Universidade do Estado da Bahia – UNEB

Departamento de Ciências Exatas e da Terra

Colegiado de Sistemas de Informação

Thomas Magnum de S. Ferreira

Técnicas Heurísticas Aplicadas na Otimização de um Modelo Híbrido de Computação Paralela

Salvador

2013

Thomas Magnum de S. Ferreira

Técnicas Heurísticas Aplicadas na Otimização de um Modelo Híbrido de Computação Paralela

Monografia apresentada à Banca Examinadora como exigência parcial para obtenção do título de bacharel em Sistemas de Informação pela Universidade do Estado da Bahia.

Orientador: Murilo do Carmo Boratto

Salvador

2013

Thomas Magnum de S. Ferreira

Técnicas Heurísticas Aplicadas na Otimização de um Modelo Híbrido de Computação Paralela/ Thomas Magnum de S. Ferreira. – Salvador, 2013-

57 p. : il. (algumas color.) ; 30 cm.

Orientador: Murilo do Carmo Boratto

Monografia (Graduação) – Universidade do Estado da Bahia – UNEB
Departamento de Ciências Exatas e da Terra
Colegiado de Sistemas de Informação, 2013.

1. Heurísticas. 2. Auto-otimização. 3. Computação Paralela. I. Murilo Boratto. II. Universidade do Estado da Bahia. III. Departamento de Ciências Exatas e da Terra. IV. Heurísticas na Otimização de um Modelo Híbrido de Computação Paralela

Thomas Magnum de S. Ferreira

Técnicas Heurísticas Aplicadas na Otimização de um Modelo Híbrido de Computação Paralela

Monografia apresentada à Banca Examinadora como exigência parcial para obtenção do título de bacharel em Sistemas de Informação pela Universidade do Estado da Bahia.

Trabalho aprovado. Salvador, 12 de dezembro de 2013:

Murilo do Carmo Boratto
Doutorando em Informática
Universidade do Estado da Bahia

Tricia Souto Santos
Mestre em Ciência da Computação
Universidade do Estado da Bahia

Marcos Ennes Barreto
Doutor em Ciência da Computação
Universidade Federal da Bahia

Salvador
2013

Agradecimentos

Em primeiro lugar a Deus por sempre ser o meu escudo e amparo, por lutar as minhas batalhas. Mesmo em momentos em que fui pressionado há violar princípios bíblicos, especialmente a guarda do sábado, operou impossíveis para que hoje alcançasse mais uma vitória e pudesse testemunhar que vale a pena ser fiel. Ao SENHOR dedico este trabalho e minha vida.

- A Maiana, minha esposa, pela compreensão em ceder o tempo de momentos que deveria estar ao seu lado para estudar e conseguir vencer mais esta etapa. Especialmente agradecer pelo carinho, paciência, motivação e por ser sempre a minha companheira.
- A minha família querida, meus pais (Joselias e Solange) e minha irmã Magda. O apoiou de vocês foi fundamental, obrigado pela educação e valores ensinados, como também por sempre acreditarem e confiarem em mim.
- Ao meu orientador, o professor Murilo Boratto, pelos incentivos, cobranças, correções, acompanhamentos e pela técnica de plantar bananeira. Esta técnica consiste em fazer o que o orientador mandar e sem questionar, é algo difícil de o aluno fazer, mas realmente funciona. Muito obrigado professor, este trabalho não seria realizado sem a sua ajuda.
- A equipe da resistência: Felippe Zacarias, Leonardo Campos, Kal Lenon, Daniel Amaral, Marcelo Tonete, Raylan e Rafael Guimarães. Meus companheiros de curso que continuaram mesmo em meio a desistência de muitos. Continuem lutando jovens, não desistam, pois assim como eu e Daniel vencemos, vocês também conseguirão.
- Ao professor Julian Quezada que teve uma participação importante na finalização desta pesquisa, ao emprestar sua literatura e sempre estar disponível para tirar dúvidas.
- A todos os meus professores, não detalharei pois são muitos os seus ensinamentos e contribuições. Podem ter certeza que cada um dos senhores contribuíram bastante para minha formação acadêmica, profissional e principalmente pessoal.

"Mas, buscai primeiro o reino de Deus,
e a sua justiça, e todas estas coisas
vos serão acrescentadas.
(Bíblia Sagrada, Mateus 6:33)

Resumo

Na computação de alto desempenho existem sistemas paralelos que utilizam uma abordagem híbrida, associando CPUs com múltiplos núcleos a unidades de processamento gráfico (GPU) para processarem informações simultaneamente. Mas para explorar o máximo desta plataforma é preciso adaptar a aplicação paralela ao ambiente utilizado. As ferramentas de auto-otimização são um meio automatizado de adaptar um determinado *software* a uma arquitetura. Antes de realizar essa adaptação, normalmente é realizada uma busca empírica por valores ótimos para parâmetros específicos da aplicação, a fim de ajustá-la às características do hardware. A busca no espaço de otimização pode ser feita através de uma busca exaustiva, a qual se aplica todas as possibilidades de otimização dentro do espaço de busca. Nesta abordagem, dependendo da quantidade de combinações, o espaço de busca pode tornar a adaptação inviável em relação ao tempo de execução do algoritmo. Uma forma de diminuir o espaço de busca e viabilizar as adaptações é realizar buscas baseadas em heurísticas. Nesta pesquisa serão apresentadas funções heurísticas que reduzem o espaço de busca e apresentam um custo computacional menor do que o método tradicional de busca exaustiva.

Palavras-chaves: Auto-otimização, computação paralela, heurísticas, multicore, GPU.

Abstract

Some high performance systems use a parallel hybrid approach, based on multicore CPUs and graphics processing units (GPU) to process information simultaneously. To explore the most of this platform is necessary to adapt the application to the parallel environment used. Auto-tuning is an automated way to adapt a particular software to an architecture, this is usually done through an empirical search for optimal values for parameters specific to an application in order to adjust them to the characteristics of hardware. The search space optimization can be performed through an exhaustive search, which applies to all the possibilities of optimization within the search space. In this approach, depending on the number of combinations, the search space can make adjustment impossible in terms of time. One way to reduce the search space and enable the adaptations is to perform searches based on heuristics. This work is presented heuristic functions that reduce the search space and have a lower computational cost than the traditional method of exhaustive search.

Key-words: Auto-tuning, parallel computing, heuristics, multicore, GPU.

Lista de ilustrações

Lista de tabelas

Lista de algoritmos

Lista de abreviaturas e siglas

API	Application Programming Interface
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
IBM	International Business Machines
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMX	MultiMedia eXtensions
MSIMD	Multiple Single Instruction Multiple Data
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
OpenGL	Open Graphics Library
RAM	Random Access Memory
SGI	Silicon Graphics International
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SMX	Streaming Multiprocessor
SPT	Sonda de Produção Terrestre
SSE	Streaming SIMD Extensions

Lista de símbolos

α	Letra grega Alfa
γ	Letra grega Gama
ε	Letra grega Épsilon
φ	Letra grega Fi
$\supset$	Símbolo de superconjunto

Sumário

1 Introdução

Com advento de novas e crescentes demandas computacionais que necessitam de máquinas com um elevado poder de processamento, amplifica-se o foco da computação de alto desempenho em sistemas distribuídos e processamento paralelo para estudo e desenvolvimento de soluções cada vez mais eficientes. Segundo Quaresma et al. (2010), a rápida evolução da tecnologia demonstra que o futuro da computação de alto desempenho tende para uma abordagem híbrida, integrando dois tipos de componentes: CPU *multicore*(múltiplos núcleos) e unidades de processamento gráfico (GPU).

A GPU tem atraído a atenção da comunidade científica devido ao seu baixo custo e seu alto poder computacional. A sua capacidade para executar centenas de *threads*, simultaneamente as torna bastante úteis para a execução de operações paralelas. Mas conseguir desempenho desses sistemas requer que as aplicações explorem diversos níveis de paralelismo simultaneamente. Portanto, a programação para estas novas arquiteturas requer novas formas de se pensar sobre o desenvolvimento de aplicações paralelas e de técnicas de otimização de desempenho.

Com o advento das arquiteturas *multicore* e multi-GPU, o estabelecimento de ferramentas e métodos de programação paralela no desenvolvimento de aplicativos é absolutamente necessário para alcançar uma boa atuação e apoiar os desenvolvedores em várias tarefas envolvidas neste processo (BORATTO et al., 2012). Dentro desse contexto, ferramentas como OpenMP (OPENMP, 2013) e CUDA (CUDA, 2013) permitem a utilização da computação de alto desempenho a um maior número de usuários, ou seja, auxiliando os iniciantes na construção de aplicativos com características como escalabilidade e produtividade.

Para desenvolver aplicações paralelas que explorem o máximo dos recursos do *hardware* disponível, o *software* deve explorar detalhes específicos do ambiente em questão e para isso é necessário um conhecimento técnico do *hardware* utilizado e uma ampla experiência em computação paralela, o que se torna um problema para usuários inexperientes. Para resolver este problema, como também prover otimizações para uma plataforma, mesmo que ela evolua ou se altere, surgiram as ferramentas de auto-otimização que são um meio automatizado de adaptar um determinado *software* a uma arquitetura (ALMEIDA, 2011).

Em ferramentas de auto-otimização, as adaptações podem ocorrer através de uma busca empírica de valores ótimos para parâmetros específicos de uma aplicação, a fim de ajustá-los às características do hardware. A busca no espaço de otimização pode ser feita através de uma forma exaustiva, a qual se aplicam todas as possibilidades de otimização

dentro do espaço de busca. Nesta abordagem, dependendo da explosão combinatória, o espaço de busca pode tornar a adaptação inviável em termos de tempo. Uma forma de diminuir o espaço de busca e viabilizar as adaptações é realizar buscas baseadas em heurísticas.

1.1 Estado da Arte

Para contextualizar o que será abordado nesta monografia é necessária uma revisão dos trabalhos disponíveis na literatura. O principal objetivo é avaliar os principais aspectos que motivam e impulsionam a criação e desenvolvimento de aplicações relacionadas com os problemas estudados. Desta forma, pode-se classificar em três grupos os problemas tratados neste trabalho.

O primeiro grupo de trabalhos relacionados estudados se refere ao uso de técnicas heurísticas em otimizações de desempenho. Em (GOMES, 2005) são utilizadas técnicas heurísticas num método de otimização continua para desenvolver uma metodologia de reconfiguração de sistemas de distribuição de energia elétrica que diminua o custo de utilização de linhas e evite perdas. A técnica heurística é utilizada para identificar as chaves dos alimentadores que teriam maior tendência a serem abertas. Este processo, caso não fosse feito com heurística, utilizaria técnicas combinacionais que demandam um alto custo computacional.

Outro trabalho estudado foi (ALOISE et al., 2002) que utiliza heurísticas baseadas no modelo de otimização por Colônia de Formigas para produzir boas soluções, em um tempo computacional aceitável. As heurísticas otimizam o itinerário de Sondas de Produção Terrestre, que são utilizadas para realizar serviços de intervenção em poços de petróleo. Em ambos os trabalhos (GOMES, 2005; ALOISE et al., 2002) ficou evidente a eficiência computacional adquirida através do uso de técnicas heurísticas na obtenção de parâmetros para os sistemas.

No segundo grupo de trabalhos relacionados foram pesquisados problemas científicos que utilizam modelos paralelos da computação de alto desempenho (BORATTO; COELHO; BARRETO, 2012; FILHO, 2012; LIMA, 2013; LIPORACE, 2013). Os projetos nesta área têm uma grande quantidade de informação a processar e requerem uma grande capacidade de processamento. A metodologia utilizada nestes estudos utilizam multiprocessadores e GPUs para atenderem à demanda de processamento existente em problemas complexos e reduzir o tempo de resposta da aplicação para um período satisfatório.

O terceiro grupo de trabalho está relacionado com a utilização e desenvolvimento de técnicas de auto-otimização para sistemas paralelos (ALMEIDA, 2011; SPENCER, 2011; RUDY, 2010). Estas técnicas permitem o desenvolvimento de rotinas que identificam parâmetros do ambiente paralelo utilizado e adaptam o algoritmo para que possa ser

executado de uma maneira mais eficiente. Desta forma proporciona aos usuários que suas aplicações paralelas tenham um desempenho próximo do ótimo, no que diz respeito ao tempo de execução, independente do conhecimento de computação paralela que o usuário contenha e das características do ambiente utilizado.

1.2 Objetivo

1.2.1 Objetivo Geral

Desenvolver técnicas de otimização que analisem o sistema utilizado e através de testes e funções heurísticas encontrem os melhores parâmetros para executar uma aplicação paralela híbrida.

1.2.2 Objetivos Específicos Divididos em Tarefas

TAREFA 1 Revisão de bibliografia e investigação de trabalhos relacionados.

TAREFA 2 Implementar um método de busca exaustiva que encontre os melhores parâmetros para o modelo paralelo híbrido.

TAREFA 3 Elaborar funções heurísticas que otimizem a busca por parâmetros ótimos.

TAREFA 4 Trabalhar com um estudo de caso específico chamado Espacialização do Relevo.

TAREFA 4 Comparar o método de busca exaustiva com as funções heurísticas implementadas.

TAREFA 5 Executar o estudo de caso com a otimização proposta em um sistema computacional de alto desempenho para avaliar os resultados.

