PARALELIZAÇÃO DO ALGORITMO DE MIGRAÇÃO SÍSMICA EM PLATAFORMAS HETEROGÊNEAS

THIAGO SANTOS FARIA XAVIER TEIXEIRA

PARALELIZAÇÃO DO ALGORITMO DE MIGRAÇÃO SÍSMICA EM PLATAFORMAS HETEROGÊNEAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais - Departamento de Ciência da Computação como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Wagner Meira Jr
Coorientador: Jairo Panetta

Belo Horizonte

05 de novembro de 2010

Teixeira, Thiago Santos Faria Xavier.
T266a Paralelização do algoritmo de migração sísmica em plataformas heterogêneas / Thiago Santos Faria Xavier Teixeira. — Belo Horizonte, 2010
xx, 60 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Orientador: Wagner Meira Jr
Coorientador: Jairo Panetta

1. Computação - Teses. 2. Algoritmos paralelos - Teses. 3. Sistemas distribuídos - Teses. I. Orientador II. Coorientador III. Título.

CDU 519.6*73(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Paralelização do algoritmo de migração sísmica em plataformas heterogêneas

THIAGO SANTOS FARIA XAVIER TEIXEIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. WAGNER MEIRA JÚNIOR - Orientador
Departamento de Ciência da Computação - UFMG

PROF. JAIRO PANETTA - Co-orientador
Departamento de Ciência da Computação - ITA

PROF. SIANG WUN SONG
Instituto de Matemática e Estatística - USP

PROF. DORGIVAL OLAVO GUEDES NETO
Departamento de Ciência da Computação - UFMG

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 05 de novembro de 2010.

Aos meus queridos pais, Nelson e Walkiria, e às minhas queridas irmãs, Maria Tereza e Ana Luiza.

Agradecimentos

Primeiramente, agradeço a Deus por iluminar meu caminho, me dar saúde e permitir que eu alcançasse esse objetivo.

Aos meus pais, Nelson e Walkiria, agradeço muito, muito mesmo. Foram eles que sempre fizeram de tudo para que eu tivesse a melhor educação possível, confiaram no meu potencial em todos os momentos e sempre me deram muita força para conquistar os meus objetivos. Às minhas irmãs, Ana Luiza e Maria Tereza, obrigado pela paciência, apoio e carinho durante todos esses anos. Obrigado à Vovó Tereza pelo carinho e, claro, pelas guloseimas, que sempre me deram mais energia para enfrentar todos os desafios. A todos os meus familiares, obrigado pela torcida.

À Gabriela, agradeço o amor, o carinho e a delicadeza de me ajudar e me apoiar em todos os momentos. Sem a sua força e compreensão nada disso seria possível.

Ao Wagner, a atenção, a amizade e os ensinamentos que foram muito além do universo acadêmico, assim como o apoio e o incentivo para terminar esse trabalho mesmo nos momentos mais incertos e difíceis. Ao Jairo, o apoio, as lições, a atenção e, principalmente, os questionamentos instigantes que fizeram com que eu melhorasse cada vez mais o trabalho. Aos professores Dorgival e Renato, que tanto me ensinaram desde os tempos de iniciação científica no Speed.

Aos colegas do Speed, especialmente George, Macambira, Coutinho e Fernando Henrique, o companheirismo e os momentos divertidos que superaram qualquer tensão e monotonia. Jamais esquecerei os incentivos, a atenção para ouvir as minhas dúvidas e o ótimo trabalho em equipe.

Agradeço a todos os professores e funcionários do DCC/UFMG que me deram totais condições de estudar temas interessantes desde a graduação. Aos colegas de graduação e de mestrado, o ambiente agradável e estimulante para a busca de novos conhecimentos.

Aos colegas da Petrobras, em especial os da Tecnologia Geofísica, que me apoiaram e me ensinaram muitos dos conceitos importantes utilizados nesse trabalho. Obrigado à Petrobras pelo incentivo e à Neiva por permitir a flexibilização do meu

horário, essencial em muitos momentos importantes.

Não poderia deixar de agradecer a todos os meus amigos, principalmente os da *turma do seu cuca*, a sincera amizade e o companheirismo. Sei que eles pouco preocupavam com os meus prazos e queriam diversão a qualquer hora e lugar, mas, mesmo assim, todos foram importantes para eu chegar até aqui.

Resumo

O petróleo é um dos recursos energéticos mais importantes e estratégicos no mundo. A demanda e o preço do barril crescentes viabilizam a produção de óleo e gás em áreas outrora inviáveis. A estratégia mais utilizada para encontrar possíveis reservas é através dos métodos sísmicos, entre os quais a migração sísmica de Kirchhoff é uma das mais populares. Entretanto, ela é computacionalmente intensiva e, por isso, melhorar o seu desempenho tem sido o foco de vários trabalhos, frequentemente através da sua paralelização.

Por outro lado, estamos observando o aparecimento de novas arquiteturas de computação paralela onde cada nó de processamento é organizado como uma hierarquia de vários processadores heterogêneos que podem ser multi-core ou many-core, como as placas gráficas (GPUs), empregadas em razão do seu poder de processamento. A despeito da sua popularidade, essas arquiteturas representam um desafio em termos de projetar programas eficientes computacionalmente.

Nesse trabalho, discutimos a paralelização do algoritmo da migração sísmica de Kirchhoff para um ambiente heterogêneo distribuído composto de CPUs multinúcleo e GPUs. Propomos e avaliamos implementações paralelas que utilizam eficientemente os processadores disponíveis. Exploramos os compromissos entre três dimensões de paralelismo, nominalmente assincronia, paralelismo de dados e paralelismo de tarefas, projetando, implementando e avaliando várias configurações possíveis. Nós também projetamos e implementamos um escalonamento dinâmico de grão mais fino para os dispositivos, possibilitando execuções mais eficientes. Os experimentos apresentaram uma aceleração de até 87 vezes com relação à execução em um único núcleo CPU. Esse resultado é decorrente da utilização eficiente das CPUs e da GPU através das estratégias apresentadas.

Abstract

Oil is one of the most important and strategic energy resources in the world. The increasing oil demand and price now support its production where it was previously unfeasible. The most common strategy to find potential reserves is through seismic methods, among which the Kirchhoff seismic migration is one of the most popular strategies. However, it is computationally intensive and improving its performance has been researched in several works, often through their parallelization.

On the other hand, we are witnessing the emergence of novel parallel computing architectures where each processing node is organized as a hierarchy of several heterogeneous processors that may be multi-core or many-core, such as Graphical Processing Units (GPUs), used as a consequence of their computing power. Such architectures are becoming commonplace and represent a challenge with respect to design computationally efficient programs.

In this thesis, we discuss the parallelization of the Kirchhoff seismic migration algorithm for execution on the aforementioned heterogeneous environment. We propose and evaluate parallel implementations that use efficiently the available processors. We exploited the trade offs among three parallelism opportunities, namely asynchrony, data parallelism and task parallelism, by designing, implementing and evaluating various possible configurations. We also devised and implemented a finer grain dynamic scheduling for the devices, supporting executions that present higher efficiency. The experiments showed an acceleration of 87 times compared to execution on a single CPU core. This was due to CPU and the GPU efficient use through the approaches presented.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O petróleo é considerado um dos mais valiosos recursos globais e a sua indústria é uma das mais importantes do mundo. O também chamado "ouro negro" é a principal matéria-prima energética e industrial do planeta e tornou-se estratégico para várias nações. Esse recurso não renovável é a única fonte de renda de muitos países e já foi causa de várias guerras [Falola & Genova, 2005].

Considerado a fonte energética mais flexível disponível atualmente, ele pode ser usado para a geração de eletricidade, fonte de calor e combustível para transporte, além de estar presente em múltiplos produtos de uso diário, como nas roupas, nos plásticos, nas borrachas, nos fertilizantes agrícolas e em muitos outros.

Desde o século XIX, a demanda por óleo não para de crescer, os países produtores alcançaram reconhecimento internacional devido ao seu valioso recurso e tornaram-se destaque pela riqueza que acumularam ao longo dos anos.

A formação do petróleo vem da deposição, no fundo de lagos e mares, de restos de animais e vegetais mortos ao longo de milhares de anos. Estes restos foram cobertos por sedimentos, que mais tarde se transformaram em rochas. A ação do calor e da alta pressão provocados pelo empilhamento dessas rochas resultaram em reações complexas, gerando petróleo. As formações rochosas de onde ele é retirado são chamadas de reservatórios. Os reservatórios não são uma cavidade gigante parecida com uma caverna preenchida com óleo. Na verdade, eles se parecem mais com uma esponja que absorve óleo e gás, onde ficam armazenados em minúsculos poros. A partir de forças tectônicas e da pressão, o petróleo se move através dos poros das rochas, geralmente em direção à superfície. Essa movimentação pode fazer com que o petróleo se acumule em grandes quantidades, particularmente quando encontra uma rocha que o aprisione, evitando que ele escape. Essas rochas selantes podem variar em tamanho e forma. Caso a migração do petróleo não seja interrompida por formações geológicas, ela continua até

alcançar a superfície.

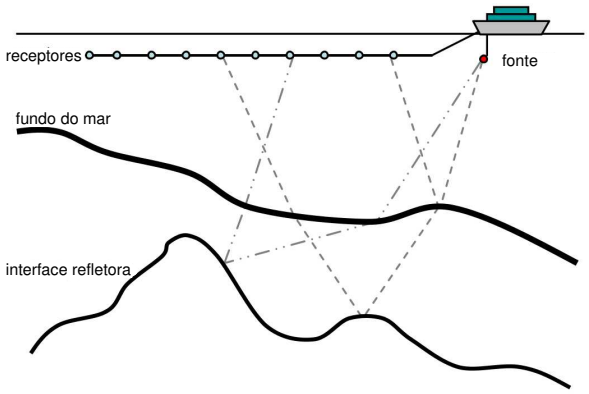
A indústria do petróleo investe bilhões de dólares para encontrar reservatórios. Como o petróleo é finito, a procura por novas reservas tem de ser contínua na medida em que as atuais estão em produção. Quanto mais reservas uma empresa possui, mais tempo de vida ela terá e será melhor avaliada pelo mercado. Além disso, a demanda e o preço do barril crescentes tornam viáveis a exploração de áreas outrora inviáveis, por muito caras.

Em geral, os objetivos da exploração consistem no mapeamento das estruturas geológicas da subsuperfície, na detecção da acumulação de hidrocarbonetos e na estimativa do tamanho das reservas em uma área. A procura por novas reservas não é uma tarefa fácil e possui uma taxa de acerto muito pequena. A perfuração de um único poço custa milhões de dólares e somente através da perfuração é que se tem certeza da existência de petróleo.

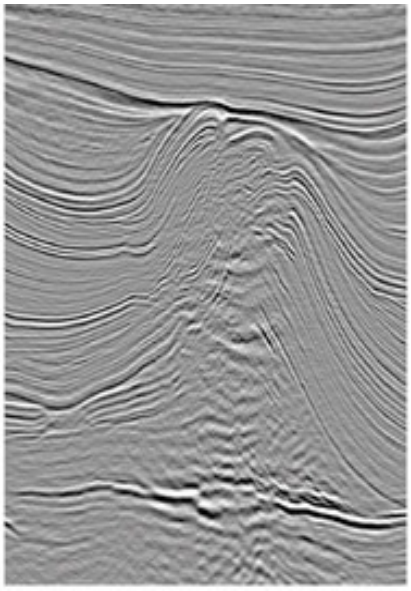
Para localizar prováveis reservas de óleo e gás e os melhores locais para perfurar utilizam-se os métodos geofísicos, sendo o método sísmico o mais importante deles. A maioria das explorações em petróleo e gás é feita usando estudos relacionados à reflexão dos sinais sísmicos (sismologia da reflexão), através dos chamados métodos sísmicos [Robinson, 1977].

