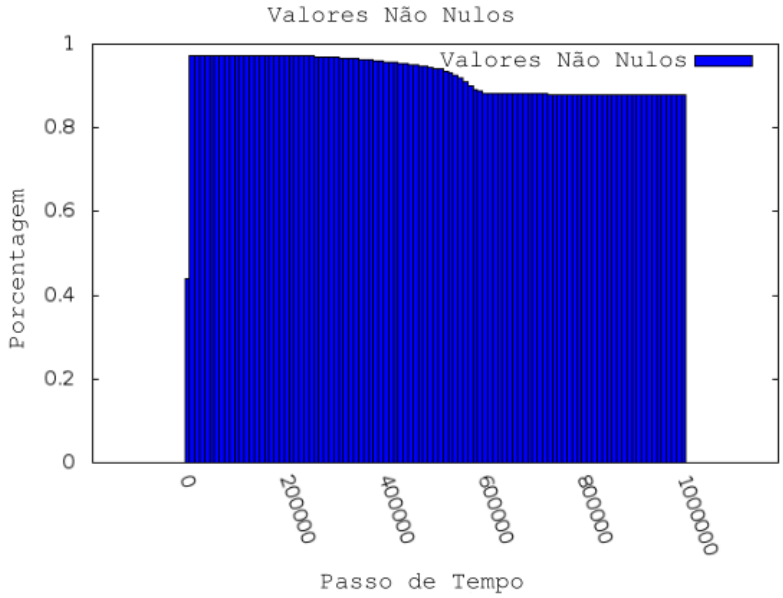

Figura 19 – Porcentagem de valores não nulos na malha. Os resultados foram coletados a cada $10,000$ passos de tempo.

5.6 Resumo do Capítulo

Esta seção apresentou uma avaliação de desempenho dos algoritmos de balanceamento de carga estático e dinâmico, usando para isso o simulador do SIH. Seis versões são usadas nos experimentos: sequencial; paralela com CPUs, sem emprego do balanceamento de carga; paralelo com GPUs, também sem balanceamento de carga; paralelo com uso simultâneo de CPUs e GPUs, também sem emprego do balanceamento de cargas; que usa todos os dispositivos do *cluster* usando um esquema de balanceamento de carga estático; e finalmente que usa todos os dispositivos do *cluster* usando um esquema de balanceamento de carga dinâmico.

Os resultados obtidos mostraram que a versão que implementa o esquema de balanceamento de carga estático possui tempo de execução maior do que a versão que executa apenas nas GPUs do *cluster*, o que indica que nesse caso a inclusão das CPUs na computação dos SIH piora o tempo de execução. Já a versão de balanceamento dinâmica é a que apresenta o menor tempo de execução.

Posteriormente, foram feitos experimentos adicionais com o objetivo de explicar melhor os resultados. Primeiro, os tempos de execução em cada dispositivo foram medidos, onde foi descoberto que os tempos da versão estática sofrem um desbalanceamento no tempo de execução, diferente da versão dinâmica, que possui os tempos de execução em todos os dispositivos aproximadamente iguais.

Como a equação de balanceamento de carga assume que o tempo de computação é diretamente proporcional à carga computada, também foi realizado um experimento

cujo objetivo foi verificar se esta relação de fato se aplica. Os resultados mostraram que o tempo de computação não cresce na mesma proporção que o aumento na carga, em especial para as CPUs. Consequentemente, as cargas calculadas pela Equação 4.6 são apenas uma aproximação, sendo necessários refinamentos para que se possa atingir os valores de carga apropriados para cada dispositivo. Esta é uma das principais limitações atuais do presente trabalho.

Como a aplicação é executada em um ambiente de memória distribuída, o tempo de comunicação envolvido no processo de rebalanceamento de cargas, realizado com maior frequência na versão dinâmica, poderia introduzir um custo adicional na execução. Um segundo experimento foi feito, sendo possível observar que o tempo em que a aplicação permaneceu ajustando as cargas era mínimo, ou seja, o número alto de ajustes da versão dinâmica não afeta o seu desempenho.