1.3 Justificativa

Segundo Almeida (2011), o desenvolvimento de aplicações de forma a atingir níveis de desempenho próximos aos níveis teóricos de uma determinada plataforma é uma tarefa que exige conhecimento técnico do ambiente de hardware, uma vez que o software deve explorar detalhes específicos da plataforma em questão. Pelo fato do software ser específico à plataforma, caso ela evolua ou se altere, as otimizações realizadas podem não explorar a nova arquitetura de forma eficiente e será necessário fazer adaptações.

Em sistemas de auto-otimização as adaptações podem ocorrer através de uma busca empírica de valores ótimos para parâmetros específicos de uma aplicação, a fim de

ajustá-los às características do hardware. A busca no espaço de otimização pode ser feita através de uma busca exaustiva, a qual se aplica todas as possibilidades de otimização dentro do espaço de busca. Nesta abordagem, dependendo da explosão combinatória, o espaço de busca pode tornar a adaptação inviável em termos de tempo. Uma forma de diminuir o espaço de busca e viabilizar as adaptações é realizar buscas baseadas em heurísticas.

1.4 Metodologia

A metodologia utilizada foi a seguinte:

- Estudo de métodos matemáticos de otimização não-linear.
- Estudo do código paralelo híbrido de espacialização do relevo com o objetivo de identificar quais os parâmetros do algoritmo têm maior influencia no tempo de execução em ambientes paralelos distintos.
- Elaborar um método de busca exaustiva que realize testes na aplicação paralela híbrida variando os valores dos parâmetros identificados anteriormente.
- Implementar funções heurísticas baseadas em métodos matemáticos de otimização que encontrem, com um custo computacional menor que o método de busca exaustiva, os valores dos parâmetros da aplicação que permitem que ela seja executada em um tempo próximo do ótimo.
- Comparar o ganho no custo computacional ao utilizar heurísticas em substituição ao método tradicional de busca exaustiva.
- Realizar experimentos em um sistema computacional de alto desempenho para validar o método proposto.

1.5 Estrutura do Trabalho:

Este trabalho está estruturado em 5 capítulos. O primeiro introduz ao leitor o problema a ser abordado.

No **capítulo 2** será abordado alguns modelos da Computação de Alto Desempenho e serão descritas as ferramentas básicas usadas neste trabalho.

No **capítulo 3** são apresentadas as heurísticas utilizadas para otimizar a busca por valores de parâmetros para a aplicação paralela híbrida.

No **capítulo 4** são relatados os resultados dos experimentos realizados.

Esta monografia é finalizada com conclusões e propostas de trabalhos futuros.

2 Descrição Teórica e Ferramentas Básicas

Serão descritos neste capítulo alguns modelos de computação de alto desempenho, como também as ferramentas matemáticas e computacionais, tanto de software com hardware, utilizadas neste trabalho.

2.1 Computação de Alto Desempenho

A contínua busca por poder computacional tem sido o principal impulsionador do desenvolvimento dos recursos de *hardware*. As atividades científicas e de engenharia requisitam cada vez mais arquiteturas de software e hardware que reduzam o tempo necessário para a solução de seus problemas que demandam altas capacidades de recursos computacionais.

Devido a essa demanda, é necessário superar as limitações físicas decorrentes da frequência dos processadores que produzem calor a níveis elevados, as limitações da miniaturização dos transistores que chegaram a um patamar que dificulta aparições de processadores seriais mais rápidos (BORATTO; COELHO; BARRETO, 2012), como também a velocidade de transmissão admitida entre a memória e CPU cujo gargalo é o barramento, mostrando que mesmo com um processador rápido este terá que esperar por alguns ciclos de *clock* pelos dados que são acessados na memória para posterior processamento.

A computação de alto desempenho se apresenta como uma alternativa para solucionar essas necessidades por recursos computacionais em que confiabilidade e velocidade de transações são fatores importantes para se manter no mercado. Elas são frequentemente utilizadas para a resolução de simulações complexas da ciência e da engenharia (MORAES, 2012), e também para a resolução de problemas com expressiva relevância científica e econômica, como por exemplo, computação da dinâmica dos fluidos, previsões climáticas, visão computacional, reconhecimento de voz e nanotecnologia.

Uma maneira para executar aplicações de alto desempenho é usar *cluster computing* (LV; CHEN, 2012). Um *cluster* é um tipo de sistema paralelo distribuído formado por uma coleção de computadores autônomos que cooperam para a realização de uma determinada tarefa como um único sistema. Um *middleware* é responsável por unificar essa coleção de computadores e oferecer ao usuário a visão de um sistema unitário podendo executar tanto aplicações sequenciais como aplicações em paralelo.

Na computação convencional, para resolver determinado algoritmo, um único processador processa um fluxo sequencial de instruções e dados previamente alocados em memória. Apenas uma instrução é executada ou um dado é manipulado por ciclo de

clock. Com o objetivo de diminuir o tempo total de processamento de um algoritmo e de obter um desempenho melhor em relação a uma versão sequencial, os algoritmos paralelos consistem em executar simultaneamente partes de uma aplicação em um mesmo processador, em uma máquina multiprocessada ou em um cluster.

Uma das diversas formas de classificar uma arquitetura paralela é utilizando a Taxonomia de Flynn. Sua classificação se baseia nos conceitos de fluxo de instruções e fluxos de dados, que são independentes entre si, gerando assim quatro categorias de máquinas (FLYNN, 1972):

Os *SISD* (*Single Instruction, Single Data*) são computadores com um único processador. Apenas uma instrução e um fluxo de dados são processados a cada momento. Ele é serial, sendo cada instrução executada uma após a outra. Nela estão contidas as máquinas que seguem o modelo tradicional de Von Neumman (KOWALTOWSKI, 1996).

O *SIMD* (*Single Instruction, Multiple Data*) é uma arquitetura desenhada para problemas específicos com alto padrão de regularidade de dados. Todas as unidades devem receber a mesma instrução no mesmo instante de tempo de modo que possam executá-las de forma simultânea. Cada unidade de processamento pode operar sobre um fluxo de dados diferente. Enquadram-se nessa categoria computadores com processamento vetorial (PITANGA, 2008). Atualmente aplicações que executam nesta arquitetura podem ser bem portadas para GPUs.

MISD (*Multiple Instruction, Single Data*): Arquitetura que assume um múltiplo fluxo de instruções sobre um único fluxo de dados, ou seja, várias instruções manipulariam um mesmo dado. Este tipo de classificação não possui representante e até mesmo *Flynn* duvidou que algum dia pudesse existir.

MIMD (*Multiple Instruction, Multiple Data*): É uma arquitetura muito usada atualmente. Cada unidade de processamento executa instruções diferentes a cada momento, e pode operar sobre um fluxo de dados diferentes. Pode ser multiprocessadores ou multicomputadores que utilizam de memória compartilhada ou distribuída. Deste grupo os clusters vêm tendo uma maior participação no cenário de máquinas paralelas.

O programa paralelo pode ser basicamente de dois tipos: de dados ou funcionais. No paralelismo de dados, os dados são particionados em subconjuntos e associados a um processo ou *thread* que executam os mesmos comandos sobre seu subconjunto de dados. Depois de processados, esses dados são combinados novamente para ser obtido o conjunto resposta (MARQUES, 2012). Essa técnica permite uma maior exploração do paralelismo em processadores *multicore*, utilizando toda a potencialidade dos *cores* que não seriam aproveitados pela aplicação tradicional. Programas deste tipo são facilmente escaláveis, pois podem ser utilizados para a resolução de problemas cada vez maiores apenas adicionando mais processos e processadores.

O paralelismo funcional divide o problema em tarefas independentes que são associadas a um processo ou *thread* e que podem operar sobre o mesmo conjunto de dados ou não (AMARAL, 2005). Algumas questões devem ser levadas em consideração nesse tipo de paralelismo, como por exemplo, a dependência entre as operações, as dificuldades de escalabilidade em relação ao tamanho do problema e a grande necessidade de comunicação e espera das funções mais lentas.

Muitos problemas podem ser resolvidos usando qualquer um dos tipos, mas o paralelismo de dados é o mais usado e o mais fácil de ser implementado. Além do tipo de paralelismo que é usado para escrever códigos paralelos, outro importante fator é o modelo de programação a ser usado. O modelo de programação de uma aplicação em paralelo define a forma de troca de dados e de sincronização dos mesmos na execução, sendo os mais usados: o Divide e Conquista (*Divide and Conquer*) e o Mestre/Escravo (*Master/Slave*).

No modelo Divide e Conquista (BORATTO, 2007), a partir de um conjunto de dados de entrada, são feitas subdivisões recursivamente de acordo com as dependências das tarefas. Como as subdivisões são independentes, não é necessária a comunicação entre os processos, apenas a comunicação entre o processo criador das subdivisões e os processos que irão realizar o trabalho. Atribui-se um ou vários segmentos consecutivos a cada processador, com as divisões escolhidas de tal modo que haja um equilíbrio da carga de trabalho. Uma vez obtidas as soluções parciais, elas são combinadas para resultar na solução do problema original.

O modelo Mestre/Escravo (GRBOVIC, 2009) é um paradigma que consegue elevado desempenho e uma boa escalabilidade, pois a comunicação só existe entre o computador denominado mestre e os outros chamados escravos. O mestre é o responsável por inicializar a execução da aplicação, distribuir os trabalhos para os processos escravos e reunir o resultado. Quando o número de escravos é grande este controle centralizado torna-se um problema, neste caso a solução é aumentar o numero de mestres onde cada um gerenciaria um grupo de escravos. Nesta abordagem o balanceamento de carga pode ser estático, se a divisão das tarefas for feita no início da computação, permitindo a participação do mestre na computação; Dinâmico, se o número de tarefas é maior que o número de escravos ou o tempo de execução das tarefas é desconhecido no início da computação.

2.1.1 Modelo de Comunicação

No sistema de comunicação de um ambiente paralelo os principais modelos são o de comunicação por memória compartilhada e memória distribuída. O modelo de memória compartilhada é um ambiente onde vários processadores compartilham o mesmo espaço de memória. Os processadores podem operar independentemente, mas compartilham recursos como a memória e barramento. Um lugar na memória não pode ser modificado por

uma tarefa, enquanto outra estiver acessando esse dado. Sua comunicação entre tarefas é rápida.

Neste modelo a interface de programação mais usada é a *OpenMP* (OPENMP, 2013), um padrão que define como os compiladores devem gerar códigos paralelos através de diretivas e funções. É composto por três componentes básicos: diretivas de compilação, variáveis de ambiente e biblioteca de função. Seu paralelismo é explícito, pois cabe ao programador identificar as tarefas para a execução em paralelo e definir os pontos de sincronização utilizando as diretivas de programação no código. Cada processo é visto como um conjunto de *threads* que se comunicam através de variáveis compartilhadas (memória compartilhada). A criação, inicialização e finalização das *threads* é feita pelo ambiente de execução.

No modelo de memória distribuída vários processadores têm seu próprio bloco de memória e estão conectados por uma rede. As memórias são independentes, neste caso as tarefas compartilham dados através do envio e recebimento de mensagens. O padrão para essa comunicação é o MPI (MPI, 2013) que provê uma base poderosa para construir programas paralelos. Apesar de acessar sua memória sem interferência e com rapidez, nesse modelo existe um *overhead* que é o de comunicação entre os processos com o envio e recebimento de dados.

O MPI assume que todos os processos são estáticos, sendo que nenhum novo processo é criado em tempo de execução. Cada processo é identificado através de um *Rank* que varia de 0 a $P-1$, onde P é o número de processadores. Neste paradigma todos os processadores executam o mesmo programa, porém executam instruções diferentes através dos desvios condicionais.

Por ser um padrão de comunicação, o MPI possibilita a utilização da passagem de mensagem em ambientes heterogêneos, visto que provê a padronização de tipos, comunicação por grupos, sincronização, controle de erros, etc. A instalação e a configuração adequada do sistema operacional são fundamentais para que os programas que utilizam o MPI sejam executados de maneira desejada.

2.2 Ferramentas de Hardware

Nesta seção será apresentado o cluster utilizado nas experimentações e uma conceituação sobre um dos seus componentes principais que é a GPU.

2.2.1 GPU

As unidades de processamento gráfico (GPU) são encontradas em muitos dispositivos como estações de trabalho, celulares e *videogames*. A GPU é um processador especializado em processamento de imagens e de gráficos 3D (BORATTO; COELHO; BARRETO, 2012). A grande demanda pela qualidade de imagens e processamento em tempo real fez com que as GPUs evoluíssem para um equipamento com alto poder computacional cuja capacidade de processamento, em certas situações, chega a ser maior que a da CPU.

A Figura 1 exibe uma comparação entre a capacidade de processamento das GPUs da NVIDIA e dos processadores da Intel, no que diz respeito às operações de ponto flutuante. Pode-se observar que, a partir de junho de 2004, as GPUs da NVIDIA apresentam um poder computacional maior que as CPUs da Intel, fato este que se acentua com o passar dos anos.