A sismologia da reflexão é um método para mapear a estrutura da subsuperfície da terra. Uma fonte é detonada na superfície. Ela pode ser por explosão de dinamite, caminhão impactador, canhão de ar comprimido ou qualquer outra maneira de transmitir energia sísmica para o solo. A figura 1.1a apresenta a aquisição sísmica no mar. As ondas sísmicas viajam da fonte até as camadas da subsuperfície no interior da terra onde são refletidas. As ondas refletidas são então gravadas por instrumentos na superfície. A estrutura da subsuperfície é desenhada a partir do conhecimento do tempo de trânsito do sinal sísmico da fonte até o receptor. As camadas sedimentares da terra tendem a ser aproximadamente horizontais, mas podem apresentar dobras, inconformidades e falhas que podem servir de armadilhas para aprisionar o petróleo e o gás.

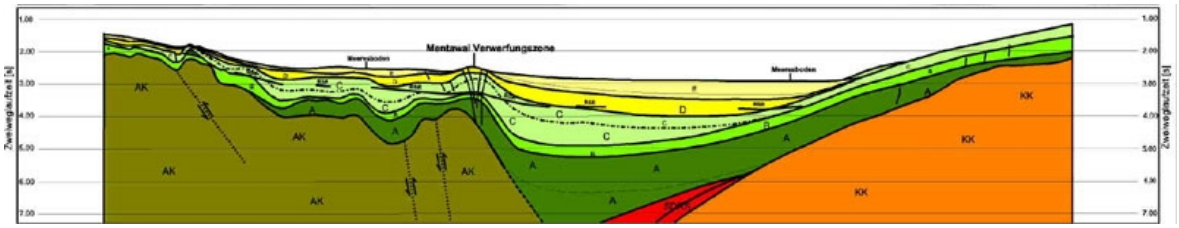
No intuito de mapear o interior da terra, o geofísico deve converter os registros sísmicos recebidos no qual os eventos foram registrados em função do tempo para registros com eventos em função da profundidade. Em outras palavras, a função em tempo registrada deve ser convertida para uma função em profundidade. As ondas sísmicas possuem velocidade altamente dependente do meio, ou seja, a velocidade das ondas sísmicas muda na medida em que viaja pelo interior da terra. Geralmente, a velocidade aumenta com a profundidade, embora existam camadas em que a velocidade possa diminuir. Dado um ponto na superfície, a velocidade encontrada em função



(a) Aquisição sísmica marítima



(b) Resultado do processamento sísmico



(c) Resultado da interpretação

Figura 1.1: Método Sísmico

da profundidade é chamada de função velocidade. Existem, portanto, duas variáveis importantes, a velocidade e o tempo de reflexão dos eventos. Com o conhecimento dessas variáveis é possível determinar a profundidade dos horizontes refletidos. No entanto, como a função velocidade varia de uma posição geográfica para outra, ela não pode ser considerada válida para todo uma área e deve ser corrigida para cada ponto local de uma região explorada.

A partir de levantamentos sísmicos de uma área são construídas imagens tridimensionais da geologia do subsolo dessa área. Enormes quantidades de dados sísmicos são registrados e processados no dia-a-dia da indústria do petróleo. Cada levantamento contém terabytes de dados que são impossíveis de serem processados sem a utilização de máquinas potentes. Após o processamento, os dados são interpretados por especialistas que decidem onde serão feitos testes de perfuração. Como exemplo, na figura 1.1b temos o resultado de um processamento sísmico e na figura 1.1c uma imagem sísmica

interpretada.

Assim, o método sísmico pode ser dividido em três etapas:

- Aquisição dos dados
- Processamento dos dados
- Interpretação dos dados

O foco deste trabalho está na etapa de processamento dos dados sísmicos, que normalmente está dividida em duas fases. A primeira corresponde ao processamento dos dados coletados para normalizar em todo o levantamento o sinal sísmico e melhorar a razão sinal/ruído. Existem centenas de procedimentos matemáticos para essa fase, para a qual especialistas em geofísica escolhem os mais apropriados para os dados em questão. A segunda fase é a mais demorada e tem como finalidade corrigir os efeitos da mudança da velocidade nos meios da subsuperfície durante a propagação da onda pelo interior da terra, produzindo imagens acuradas da subsuperfície. A Migração de Kirchhoff em Tempo (KTM) é um dos métodos mais utilizados na indústria para essa fase de imageamento dos dados sísmicos. Cada uma dessas fases necessita de um processo iterativo para melhorar as imagens geradas resultantes de complicadas estruturas do subsolo.

Os principais atributos do processamento de dados sísmicos são o tamanho dos dados e a complexidade computacional dos algoritmos. Um levantamento sísmico 3D geralmente resulta em terabytes de dados e exige enormes recursos computacionais. Além disso, a exigência por dados mais complexos tem aumentado na medida em que a exploração tem sido feita em áreas mais complicadas e profundas. Consequentemente, os algoritmos também se tornaram mais complexos e a demanda computacional cresceu vertiginosamente. O processamento de tais conjuntos de dados e a concorrência acirrada entre as empresas tem gerado uma grande demanda por processamento de alto desempenho na indústria do petróleo.

Inicialmente, apenas os mainframes e os supercomputadores vetoriais eram utilizados para o processamento sísmico devido a sua grande capacidade e contínuo avanço tecnológico. Entretanto, os custos de um supercomputador desses eram muito altos, o que dificultava o acesso dos centros de processamento a esse tipo de máquina. Ao mesmo tempo, os processadores de baixo custo utilizados em computadores pessoais também avançavam muito tecnologicamente, mas ainda assim não eram suficientemente potentes para tarefas computacionalmente intensivas. Então, surgiu a ideia de utilizar vários computadores pessoais interconectados por uma rede e funcionando como se fos-

sem uma única máquina, os aglomerados[1]. Através dessa abordagem a computação paralela se tornou mais acessível por ter menor custo que os mainframes e ser mais escalável na medida em que novos computadores podem ser adicionados sem grandes mudanças. Além disso, melhorias ocorridas ao longo dos anos nos processadores e nas redes de interconexão fizeram com que os aglomerados se tornassem a arquitetura dominante nos atuais centros de processamento de alto desempenho.

Essa mudança para arquitetura de memória distribuída, como os aglomerados, trouxe também vários desafios, uma vez que a sua programação é mais difícil. O programador precisa cuidar explicitamente da divisão de tarefas entre os nós e dos mecanismos de sincronização entre os processadores. A paralelização de algoritmos para processamento sísmico em aglomerados tem recebido bastante ênfase e apresentado bons resultados [Abdelkhalek et al., 2009; Panetta et al., 2007; Almasi et al., 1992].

O desempenho dos aglomerados aumenta na medida em que o desempenho dos nós melhora e o número de componentes dos processadores desses nós aumenta a cada ano, seguindo a constatação de Moore [1965]. Tradicionalmente, o projeto dos processadores sempre focou no aumento do desempenho da execução sequencial, explorando ao máximo o paralelismo no nível de instrução (ILP) em conjunto com o aumento da frequência de funcionamento dos processadores. Complexas técnicas como predição de desvios, emissão de múltiplas instruções, especulação e execução fora de ordem são conduzidas a taxas de *clock* extremamente altas, resultando em aumento na complexidade do projeto e excessiva dissipação de energia e calor. No entanto, como resultado do baixo ILP das cargas de trabalho típicas, essas técnicas bastante utilizadas atingiram um ponto em que rapidamente reduziram os ganhos de desempenho. Grandes cargas de trabalho e baixa localidade de referência levaram a uma alta taxa de falha em cache, enquanto a dependência de dados entre instruções prejudica a predição de desvios e a leitura alinhada da memória resultando na execução de várias instruções, desnecessariamente desperdiçando energia. Dessa maneira, o tempo de espera pela memória tornou-se crítico como gargalo para o desempenho do sistema, com os processadores mais modernos gastando por volta de 85% dos ciclos esperando pelos dados vindo da memória [Wulf & McKee, 1994; Wilkes, 2001].

Esses desafios exigiram uma mudança radical no projeto dos microprocessadores para que superassem as barreiras de desempenho, de consumo de energia e dissipação de calor. Na medida em que o tamanho físico dos componentes dos processadores diminuía e o número de transistores por área aumentava, a solução encontrada foi explorar o paralelismo e o compartilhamento de recursos através dos multiprocessadores, ou seja,

[1] *clusters*.

vários processadores por *chip*, com o foco maior em processar mais tarefas por tempo do que acelerar o processamento de uma única tarefa. O número de processadores por *chip* tem aumentado e segue essa tendência a cada nova geração da tecnologia de semicondutores, ou seja, a cada 18/24 meses. Além disso, as novas arquiteturas tendem a projetos mais simples e de baixa frequência, capazes de alcançar níveis sem precedentes de desempenho por área e por watt [Knight, 2005; Horowitz & Dally, 2004].

No passado, os programadores confiavam nos avanços do hardware, da arquitetura e dos compiladores para aumentar o desempenho a cada 18 meses sem ter que alterar nenhuma linha de código das suas aplicações. Atualmente, para os programadores aumentarem o desempenho significativamente, eles precisam reescrever seus programas para obter as vantagens dos múltiplos processadores. Mais além, para garantir essa melhora de desempenho histórica, eles terão de modificar seus códigos a cada vez que o número de multiprocessadores dobrar [Patterson & Hennessy, 2008].

Os multiprocessadores se tornaram o foco da indústria e surgiram diversas implementações como os multiprocessadores simétricos (SMP) Core 2 da Intel e Athlon X2 da AMD [Peng et al., 2007], o orientado a threads UltraSparc da Sun [Leon & Sheahan, 2006] e o multinúcleo heterogêneo STI Cell [Kahle et al., 2005]. Atualmente, todos esses multiprocessadores são amplamente utilizados e servem de base para os maiores supercomputadores do mundo. Um outro tipo de multiprocessador tem recebido grande atenção dos pesquisadores em computação de alto desempenho: as unidades gráficas de processamento (GPU). Essa atenção ocorreu devido ao baixo custo por *flop/s*[2] nominal, ao grande poder computacional e às melhorias nos ambientes de programação desses aceleradores. Esses dispositivos são altamente paralelos, compostos de centenas de processadores programáveis e grande largura de banda para a memória. Além disso, as GPUs podem ser adicionadas facilmente nas arquiteturas atuais [Harris, 2008].

Originalmente, a GPU foi direcionada para os jogos e para as aplicações gráficas, que são aplicações naturalmente paralelas. Com o passar do tempo, as GPUs têm se desenvolvido gradualmente de um específico e limitado controlador gráfico para um processador paralelo programável. Essa evolução se deu pela mudança lógica no *pipeline* gráfico para incorporar elementos programáveis em conjunto com uma implementação em hardware menos especializada e mais programável. Por fim, acabou-se por unificar diferentes elementos programáveis do *pipeline* em hardware em um único conjunto de vários processadores genéricos. Nas GPUs atuais todo o processamento da geometria, pixels e vértices das imagens acontecem num mesmo tipo de processador. Essa unificação simplificou o projeto da arquitetura e permitiu enorme escalabilidade,

[2] Operações de ponto flutuante por segundo.

na qual o aumento da quantidade de processadores programáveis melhora, na mesma proporção, a capacidade de processar mais tarefas. Além disso, tornou a GPU mais acessível para a programação de aplicações não-gráficas, pois eliminou a necessidade de utilizar as abstrações de processamento gráfico para programar a GPU. A unificação dos processadores também simplificou o balanceamento de carga, pois qualquer estágio do processamento pode se beneficiar de qualquer processador.