Finalmente, verificou-se que o número de pontos nulos na malha afeta o tempo de computação de alguns dispositivos, o que faz com que a aplicação tenha um comportamento irregular, mesmo tratando-se de uma aplicação que a princípio, por suas características, seria classificada como regular.

6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho propôs dois algoritmos de balanceamento de carga para emprego em um ambiente híbrido de memória distribuída (*cluster*), composto por CPUs e GPUs. A idéia principal é dividir os dados a serem computados por aplicações que implementem paralelismo de dados, para serem computados simultaneamente pelas CPUs e GPUs.

A quantidade de dados que será designada para CPUs e GPUs depende de suas respectivas capacidades computacionais, medidas em tempo de execução. No algoritmo estático, após a divisão de dados ser estabelecida, a mesma é mantida até o final da computação. Já o algoritmo de balanceamento de carga dinâmico mede o tempo de execução e atualiza os dados que cada dispositivo recebe para realizar computações durante toda execução da aplicação, em intervalos de tempo pré-definidos. O modelo computacional do Sistema Imune Humano (SIH) foi usado para avaliar ambos os algoritmos.

Os resultados mostraram que os algoritmos de balanceamento de carga foram efetivos no intuito de dividir o trabalho entre os dispositivos heterogêneos do *cluster*, resultando em ganhos de desempenho de até 714 vezes quando comparado com o algoritmo sequencial. Comparado à versão que distribuiu a carga igualmente entre todos os dispositivos, os ganhos de desempenho foram de até 3.3 vezes. Ao comparar os resultados com a versão que utiliza somente as GPUs do sistema, foi observado que apenas o esquema de balanceamento dinâmico apresenta melhoras no desempenho, executando $11s$ mais rápido (cerca de 2%).

As principais limitações deste trabalho estão relacionadas ao fato de que a equação que é usada para atribuir a carga aos dispositivos assume que o tempo de computação aumenta na mesma proporção que o aumento na carga, e que o tempo de computação de um conjunto de instruções é igual para qualquer conjunto de dados. A presença de zeros como operandos para em algumas instruções aritméticas ilustra essa situação. O algoritmo dinâmico foi capaz de se adaptar à mudanças que ocorrem durante toda a simulação quando uma quantidade grande de valores nulos inicialmente presentes na computação são substituídos por valores não nulos. Já o esquema estático de balanceamento de carga não é capaz de ajustar essas cargas durante a simulação, o que tem um impacto negativo no desempenho. Uma última limitação é o fato de que os algoritmos de balanceamento de carga foram avaliados usando uma única aplicação, de modo que algumas conclusões apresentadas nesse trabalho não podem ser generalizadas para outros contextos.

Como trabalho futuro, planeja-se usar outros *benchmarks* para avaliar os dois esquemas de balanceamento de carga, contemplando tanto aplicações regulares quanto irregulares, a fim de determinar em quais casos um esquema de balanceamento possa apresentar uma eficiência maior que o outro. Planeja-se também aumentar o número de passos de simulação do SIH para 1.000.000, já que foi mostrado que a quantidade de