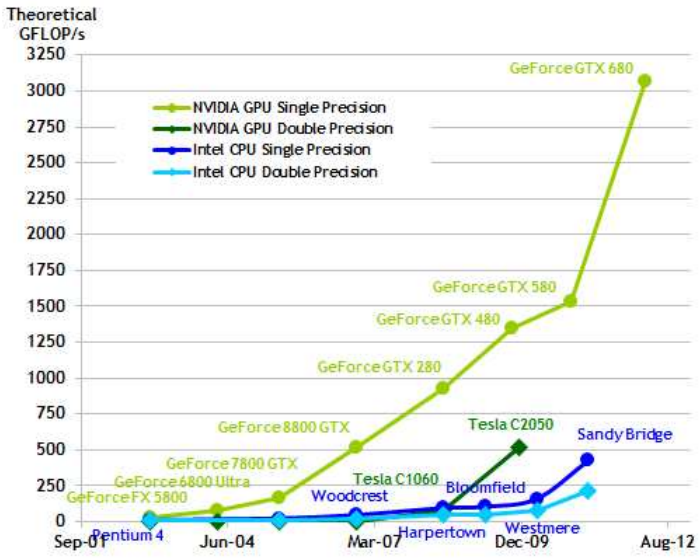


Figura 1 – Evolução dos GFLOPS das GPUs e CPUs.

Fonte: (SOUZA, 2011)

Um fato que explica essa diferença no poder de processamento é que a CPU foi desenvolvida para executar uma *thread* de instruções sequenciais em sua velocidade máxima. Nas atuais arquiteturas *multicore* as CPUs podem rodar até duas *threads* por núcleo, entretanto a GPU pode executar paralelamente milhares de *threads*. Conforme pode ser visto na Figura 2, as GPUs possuem mais transistores dedicados aos cálculos e poucos

para *caching* de informação ou controle de fluxos altamente complexos, diferente da CPU que reserva boa parte de seus transistores para controle de fluxo e cache de dados (VASCONCELLOS, 2009).

Figura 2 – Disposição dos transistores entre CPU e GPU.

Fonte: (SOUZA, 2011)

Para descrever suas GPUs a NVIDIA introduziu um termo relativamente novo, o *SIMT* (VASCONCELLOS, 2009), porém outros autores preferem se referir as GPUs como arquiteturas *MSIMD*, isto é múltiplas unidades *SIMD* (PEREIRA, 2012). Como uma GPU possui vários processadores que executam instruções independentes em paralelo, cada processador é visto como uma unidade *SIMD*, que se difere das extensões vetoriais *MMX*, *SSE* e *SSE2* encontradas nos processadores de propósito gerais, da intel, por exemplo, também são chamadas por muitos autores de *SIMD*, por permitirem aos processadores a realização de operações sobre diferentes dados. A diferença é que as GPUs podem desativar *threads* que pertencem a um processador, modelando desta forma os desvios condicionais, o que não ocorre nas extensões vetoriais dos processadores.

A demanda por alta definição tem impulsionado o desenvolvimento de arquiteturas de GPU cada vez mais paralelizáveis. Em 2009 a NVIDIA lançou a arquitetura *Fermi* com 3 bilhões de transistores e 512 núcleos CUDA, tornando o coprocessamento entre CPU e GPU mais difundido e atendendo todo o espectro de aplicativos da computação (NVIDIA, 2009).

A mais recente arquitetura *Kepler* (Figura 3), produzida em 28 nanômetros, aumenta o desempenho da GPU e contém 7,1 bilhões de transistores. Além de ser mais econômica do que a versão anterior, possui 15 multiprocessadores denominados *SMX*, 6 controladores de memória de 64 bits, um reforçado sistema de memória com capacidade de cache adicional e maior largura de banda em cada nível da hierarquia de memória (NVIDIA, 2012).

Um *SMX* da Arquitetura Kepler contém 192 núcleos CUDA com precisão simples, e cada núcleo possui unidade lógica e aritmética com *pipeline* completo em ponto flutuante e inteiro. Como para muitas aplicações de alto desempenho a característica

Figura 3 – Arquitetura Kepler.

Fonte: (NVIDIA, 2012)

de precisão dupla é muito importante, esta arquitetura possui 32 unidades de leitura e escrita (*LD/ST*), 64 unidades de precisão dupla (*DP unit*) e 32 unidades de funções especiais (*SFU*) que são responsáveis por executar uma rápida aproximação de funções como inversa, raiz quadrada da inversa, seno e cosseno (NVIDIA, 2012).

2.2.1.1 GPGPU

A demanda cada vez maior por qualidade de aplicações gráficas, jogos e imagens fez com que os equipamentos que atendessem esses requisitos fossem fabricados em larga escala, reduzindo o seu custo e tornando-se mais acessíveis. Então o uso da GPU em outros contextos tornou-se viável, o que permitiu o surgimento do termo GPGPU.

O conceito de GPGPU é de explorar o uso da GPU em problemas que não estão necessariamente relacionados ao processamento gráfico. Nem todo tipo de problema pode ser resolvido usando GPUs, porém existem vários casos de aplicações que conseguem um aumento de desempenho considerável utilizando este *hardware* (SOUZA, 2011). O desafio é identificar os problemas que podem ser mapeados para esta plataforma que inicialmente fora pensada para processamento gráfico.

As abordagens iniciais para a programação em GPU exigiam dos programadores um conhecimento sobre como mapear os cálculos para problemas que poderiam ser representados por triângulos e polígonos. Mesmo para os que sabiam programação gráfica essa era uma atividade árdua. Outra característica que retirava a flexibilidade de programar aplicações não gráficas para GPU era a incapacidade de escrita em qualquer segmento da

DRAM da GPU, apesar da leitura poder ser efetuada.

Algumas APIs de programação foram desenvolvidas para a programação em GPU. O OpenGL é um padrão aberto que fornece um conjunto de funções para a criação de aplicações gráficas, expondo todos os recursos do *hardware* do vídeo (OPENGL, 2013). Já o DirectX é um padrão proprietário mantido pela Microsoft para utilização em sua plataforma, utiliza o HLSL (MSDN, 2013) para o desenvolvimento de *shaders*. O *Cg*, ou *C* for *Graphics*, é uma linguagem desenvolvida pela NVIDIA baseada em C ANSI para suporte ao design gráfico e para o desenvolvimento de algoritmos para *pixel* e *shaders* (FERNANDO, 2003).

Porém, com o desenvolvimento das GPUs para propósito geral, os ambientes que mais se destacaram foram o OpenCL e o CUDA. O OpenCL é uma arquitetura para escrever programas que funcionam em ambientes heterogêneos, CPU e GPU (OPENCL, 2013), provendo suporte tanto ao paralelismo de dados quanto de tarefas, incluindo também uma linguagem para a programação de funções que é bastante parecida com a programação CUDA que será apresentada na Seção 2.3.2.

2.2.2 Descrição do Cluster Utilizado

O ambiente que será caracterizado foi utilizado para experimentar e obter os resultados das soluções desenvolvidas. A escolha foi feita considerando as necessidades de computação da aplicação paralela a ser estudada.

2.2.2.1 ELEANORRIGBY

A plataforma **ELEANORRIGBY** (eleanorrigby.iteam.upv.es) está formada por um nó com 24 processadores Intel Xeon X5680 com *clock* de 3.33 GHz e 96 GB DDR3 de memória principal. Cada processador contém 6 núcleos com 12 MB de memória cache. A plataforma também contém 2 GPUs NVIDIA Tesla C2070 com 14 *stream multiprocessors* (SM) e 32 *stream processors* (SP) cada um com um total de 448 núcleos, tem 16 unidades de *load/store*, com *32K-word* no registrador, 64K de memória RAM configurável. Cada SM contém unidades simples e duplas de ponto flutuante. Sendo que cada operação segue o IEEE 754-2008 *floating-point*. Esta plataforma pertence ao Grupo de Computação Interdisciplinar (INTERDISCIPLINARY..., 2013) da Universidade Politécnica de Valencia, Espanha.

2.3 Ferramentas de Software

As ferramentas de software usadas para este trabalho podem ser divididas em dois grupos diferentes: linguagens de programação e bibliotecas.

Como linguagem de programação foi utilizado o ANSI C/C++ para a programação sequencial. Para o paradigma de memória compartilhada foi utilizada as bibliotecas OpenMP (OPENMP, 2013) e CUDA (CUDA, 2013).

2.3.1 OpenMP

O padrão OpenMP é desenvolvido e mantido pelo grupo *Architecture Review Board* (ARB), formado por fabricantes de *software* e *hardware*, tais como *SUN Microsystems*, SGI, IBM, Intel, dentre outros, que, no final de 1997, reuniram esforços para criar um padrão de programação paralela para arquiteturas de memória compartilhada (SENA; COSTA, 2008). O *Open* significa que é um padrão e está definido por uma especificação de domínio público, enquanto *MP* são as siglas de *Multi Processing*. O OpenMP consiste em uma API e um conjunto de diretivas que permite a criação de programas paralelos com compartilhamento de memória através da implementação automática e otimizada de um conjunto de threads. Suas funcionalidades podem atualmente ser utilizadas nas linguagens Fortran 77, Fortran 90, C e C++.

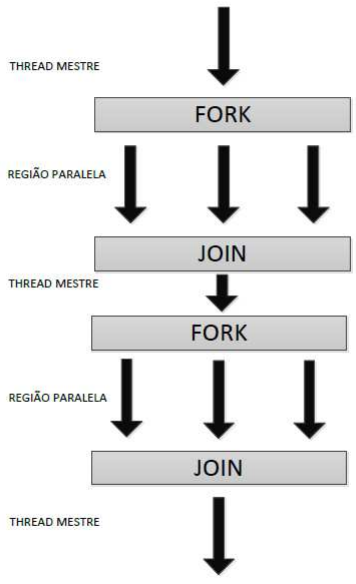


Figura 4 – Modelo de Programação OpenMP.

Fonte: (FILHO, 2012)

Conforme pode ser visto na Figura 4, o paralelismo no OpenMP é baseado em *threads* através do padrão *fork-join*. Todos os programas OpenMP iniciam com uma única *thread*, denominada de *thread* mestre. A *thread* mestre executa suas tarefas sequencial-

mente até que a primeira região paralela é encontrada. Então um *fork* é realizado criando um conjunto de *threads*, assim as instruções originalmente delimitadas no código pela diretiva são executadas em paralelo nas diversas *threads* do conjunto. Quando as *threads* filhas completam a execução das instruções na região paralela elas sincronizam e terminam, restando por fim apenas a *thread* mestre novamente (FILHO, 2012).

O OpenMP é constituído de variáveis de ambiente, diretivas de compilação e bibliotecas de serviço, que juntas integram todas as funcionalidades disponíveis no padrão. Na Figura 5 é ilustrada a sintaxe desses elementos para a linguagem C/C++.

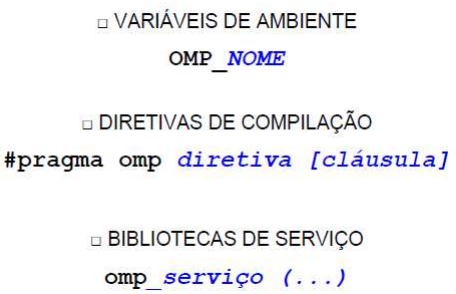


Figura 5 – Elementos da interface OpenMP.

Fonte: (FILHO, 2012)

As diretivas consistem em linhas de código com significado "especial" para o compilador. Por exemplo, nas linguagens C e C++ as diretivas OpenMP são identificadas pelo *#pragma omp*. Algumas das diretivas do OpenMP são: *parallel, section, single, master, critical, ush, ordered, barrier e atomic*. Essas diretivas especificam o compartilhamento de trabalho entre threads ou instruções de sincronização.

2.3.2 CUDA

Enxergando a oportunidade de crescimento da utilização das GPUs, a NVIDIA desenvolveu uma arquitetura de propósito geral chamada *Compute Unified Device Architecture* (CUDA), que visa tirar proveitos do poder de processamento paralelo das GPUs sem que o programador necessite conhecer de computação gráfica (NVIDIA, 2013). A Figura 6 ilustra a estrutura de desenvolvimento fornecida pela NVIDIA. Esta é composta por *drives* de acesso ao *hardware*, bibliotecas matemáticas e de *runtime*, além de um compilador com ferramentas de otimização e depuração de aplicativos.

A programação utilizando o CUDA define o conceito de *kernels* que são funções executadas N vezes em paralelo por N *threads* na GPU. Quando um *kernel* é chamado pela CPU o fluxo de execução sai da CPU, que é conhecida como *host*, e segue na GPU, conhecida como *device*. Esta programação é orientada a *threads* e o gerenciamento automático destas *threads* reduz a complexidade da programação e oferece a CPU uma visão

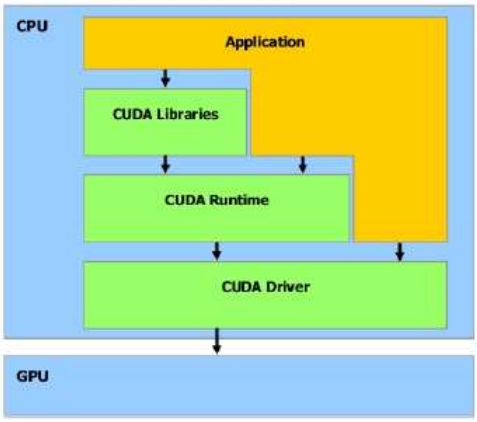


Figura 6 – Pilha de Software do CUDA.