A utilização de GPUs cria um ambiente heterogêneo em cada nó do aglomerado, na medida em que a GPU precisa da intermediação de uma CPU para funcionar. Com o número de processadores por *chip* da CPU aumentando, essa plataforma heterogênea pode ser considerada um aglomerado de multiprocessadores assimétricos, sendo um desafio utilizar eficientemente os recursos disponíveis. A fim de se beneficiar de todo esse poder de processamento, as aplicações deverão distribuir as tarefas de maneira balanceada. Isso leva a criação de novos métodos e estratégias para distribuir a carga da aplicação para executar nas unidades de processamento específicas e com isso atingir o máximo de desempenho possível sem perder a flexibilidade.

A tendência do desenvolvimento de novas arquiteturas segue aumentando os recursos disponíveis para a computação de alto desempenho, através da adição de novos núcleos. Assim, o desempenho global dos novos multiprocessadores está aumentando drasticamente, mesmo sem poder melhorar o desempenho individual dos processadores. O poder de processamento adicional fornecido tanto pelos multiprocessadores CPU quanto pelas GPUs programáveis não poderá mais ser alcançado apenas recompilando a aplicação, sendo necessárias mudanças tanto nos algoritmos quanto na arquitetura de software.

A paralelização eficiente e escalável da migração sísmica possui grande importância para a indústria do petróleo, uma vez que, quanto mais rápido os resultados forem obtidos, mais regiões podem ser exploradas na busca por óleo e gás. Assim, se torna evidente o interesse e o desafio do ponto de vista acadêmico de estudar e fornecer soluções para indústria sobre como explorar de maneira eficiente essas novas arquiteturas.

1.1 Motivações

A primeira motivação para este trabalho é a disponibilidade de novas arquiteturas paralelas, pois os esforços para adicionar cada vez mais núcleos estão criando arquiteturas diversificadas, seja em multiprocessadores ou em aceleradores. Como exemplo, temos as unidades de processamento gráfico (GPU) altamente paralelas, que rapidamente ganharam maturidade como um poderoso mecanismo para aplicações de alta demanda

computacional.

O surgimento de novas arquiteturas paralelas está transformando cada nó de processamento em uma máquina paralela de memória compartilhada com os mesmos desafios e desempenho de máquinas super avançadas de alguns anos atrás. As máquinas se transformaram em ambientes hierárquicos e heterogêneos, onde cada nó pode ter múltiplos processadores diferentes e cada processador pode ter múltiplos núcleos diversificados. As plataformas distribuídas originadas da união de várias dessas máquinas também podem ser heterogêneas, uma vez que os nós podem ser diferentes.

A segunda motivação é a utilização eficiente dessas novas arquiteturas, em que os desenvolvedores precisarão se preocupar com a escalabilidade e levar em conta as características de cada arquitetura. O conhecimento das características de cada arquitetura será necessário para obter o máximo de desempenho em cada caso. Já a escalabilidade implica em preparar a aplicação para a inclusão futura de mais processadores, sem a necessidade de reescrever todo o código. O ideal é utilizar uma abordagem versátil que possa se adaptar a cada configuração.

Um fator crítico para a utilização eficiente dessas arquiteturas é a alocação de tarefas entre as unidades de processamento. Além disso, é muito complicado implementar uma nova estratégia de alocação de tarefas para cada possível configuração em plataformas multiaceleradas, multiprocessadas, multinúcleo e heterogêneas. Assim, uma abordagem de alocação de tarefas que possa se adaptar para vários tipos de nós torna-se bastante interessante.

Por fim, a terceira motivação é a relevância da migração sísmica, que é muito utilizada e, no processamento dos dados sísmicos, é um dos algoritmos de maior demanda computacional. Assim, os centros de processamento sísmico têm investido muitos recursos financeiros nessas novas arquiteturas com o objetivo de acelerar os resultados. Grande parte desse volume de investimento tem se concentrado na compra e utilização de ambientes heterogêneos compostos de multiprocessadores e múltiplas GPUs. Dessa maneira, a utilização eficiente e flexível de todos os recursos desses ambientes é um desafio muito interessante.

1.2 Objetivos

O objetivo desse trabalho é estudar, implementar e avaliar estratégias de paralelização da migração sísmica em ambientes heterogêneos para obter o máximo de eficiência e desempenho na utilização dos recursos computacionais disponíveis.

1.3 Contribuições

A principal contribuição desse trabalho foi a paralelização do algoritmo da migração sísmica de Kirchhoff para um ambiente heterogêneo distribuído composto de CPUs multinúcleo e GPUs. Apesar de as CPUs e as GPUs terem sido o alvo da implementação, ela pode ser facilmente generalizada para outros dispositivos e ambientes com características semelhantes. A implementação foi feita de maneira escalável para permitir a utilização concorrente de vários multiprocessadores e múltiplos aceleradores diferentes.

Três dimensões de paralelismo foram desenvolvidas para atingir os objetivos mencionados anteriormente. Essas três dimensões utilizaram assincronia, paralelismo de dados e paralelismo de tarefas para alcançar uma significativa melhora no desempenho.

Outra contribuição importante foi a implementação de um escalonamento dinâmico de tarefas de grão mais fino entre as unidades de processamento. Ao contrário de outras abordagens anteriores, essa contribuição permitiu que não houvesse desbalanceamento de carga entre as múltiplas unidades heterogêneas e que todas elas operassem na máxima capacidade de processamento.

Por fim, foi publicado um artigo [Panetta et al., 2009] com resultados do trabalho de pesquisa e desenvolvimento da migração sísmica em ambientes heterogêneos. Esse mesmo artigo foi convidado para ser estendido para a revista *International Journal of Parallel Programming* e o texto submetido foi aceito para publicação.

1.4 Organização

Este texto está organizado da seguinte forma: o capítulo 2 apresenta os principais trabalhos relacionados à paralelização de algoritmos de processamento sísmico. No capítulo seguinte (3) é apresentado o algoritmo da migração sísmica. No capítulo 4 são apresentadas as estratégias utilizadas para a paralelização da migração sísmica para múltiplas unidades de execução heterogêneas. A avaliação da abordagem implementada é apresentada no capítulo 5. Por fim, no capítulo 6 são apresentadas as conclusões e os possíveis trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Neste capítulo discutimos os trabalhos relacionados, que foram divididos em duas seções para melhor apresentação. A seção 2.1 apresenta trabalhos sobre a paralelização dos algoritmos de migração sísmica para diversas arquiteturas. A seção seguinte (2.2) apresenta alguns ambientes de suporte à execução distribuída, incluindo o Anthill [Ferreira et al., 2005], que foi utilizado nesse trabalho.

2.1 Paralelização da migração sísmica

A migração sísmica vem sendo discutida há bastante tempo pela comunidade científica e há diversas abordagens. Por ser um algoritmo bastante intensivo em termos de processamento, a sua paralelização também foi tema de vários trabalhos, como mostrado a seguir.

A utilização de aglomerados de computadores se tornou o foco principal dos centros de processamento sísmico e as dificuldades encontradas para a programação desses ambientes de memória distribuída foram abordadas em vários trabalhos, como em Madisetti & Messerschmitt [1991]; Panetta et al. [2007]; Zhao et al. [2008]; Danek [2009]; Dai [2005].

O artigo de Zhao et al. [2008] apresenta o *Distributed Execution Control Framework* (DECF) para a migração sísmica, que foi desenvolvido para a companhia chinesa de petróleo. Os requisitos para a construção desse arcabouço de execução distribuída foram:

- facilidade de programação por parte dos geofísicos, que consideram a programação distribuída e paralela muito complicada;

- compatibilidade com programas sequenciais, pois uma grande quantidade de módulos são escritos em linguagens de programação sequenciais, como C e Fortran;
- apropriado para aplicação em aglomerados que utilizam multiprocessadores homogêneos e heterogêneos e aceleradores, com base na percepção que o processamento sísmico deve adotar essas novas arquiteturas para ganhar maior poder de processamento.

O DECF escalona os módulos de processamento escritos em linguagem sequencial para executarem simultaneamente em múltiplos processadores. Ele deixa o escalonamento dos recursos computacionais transparente para os usuários, possui tolerância a falhas tanto de máquinas quanto da rede de interconexão e faz a transmissão de dados entre os módulos de processamento.

O DECF apresentou um speedup bem próximo do ótimo a partir da sobreposição entre comunicação e processamento e utilização de múltiplas *threads* por processo. Além disso, o algoritmo se apresentou bastante escalável e para trabalhos futuros os autores pensam em utilizar aceleradores como a GPU.

No trabalho de Panetta et al. [2007] é apresentada a implementação do algoritmo da migração de Kirchhoff utilizada no ambiente de produção da Petrobras. Os seus objetivos são apresentar uma importante aplicação que demanda processamento de alto desempenho, o seu desenvolvimento e suas características de utilização na produção, assim como as suas características computacionais e desempenho. Além disso, a adequação da aplicação em novas arquiteturas é explorada pela preocupação com o consumo e dissipação de energia.

A utilização da migração sísmica na Petrobras tem sua importância desde a década de 70. No início, eram utilizadas mainframes e aceleradores vetoriais para o processamento e os dados eram compactados em 100 vezes para reduzir a demanda de processamento e se adequar à capacidade das máquinas disponíveis. Nos anos 90, foram adquiridas máquinas com 32 processadores RISC e memória compartilhada, o que, a partir da paralelização utilizando OpenMP, fez com que o tempo de execução reduzisse bastante. A partir de 1998, a utilização de aglomerados de computadores já era bastante significativa na indústria do petróleo. Em 1999, foi adquirido um aglomerado de 72 processadores e o algoritmo foi adaptado para memória distribuída, o que resultou em uma segunda versão implementada em MPI. Desde então, a utilização de aglomerados não parou de crescer alcançando em 2006 um total de 5.000 processadores disponíveis para a produção.

Nesse mesmo trabalho são apresentadas avaliações e otimizações feitas, como a vetorização das operações de ponto flutuante e reuso da cache L2 no laço mais interno

da aplicação para melhorar ainda mais o desempenho. De acordo com os autores, a migração sísmica possui uma alta "intensidade computacional", definida como o número operações de ponto flutuante por referência à memória.

Foram apresentados resultados para as arquiteturas IBM Blue Gene [Moreira et al., 2007], Intel x86 e STI Cell [Kahle et al., 2005]. A implementação é portátil, eficiente e escalável. Sua escalabilidade foi testada para até 8192 processadores no Blue Gene. A implementação no x86 de 4 núcleos e no Cell obtiveram ganhos expressivos de preço por desempenho e desempenho por watt com relação a máquinas x86 de um único núcleo, demostrando que o processamento sísmico pode se beneficiar de abordagens que utilizam o paralelismo multinível.

Em Almasi et al. [1992] é relatado um ganho significativo de desempenho para a migração sísmica em um conjunto interconectado de estações de trabalho IBM RISC/6000. Com o aumento do número de nós os ganhos foram limitados pela rede de interconexão, portanto também foram apresentados resultados preliminares sobre o uso de conexão através de fibra-ótica. Além disso, foi proposto um modelo teórico que permite a previsão dos ganhos de desempenho a partir do conhecimento da razão entre computação e comunicação, que pode ser determinada empiricamente antes de o programa ser paralelizado.