posições nulas na malha se estabiliza após 100.000 passos de tempo, o que pode resultar em um tempo de computação mais estável, favorecendo o uso do esquema de balanceamento de carga estático. Outro ponto de comparação entre os esquemas estático e dinâmico está na distribuição uniforme de pontos de infecção na malha, o que resultaria em tempos de computação mais estáveis ao longo da simulação, supostamente favorecendo o esquema de balanceamento estático. Outro foco para pesquisas futuras está em utilizar dispositivos híbridos de memória verdadeiramente compartilhada, onde todos os dispositivos tem acesso ao mesmo espaço lógico de memória, descartando a necessidade de operações de leitura e cópia de dados. Este é o caso, por exemplo, da interface HUMA (*Heterogeneous Unified Memory Access*), presente nas versões mais atuais de APUs AMD. Tais dispositivos, quando localizados em uma mesma máquina, eliminariam os custos de comunicação durante a alocação das cargas, melhorando a eficiência. Outra melhoria está no uso de redes mais rápidas para enviar dados, com é o caso da rede *Infiniband*. A definição de alguns dos parâmetros computacionais do SIH é definido pelo usuário através de métodos exaustivos, a fim de se obter os valores que resultem no melhor desempenho para a aplicação. Uma possível melhoria está na utilização de métodos para definir esses parâmetros em tempo de execução de forma iterativa, o que facilitaria a utilização do balanceador de carga pelo usuário, assim como a possibilidade de ajustes às irregularidades no tempo de computação dos dispositivos que possam ocorrer durante a execução.

REFERÊNCIAS

[1] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. : Pract. Exper. 23(2), 187–198 (Feb 2011), http://dx.doi.org/10.1002/cpe.1631

[2] Bernabé. Gregorio, Cuenca. Javier, G.D.: Optimization techniques for 3d-fwt on systems with manycore gpus and multicore cpus. In: Computer Engineering Department, University of Murcia (Spain). International Conference on Computational Science, ICCS 2013 - Procedia Computer Science 18 (2013) 319 – 328 (2012)

[3] Carastan-Santos. Danilo, Y. de Camargo. Raphael, C.M.J.D.e.a.: Finding exact hitting set solutions for systems biology applications using heterogeneous gpu clusters. In: Universidade Federal do ABC, Center of Mathematics, Computing and Cognition, Santo André, Brazil. Elsevier – Future Generation Computer Systems 67 (2017) 418–429 (2017)

[4] Danilo, C.S.: Um Algoritmo Exato em clusters de GPUs para o Hitting Set Aplicado à Inferência de Redes de Regulação Gênica. Master's thesis, Universidade Federal do ABC (2015)

[5] F Borelli. Fabrizio, Y de Camargo. Raphael, C.M.J.D.e.a.: Gene regulatory networks inference using a multi-gpu exhaustive search algorithm. In: 14(Suppl 18):S5 - Las Vegas, NV, USA. 23-25 February 2012. Second IEEE International Conference on Computational Advances in Bio and Medical Sciences (ICCABS 2012) - BMC Bioinformatics 2013 (2012)

[6] Hafez, M.M., Chattot, J.J.: Innovative Methods for Numerical Solution of Partial Differential Equations. World Scientific Pub Co Inc, 1st edn. (2002)

[7] Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fifth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edn. (2011)

[8] J.I, Agulleiro, F.V.E.G.J.F.: Hybrid computing: Cpu+gpu co-processing and its application to tomographic reconstruction. In: Supercomputing and Algorithms Group, Associated Unit CSIC-UAL, University of Almeria, 04120 Almeria, Spain. Ultramicroscopy 115 109-114 - Elsevier (2012)

[9] Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1st edn. (2013)

[10] LeVeque, R.: Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics Classics in Applied Mathematics). Society for Industrial and Applied Mathematics, 1st edn. (2007)

[11] Lewis, Bil., J.B.D.: Threads primer: a guide to multithreaded programming. Prentice Hall, first edn. (1995)

[12] Lu.n, Fengshun, S.J.C.X.Z.X.: Cpu/gpu computing for long-wave radiation physics on large gpu clusters. In: Computers & Geosciences 41 (2012) 47-55 - College of

Computer, National University of Defense Technology, 410073 Changsha, Hunan, China. Elsevier (2012)

[13] Maheshwari, P.: A dynamic load balancing algorithm for a heterogeneous computing environment. In: School of Computing and Information Technology Griffith University, Brisbane, Queensland, Australia 4111. Proceedings of the 29th Annual Hawaii International Conference on System Sciences (1996)

[14] Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, 1st edn. (2004)