Fonte: (NVIDIA, 2013)

de que a GPU é um coprocessador capaz de processar um número muito grande de *threads* em paralelo (NVIDIA, 2013).

A organização e distribuição dos dados no *hardware* e entre as *threads* é realizada através do conceito de bloco e *grid*. Conforme pode ser visto na Figura 7, o bloco é um lote de *threads* que podem cooperar juntas pelo compartilhamento eficiente dos dados através de uma memória de acesso rápido e sincronização de suas ações para o acesso a memória (NVIDIA, 2013).

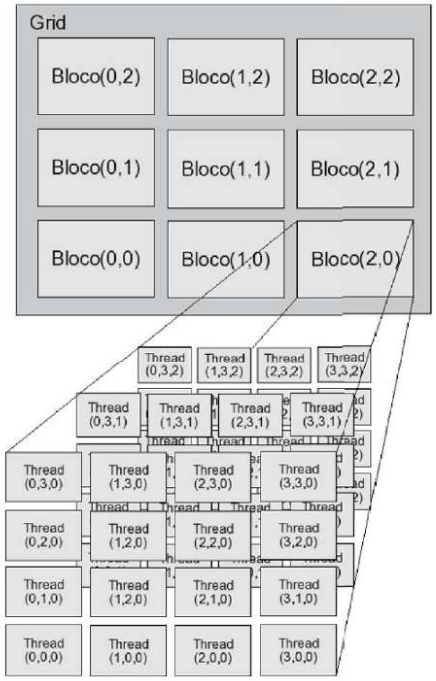


Figura 7 – Organização das *Threads* na GPU.

Fonte: (NVIDIA, 2013)

Cada *thread* no bloco possui um identificador único tridimensional que pode ser acessado através da variável *threadIdx* (*threadIdx.x*, *threadIdx.y*, *threadIdx.z*). Múltiplos blocos são organizados em *grids*, os quais são uma estrutura completa de distribuição das *threads* que executam uma função. Neles são definidos o número total de blocos e de *threads* que serão gerenciados pela GPU. Os *grids* podem ter uma ou duas dimensões e os blocos são acessados nos *grids* através de seu próprio identificador bidimensional (*blockIdx.x*, *blockIdx.y*).

Um grid é executado na GPU pelo escalonamento de blocos para execução nos multiprocessadores. Cada multiprocessador processa um bloco após o outro colocando-os como ativo. Internamente os blocos são separados em *Warps*, que são grupos contendo 32 *threads* controladas pelo gerenciador dentro de cada multiprocessador, sendo executadas no modelo *SIMD* (NVIDIA, 2013). A ordem de execução dos *warps* dentro de um bloco é indefinida, porém sua execução pode ser sincronizada para coordenar o acesso à memória global ou compartilhada, mas não é possível sincronizar *threads* de blocos diferentes.

A GPU possui um sistema próprio de memória, sendo que a *thread* possui acesso a níveis diferentes de memória. Cada *thread* possui uma memória local privada, que não pode ser acessada por nenhuma outra *thread*, ela é uma abstração de uma memória local e fica localizada na memória global. É utilizada quando o conjunto de dados ultrapassa o limite de memória que se localiza no multiprocessador e possui mesmo tempo de vida da *thread*.

Na Figura 8 é apresentada a hierarquia de acesso a memória em um multiprocessador da GPU. Cada multiprocessador executa um bloco e possui uma memória compartilhada que é visível às *threads* do mesmo bloco, servindo para troca de informações entre elas. Cada *thread* é executada em um processador e possui acesso a memória global situada na DRAM que suporta apenas leitura ou leitura-escrita de dados, mas esse acesso é mais lento se comparado com o acesso aos registradores ou a memória compartilhada. Existem também dois espaços adicionais de memória acessíveis a todas as *threads*, porém esses espaços são apenas de leitura, que são os espaços de memória de constantes e de textura (NVIDIA, 2013).

Quando um *kernel* é executado a função só consegue acessar a memória presente no *device*, isso quer dizer que ela não consegue acessar diretamente a memória do *host*. Como a *thread* inicial está no *host* os dados estão na sua memória RAM, então antes de fazer a chamada do *kernel* é necessário copiar os dados do *host* para a memória do *device*, desta maneira a função poderá acessar os dados diretamente. É comum programas CUDA possuirem as funções *cudaMalloc()* e *cudaFree()*, pois elas são responsáveis por alocar e liberar espaço na memória global do *device*. O fluxo consiste em alocar memória no *device*, copiar os dados do *host* para o *device* e fazer a chamada do *kernel*. Depois de terminada a computação, os dados são copiados do *device* para o *host* e a memória

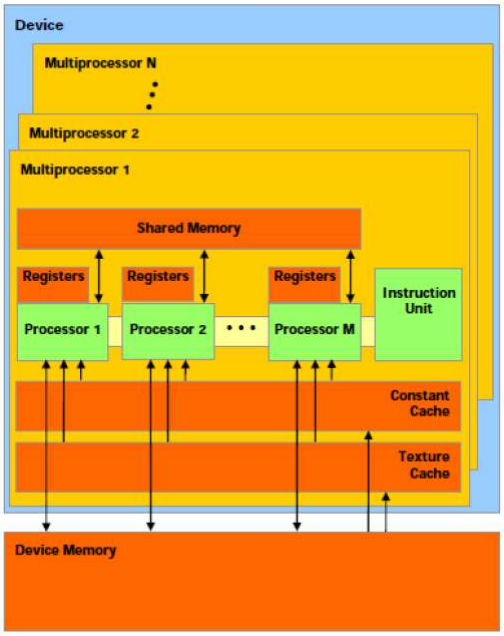


Figura 8 – Hierarquia de memória CUDA.

Fonte: (NVIDIA, 2013)

do *device* é liberada. A função *cudaMemCpy()* é responsável pela transferência de dados entre a memória do *device* e a memória do *host* (NVIDIA, 2013).

A operação do CUDA se dá a partir de extensões na linguagem C e utiliza qualificadores de funções que definem onde a função será executada, se no *host* ou no *device*, como também onde ela será chamada (NVIDIA, 2013). Esses qualificadores podem ser:

- ___device___ : este qualificador declara que a função é executada no *device* e só poderá ser chamada pela mesma. Funções deste tipo não aceitam recursão nem argumentos variáveis.
- ___global___ : define a função *kernel*, que será executada no *device* e obrigatoriamente será chamada a partir do *host*. Esta função deve obrigatoriamente retornar *void* e receber como parâmetro a dimensão do grid e do bloco que será usado na execução da função. Sua chamada é assíncrona, a execução na CPU continua mesmo que não tenha terminado na GPU.
- ___host___ : define a função que será chamada e executada apenas pelo *host*.

Os qualificadores de variável especificam a localização da variável na memória do *device* (NVIDIA, 2013). Caso alguma variável declarada no código do *device* não possua nenhum qualificador, ela será alocada no registrador, contudo em alguns casos o compilador pode escolher coloca-la na memória local, o que pode causar um prejuízo no desempenho. Os qualificadores de variáveis são:

- ___device___ : define uma variável que reside na memória global do *device*. É acessível a todas as *threads* de um grid e também pode ser acessada pelo *host* através da biblioteca de *runtime* do CUDA. Possui o mesmo tempo de vida da aplicação.

- ___shared___: define uma variável que reside na memória compartilhada de um bloco de *threads*. É acessível apenas pelas *threads* de um mesmo bloco, possuindo o mesmo tempo de vida do bloco.

- ___constante___: define uma variável que reside na memória constante, sendo acessível por todas as *threads* de um grid apenas para leitura. Pode ser acessada também pelo *host* através da biblioteca de *runtime*, podendo ser lida e escrita.

2.4 Ferramentas Matemáticas

As ferramentas matemáticas usadas neste trabalho foram métodos de otimização que utilizam a técnica de eliminação de região para encontrar o ótimo em funções de uma única variável. Estes métodos não requerem que a função seja diferenciável ou contínua. Simplesmente assumem que o mínimo está em um certo intervalo [A,B] da função g a ser minimizada e que esta tem uma característica unimodal.

Conforme Minoux (1986), uma função é unimodal em [A,B] quando tem um único mínimo local, mas isto não quer dizer que necessariamente ela seja diferenciável ou contínua.

Portanto uma função é unimodal quando o valor da função g pode ser calculado em quatro pontos a,b,c,d do intervalo [a,d] e há sempre um subintervalo que não contém o ótimo e pode ser ignorado. Desta forma obtemos o intervalo reduzido, no qual o procedimento pode ser recomeçado (MINOUX, 1986).

A partir de um intervalo [a,b] que contém o ótimo, é possível encontra-lo através da utilização de um dos métodos que serão apresentados.

2.4.1 Método Dicotomia

O método consiste em partir de um intervalo $[\alpha_{min}, \alpha_{max}]$ e então ir reduzindo progressivamente este intervalo por bisseções até encontrar um intervalo final com largura $< \varepsilon$, sendo o tamanho de ε escolhido anteriormente e suficientemente pequeno.

Este método nos permite, a cada passo, reduzir pela metade o intervalo que contém o ótimo através do cálculo da função g com dois novos pontos. Segundo Minoux (1986), através da realização de n cálculos da função g, nós podemos reduzir o cumprimento do intervalo em uma proporção de $2^{\frac{(n-3)}{2}}$.

A Figura 9 ilustra o funcionamento do método na busca do valor mínimo da função. Começamos com o intervalo $[a_0, b_0]$. Tomamos o ponto médio $c_0 = \frac{(a_0+b_0)}{2}$, depois os dois pontos $d_0 = \frac{(a_0+c_0)}{2}$, $e_0 = \frac{(c_0+b_0)}{2}$, então nós obtemos cinco equidistantes pontos com $\gamma = \frac{(b_0-a_0)}{4}$.

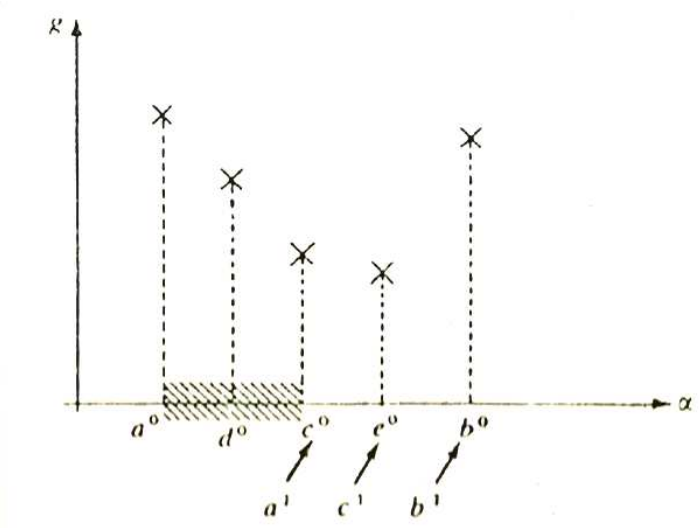


Figura 9 – Ilustração da diminuição no intervalo de busca através do método dicotomia.

Fonte: (MINOUX, 1986)

Fazendo uso de unimodalidade, podemos sempre eliminar dois dos quatro subintervalos (porque o ótimo não está dentro deles) e somente dois intervalos adjacentes permanecem $[a_1, c_1]$ e $[c_1, b_1]$. Portanto, somos levados a um problema com o segmento $[a_1, b_1]$ com metade do tamanho. Na próxima etapa haverá apenas dois novos cálculos da função g para serem realizados, nos pontos $d_1 = \frac{(a_1+c_1)}{2}$ e $e_1 = \frac{(c_1+b_1)}{2}$, e assim por diante até que o intervalo atinja a largura de ε desejado.

A Tabela 1 mostra a taxa de redução do tamanho do intervalo em relação ao número de cálculos da função g. A convergência global do método para funções unimodais decorre da definição, a taxa de convergência é linear com um valor de $\frac{1}{\sqrt{2}} \simeq 0.7$ (MINOUX, 1986).

Tabela 1 – Taxa de redução do intervalo de busca aplicando dicotomia.

n	$\frac{b_n-a_n}{b_0-a_0} = 1/2^{\frac{(n-3)}{2}}$
17	10^{-2}
23	10^{-3}
29	10^{-4}
42	10^{-6}

Fonte: (MINOUX, 1986)

2.4.2 Método Seção Áurea

Seguindo o mesmo princípio descrito na subseção 2.4.1 (eliminação de um subintervalo para cada ponto recém-calculado), o método da seção áurea busca a solução ótima reduzindo o intervalo de busca em uma razão de 0,618, em cada iteração. Essa razão corresponde a parte fracionária da proporção áurea $\varphi \cong 1{,}618$. As reduções de intervalos acontecem até que o mesmo atinja uma tolerância especificada (ε). Sendo que os comprimentos dos intervalos sucessivos seguem em uma determinada proporção, de modo que, na fase $K+1$ a disposição relativa dos pontos seja a mesma da fase K.

Seja *g(x)* uma função com um único mínimo no intervalo [a, b]. O método consiste em criar uma sequência (x_n) que converge para o mínimo da função. A sequência (x_n) será dada por $x_n = \frac{a_n+b_n}{2}$ onde $[a_0, b_0] \supset [a_1, b_1] \supset ...[a_n, b_n] \supset \ldots$.