O trabalho de Dai [2005] apresenta uma análise sobre a paralelização do algoritmo da migração de Kirchhoff em tempo para um aglomerado de máquinas "Beowulf". A implementação da paralelização foi feita utilizando MPI através da abordagem mestre/escravo com a atribuição de tarefas feita de acordo com a demanda dos escravos. Essa abordagem alcançou um melhor balanceamento de carga entre os nós do aglomerado. Além disso, analisou a granularidade das tarefas e conseguiu um ganho de desempenho próximo do ótimo a partir dos resultados dessa análise.

O artigo de Madisetti & Messerschmitt [1991] apresenta um estudo sobre os algoritmos de migração sísmica com o objetivo de definir um arcabouço geral para analisá-los em vários paradigmas de computação paralela. As exigências computacionais e de comunicação desses algoritmos foram discutidas e diversas técnicas de otimização foram propostas. A limitação de memória, gargalos de entrada-e-saída e os compromissos computacionais em um multiprocessador hipercubo também foram analisados.

Além dos aglomerados de computadores, a utilização de aceleradores como a GPU [Panetta et al., 2009; Abdelkhalek et al., 2009; Danek, 2009] e a FPGA [He et al., 2004; Fu et al., 2009] estão atraindo bastante atenção.

No trabalho de Abdelkhalek et al. [2009] é discutido como a modelagem sísmica e um outro algoritmo de migração, a *Migração Reversa no Tempo* (RTM), podem se beneficiar da GPU para alcançar alto desempenho.

Os autores precisaram fazer diversas otimizações para que a utilização da GPU fosse a melhor possível. O acesso à memória global estava penalizando o desempenho da aplicação, o que motivou a utilização da memória compartilhada da GPU, que é mais rápida, mas menor. A utilização da memória compartilhada e o número de registradores limitou o número de threads utilizadas, o que fez com que o algoritmo fosse dividido em duas etapas. Essa divisão possibilitou a utilização de mais threads o que resultou em 100% de ocupação da GPU.

Os resultados alcançados pela GPU apresentaram ganhos de 10 vezes para RTM e 30 vezes para a modelagem sísmica quando comparados com o desempenho da CPU, justificando a utilização das GPUs em grande escala na indústria, especialmente quando a precisão dupla não é necessária. Para a CPU, os autores indicaram um gargalo no acesso a memória da série Intel Xeon 5400 (veja a figura 4.1b), o que prejudica a escalabilidade na utilização de mais núcleos por multiprocessador.

Outro trabalho relacionado é o de Panetta et al. [2009], onde é apresentado o desempenho da migração sísmica de Kirchhoff em tempo para um ambiente heterogêneo composto de 64 GPUs e 256 cores de CPU. A cooperação entre CPU e GPU é o foco do trabalho ao invés de colocar todo o processamento apenas na GPU. A ociosidade da GPU foi eliminada através de múltiplas requisições de núcleos de CPU. Os resultados apresentados demonstram que o desempenho do uso cooperativo de CPU e GPU reduz o tempo de execução por um fator de 59 e foi a melhor alternativa comparado ao uso só de CPU ou só de GPU.

Danek [2009] apresenta a modelagem do campo de ondas sísmicas utilizando um aglomerado com sistema operacional Linux e GPUs. A modelagem do campo de ondas tem sido estudada desde a década de 70, no entanto, os recursos computacionais disponíveis não permitiram a execução de modelos complexos. Inclusive nas máquinas atuais um único modelo pode levar horas se executado sequencialmente. Entretanto, a modelagem do campo de ondas pode ser facilmente paralelizada e se encaixa muito bem no paradigma utilizado pela GPU. O estudo consiste na solução por diferenças finitas de equações de ondas elásticas e acústicas aplicadas a duas condições de borda.

As implementações feitas pelos autores utilizaram OpenGL ao invés de CUDA, pois CUDA não suportava todas as GPUs testadas. Problemas como gargalo de comunicação entre CPU e GPU foram levados em conta, além da preocupação com os desvios condicionais no código implementado para GPU. No pior caso, o desempenho da GPU chegou a ser 3 vezes melhor que o da CPU utilizada. Os autores ainda consideram o fato de utilizar a cooperação entre CPU e GPU dado que as tarefas de grão fino, em que o tempo de transferência dos dados para a memória da GPU é maior que o de processamento, não são adequadas para execução na GPU. Para eles, a melhor

solução seria uma implementação mista, ou seja, que utiliza CPU e GPU de maneira a não desperdiçar ciclos nas unidades de processamento.

Comparado aos microprocessadores convencionais, as FPGAs são uma arquitetura computacional diferente baseada em fluxos. A maioria das plataformas com coprocessadores reconfiguráveis de propósito geral propostas nos últimos anos foram focadas no processamento de sinal em tempo real orientado a fluxos de entrada e saída. Assim como as GPUs, as FPGAs são facilmente anexadas às arquiteturas atuais e podem ser substituídas por outras melhores no futuro sem modificações para a máquina hospedeira.

O grande obstáculo do processamento de dados sísmicos é o grande volume de dados em conjunto com complexos algoritmos de processamento. O artigo de He et al. [2004] apresenta uma nova plataforma baseada em FPGA para acelerar o algoritmo da migração sísmica. Ao invés de utilizar um sistema computacional comum, foi projetado um novo dispositivo que é anexado às estações de trabalho convencionais. O núcleo do algoritmo, que consome mais de 90% do tempo de execução, foi ajustado cuidadosamente para maximizar a aceleração na plataforma SPACE proposta. As operações de gerenciamento de entrada e saída, de processos e da comunicação foram deixadas para a CPU.

Os autores projetaram um módulo aritmético (AM) especial dentro da FPGA para calcular o tempo de trânsito e fazer as operações de adição. Para atingir uma alta taxa de dados processados por segundo foi implementado um *pipeline* entre os estágios. Dentro de uma FPGA vários AMs podem ser utilizados para processar simultaneamente conjuntos de dados distintos. Além disso, várias FPGAs podem ser colocadas em um única máquina, aumentando significativamente o poder computacional.

A grande motivação de utilizar recursos computacionais reconfiguráveis para o processamento de dados sísmicos é que o mesmo recurso pode ser reconfigurado de maneira otimizada para diferentes algoritmos utilizados em diferentes estágios do processamento. As simulações de desempenho mostraram que o algoritmo de migração na plataforma SPACE foi por volta de 10 vezes mais rápida do que na CPU utilizada sem perder precisão. Como trabalhos futuros os autores pretendem utilizar a plataforma com FPGAs na migração em profundidade e em outros métodos de migração.

Além da capacidade de desempenhar computações de maneira paralela, as FPGAs também suportam diferentes representações numéricas de acordo com a necessidade da aplicação. Como todas as unidades de processamento e conexões na FPGA são reconfiguráveis, é possível utilizar diferentes representações numéricas, que levam a diferentes complexidades para as unidades aritméticas e, consequentemente, para os custos e o desempenho do circuito resultante. A utilização de uma representação

numérica específica pode resultar em ganhos significativos de desempenho ou redução de custos no hardware utilizado. Existe um compromisso claro entre a precisão da representação numérica utilizada e a velocidade de processamento [Deschamps et al., 2006].

O trabalho de Fu et al. [2009] implementou uma ferramenta que faz um estudo automático sobre o uso de diferentes formatos numéricos e encontra a menor precisão que gera resultados sísmicos bons o suficiente para interpretação. Com base na precisão encontrada, foram implementados em FPGA dois algoritmos importantes para o processamento sísmico. Os resultados encontrados apontam para a aceleração de 5 a 7 vezes incluindo o tempo de transferência de dados entre a FPGA e a CPU. Além disso, os autores mostram que colocando 7 núcleos paralelos na FPGA a aceleração pode alcançar até 48 vezes.

2.2 Ambientes de suporte à execução distribuída

Nesta seção apresentamos três ambientes de suporte à execução distribuída que proveem abstrações para a programação paralela de ambientes distribuídos, cujo objetivo é facilitar para o programador a utilização eficiente das arquiteturas paralelas. Os ambientes apresentados são o Anthill, o MapReduce e o Dryad.

2.2.1 Anthill

O *Anthill* (Formigueiro) [Ferreira et al., 2005] é um ambiente para desenvolvimento e execução de aplicações distribuídas escaláveis. O ambiente foi desenvolvido tendo em mente aplicações paralelas não regulares, intensivas em processamento e em entrada-e-saída de dados (E/S). Nessas aplicações, que manipulam grandes volumes de dados, estes se encontram distribuídos em várias máquinas do sistema.

O sucesso desse enfoque depende da facilidade com que a aplicação possa ser dividida em etapas que sejam passíveis de execução em nós diferentes do sistema. Cada etapa dessa forma executará parte das transformações sobre os dados, iniciando com o conjunto de dados de entrada, até que se atinja o conjunto de dados de saída.

No modelo de programação *filtro-fluxo* utilizado pelo *Anthill*, o processamento é dividido em tarefas que operam sobre os dados que fluem pelo sistema. Cada filtro implementa uma tarefa que transforma os dados segundo a necessidade da aplicação e se comunica com outros filtros pelos canais de comunicação responsáveis pela transmissão contínua de dados (streams ou fluxos). Essas duas abstrações podem ser combinadas formando grafos arbitrários que representam o processamento da aplicação.

Usando esse modelo, criar uma aplicação no *Anthill* é um processo de decomposição do processamento em filtros (veja figura 2.1). Nesse processo, a aplicação é modelada como uma computação no modelo *dataflow* dividida em uma rede de filtros que transformam os dados. Durante a execução, o processo definido para cada filtro é instanciado em diferentes nós do ambiente distribuído. A esses processos dá-se o nome de cópias transparentes ou instâncias de um filtro. Dessa forma, cada estágio do processamento pode ser distribuído por muitos nós de uma máquina paralela e os dados que devem fluir por aquele filtro podem ser particionados pelas cópias transparentes, produzindo o paralelismo de dados desejado.

Além disso, para muitas aplicações, a execução consiste em múltiplas iterações da mesma cadeia de filtros. A aplicação se inicia com um conjunto de soluções possíveis que passam pelos filtros, sendo melhoradas e em alguns casos criando novas soluções que devem ser processadas novamente. Uma característica interessante nesse caso é que há muitas oportunidades para execução assíncrona, já que diversas soluções podem estar sendo testadas independentemente ao longo da cadeia de filtros a cada momento.

Muitas vezes é preciso garantir certa localidade de processamento: dados que compartilham uma certa característica na semântica da aplicação podem ter que ser processados pela mesma cópia transparente. Isso ocorre sempre que há algum tipo de estado local associado com algumas instâncias dos dados, ou quando há dependência de dados no processamento, como pode ocorrer em qualquer processo iterativo. Para esse fim, o *Anthill* fornece o fluxo identificado (*labeled stream*) que permite exatamente que a cópia de destino de cada instância dos dados seja determinada em função de alguma propriedade (um identificador) derivado do seu conteúdo.

Esse modelo de programação, dessa forma, nos permite extrair paralelismo das aplicações através de três possibilidades paralelismo de dados, de tarefas e assincronia.

Existe também a implementação de uma abordagem multinível do paradigma filtro-fluxo identificado [Teodoro et al., 2008]. O ambiente de execução do filtro foi substituído de orientado a processos para orientado a eventos. Essa nova versão permite que os programadores expressem eficientemente oportunidades de paralelismo para cada nó de processamento através de uma maior abstração de programação. Essa abordagem é bastante eficiente para plataformas que proveem múltiplos níveis de paralelismo, tanto inter quanto intra-nó.