[15] Mittal, Sparsh. Vetter, J.S.: A survey of cpu-gpu heterogeneous computing techniques. In: Universidade Federal do ABC, Center of Mathematics, Computing and Cognition, Santo André, Brazil. ACM Comput. Surv. 47, 4, Article 69 (July 2015), 35 pages (2017)

[16] Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide. Addison-Wesley Professional, 1st edn. (2011)

[17] do Nascimento, T.M., dos Santos, R.W., Lobosco, M.: Dynamic load balancing algorithm for heterogeneous clusters. In: Parallel Processing and Applied Mathematics - PPAM 2017, Lublin, Poland, September 10-13, 2017, Proceedings. pp. 166 – 175 (2017)

[18] do Nascimento, T.M., dos Santos, R.W., Lobosco, M.: Performance evaluation of two load balancing algorithms on a hybrid parallel architecture. In: Parallel Computing Technologies - PaCT 2017, Nizhni Novgorod, Russia, September 4-8, 2017, Proceedings. p. to appear (2017)

[19] Pacheco, P.: An Introduction to Parallel Programming. Morgan Kaufmann, 1st edn. (2011)

[20] Panetta, Jairo, T.T.R.P.d.S.F.P.: Accelerating kirchhoff migration by cpu and gpu cooperation. In: Tecnologia Geofísica – Petróleo Brasileiro SA, PETROBRAS – Rio de Janeiro, Brazil. 21st International Symposium on Computer Architecture and High Performance Computing (2009)

[21] Peters Xavier, M.: Implementação Paralela em um Ambiente de Múltiplas GPUs de um Modelo 3D do Sistema Imune Inato. Master's thesis, Universidade Federal de Juiz de Fora (2013)

[22] Pigozzo, A.B., Macedo, G.C., Santos, R.W., Lobosco: On the computational modeling of the innate immune system. BMC Bioinformatics - 14 Suppl 6 - S7 (2013)

[23] Pigozzo, A.B.: Implementação Computacional de um Modelo Matemático do Sistema Imune Inato. Master's thesis, Universidade Federal de Juiz de Fora (2011)

[24] Rocha, P., Xavier, M., Pigozzo, A., de M. Quintela, B., Macedo, G., dos Santos, R., Lobosco, M.: A three-dimensional computational model of the innate immune system. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A., Taniar, D., Apduhan, B. (eds.) Computational Science and Its Applications - ICCSA 2012, Lecture Notes in Computer Science, vol. 7333, pp. 691–706. Springer Berlin Heidelberg (2012)

[25] Sanders, J.: Cuda by Example: An Introduction to General-Purpose GPU. Addison-Wesley Professional, 1st edn. (2010)

[26] Sant'Ana, L., Cordeiro, D., Camargo, R.: Plb-hec: A profile-based load-balancing algorithm for heterogeneous cpu-gpu clusters. In: 2015 IEEE International Conference on Cluster Computing. pp. 96–105 (Sep 2015)

[27] Walter, A., S.: Partial Differential Equations: An Introduction. Wiley, second edn. (2007)

APÊNDICE A – PARÂMETROS DA SIMULAÇÃO – CONDIÇÕES INICIAIS

Condição inicial	Valor	Unidade
$LPS(x, y, z, 0)$	10^6 se $z \geq 0,8 \times Z$; 0 caso contrário	$\frac{partícula}{mm^3}$
$MR(x, y, z, 0)$	10^4	$\frac{partícula}{mm^3}$
$MA(x, y, z, 0)$	0	$\frac{partícula}{mm^3}$
$N(x, y, z, 0)$	0	$\frac{partícula}{mm^3}$
$ND(x, y, z, 0)$	0	$\frac{partícula}{mm^3}$
$CH(x, y, z, 0)$	0	$\frac{partícula}{mm^3}$
$G(x, y, z, 0)$	0	$\frac{partícula}{mm^3}$
$CA(x, y, z, 0)$	0	$\frac{partícula}{mm^3}$

Tabela 12 – Condições iniciais do SIH.