Assim seja $a_0 = a$ e $b_0 = b$,

$a_1 = b_0 - 0.618(b_0 - a_0)$ e $b_1 = a_0 + 0.618(b_0 - a_0)$,

Se $g(a_1) < g(b_1)$, então $a_2 = a_1$ e $b_2 = a_1 + 0.618(b_1 - a_1)$,

Se $g(a_1) \geq g(b_1)$, então $a_2 = b_1 - 0.618(b_1 - a_1)$ e $b_2 = b_1$.

Novamente

Se $g(a_2) < g(b_2)$, então $a_3 = a_2$ e $b_3 = a_2 + 0.618(b_2 - a_2)$,

Se $g(a_2) \geq g(b_2)$, então $a_3 = b_2 - 0.618(b_2 - a_2)$ e $b_3 = b_2$.

E o processo segue até que $\mid x_n - x_{n-1} \mid$ seja menor que um certo ε fixado como critério de parada (BERGAMASHI, 2010).

2.4.3 Método Fibonacci

O método de Fibonacci é um procedimento de busca linear utilizado para minimizar funções unimodais sobre um intervalo limitado. De forma análoga ao método da seção áurea, são realizadas duas avaliações de função na primeira iteração e uma avaliação de função nas iterações seguintes. Contudo, diferentemente do método da seção áurea que reduz o intervalo de incerteza na razão 0,618, o método Fibonacci reduz o intervalo de acordo com a sequência de números inteiros designados por números de Fibonacci F_i que têm a particularidade de serem definidos pelas seguintes equações:

$$F_0 = F_1 = 1, \tag{2.1}$$

$$F_i = F_{i-1} + F_{i-2}, i \geq 2. \tag{2.2}$$

Segundo Araújo (2009), para cada iteração k em um intervalo de incerteza $[a_k, b_k]$, com n sendo o número total de avaliações de funções planejadas, os pontos y_k e z_k são

selecionados da seguinte forma:

$$y_k = a_k + \frac{F_{n-k-1}}{F_{n-k+1}}(b_k - a_k), k = 1, ..., n-1, \tag{2.3}$$

$$z_k = a_k + \frac{F_{n-k}}{F_{n-k+1}}(b_k - a_k), k = 1, ..., n-1. \tag{2.4}$$

O novo intervalo de incerteza $[a_{k+1}, b_{k+1}]$ é equivalente a $[y_k, b_k]$ se f(y) > f(z) ou equivalente a $[a_k, z_k]$ se f(y) $\leq$ f(z).

2.5 Considerações do capítulo

Este capítulo teve como objetivo explicar e fundamentar aspectos importantes abordados nesta pesquisa. Iniciou-se com as classificações das arquiteturas paralelas, os tipos de paralelismo, como também os modelos de programação e comunicação.

Na sequência houve uma explanação sobre GPU e o ambiente de alto desempenho utilizado nas experimentações. Também foram abordadas as bibliotecas CUDA e OpenMP que atuam sobre o paradigma de memória compartilhada na aplicação utilizada como estudo de caso.

Encerrou-se então com métodos matemáticos de otimização que guiaram o desenvolvimento das heurísticas utilizadas no processo de busca por parâmetros ótimos. Essas heurísticas serão apresentadas detalhadamente no próximo capítulo.

3 Heurísticas Aplicadas na Otimização do Modelo Paralelo

Serão descritas neste capítulo as heurísticas que foram utilizas para diminuir o espaço de busca por parâmetros ótimos para um modelo paralelo híbrido. Para validação foi aplicado um estudo de caso específico chamado Espacialização do Relevo.

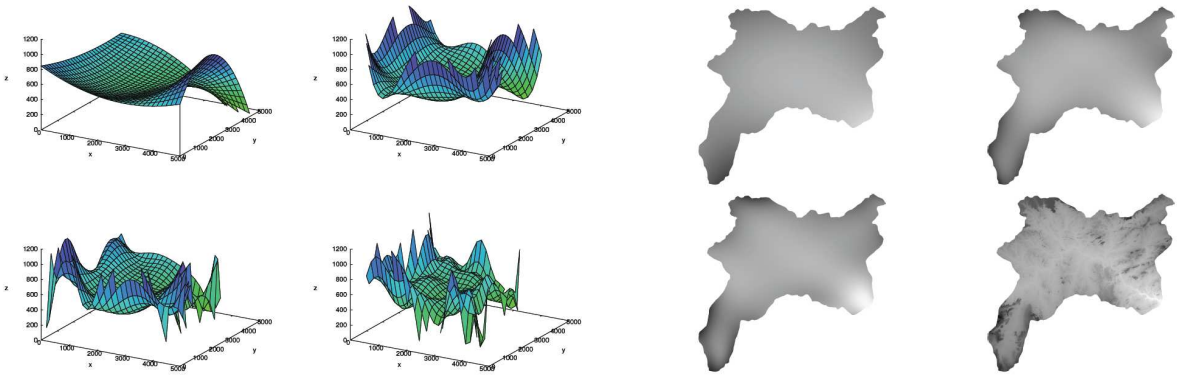
3.1 Estudo de Caso: Espacialização do Relevo

3.1.1 Introdução

A Espacialização do Relevo é um instrumento muito utilizado na representação do relevo e sua grande importância deve-se ao fato de possibilitar a descrição de fenômenos por meio de modelos matemáticos a partir de uma amostra de dados. Graficamente, equivale a identificar a curva ou superfície que melhor se ajusta aos pontos de um diagrama de dispersão. Um problema identificado nessa representação foi a alta capacidade de processamento e armazenamento, pois a representação com uma maior fidelidade requer um polinômio de alto grau (BORATTO et al., 2012).

Para estimar os coeficientes do polinômio é realizada uma regressão polinomial, que é um tipo de interpolação global e como tal, exige um poder computacional significativo para ser realizada em um conjunto de dados muito grande. O relevo da área escolhida é representado por um conjunto de pontos obtido pelo modelo digital de elevação na forma de uma grade regular (BORATTO; COELHO; BARRETO, 2012). As informações relativas ao relevo estão armazenadas em um arquivo texto e para cada medição realizada existe um valor que corresponde à altitude na respectiva posição. As linhas e colunas do arquivo representam a latitude e longitude onde a medição foi realizada.

A Figura 10 apresenta a Espacialização do Relevo com diferentes graus de polinômios. Pode-se notar que a fidelidade da representação é diretamente proporcional ao grau do polinômio. Segundo Boratto, Coelho e Barreto (2012) o grau considerado ótimo para representar o Vale do Rio São Francisco seria o grau 500, no qual seria ajustado com uma correlação $R^2 = 0{,}998$, mas seriam necessários aproximadamente 45 anos para gerar todos os coeficientes das matrizes do sistema de equações lineares através da computação sequencial, logo torna-se fundamental a aplicação de modelos de computação de alto desempenho na Espacialização do Relevo.



(a) Visão 3D (b) Visão em Escala de Cinza

Figura 10 – Espacialização do Relevo para os graus 2, 4, 6, 20 do polinômio.

Fonte: (BORATTO; COELHO; BARRETO, 2012)

3.1.2 O modelo Paralelo Híbrido

A aplicação híbrida da Espacialização do Relevo foi implementada na linguagem C fazendo uso do CUDA e do padrão OpenMP. Os dados de entrada são representados por matrizes que contêm as coordenadas geográficas de latitude, longitude e altitude da região a ser espacializada.

No processo de regressão polinomial, a solução final resume-se na representação matricial $Ax = b$, onde A é uma matriz representada por somatórios independentes. A construção da matriz A é a parte mais custosa de todo o processo, mas em virtude da independência dos seus somatórios, pode-se calculá-los simultaneamente, o que proporciona uma grande possibilidade de paralelismo. Para explorar esse paralelismo, o código particiona esses somatórios em pedaços que serão distribuídos entre os recursos computacionais disponíveis.

O código paralelo divide os somatórios que deverão ser feitos entre os recursos disponíveis no ambiente que será utilizado, podendo este ser heterogêneo, composto por multi-core e multi-GPU. A Figura 11 ilustra o particionamento das tarefas em um ambiente com duas GPUs e doze CPUs. A proporção destinada as GPUs é dividida entre as placas gráficas disponíveis e o restante destinado à CPU é particionado entre as *threads* que executarão em cada processador.

O particionamento da carga de trabalho se dá através de uma estratégia estática para enviar dados e tarefas para os núcleos de CPU e para as múltiplas GPUs. A percentagem de carga de trabalho fornecida a cada sistema é uma entrada do algoritmo e é de responsabilidade do usuário informar um valor proporcional a potência de computação dos sistemas, para que a execução utilize o máximo dos recursos disponíveis. Uma vez dada a percentagem desejada de computação, o tamanho dos dados é calculado antes da execução do código representado pelo Algoritmo 1.

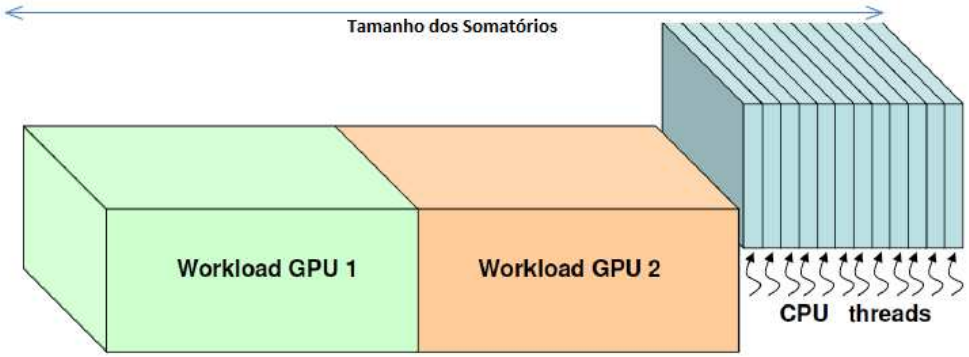


Figura 11 – Divisão dos cálculos a serem realizados pelas GPUs e CPUs na Espacialização do Relevo.

Fonte: (BORATTO; COELHO; BARRETO, 2012)

No Algoritmo 1 é controlada a execução no ambiente paralelo híbrido. Na primeira linha deste algoritmo é informada a quantidade de *threads* que serão utilizadas na região paralela. Esta quantidade é atribuída a partir da variável *nthreads* que corresponde à soma do número de GPUs e CPUs que serão utilizados no processamento. A região paralela é então iniciada na linha 3, onde serão inicializadas as *nthreads* que executarão as instruções das linhas 4 a 14.

Algoritmo 1: Trecho de código híbrido: OpenMP + CUDA.

```
1: omp_set_num_threads(nthreads);
2:
3: #pragma omp parallel {
4: int thread_id = omp_get_thread_num();
5:
6: if( thread_id < num_gpus ) { /* processamento na GPU */
7:      gpu_id = thread_id;
8:      first = thread_id * sizeGPU;
9:      matrixGPU(...);
10: }
11:  else{                        /* processamento na CPU */
12:      cpu_thread_id = thread_id - num_gpus;
13:      first = num_gpus*sizeGPU + cpu_thread_id*sizeThr;
14:      matrixCPU(...);
15:     }
16:}
```

No momento de inicialização da região paralela, cada *thread* receberá um identificador único (*id*), podendo este ser acessado através da função *omp_get_thread_num()* (linha 4). A estrutura condicional presente na linha 6 tem como objetivo identificar se a *thread* que está sendo executada realizará o processamento na CPU ou GPU. Este controle é feito pelo *id* e destinando as primeiras *threads* de execução para se comunicarem com as GPUs que realizarão o processamento, enquanto as restantes serão utilizadas para processarem nas CPUs.

Após o reconhecimento da *thread* como destinada a CPU ou GPU, é realizado um cálculo para encontrar a variável *first* (linhas 8 e 13). Esta variável indica onde cada *thread_id* irá trabalhar em relação ao seu pedaço do somatório. Então são chamadas as funções *matrixGPU* e *matrixCPU* que são responsáveis pela computação do montante parcial destinado a cada recurso computacional. As funções *matrixGPU* e *matrixCPU* recebem como parâmetros as matrizes que devem calcular e produzem um *array* linear A_[thread_id]. O resultado da matriz A da regressão polinomial é o somatório desses *arrays* A_[0] + A_[1] + ... + A_[$nthreads - 1$].

Segundo Boratto, Coelho e Barreto (2012) o sistema atinge uma alta escalabilidade ao particionar a carga de trabalho de forma proporcional à potência de computação. Porém, cabe ao usuário informar quanto deste trabalho deve ser destinado a cada sistema de computação (CPU e GPU), tarefa essa que não é trivial, pois exige um profundo conhecimento em computação paralela e do ambiente computacional a ser utilizado. Além do mais, esse parâmetro irá variar de um ambiente para outro devido as especificidades de cada um.

Ferramentas de auto-otimização auxiliam a encontrar os melhores parâmetros para as aplicações em cada ambiente. Esses valores normalmente são encontrados utilizando a aplicação como *benchmark* (WILLIAMS, 2008). Esta técnica consiste em executar a aplicação com diferentes parâmetros até encontrar o que proporciona um melhor desempenho. Esta escolha pode ser feita através de um método de exploração que restrinja as possibilidades do espaço de busca.