Uma abordagem escalável e eficiente do Anthill para ambientes heterogêneos é apresentada em Teodoro et al. [2009b]. Essa abordagem permite ao programador focar em partes específicas do código ao invés da aplicação inteira, facilitando a construção de implementações alternativas para um mesmo filtro, que podem operar sem problemas em dados de um mesmo fluxo. A ideia é que cada implementação diferente do

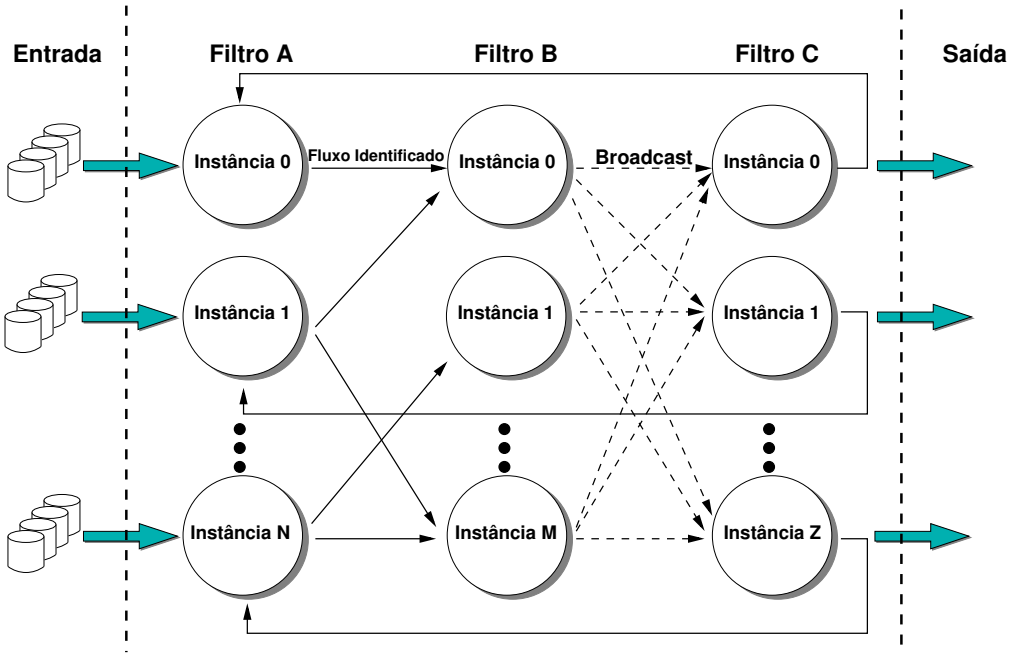


Figura 2.1: Fluxo de execução no ambiente Anthill.

filtro execute em diferentes dispositivos do nó computacional, com o objetivo de explorar todos os recursos heterogêneos disponíveis em máquinas de memória distribuída. Além disso, foram integrados módulos adicionais no arcabouço de execução do filtro para permitir ao Anthill escolher automaticamente entre as alternativas disponíveis e mandar os eventos para dispositivos diferentes concomitantemente.

Foram implementadas duas políticas de escalonamento de eventos entre os dispositivos disponíveis: (a) *first-come first-served* (FCFS) e (b) *dynamic weighted round robin* (DWRR). A primeira política é a padrão e nela o próximo evento a ser processado será o mais antigo enfileirado, utilizando o primeiro dispositivo que ficar disponível. A segunda política divide os eventos de acordo com um peso que varia durante a execução, que é específico para cada dispositivo. A partir disso, o peso é usado para ordenar a fila de eventos de cada dispositivo e quando o dispositivo estiver disponível o evento que ele irá processar será o de maior peso.

A aplicação de análise de imagens do neuroblastoma foi implementada utilizando o Anthill orientado a eventos para uma plataforma heterogênea utilizando CPUs e GPUs. As políticas implementadas reduziram o tempo de execução à metade em comparação com a utilização somente de GPUs.

Em Teodoro et al. [2009a] foram apresentadas duas aplicações de mineração de dados utilizando o Anthill orientado a eventos para um ambiente heterogêneo composto de CPUs e GPUs. O resultado alcançou uma aceleração de até 38 vezes para utilização

combinada de CPUs e GPUs. Os experimentos mostraram que foi possível acelerar a execução a partir da utilização de todos os recursos de um nó.

Teodoro et al. [2010] mostra várias otimizações em tempo de execução para ambientes heterogêneos utilizando o Anthill. As contribuições incluem:

- um módulo para estimar o desempenho relativo de tarefas baseado nos parâmetros de entrada com objetivo de poder definir com a maior precisão possível qual o melhor dispositivo para a execução;
- um algoritmo para coordenar de maneira eficiente as transferências de dados assíncronas entre a CPU e a GPU e sobrepor completamente as etapas de comunicação e processamento, evitando ciclos ociosos nos processadores;
- uma política de escalonamento de tarefas intra-filtro dinâmica para explorar a heterogeneidade das tarefas;
- uma política de distribuição de carga dinâmica e eficiente para otimizar computações do tipo filtro-fluxo em ambientes distribuídos e heterogêneos.

2.2.2 MapReduce

O MapReduce é um modelo de programação e uma implementação para o processamento e geração de grandes conjuntos de dados [Dean & Ghemawat, 2008]. Os usuários especificam uma função de mapeamento que processa um par chave/valor para gerar um conjunto intermediário de pares chave/valor e uma função de redução que mescla todos os valores intermediários associados com a mesma chave intermediária.

Os programas escritos nesse estilo funcional são automaticamente paralelizados e executados em um enorme conjunto de máquinas comuns. O sistema de execução cuida dos detalhes de particionamento dos dados de entrada, escalonamentos do programa no conjunto de máquinas, trata falhas de máquinas e gerencia a comunicação entre os nós. Isso permite aos programadores sem nenhuma experiência em sistemas paralelos e distribuídos utilizarem os recursos de um grande sistema distribuído. A implementação feita é altamente escalável e já foi testada para milhares de máquinas. Execuções típicas utilizam terabytes de dados.

Como mostrado na figura 2.2, o ambiente do MapReduce divide o arquivo de entrada em M partes de tamanhos iguais (parâmetro do usuário) e inicia várias cópias do programa em um aglomerado de máquinas. Uma das cópias é o mestre e fica responsável por atribuir as M tarefas de mapear e as N tarefas de redução para as outras cópias, os escravos. Os escravos leem do disco o dado de entrada para a tarefa

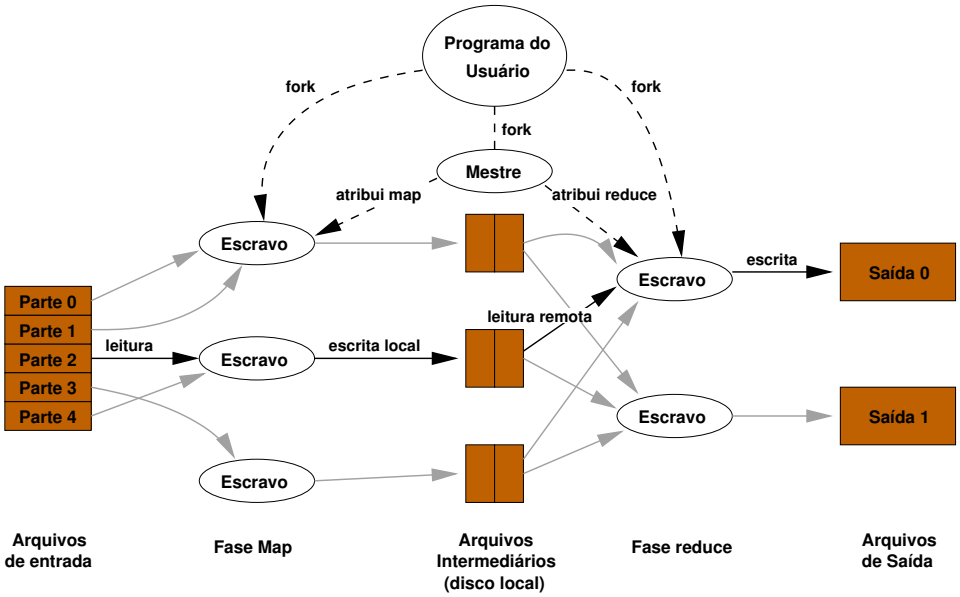


Figura 2.2: Fluxo de execução no MapReduce.

recebida e escrevem o resultado também no disco. Quando todas as tarefas estão terminadas a execução é retornada para o programa do usuário.

Várias tarefas reais podem ser encaixadas nesse modelo, além de várias arquiteturas de máquinas paralelas. O trabalho de Kruijf & Sankaralingam [2009] mostra que esse modelo é bastante adequado para aplicações que se adaptaram bem à arquitetura do processador STI Cell. Nessas aplicações foram alcançados ótimos resultados para o tempo de execução.

Ranger et al. [2007] faz uma avaliação do modelo do MapReduce para sistemas compostos de vários multiprocessadores. Os autores apresentam o *Phoenix*, uma implementação do MapReduce para sistemas de memória compartilhada que inclui uma interface de programação e um sistema de execução eficiente. O Phoenix gerencia automaticamente a criação de threads, escalonamento dinâmico de tarefas e particionamento dos dados. Os resultados encontrados foram comparados com a implementação paralela manual utilizando POSIX threads. O Phoenix apresentou resultados muito bons com relação ao tempo de execução e similares aos da paralelização manual.

O artigo de Tian et al. [2009] apresenta um escalonador dinâmico para o MapReduce avaliado em cargas de trabalho heterogêneas reduzindo o tempo de execução em cerca de 30%.

A utilização de aceleradores no MapReduce também tem sido alvo de estudos. Fang et al. [2010] apresenta o sistema de execução *Mars* para a utilização de GPUs no modelo MapReduce. O Mars esconde a complexidade da programação em GPUs

através da interface simples e familiar do MapReduce. Ele gerencia automaticamente o particionamento de tarefas, distribuição dos dados e a paralelização entre os processadores. Foram avaliadas 6 aplicações em um conjunto de computadores composto de CPUs multiprocessadas e GPUs conseguindo uma aceleração de até 72 vezes com relação à utilização do Phoenix somente em CPUs.

2.2.3 Dryad

O Dryad [Isard et al., 2007] é um ambiente de execução distribuída de alto desempenho e propósito geral. Uma aplicação no Dryad combina vértices computacionais com canais de comunicação para formar um grafo direto acíclico. Ele lida com problemas difíceis para a criação de aplicações concorrentes, escaláveis e distribuídas, como: escalonamento dos recursos, otimização do nível de concorrência dentro do computador, recuperação de falhas nos nós ou na comunicação e o envio de dados aonde ele é necessário. Ainda suporta vários mecanismos diferentes de transporte de dados entre os vértices de computação e a construção e o refinamento explícito de grafos *dataflow*.

Para facilitar a programação, o Dryad possui uma linguagem simples para a descrição de grafos que permite ao desenvolvedor a construção e otimização de grafos para explorar todas as características do seu mecanismo de execução. Também foram construídas abstrações de programação de alto nível em cima do Dryad para alguns domínios específicos de aplicações, o que permitiu a utilização mais fácil por alguns especialistas interessados em rápida prototipação das suas aplicações.

Os resultados relativos ao tempo de execução apresentados mostraram um excelente desempenho do Dryad tanto para a execução em um computador multiprocessado quanto para um enorme conjunto composto de milhares de computadores usando aplicações reais e não triviais.

2.3 Sumário

A utilização de aglomerados de computadores para o processamento sísmico já se tornou comum e, no momento, os trabalhos estão se concentrando no uso de aceleradores, como a GPU, que resultam em ambientes heterogêneos. Porém, somente Panetta et al. [2009] abordou o uso cooperativo de um ambiente heterogêneo composto de CPUs e GPUs.