Parâmetro	Valor	Unidade
Δt	$0,000001$	dias
Δx	$0,1$	mm
Δy	$0,1$	mm
Δz	$0,1$	mm
μ_{LPS}	0	$dias^{-1}$
λ_{LPS_N}	$0,55$	$\frac{mm^3}{dias \times partícula}$
λ_{LPS_MA}	$0,8$	$\frac{mm^3}{dias \times partícula}$
σ_{LPS_MR}	$0,1$	$\frac{mm^3}{dias \times partícula}$
γ_{CA}	1	$\frac{mm^3}{dias \times partícula}$
d_{LPS}	2	$\frac{mm^2}{dias}$
μ_{MR}	$0,033$	$dias^{-1}$
σ_{LPS_MR}	$0,1$	$\frac{mm^3}{dias \times partícula}$
γ_{CA}	1	$\frac{mm^3}{partícula}$
M^{max}	6	$\frac{partícula}{mm^3}$
$P^{max}_{MR\_CH}$	$0,1$	$dias^{-1}$
$P^{min}_{MR\_CH}$	$0,01$	$dias^{-1}$
η_{MR_CH}	1	$\frac{partícula}{mm^3}$
$P^{max}_{MR\_G}$	$0,5$	$dias^{-1}$
$P^{min}_{MR\_G}$	0	$dias^{-1}$
η_{MR_G}	1	$\frac{partícula}{mm^3}$
d_{MR}	$4,320$	$\frac{mm^2}{dias}$
q_{CH_MR}	$3,6$	$\frac{mm^2}{dias}$
μ_{MA}	$0,07$	$dias^{-1}$
σ_{LPS_MR}	$0,1$	$\frac{mm^3}{dias \times partícula}$
γ_{CA}	1	$\frac{mm^3}{partícula}$
d_{MA}	3	$\frac{mm^2}{dias}$
q_{CH_MA}	$4,32$	$\frac{mm^2}{dias}$
μ_N	$3,43$	$dias^{-1}$
N^{max}	8	$\frac{partícula}{mm^3}$
$P^{max}_{N\_CH}$	$11,4$	$dias^{-1}$
$P^{min}_{N\_CH}$	$0,0001$	$dias^{-1}$
η_{N_CH}	1	$\frac{patícula}{mm^3}$
d_N	$12,096$	$\frac{mm^2}{dias}$
q_{CH_N}	$14,4$	$\frac{mm^2}{dias}$

Tabela 13 – Parâmetros do SIH.

Parâmetro	Valor	Unidade
μ_{CH}	7	$dias^{-1}$
β_{LPS_N}	1	$\frac{mm^3}{dias \times partícula}$
ω_{CH}	3,6	$\frac{partícula}{mm^3}$
κ_{CA}	1	$\frac{mm^3}{partícula}$
β_{LPS_MA}	0,8	$\frac{mm^3}{dias \times partícula}$
d_{CH}	9,216	$\frac{mm^2}{dias}$
μ_N	3,43	$dias^{-1}$
λ_{ND_MA}	2,6	$\frac{mm^3}{dias \times partícula}$
d_{ND}	0,144	$\frac{mm^2}{dias}$
μ_G	5	$dias^{-1}$
α_{N_G}	0,6	$adimensional$
N^{max}	8	$\frac{partícula}{mm^3}$
$P^{max}_{N\_CH}$	11,4	$dias^{-1}$
$P^{min}_{N\_CH}$	0,0001	$dias^{-1}$
d_G	9,216	$\frac{mm^2}{dias}$
μ_{CA}	4	$dias^{-1}$
β_{MR_ND}	1,5	$\frac{mm^3}{dias \times partícula}$
ω_{CA}	3,6	$\frac{partícula}{mm^3}$
β_{MA}	1,5	$adimensional$
d_{CA}	9,216	$\frac{mm^2}{dias}$

Tabela 14 – Parâmetros do SIH.