Segundo Almeida (2011), a busca no espaço de otimização pode ser feita através de três abordagens principais: busca exaustiva, *hill climbing* e heurísticas. A mais simples e direta é a busca exaustiva, a qual testa todas as possibilidades dentro do espaço de busca. Nesta abordagem, a depender da quantidade de combinações, o espaço de busca pode tornar a adaptação inviável em relação ao tempo de execução do algoritmo.

O método *hill climbing* efetua um busca linear dentro de todo o espaço de busca de um determinado parâmetro, mantendo fixos os outros. Entretanto, devido as interações implícitas entre parâmetros, este método pode resultar em escolhas de valores muito distantes do ótimo global (ALMEIDA, 2011). A terceira técnica é utilizar heurísticas para reduzir o espaço de busca e o custo computacional do processo de auto-otimização.

3.2 Aplicação de Métodos Matemáticos na Otimização

As próximas subseções apresentarão as heurísticas que foram construídas para otimizar a busca por parâmetros ótimos para aplicações paralelas híbridas. O objetivo é encontrar estes parâmetros de uma maneira mais eficiente do que o método tradicional de busca exaustiva. Para atingir este objetivo, os algoritmos foram implementados baseados

nos métodos matemáticos de otimização apresentados na Seção 2.4.

Os algoritmos visam encontrar o parâmetro que proporciona a aplicação paralela ser realizada em menor tempo de execução.

3.2.1 Algoritmo Dicótoma

O Algoritmo 2 foi implementado, baseando-se no método de busca Dicotomia (seção 2.4.1). Este método encontra o ótimo reduzindo progressivamente o intervalo de busca, através de bisseções, até que este atinja uma *precisão* desejada (margem de tolerância).

Algoritmo 2: Algoritmo baseado no método de busca Dicotomia.

```
 1: a = inicioIntervalo;          b = fimIntervalo;
 2: subintervalo = ((b - a)/4);
 3: Se(subintervalo > precisão){
 4:    p1 = a;                       t1 = executa(p1);
 5:    p2 = a + subintervalo;        t2 = executa(p2);
 6:    p3 = a + (subintervalo*2);    t3 = executa(p3);
 7:    p4 = a + (subintervalo*3);    t4 = executa(p4);
 8:    p5 = b;                       t5 = executa(p5);
 9:    Se(t1 >= t5){
10:         Se(t2 >= t5){
11:             a = p3;
12:             t1 = t3;   t3 = t4;
13:         }
14:         Senão{
15:             a = p2;     b = p4;
16:             t1 = t2;    t5 = t4;
17:         }
18:     }
19:     Senão{
20:         Se(t4 >= t1){
21:            b = p3;
22:            t3 = t2;    t5 = t3;
23:         }
24:         Senão{
25:            a = p2;     b = p4;
26:            t1 = t2;    t5 = t4;
27:         }
28:     }
29:     subintervalo=((b - a)/4);
30:     Enquanto(subintervalo > precisão){
31:        p1 = a;
32:        p2 = a + subintervalo;
33:        p3 = a + (subintervalo*2);
34:        p4 = a + (subintervalo*3);
35:        p5 = b;
36:        t2 = executa(p2);
37:        t4 = executa(p4);
38:         /* Trecho de código da linha 9 a 28 */
39:         subintervalo=((b - a)/4);
40:     }
41: }
```

O método inicia definindo os cinco pontos (p) a serem analisados. Os pontos 1 e 5 correspondem respectivamente ao início e fim do intervalo. O ponto 3 é o médio entre o $p1$ e $p5$, o ponto 2 é o médio entre o 1 e o 3, assim como o 4 é entre o $p3$ e o $p5$. Estes pontos são equidistantes conforme exigido pelo modelo matemático.

As variáveis $inicioIntervalo$ e $fimIntervalo$ representam o intervalo inicial de busca e devem ser definidas antes da execução do código representado pelo Algoritmo 2. Os valores dessas variáveis serão atribuídos respectivamente as variáveis a e b (linha 1) que representarão o início e fim do intervalo em cada processo iterativo.

A distância entre cada ponto é calculada e atribuída a variável *subintervalo* (linha 2) e é fundamental para o controle de parada do algoritmo. O algoritmo prossegue com as divisões somente quando o *subintervalo* é maior que a *precisão* definida como margem de tolerância, este controle é feito pelas condicionais das linhas 3 e 30.

Nas linhas 4 a 8 são definidos e executados os pontos que orientarão em quais subintervalos a busca será aprofundada. Para ter o tempo de execução da aplicação paralela com o parâmetro testado, é utilizada uma função chamada *executa* que se encarrega de fazer a experimentação no ambiente paralelo e retornar o tempo gasto.

De posse dos tempos de execução dos cinco pontos diferentes, cabe analisar quais subintervalos têm valores maiores e eliminá-los prosseguindo a busca pelo mínimo nos que restaram. Conforme o método matemático, podemos eliminar dois dos quatro subintervalos, permanecendo somente com os dois adjacentes que devem conter o ótimo.

A análise e estabelecimento dos subintervalos que permanecerão é realizada no bloco que está entre as linhas 9 a 28. Caso os tempos de execução dos pontos 1 e 2 ($t1$ e $t2$) sejam maiores que o tempo do ponto 5 ($t5$), indica que o a valor mínimo não está nos dois primeiros subintervalos ($[p1, p2]$ e $[p2, p3]$). Então devemos aprofundar as busca nos subintervalos $[p3, p4]$ e $[p4, p5]$. Desta forma, o próximo intervalo de busca será do ponto 3 (linha 11) ao ponto 5. Reduzindo pela metade o espaço de busca a cada iteração.

A linha 15 é executada quando devem ser eliminados os subintervalos $[p1, p2]$ e $[p4, p5]$. Enquanto a linha 21 restringe o próximo intervalo de busca a $[p1, p3]$ eliminando os subintervalos $[p3, p4]$ e $[p4, p5]$. Caso os tempo de execução dos pontos 1 e 5 ($t1$ e $t5$) sejam os maiores, então os subintervalos $[p1, p2]$ e $[p4, p5]$ serão eliminados na linha 25 no momento em que atribuímos $p2$ a variável a (início do intervalo) e $p4$ a variável b (fim do intervalo).

Na linha 29 é atualizado o comprimento que cada subintervalo terá na próxima iteração caso seja afirmativa a condicional da linha 30. Segue-se então estabelecendo os próximos cinco pontos que orientarão a busca. Destes cinco, três estavam presentes na iteração anterior, então só precisamos executar aplicação paralela com os dois novos pontos (linhas 36 e 37).

Para não precisar executar novamente os pontos da iteração anterior é necessário guardar estes valores conforme a posição que eles ficarão na próxima iteração. Para realizar esse controle as atribuições dos valores foram feitas dentro do bloco que analisa e estabelece os subintervalos para iteração seguinte (linhas 9 a 28). Mas especificamente nas linhas 12, 16, 22 e 26 que fazem as atribuições necessárias de acordo com os subintervalos que permaneceram.

quando os subintervalos de busca atingirem um tamanho menor que a *precisão* informada no inicio do processo, são então encerradas as iterações e basta agora identificar qual dos pontos da última iteração proporcionou um menor tempo de execução, assim tem-se o valor ótimo para o parâmetro naquele ambiente.

3.2.2 Algoritmo Seção Áurea

O Algoritmo 3 fundamenta-se no método matemático da Seção Áurea (Seção 2.4.2). A técnica utilizada é reduzir sucessivamente o intervalo de busca a uma proporção de $0,618$ até atingir uma tolerância especificada (*precisão*).

A cada iteração o algoritmo utilizará dois pontos (p) para orientar onde deve aprofundar a busca. Estes pontos são calculados a partir do intervalo de busca e de um dx, que corresponde à distância relativa à razão áurea. O primeiro ponto ($p1$) é calculado somando o início do intervalo com o dx (linha 4), enquanto o segundo ponto ($p2$) é a subtração entre o fim do intervalo e o dx (linha 5).

Antes de ser executado o código representado pelo algoritmo 3, deve ser definido o intervalo inicial de busca (variáveis $inicioIntervalo$ e $fimIntervalo$), como também a *precisão* para o resultado. O processo iterativo de busca é realizado enquanto o dx é maior que a tolerância especificada (linhas 6 e 9).

Nas linhas 16 e 24 é utilizada a função $executa()$ para obter o tempo que foi gasto pela aplicação paralela com o parâmetro testado. Se o tempo do ponto 1 ($t1$) é menor que o do ponto 2 ($t2$), o início do intervalo (a) será o valor de $p2$ (linha 11), senão o b (fim do intervalo) receberá o valor de $p1$ (linha 19).

Após a redução de intervalo realizada pela linha 11 ou 19, são calculados os novos valores de dx, $p1$ e $p2$. Mas dos dois pontos que participarão da próxima iteração, um ponto tem o mesmo valor de outro que estava presente na iteração anterior e pode-se otimizar ainda mais reutilizando o seu tempo sem realizar uma nova execução. Esta manobra exige cautela pois o posicionamento dos pontos é diferente, por exemplo, caso o ponto 1 tenha permanecido, ele será o ponto 2 na próxima iteração (linha 15).

O processo iterativo prossegue enquanto o dx possui um valor maior que a *precisão*, quando ele passa a ser menor ou igual, o processo é encerrado e o código executa a linha 27, que faz uma comparação entre os tempo do ponto 1 e 2, retornando o ponto que obteve

Algoritmo 3: Algoritmo baseado no método de busca Seção Áurea.

```
 1: a = inicioIntervalo;
 2: b = fimIntervalo;
 3: dx = 0.618 * (b - a);
 4: p1 = a + dx;
 5: p2 = b - dx;
 6: Se(dx > precisão){
 7:     t1 = executa (p1);
 8:     t2 = executa (p2);
 9:     Enquanto(dx > precisão){
10:         Se(t1 < t2){
11:             a = p2;
12:             dx = 0.618 * (b - a);
13:             p1 = a + dx;
14:             p2 = b - dx;
15:             t2 = t1;
16:             t1 = executa (p1);
17:         }
18:         Senão{
19:             b = p1;
20:             dx = 0.618 * (b - a);
21:             p1 = a + dx;
22:             p2 = b - dx;
23:             t1 = t2;
24:             t2 = executa (p2);
25:         }
26:     }
27:     Se(t1 < t2){
28:         retorna(p1);
29:     }
30:     Senão{
31:         retorna(p2);
32:     }
33:}
```

o menor tempo. Finalizando assim a execução do algoritmo com o retorno do ótimo.

3.2.3 Algoritmo Fibonacci

Para desenvolver este algoritmo foi tomado como base o método de busca Fibonacci (Seção 2.4.3). Neste método as reduções do intervalo de busca são realizadas de acordo com a sequência de números inteiros de Fibonacci (ARAÚJO, 2009).

O primeiro passo do Algoritmo 4 é criar a sequência de números de Fibonacci (Equação 2.1) e calcular qual será o primeiro número da sequência a ser utilizado (linhas 1 e 2). O cálculo do Fibonacci inicial ($f0$) consiste em resolver a Equação 3.1 e verificar qual o número da sequência de Fibonacci que é mais próximo do resultado obtido; este número será o $f0$.

$$f_0 = \frac{(fimIntervalo - inicioIntervalo)}{precisão}. \tag{3.1}$$

O valor da variável *precisão* é a margem de tolerância desejada que deve ser especificada antes do inicio do algoritmo, juntamente com a definição do intervalo inicial de busca ($inicioIntervalo, fimIntervalo$). De posse desses dados, a linha 4 inicializa a variável E que consiste em um ajuste da *precisão* para que fique proporcional ao intervalo de busca e o Fibonacci inicial.

Algoritmo 4: Algoritmo baseado no método de busca Fibonacci.

```
 1: F[] = inicializa serie Fibonacci;
 2: f0 = determina Fibonacci inicial;
 3: posicaoF = busca a posicao de f0 no array F[];
 4: E = (fimIntervalo - inicioIntervalo)/f0;
 5: a = inicioIntervalo;
 6: b = fimIntervalo;
 7: dx = F[--posicao] * E;
 8: p1 = b - dx;
 9: p2 = a + dx;
10:
11: Se(dx > precisao){
12:     t2 = executa (p2);
13:     t1 = executa (p1);
14:     Enquanto(dx > precisão){
15:         Se(t2 < t1){
16:             a = p1;
17:             dx= F[--posicaoF] * E;
18:             p1 = b - dx;
19:             p2 = a + dx;
20:             t1 = t2;
21:             t2 = executa (p2);
22:         }
23:         Senão{
24:             b = p2;
25:             dx= F[--posicaoF] * E;
26:             p1 = b - dx;
27:             p2 = a + dx;
28:             t2 = t1;
29:             t1 = executa(p1);
30:         }
31:     }
32:     Se(t2 < t1){
33:         retorna(p2);
34:     }
35:     Senão{
36:         retorna(p1);
37:     }
38: }
```

O método utiliza a sequência de Fibonacci de modo decrescente, partindo do $f0$ e indo em direção ao primeiro número da sequência. No Algoritmo 4 a sequência foi inicializada em um vetor unidimensional ($F[]$) e faz uso de uma variável chamada *posicaoF*

para controlar em que posição deste vetor está o número da sequência de Fibonacci que será aplicado para definir os pontos de busca.