Vários estudos apresentaram algoritmos paralelos para processamento sísmico, mas nenhum deles explorou estratégias como o paralelismo de dados, de tarefas e assincronia em conjunto para reduzir o tempo de execução.

O Anthill é um ambiente bastante versátil e eficiente para a execução distribuída, sendo que também foi estudado para ambientes heterogêneos [Teodoro et al., 2010, 2009b,a]. O paradigma filtro-fluxo identificado já se mostrou bastante eficiente em vários tipos de aplicações [Santos et al., 2007; Araujo et al., 2006; Teodoro et al., 2006; Veloso et al., 2004; Teodoro et al., 2009b,a] e poderá obter também um excelente desempenho no processamento sísmico, principalmente em ambientes heterogêneos.

O MapReduce é bastante interessante para o processamento de dados em larga escala e semelhante ao Anthill em alguns pontos, como: arquitetura, modo de paralelização e escalonamento. No entanto, o mesmo possui limitações importantes, como: a troca de mensagens via disco, as sincronização entre as etapas e a utilização fixa de duas etapas.

O Dryad também é um ambiente bastante interessante e, como o Anthill, é mais genérico que o MapReduce. Ao contrário do último, em que cada fase (map e reduce) possui somente uma entrada e uma saída, o Dryad processa grafos, na qual os vértices podem ter múltiplas entradas e múltiplas saídas.

Considerando o apresentado, vemos que pode ser interessante investigar a paralelização de algoritmos de processamento sísmico no ambiente Anthill. Além disso, o surgimento de ambientes heterogêneos torna ainda mais importante a utilização em conjunto de novas estratégias para explorar os recursos computacionais disponíveis.

Capítulo 3

Migração Sísmica

Neste capítulo apresentamos o algoritmo da migração sísmica de Kirchhoff. O procedimento descrito é um resumo das características consideradas pertinentes. Para mais informações veja Claerbout [1985] e Yilmaz [1987].

O método sísmico pode ser dividido em três etapas: aquisição, processamento e interpretação. A primeira etapa, a aquisição dos dados, pode ser tanto em mar quanto em terra. As figuras 1.1a e 3.1 representam o processo de aquisição no mar. A fonte gera ondas periódicas que são refletidas nas interfaces das camadas que formam o subsolo e depois são coletadas pelos receptores. Um traço é o conjunto de sinais originados a partir de uma mesma fonte e coletados por um mesmo receptor. Um traço é composto por valores de amplitude denominados amostras, coletados em um intervalo de tempo constante.

O processamento dos dados sísmicos, que corresponde à segunda etapa, tem como objetivo extrair informações do subsolo a partir dos dados adquiridos. A indústria do petróleo utiliza pacotes de software comerciais disponíveis para esse processamento. Esses pacotes contém centenas de módulos sísmicos (como filtros, redução de ruídos etc) e funcionalidades como uma ferramenta automática para executar em sequência vários módulos sobre os mesmos dados. Profissionais especializados em processamento sísmico selecionam os módulos mais apropriados para a aquisição e para a área alvo. Esse processamento demanda meses de trabalho de uma equipe inteira e o seu tempo de execução é totalmente dominado pelo módulo de migração sísmica. A terceira etapa corresponde à interpretação dos dados por especialistas, que decidem onde serão feitos os testes de perfuração.

A migração é o processo de produzir uma imagem do subsolo consistente com os dados adquiridos, através do posicionamento correto das superfícies refletoras. É um problema inverso, pois produz os parâmetros do modelo a partir de dados observa-

dos. Como em muitos outros problemas inversos, soluções consistentes podem requerer múltiplas execuções, considerando que muitas vezes a própria saída é utilizada como entrada, tal como a distribuição da velocidade da onda sísmica na subsuperfície. A migração sísmica de Kirchhoff, apresentada a seguir, é um dos métodos de migração mais utilizados para o processamento dos dados.

3.1 O algoritmo

A migração sísmica de Kirchhoff é apresentada no algoritmo 1. O texto em negrito entre parênteses define a nomenclatura que usaremos posteriormente.

Algoritmo 1: Algoritmo da Migração Kirchhoff

```
for todos offsets (laço do offset) do
    for todos os blocos de saída desse offset (laço do bloco de saída) do
        Zera o bloco de saída;
        for traços de entrada desse offset (laço dos traços de entrada) do
            Lê traço de entrada;
            Filtra traço de entrada;
            for traços de saída dentro da abertura (laço de migração) do
                for amostras do traço de saída (laço de contribuição) do
                    Calcula o tempo de trânsito;
                    Calcula a correção de amplitude;
                    Calcula os índices no traço de entrada filtrado e interpola
                    bilinearmente, o resultado é corrigido pela amplitude
                    calculada;
                    Acumula a contribuição na amostra de saída;
                end
            end
        end
        Grava o bloco de saída resultante em disco;
    end
end
```

O dado de entrada do algoritmo da migração sísmica é o conjunto de todas as amostras coletadas em uma aquisição. Esse conjunto é representado por $S(t, x, y, o)$, em que t é o tempo de trânsito do sinal (fonte até a interface no subsolo e a volta para o receptor), x e y são as coordenadas do *midpoint* na superfície (metade do caminho entre a fonte e o receptor) e o é a distância da fonte para o receptor ou *offset*. Todas as amostras que possuem os mesmos x, y e o fazem parte de um mesmo *traço sísmico*. Considerando que a fonte se move e dispara ondas periodicamente, a aquisição de dados

possui redundância. A redundância é desejada e utilizada para melhorar a qualidade do sinal. Uma aquisição comum pode conter até centenas de terabytes de dados.

A região geográfica de interesse (área de saída) é usualmente um subconjunto da área adquirida (área de entrada), como na figura 3.2. Os dados de entrada, por serem muito grandes, são processados por *offset* (**laço de offset**).

É crítico observar que cada traço de saída é calculado independentemente, pois sua computação depende apenas dos traços de entrada. Logo, trata-se de uma computação embaraçosamente paralela.

Como os traços sísmicos do volume de saída podem ser migrados de maneira independente, eles são, para cada *offset*, processados em blocos de tamanho parametrizado. Esses blocos são chamados de *blocos de saída* (**laço do bloco de saída**). Essa divisão por blocos é a maneira mais comum de paralelização da migração e, como os blocos possuem cargas de trabalho diferentes, a atribuição dos blocos é feita de maneira dinâmica utilizando uma estratégia do tipo mestre/escravo. A escolha do tamanho do bloco procura se adequar à quantidade de memória principal disponível e influi na granularidade do paralelismo.

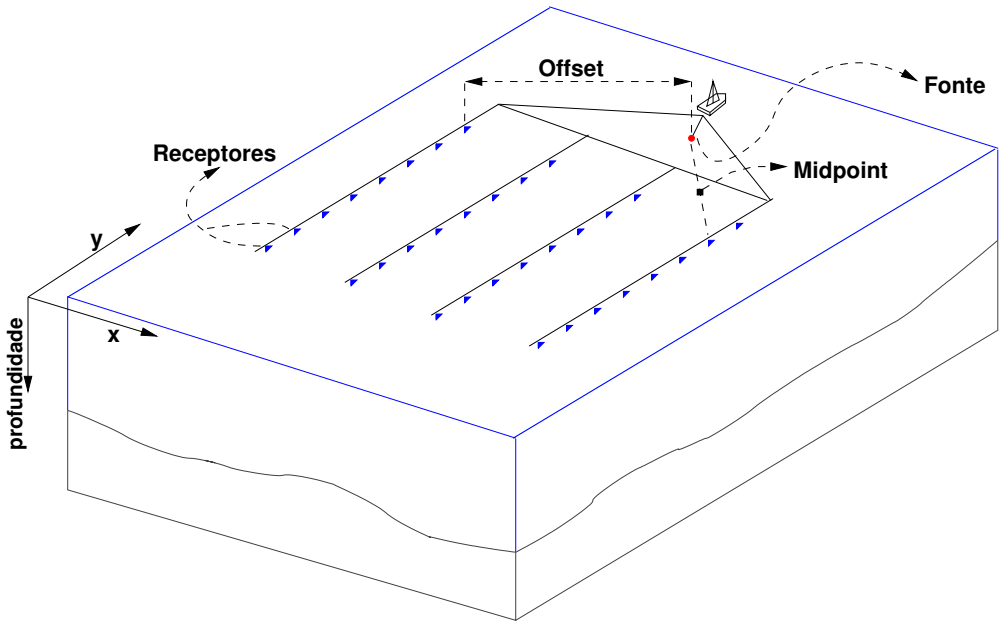


Figura 3.1: Aquisição dos dados símicos no mar.

Um outro parâmetro definido pelo usuário é a *abertura* (veja figura 3.2) do traço de entrada, que representa o volume da superfície máxima de contribuição desse traço. Essa abertura é definida por uma elipse centrada nas posições x e y do traço e dimensões a_x e a_y. Quando houver interseção entre a superfície correspondente ao volume da abertura e a superfície correspondente ao volume do bloco de saída, significa que haverá

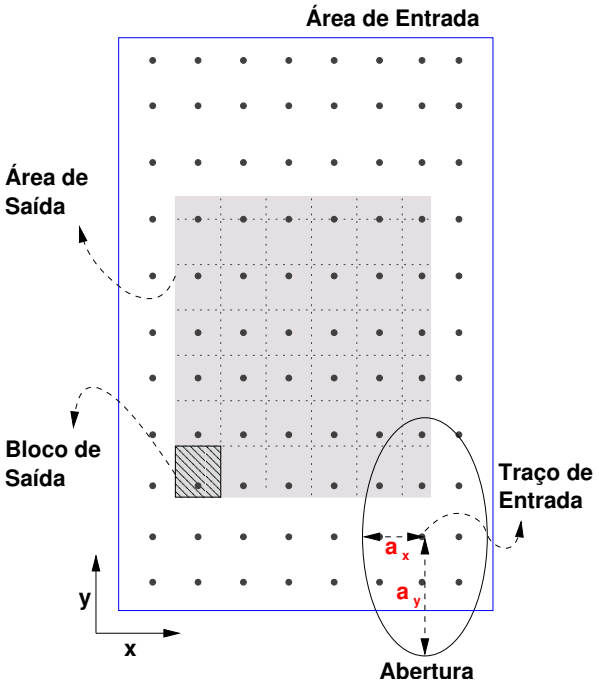


Figura 3.2: Atributos da migração sísmica.

migração do traço de entrada nos traços, interceptados pela abertura, do bloco de saída. As dimensões da abertura influenciam diretamente no tempo de processamento, pois os blocos de saída receberão a contribuição de mais traços de entrada na medida em que a abertura aumenta.

Assim, para cada traço de entrada são lidas apenas as suas coordenadas (**laço dos traços de entrada**) e, a partir da abertura, é verificado se ele contribui para algum traço do bloco de saída. Em caso positivo, as amostras do traço são lidas e filtradas. Ocorre, então, a contribuição desse traço de entrada filtrado para todos os traços no bloco de saída dentro da abertura (**laço de migração**), sendo que essa contribuição é feita para cada amostra (**laço de contribuição**).

Uma das operações de maior custo do *laço de contribuição* é a interpolação bilinear executada no traço de entrada filtrado. A interpolação é utilizada para mapear quais amostras do traço de entrada filtrado devem contribuir para uma determinada amostra do traço de saída. As amostras dos traços de entrada filtrados são armazenadas em posições discretas, já as posições do tempo de trânsito e do filtro são calculadas em ponto flutuante. Dessa maneira, é utilizado uma interpolação bilinear das quatro amostras discretas mais próximas, correspondentes ao tempo de trânsito e ao filtro calculados, para encontrar a amostra de entrada que contribuirá para a amostra de saída.