Assim como nos outros algoritmos implementados, as variáveis a e b especificam o intervalo atual do processo de busca (linhas 5 e 6). Como no algoritmo da Seção Áurea, os primeiros dois pontos são definidos a partir das variáveis a, b e dx. Mas a forma de calculo do dx é diferente, pois neste método o valor é obtido através da multiplicação do número da sequência Fibonacci ($F[]$) pela precisão ajustada (E).

Após definir os primeiro dois pontos (linhas 8 e 9), a função $executa()$ é utilizada para que seja obtido o tempo de execução da aplicação paralela com estes dois parâmetros (linhas 12 e 13). Caso o tempo do ponto 1 ($t1$) seja maior, reduzimos o intervalo de busca desconsiderando o subintervalo $[a, p1]$ ao fazer a atribuição $a = p1$ na linha 16. Caso o tempo do ponto 2 ($t2$) seja maior, o subintervalo desconsiderado será $[p2, b]$ (linha 24).

Nas linhas 17 a 19, como nas linhas 25 a 27 são calculados os novos valores de dx, $p1$ e $p2$ que serão utilizados na próxima iteração. Mas somente será necessário realizar uma execução, pois um dos pontos já foi executado na iteração anterior. Este controle é realizado tanto nas linhas 20 e 21 quanto nas 28 e 29.

Quando dx atinge um valor menor ou igual à $precisão$ especificada, o processo iterativo é encerrado. As linhas 32 a 37 verificam o ponto que proporcionou o menor tempo de execução e então o retornam como solução final do processo de busca.

3.3 Considerações do capítulo

Neste capítulo foram apresentados métodos heurísticos, baseados em modelos matemáticos, que otimizam a busca por parâmetros ótimos para uma aplicação em um determinado ambiente. Estas heurísticas são representadas pelos Algoritmos 2, 3 e 4, e foram implementados na linguagem C, utilizando como validação o estudo de caso apresentado na Seção 3.1. Os resultados dos experimentos serão apresentados no próximo capítulo que também realizará uma comparação entre esses algoritmos e o método tradicional de busca exaustiva.

4 Resultados Experimentais

Serão descritos neste capítulo os resultados das buscas por parâmetros ótimos realizadas pelas heurísticas desenvolvidas.

4.1 Introdução

No capitulo anterior foi apresentado um estudo de caso chamado Espacialização do Relevo que utiliza múltiplas GPUs e CPUs. A partir deste estudo de caso, aplicou-se heurísticas a fim de obter o melhor conjunto de parâmetros de execução para o processamento massivo de dados.

Descobrir a melhor proporção de trabalho não é algo trivial e exige um alto conhecimento de computação paralela como também do ambiente a ser utilizado. Para encontrar este parâmetro de uma maneira mais eficiente, ao invés da aplicação da busca exaustiva, foram desenvolvidas três heurísticas baseadas em métodos matemáticos de otimização. Estas heurísticas foram explicadas através dos algoritmos apresentados na Seção 3.2 e os seus resultados serão apresentados neste capítulo.

4.2 Análise dos Resultados

Os experimentos apresentados utilizaram o estudo de caso da Espacialização do Relevo para um polinômio de grau 25. As execuções foram realizados no cluster ELEANORRIGBY (Seção 2.2.2.1) utilizando 24 CPUs e uma GPU. O objetivo foi encontrar a proporção da carga de trabalho destina à GPU (*workload*) que proporcione a aplicação ter um menor tempo de execução.

4.2.1 Busca Exaustiva

Para analisar o comportamento da aplicação e servir de comparação, foi desenvolvido um método que faz uso da técnica de busca exaustiva para encontrar o melhor valor para o parâmetro de *workload*. Esta técnica faz experimentações com todas as combinações possíveis e ao final apresenta o ótimo encontrado.

A Figura 12 apresenta o comportamento da aplicação quando variamos a sua distribuição da carga de trabalho. O tempo de execução decresce enquanto o *workload* (carga de trabalho destinada a GPU) aumenta, mas ao passar de 80% o tempo começa a subir.

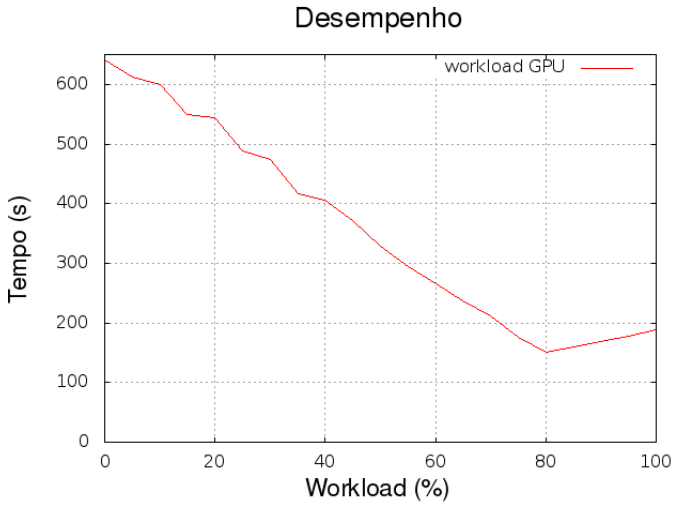


Figura 12 – Desempenho da Espacialização do Relevo ao variar o *workload*

Fonte: Elaborado pelo autor.

Tabela 2 – Resultados da busca exaustiva com precisão de 5%.

Iteração	Workload	Tempo(s)
1	0	641.53
2	5	612.02
3	10	599.98
4	15	549.74
5	20	553.46
6	25	488.12
7	30	474.42
8	35	416.59
9	40	405.34
10	45	372.38
11	50	328.24
12	55	294.93
13	60	266.33
14	65	237.79
15	70	211.65
16	75	175.07
17	**80**	**150.88**
18	85	159.89
19	90	168.39
20	95	177.07
21	100	187.66

Fonte: Elaborado pelo Autor.

A Tabela 2 demonstra os resultados da utilização do método de busca exaustiva. Exigindo uma precisão de 5%, foi necessário realizar vinte e uma combinações para ter o resultado. O parâmetro que teve um menor tempo de execução foi o que destinou 80% do processamento para a GPU, deixando os outros 20 para serem calculados nas CPUs.

Através do método de busca exaustiva é possível encontrar o parâmetro ótimo, o problema é que a depender da quantidade de combinações necessárias o método pode torna-se inviável. Caso fosse exigida uma precisão de 2%, seria necessário realizar 51 execuções, o que tornaria o processo muito demorado. Para diminuir o processo de busca foram desenvolvidas três heurísticas chamadas de: Dicotomia, Seção Áurea e Fibonacci.

4.2.2 Heurística Dicotomia

A busca heurística Dicotomia foi apresentada detalhadamente na Seção 3.2.1 e os seus resultados para uma precisão de 5% estão na Tabela 3. A cada iteração o processo utiliza cinco pontos do intervalo para dividir o espaço de busca em 4 subintervalos. O objetivo é ignorar dois subintervalos que contém os maiores valores e aprofundar a busca nos outros dois subintervalos adjacentes que contém valores menores.

Tabela 3 – Resultados do algoritmo Dicotomia com precisão de 5%.

Iteração	Intervalo	Tempo(s)
1	0	844.00
	25	617.53
	50	404.47
	75	224.02
	100	186.41
2	50	404.47
	62	373.91
	75	224.02
	87	164.05
	100	186.41
3	75	224.02
	81	153.66
	87	164.05
	93	174.52
	100	186.41
4	**81**	**153.66**
	84	158.74
	87	164.05
	90	169.44
	93	174.52

Fonte: Elaborado pelo Autor.

A cada iteração, os cinco pontos equidistantes devem ser utilizados como parâmetro da aplicação paralela e executados no *cluster* para obter-se os tempos. Na primeira iteração os pontos (0 e 25) tiveram um tempo maior, então os subintervalos que eles representam ([0 - 25][25 - 50]) foram ignorados na próxima busca, que se aprofundou no intervalo [50 - 100]. Esta mesma estratégia foi aplicada em todas as iterações, proporcionando que o espaço de busca seja reduzido pela metade a cada passo.

A partir da segunda iteração pode-se notar que dos cinco pontos analisados, três pertencem à iteração anterior. Então não é necessário executá-los para obter os tempos, basta utilizar os valores anteriores. Esta tarefa exige cuidado, pois os valores dos pontos são os mesmos mas não a sua disposição no intervalo. O Algoritmo 2 que implementa essa heurística aplica um controle que permite reutilizar os valores e evitar novas execuções.

Para obter o total de execuções do método heurístico Dicotomia basta somar as 5 execuções da primeira iteração com as 2 execuções de cada uma das 3 iterações seguintes, obtendo um total de 11 execuções. Este valor é quase a metade das execuções realizadas pelo método de busca exaustiva.

4.2.3 Heurística Seção Áurea

A segunda heurística desenvolvida foi a Seção Áurea, nessa técnica são utilizados dois pontos do intervalo para analisar onde será aprofundada a próxima busca. O primeiro ponto equivale ao início do intervalo mais um dx, enquanto o segundo ponto corresponde ao fim menos o dx. O cálculo do dx é realizado envolvendo a razão Áurea e o intervalo de busca, este cálculo, assim como mais detalhes da heurística estão presentes na Seção 3.2.2.

Tabela 4 – Resultados do algoritmo Seção Áurea com precisão de 5%.

Iteração	Intervalo	dx	workload	Tempo(s)
1	0 - 100	61.8	61	291.30
			38	439.67
2	38 - 100	38.1	76	184.17
			61	291.30
3	61 - 100	23.6	85	160.25
			76	184.17
4	76 - 100	14.5	90	169.24
			85	160.25
5	76 - 90	9.0	85	160.25
			81	153.17
6	76 - 85	5.5	81	153.17
			79	162.27
7	79 - 85	3.4	83	156.92
			81	**153.17**

Fonte: Elaborado pelo Autor.

Na Tabela 4 são apresentados os resultados da utilização da heurística Seção Áurea para encontrar o melhor *workload* para o ambiente experimentado. Estes dados foram gerados a partir da utilização do Algoritmo 3 com uma precisão (margem de tolerância) de 5%.

Na iteração 1, os pontos analisados foram 61 e 38. Para serem obtidos os tempos destes pontos, são realizadas execuções no ambiente paralelo utilizando o ponto como o *workload* a ser testado. Nesta primeira iteração o ponto 61 apresentou um menor tempo, então o processo de busca será aprofundado nas suas proximidades, enquanto o subintervalo que vai do início ao ponto 38 será ignorado na próxima iteração por conterem valores maiores.

A partir da segunda iteração é perceptível que sempre há um ponto que foi analisado na etapa anterior. Diante deste fato não necessitamos de duas novas execuções, podemos reutilizar o tempo do ponto que se repete e realizar a execução somente com o novo. Então o total de execuções corresponde a: duas na primeira iteração e uma nas demais, totalizando 8 execuções.

Na Tabela 4 também podemos notar a mudança do intervalo de busca a cada passo, na iteração 7 o processo é encerrado pois o *dx* atinge um valor menor que a *precisão* solicitada, então é retornado o valor ótimo de *workload* que neste caso foi 81.

4.2.4 Heurística Fibonacci

A heurística Fibonacci também utiliza uma técnica de eliminação de região para encontrar o ótimo, mantendo uma simetria baseada na sequência de números inteiros de Fibonacci. Os detalhes desta heurística são explicados na Seção 3.2.3.

Tabela 5 – Resultados do algoritmo Fibonacci com precisão de 5%.

Iteração	Intervalo	dx	workload	Tempo(s)
1	0 - 100	61.9	38	431.60
			61	282.07
2	38 - 100	38.0	61	282.07
			76	171.91
3	61 - 100	23.8	76	171.91
			85	159.72
4	76 - 100	14.2	85	159,72
			90	168.41
5	76 - 90	9.5	80	150.83
			85	159,72
6	**76 - 85**	**4.7**	**80**	**150.83**
			80	**150.83**

Fonte: Elaborado pelo Autor.

A Tabela 5 apresenta os resultados da busca pelo *workload* ótimo com a utilização da heurística Fibonacci. Esta heurística, de forma análoga a Seção Áurea, realiza duas execuções na primeira iteração e uma execução nas seguintes, mas encontra o ótimo em

menos iterações. Além do fato de a última iteração apresentar dois pontos iguais, desta forma, foram necessárias somente 6 execuções para encontrar o ótimo.

4.3 Considerações do capítulo

Neste capítulo foram apresentados os resultados das busca baseadas em heurísticas e da busca exaustiva. A Figura 13 compara a quantidade de execuções que cada método teve que realizar para obter o parâmetro ótimo. É perceptível que as heurísticas apresentaram uma eficiência superior ao método de busca exaustiva.

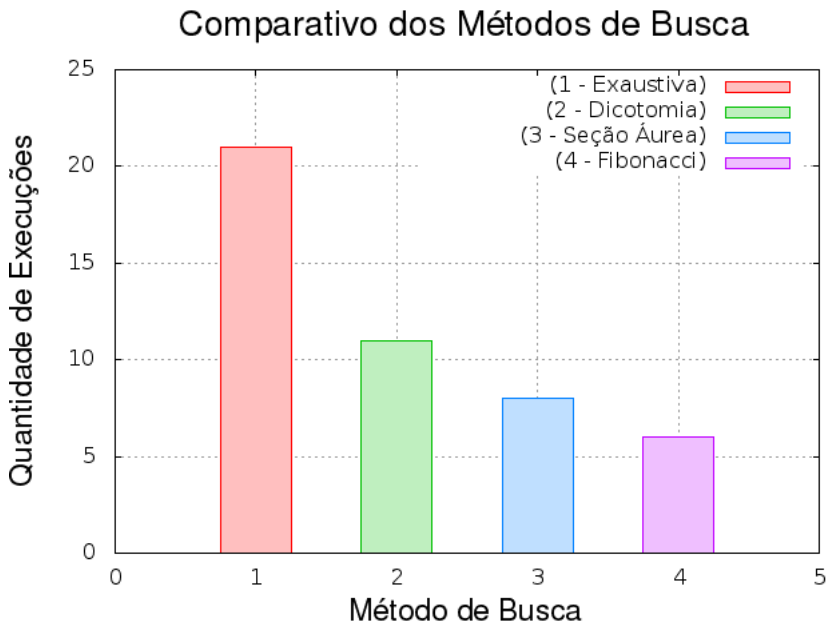


Figura 13 – Comparativo entre a quantidade de execuções dos métodos de busca.

Fonte: Elaborado pelo autor.

Outra maneira de comparar os métodos é através do *speedup*. *Speedup* é uma grandeza adimensional que representa o quanto um algoritmo é mais rápido do que outro. Essa medida é calculada através da Equação 4.1. Nesta equação o resultado (S) representa o quanto o algoritmo 2 é mais rápido do que o 1.

$$S = \frac{tempo_{algoritmo_1}}{tempo_{algoritmo_2}} = \frac{t_1}{t_2}. \tag{4.1}$$

A Figura 14 apresenta um gráfico de *speedup*, para obter o tempo de cada método foram somados os tempos das execuções que foram feitas ao buscar o parâmetro ótimo. Neste gráfico podemos perceber o quanto os métodos heurísticos foram mais rápidos do que a busca exaustiva, com um destaque especial para a heurística Fibonacci que foi 5 vezes mais rápida do que o método de busca exaustiva.

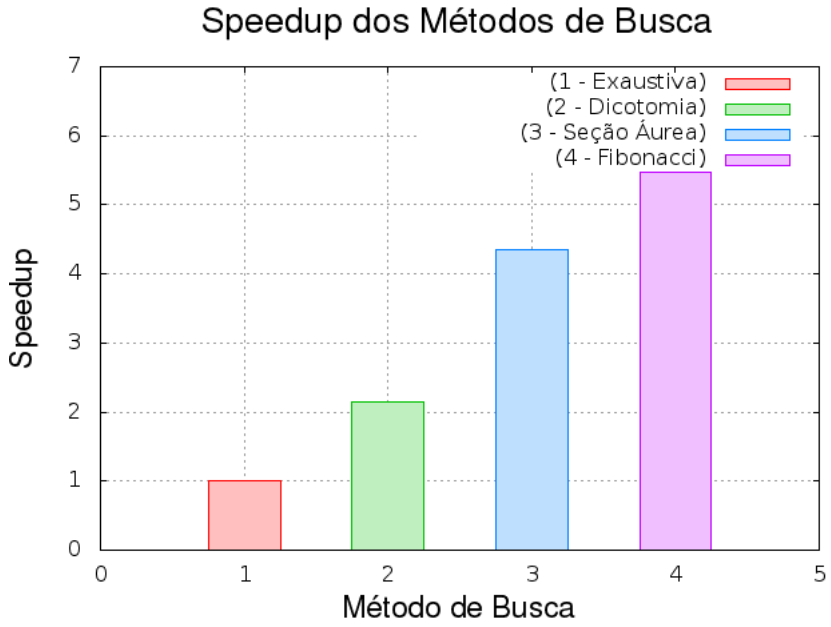


Figura 14 – Speedup dos métodos de busca.

Fonte: Elaborado pelo autor.

5 Considerações Finais

Dentro do cenário da computação de alto desempenho, há aplicações paralelas que utilizam uma abordagem híbrida, integrando CPUs e GPUs no processamento das informações. Para que essas aplicações sejam executadas de maneira eficiente é necessário ajustá-las ao ambiente de execução, o que normalmente é realizado por ferramentas de auto-otimização.

As ferramentas de auto-otimização buscam quais os melhores parâmetros que propiciam à aplicação utilizar o máximo dos recursos disponíveis. Neste processo de busca, são aplicados métodos que exploram as possibilidades existentes. O método mais simples e direto é a busca exaustiva, que testa todas as combinações possíveis. A depender da quantidade de combinações, este método pode tornar a adaptação inviável em relação ao tempo de execução do algoritmo. Uma forma de reduzir o espaço de busca é através do uso de heurísticas.

Nesta pesquisa foram desenvolvidas heurísticas que buscam o melhor parâmetro para uma variável da aplicação. Para fazer uma melhor exploração do espaço de busca, foram utilizadas técnicas de eliminação de região aplicadas por métodos matemáticos de otimização. Estes métodos encontram o ótimo através de reduções criteriosas no intervalo de busca de funções que contém uma variável e um único mínimo no intervalo especificado.

Para validação, as heurísticas foram aplicadas em um estudo de caso chamado Espacialização do Relevo. O objetivo foi encontrar a melhor distribuição da carga de trabalho entre CPUs e GPUs. As heurísticas se mostraram eficientes no processo de busca do parâmetro ótimo e chegaram a ser cinco vezes mais rápidas do que o método de busca exaustiva.

Por fim, para continuidade deste trabalho, sugere-se uma adaptação destas heurísticas para que realizem buscas por parâmetros ótimos de várias variáveis simultaneamente.

Referências

ALMEIDA, A. V. *Uso de auto-tuning para otimizaçãoo de decomposição de domínios paralelos*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2011. Citado 4 vezes nas páginas 16, 17, 18 e 40.

ALOISE, D. et al. Heurísticas de colônia de formigas com path-relinking para o problema de otimização da alocação de sondas de produção terrestre–spt. *XXXIV Simpósio Brasileiro de Pesquisa Operacional*, 2002. Citado na página 17.

AMARAL, P. R. C. *Programação paralela utilizando Cluster Beowulf e o Sistema PVM*. Dissertação (Graduação) — DCET, Universidade do Estado da Bahia, Bahia, 2005. Citado na página 22.

ARAÚJO, T. M. U. *Métodos híbridos baseados em continuos-grasp aplicados à otimização global contínua*. Dissertação (Mestrado) — Universidade Federal da Paraíba, 2009. Citado 2 vezes nas páginas 35 e 44.

BERGAMASHI, F. B. Mini curso: Métodos numéricos para encontrar mínimos e máximos de uma função. *II Colóquio de Matemática, UESB*, 2010. Citado na página 35.

BORATTO, M. *Parametrizacíon en Esquemas Paralelos Divide y Vencerás*. Dissertação (Mestrado) — Universidade Politécnica de Valencia, 2007. Citado na página 22.

BORATTO, M.; COELHO, L.; BARRETO, M. Distributed and parallel computing on multicore and multi-gpu systems. *Simpósio em Sistemas Computacionais (WSCAD-SSC)*, XIII, 2012. Citado 7 vezes nas páginas 17, 20, 24, 37, 38, 39 e 40.

BORATTO, M. et al. *Pesquisas Aplicadas em Modelagem Matemática*. [S.l.]: Biblioteca Universitária Mario Osório Marques, 2012. Citado 2 vezes nas páginas 16 e 37.

CUDA: Site. 2013. Disponível em: <http://www.nvidia.com.br/object-/cuda_home_new_br.html>. Acesso em: 20 out. 2013. Citado 2 vezes nas páginas 16 e 28.

FERNANDO, M. J. K. R. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-wesley longman. [S.l.], 2003. Citado na página 27.

FILHO, J. N. *Espacialização da Temperatura para o Pólo de Desenvolvimento Petrolina/Juazeiro, utilizando Computação de Alto Desempenho*. Dissertação (Graduação) — Universidade Federal do Vale do São Francisco, 2012. Citado 3 vezes nas páginas 17, 28 e 29.

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

GOMES, F. V. *Reconfiguração de Sistemas de Distribuição Utilizando Técnicas de Otimização Contínua e Heurística para Minimização de Custos*. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2005. Citado na página 17.

GRBOVIC, P. J. Master/slave control of input-series-and output-parallel-connected converters: Concept for low-cost high-voltage auxiliary power supplies. *Power Electronics, IEEE Transactions*, v. 24, 2009. Citado na página 22.

INTERDISCIPLINARY Computation and Communication Group (Universidad Politénica de Valencia): Site. 2013. Disponível em: <http://www.inco2.upv.es/>. Acesso em: 25 out. 2013. Citado na página 27.

KOWALTOWSKI, T. Von neumann: suas contribuições à computação. *Estudos Avançados*, SciELO Brasil, v. 10, p. 237–260, 1996. Citado na página 21.

LIMA, D. S. *Estratégia paralela exata para o alinhamento múlltiplo de sequências biológicas utilizando Unidades de Processamento Gráfico (GPU)*. Dissertação (Mestrado) — Universidade de Brasília, 2013. Citado na página 17.

LIPORACE, F. dos S. Programação paralela com gpu aplicada ao registro automático de imagens de satélites. *XVI Simpósio Brasileiro de Sensoriamento Remoto - SBSR, Foz do Iguaçu, PR*, 2013. Citado na página 17.

LV, Y.; CHEN, W. Heterogeneous clustering computing based on parallel task distribution. In: *Network Computing and Information Security*. [S.l.]: Springer, 2012. p. 485–491. Citado na página 20.

MARQUES, E. R. da S. V. F. *Single operation multiple data - paralelismo de dados ao nível da sub-rotina*. Dissertação (Mestrado) — Universidade Nova de Lisboa, 2012. Citado na página 21.

MINOUX, M. *Mathematical Programming*. [S.l.]: Bordas Dunod Gauthier Villars, 1986. Citado 2 vezes nas páginas 33 e 34.

MORAES, S. R. dos S. *Computação paralela em um cluster de GPU aplicado a problema da esgenharia nuclear*. Dissertação (Mestrado) — Instituto de Engenharia Nuclear, 2012. Citado na página 20.

MPI: Site. 2013. Disponível em: <http://www.mcs.anl.gov/research/projects/mpi/>. Acesso em: 23 out. 2013. Citado na página 23.

MSDN, M. *Programming Guide for HLSL (Windows)*: Site. 2013. Disponível em: <http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635(v=vs.85).aspx>. Acesso em: 25 out. 2013. Citado na página 27.

NVIDIA. *Whitepaper NVIDIA Next Generation CUDA Compute Architecture: Fermi*. Santa Clara, CA, USA, 2009. Disponível em: <http://www.nvidia.com.br/content/PDF-/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Citado na página 25.

NVIDIA. *Whitepaper NVIDIA Next Generation CUDA Compute Architecture: Kepler GK110*. Santa Clara, CA, USA, 2012. Disponível em: <http://www.nvidia.com/content-/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. Citado 2 vezes nas páginas 25 e 26.

NVIDIA. *CUDA Programming Guide*: Site. 2013. Disponível em: <http://docs.nvidia-.com/cuda/cuda-c-programming-guide/>. Acesso em: 25 out. 2013. Citado 4 vezes nas páginas 29, 30, 31 e 32.

OPENCL: Site. 2013. Disponível em: <http://www.khronos.org/opencl/>. Acesso em: 25 out. 2013. Citado na página 27.

OPENGL: Site. 2013. Disponível em: <http://www.opengl.org/>. Acesso em: 25 out. 2013. Citado na página 27.

OPENMP: Site. 2013. Disponível em: <http://openmp.org/wp/>. Acesso em: 20 out. 2013. Citado 3 vezes nas páginas 16, 23 e 28.

PEREIRA, F. M. Q. Técnicas de otimização de código para placas de processamento gráfico. *III Escola de Verão em Computação*, 2012. Citado na página 25.

PITANGA, M. *Construindo Supercomputadores com Linux*. 3. ed. Rio de Janeiro: [s.n.], 2008. Citado na página 21.

QUARESMA, P. J. et al. Tomo-gpu: um ambiente de resolução de problemas destinado à análise de dados tomográficos relativos à caracterização estrutural de materiais. *Revista de Ciências da Computação*, v. 5, 2010. Citado na página 16.

RUDY, G. *CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation*. Dissertação (Mestrado) — University of Utah, 2010. Citado na página 17.

SENA, M. R.; COSTA, J. C. Tutorial openmp c/c++. *Programa Campus Ambassador HPC*, 2008. Citado na página 28.

SOUZA, T. T. P. Introdução a computação de alto desempenho utilizando gpu. *Seminário de Programação em GPGPU - Universidade de São Paulo*, 2011. Citado 3 vezes nas páginas 24, 25 e 26.

SPENCER, B. A general auto-tuning framework for software performance optimisation. *Balliol College, University of Oxford, Third Year Project Report*, 2011. Citado na página 17.

VASCONCELLOS, F. B. *Programando Com GPUs: Paralelizando o Método Lattice-Boltzmann Com CUDA*. Dissertação (Graduação) — Universidade Federal do Rio Grande do Sul, 2009. Citado na página 25.

WILLIAMS, S. W. *Auto-tuning Performance on Multicore Computers*. Tese (Doutorado) — EECS Department, University of California, Berkeley, 2008. Citado na página 40.