A figura 3.3 apresenta a operação de interpolação bilinear em um traço de entrada

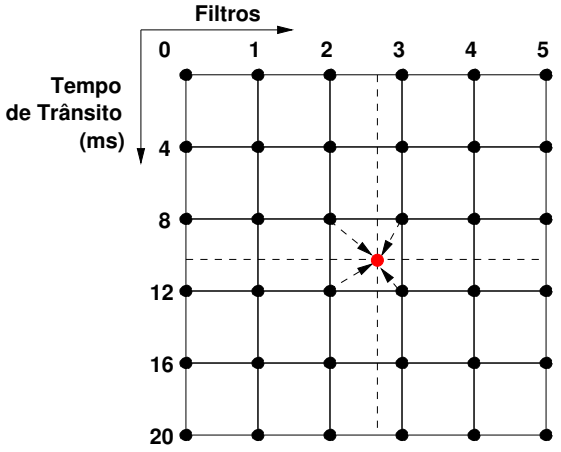


Figura 3.3: Interpolação bilinear em um traço de entrada filtrado.

filtrado com seis filtros e amostras coletadas em intervalos de quatro milissegundos. O ponto alvo corresponde às coordenadas em um traço de entrada filtrado, calculadas em ponto flutuante, para uma determinada amostra de saída. Como descrito anteriormente, a amostra que será utilizada para contribuição é resultante da interpolação das quatro amostras mais próximas.

O algoritmo da migração pode usar tanto um campo de velocidade simplificado quanto outro mais complexo, que leva em conta os detalhes de deformação da frente de onda. O primeiro caso é chamado de *migração em tempo* e produz como resultado uma imagem $T(\alpha, x, y, o)$, onde α representa o tempo de trânsito vertical. O segundo caso é chamado de *migração em profundidade* e produz uma imagem $T(z, x, y, o)$ onde z representa a profundidade.

Formalmente, o algoritmo pode ser descrito da seguinte maneira. Uma contribuição para uma amostra na imagem $T(z, x, y, o)$ é gerada por uma amostra de entrada $S(t, x', y', o)$ sempre que o tempo de trânsito t da amostra seja a soma dos tempos tempo de trânsito da fonte até o ponto alvo (z, x, y, o) no subsolo e dele até o receptor. O conjunto de todos os possíveis traços de entrada (x', y', o) que podem contribuir para um traço de saída estão dentro de uma elipse de eixos a_x e a_y (aberturas) centrada no ponto (x, y, o). Os traços de entrada são filtrados para atenuar o álias espacial. As amplitudes amostrais são corrigidas para contabilizar o espalhamento de energia durante a propagação. A migração Kirchhoff em profundidade é o resultado, para cada amostra de saída (z, x, y, o), do somatório

$$T_{z,x,y,o} = \sum \beta(\alpha S^f_{t,x',y',o} + (1-\alpha) S^f_{t+\Delta t,x',y',o}) \tag{3.1}$$

em que a soma das contribuições é feita por todos os traços de entrada (x', y', o) dentro

da abertura elíptica, f denota o filtro selecionado, β denota a correção da amplitude, t é o tempo de trânsito correspondente a z e α é o peso da interpolação necessária para mapear o tempo de trânsito em ponto flutuante computado para amostras discretas.

A principal diferença entre migração em tempo e em profundidade é o cálculo do tempo de trânsito. A migração em profundidade calcula o tempo de trânsito através da adição dos tempos de trânsito pré-computados da fonte para o alvo e do alvo para o receptor (utiliza grandes matrizes de dados pré-computados), necessitando de estimativas confiáveis para o campo de velocidades. Por outro lado, a migração em tempo baseia-se em aproximações analíticas do tempo de trânsito que dependem apenas de estimativas do campo de velocidade média. Consequentemente, a migração em profundidade precisa de muito mais memória e tempo de processamento do que a migração em tempo.

Cada instância do laço mais interno (laço de contribuição) necessita da ordem de dezenas de KB de memória enquanto o próximo laço do aninhamento (laço de migração) precisa da ordem de dezenas de MB. As referências de memória do laço mais interno são o traço de entrada filtrado (leitura), o traço de saída (leitura, modificação, escrita), a velocidade (leitura) e dados temporários. O laço de migração precisa de outro traço filtrado e velocidades diferentes para executar o laço interno. Dessa maneira, essa aplicação possui uma alta intensidade computacional, definida como o número de operações de ponto flutuante por referência à memória.

Etapa	*Tempo de Execução (s)*	*%*
Filtragem	513,27	1,29
Laço de Migração	39.351,60	98,70
Outros	3,05	0,01
Total	39.867,92	100,00%

Tabela 3.1: Tempo de execução de um bloco em uma CPU.

O algoritmo da migração possui duas etapas que dominam o seu tempo de execução: a filtragem e o laço de migração. Como exemplo, temos na tabela 3.1 uma execução na CPU utilizando um bloco de saída representativo semelhante aos coletados para uma migração real. O tempo do laço de migração é significativamente dominante, cerca de $98,70\%$ do tempo total.

Uma métrica de velocidade específica da aplicação é o *número de amostras contribuídas por segundo*, definida como a taxa de execução das iterações do laço mais interno. Nesse trabalho, os estudos se restringiram à migração em tempo e toda a computação utilizou precisão simples, como em toda a indústria do petróleo.

Capítulo 4

Paralelização da Migração Sísmica

Neste capítulo são apresentados os detalhes da paralelização implementada para a migração sísmica de Kirchhoff em tempo descrita no capítulo 3. Foram implementados, utilizando CPU e GPU, três dimensões de paralelismo: pelo dado de saída, de tarefas e pelo dado de entrada. A implementação final foi baseada no Anthill (seção 2.2.1) e utilizou também OpenMP [Chapman et al., 2007], POSIX threads [Nichols et al., 1996] e CUDA [Garland et al., 2008].

O capítulo está organizado da seguinte forma: a seguir apresentamos as arquiteturas de CPU e GPU utilizadas, depois as três dimensões de paralelismo e, por último, os detalhes de implementação.

4.1 As arquiteturas utilizadas

A paralelização proposta nesse trabalho não se restringe a nenhuma arquitetura específica. No entanto, o conhecimento da arquitetura alvo permite o uso de otimizações que podem aumentar significativamente o desempenho das aplicações.

Nesta seção, apresentamos de maneira geral algumas arquiteturas paralelas bastante comuns incluindo as que foram utilizadas para a experimentação.

4.1.1 A arquitetura CPU

Existem diversos exemplos de CPUs multiprocessadas no contexto atual, pois várias são as formas utilizadas para integrar múltiplos processadores em um mesmo *chip*. Uma das principais diferenças entre as diversas arquiteturas é que elas têm vários graus de compartilhamento de caches em níveis diferentes.

Assim, para atingir bons níveis de desempenho é necessário que as aplicações sejam explicitamente implementadas para cada caso. O uso eficiente dessa hierarquia requer maximizar a localidade de referência na utilização dos dados pelos núcleos dos multiprocessadores. Enquanto o uso eficiente da hierarquia de memória é importante nos processadores sequenciais, nos multiprocessadores essa importância aumenta drasticamente.

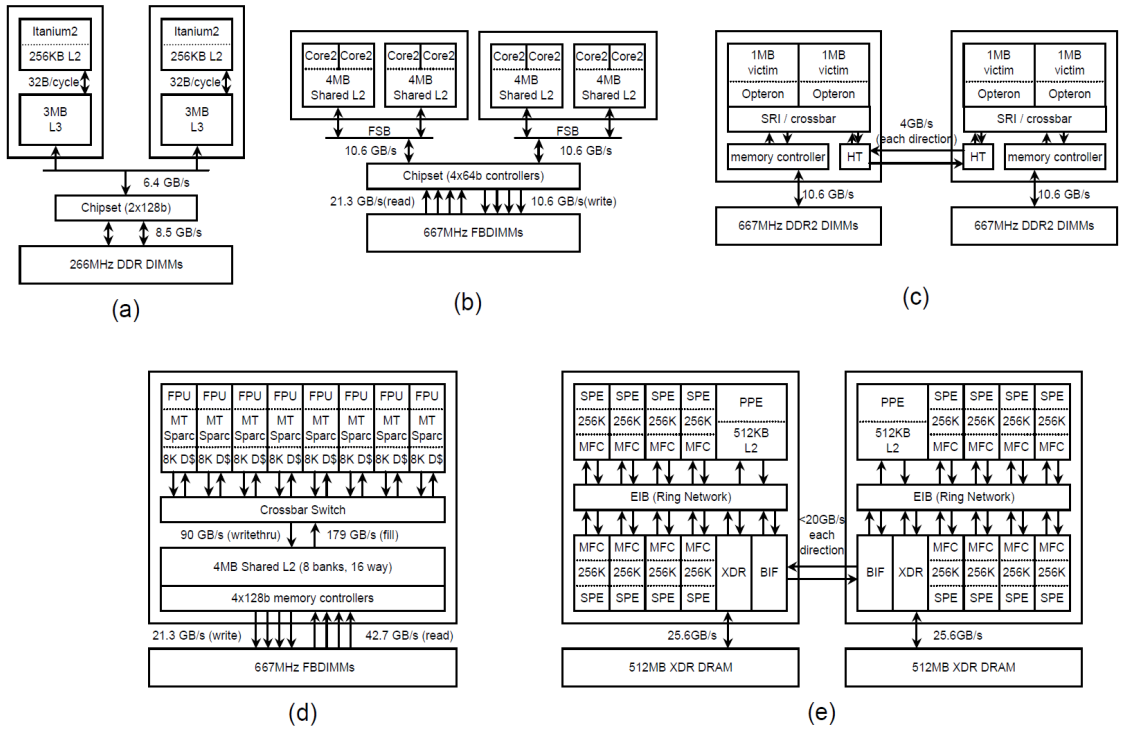


Figura 4.1: Visão geral da arquitetura de (a) dois Itanium2, (b) dois quad-core Intel Clovertown, (c) dois dual-core AMD Opteron X2, (d) um octa-core Sun Niagara 2 e (e) dois octa-core STI Cell.

A figura 4.1 apresenta uma visão geral da arquitetura de cinco multiprocessadores. Entre eles há uma grande diferença na hierarquia das caches utilizadas, o que certamente acarreta diferentes abordagens para aproveitar da melhor maneira a hierarquia disponível.

Além disso, outras otimizações como utilização das unidades vetoriais e acesso alinhado tem implicações importantes no desempenho das aplicações em arquiteturas multiprocessadas. Os pesquisadores têm estudado extensivamente as implicações de cada modelo de cache e outras otimizações no desempenho das aplicações [Savage & Zubair, 2008; Marathe et al., 2006; Datta et al., 2008; Williams et al., 2008; Liu et al., 2008].

4.1.2 A arquitetura GPU

Inicialmente, a *Graphics Processing Unit* (GPU) foi projetada para um classe específica de problemas, o processamento de imagens. Nos últimos anos, um número crescente de pesquisadores identificou aplicações de outras classes com características semelhantes, que foram aceleradas pela GPU [Owens et al., 2008].

A GPU é um dispositivo altamente paralelo que rapidamente ganhou espaço para aplicações computacionalmente intensivas. E mesmo possuindo arquitetura e modelo de programação bem diferentes da maioria dos processadores convencionais, o desempenho alcançado pela GPU a torna potencialmente importante nos sistemas computacionais atuais e futuros.

Tão importante no desenvolvimento da GPU como um dispositivo de propósito geral (GPGPU), está o avanço dos modelos e ferramentas de programação. O desafio, tanto para a indústria como para a academia, é conciliar o acesso de baixo-nível ao hardware para obter desempenho com a criação de ferramentas e modelos de programação de alto-nível que permitam ao programador flexibilidade e produtividade no uso das GPUs.

As unidades programáveis da GPU seguem o modelo de programação único-programa-múltiplos-dados (SPMD). Para eficiência, a GPU processa vários elementos em paralelo. Cada elemento é independente dos outros e, no modelo inicial, não existe comunicação entre eles. Os elementos podem ler dados de uma memória global compartilhada e também escrever nessa memória de maneira arbitrária. O modelo é adequado para programas regulares, sem desvios, em que a mesma instrução é aplicada para múltiplos dados (SIMD).

Uma GPU muito utilizada atualmente em computação de alto desempenho é a Nvidia Tesla C1060 (figura 4.2), que pode ser vista como um conjunto de 240 *streaming processors* (SP), organizados em 30 multiprocessadores (SM). Cada SP pode executar simultaneamente 128 threads, resultando em 30.720 threads concorrentes por GPU. Além disso, possui 16 KB de memória compartilhada por SM, 4 GB de memória global interna e 102 GB/segundo de largura de banda da memória.

CUDA é uma interface de alto-nível da Nvidia para a execução em GPU [Garland et al., 2008]. Ela provê uma sintaxe baseada na linguagem C. Expõe dois níveis de paralelismo (dados e threads) e uma múltipla hierarquia de memória: registradores por thread, memória rápida compartilhada por bloco de threads, memória global e memória principal do nó. Além disso, CUDA permite a utilização de ponteiros, leitura/escrita em qualquer posição na memória e sincronização entre as threads de um mesmo bloco. Entretanto, toda essa flexibilidade e potencial ganho de desempenho obriga o progra-

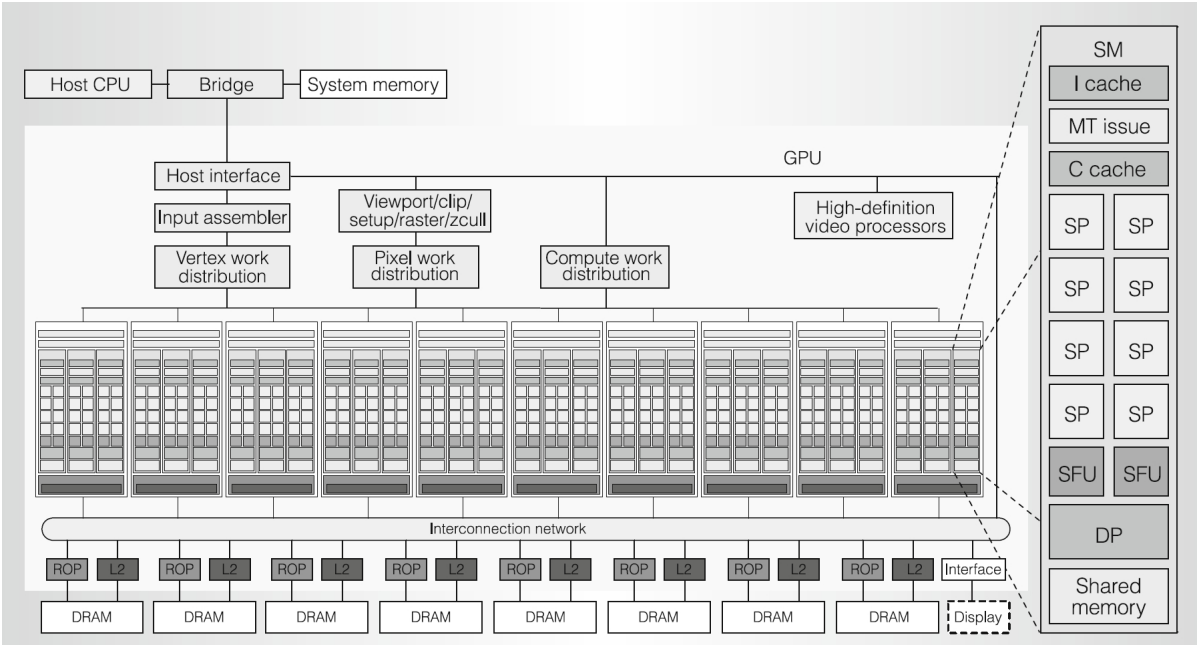


Figura 4.2: Arquitetura da GPU.

mador a conhecer os detalhes do hardware, como: uso de registradores, escalonamento de threads e blocos e o padrão de comportamento de acesso à memória.

4.2 As três dimensões de paralelismo

A busca por acelerar o algoritmo da migração sísmica motivou a implementação da migração em um ambiente heterogêneo composto de multiprocessadores e aceleradores. Além disso, a disponibilidade de conjuntos de máquinas interconectadas levou a implementação da migração para o ambiente distribuído.

A seguir são descritas as três dimensões que foram desenvolvidas para aproveitar ao máximo esse ambiente através da cooperação entre os dispositivos paralelos.

A figura 4.3 apresenta um diagrama com a visão geral das três dimensões de paralelismo.

4.2.1 Paralelismo pelo dado de saída

O *paralelismo pelo dado de saída* é o que ocorre pela execução simultânea de iterações do *laço de migração* (algoritmo 1) e foi denominada assim porque essas iterações geram contribuições que são armazenadas nos traços de saída.

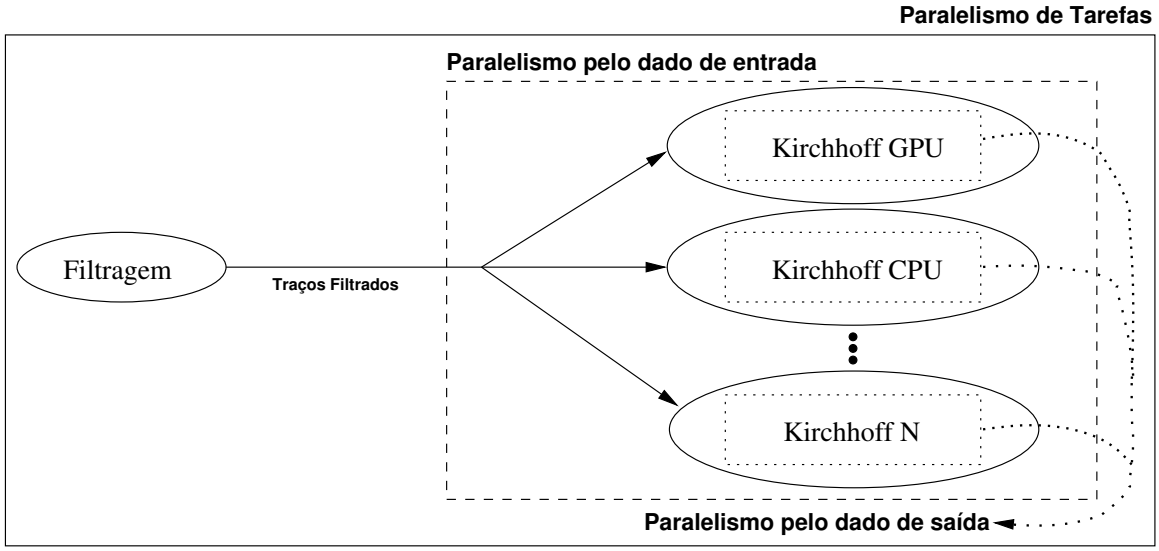


Figura 4.3: Visão geral das três dimensão de paralelismo implementadas.

Como apresentado na seção 3.1, o dado de saída é composto de traços sísmicos, que podem ser organizados de várias maneiras. A maneira mais comum é a divisão da área de saída em subáreas de tamanho parametrizado. Cada subárea é chamada de bloco porque, além das duas dimensões definidas pelo usuário, existe a terceira dimensão em que estão as amostras de cada traço.

Da mesma maneira que o processamento de cada bloco é independente dos demais, cada um dos traços do bloco de saída tem o seu processamento independente. Isso levou à paralelização intra-bloco do *laço de migração* através da utilização de threads tanto na abordagem para a CPU quanto na abordagem para a GPU.

A seguir estão explicadas como foram feitas as implementações paralelas na CPU e na GPU para o laço de migração, com o objetivo de aproveitar as unidades paralelas disponíveis nesses multiprocessadores.

4.2.1.1 Usando CPU

A paralelização do laço de migração na CPU foi implementada utilizando OpenMP, em que cada thread/processador fica responsável por realizar a migração em um conjunto de traços do bloco de saída.

A contribuição de um traço de entrada no traço de saída ocorre de maneira irregular porque depende da posição do primeiro com relação ao segundo. Assim, a contribuição de um traço de entrada pode ser desigual ante ao bloco e gerar desbalanceamento de carga entre as threads, uma thread pode ficar responsável por um

conjunto de traços que é contribuído enquanto as outras threads recebem traços de saída que não resultam em processamento.

Os traços de saída que cada thread vai processar podem ser definidos estaticamente ou dinamicamente. A definição estática está mais sujeita ao problema do desbalanceamento, mas não possui o custo extra do escalonamento dinâmico. A definição dinâmica tende a ser mais justa, porque define quais traços cada thread vai processar durante a execução. Entretanto, esse tipo de definição possui o custo extra do escalonamento que aumenta com tarefas de grão mais fino. Nesse caso, a tarefa consiste no conjunto de traços que será designado estaticamente ou dinamicamente para as threads.

4.2.1.2 Usando GPU

Para conseguir um bom desempenho da migração sísmica na GPU é necessário gerar o máximo possível de threads idênticas e independentes para a execução do laço de migração, assim como manter os dados o máximo possível na memória da GPU e sobrepor processamento e comunicação.

Com intuito de reduzir o tráfego de dados entre as memórias da CPU e da GPU, cada bloco de saída é criado na memória da GPU somente uma vez no início da execução e reside lá até o fim do *laço dos traços de entrada*, quando o resultado é copiado de volta para ser gravado em disco. Além disso, para tentar sobrepor a comunicação e o processamento foi utilizado um buffer duplo na GPU para a transferência dos traços filtrados. Logo, enquanto um traço está sendo migrado o próximo já está sendo copiado para a GPU.

Cada amostra do bloco de saída recebe, no máximo, uma contribuição de cada traço de entrada. Portanto não existe nenhuma condição de corrida para uma execução do laço de migração, pois este laço é executado para um traço de entrada por vez. Consequentemente, as iterações do laço de migração são totalmente independentes e podem ser mapeadas em múltiplas threads. Uma thread pode ser definida como a computação da contribuição de um único traço de entrada para uma única amostra de saída. No entanto, essa abordagem é ruim porque não considera o fato de que algumas computações do laço mais interno são válidas para um conjunto de amostras de saída consecutivas e podem ser feitas uma vez só. Para reaproveitar essas computações, foi definido que cada thread seria responsável por um conjunto consecutivo de amostras de saída. Mesmo com essa diminuição, o número de threads para um bloco de saída comum passa de 1 milhão.

Na GPU, as threads possuem tarefas de grão bem mais fino que na CPU. Enquanto cada thread na CPU é responsável por um conjunto de traços, na GPU é

responsável por uma parte do traço, ou seja, um conjunto de amostras.

A computação mais interna do algoritmo necessita de uma interpolação bilinear das amostras do traço de entrada. Na GPU, essa interpolação é semelhante a uma operação gráfica de textura e utiliza hardware especializado. Ela é feita em conjunto com uma leitura consecutiva de um vetor em cache na memória de textura. Um processamento utilizando 1024 threads por SM e boa utilização da cache permite sobrepor quase que completamente as latências dessa operação.

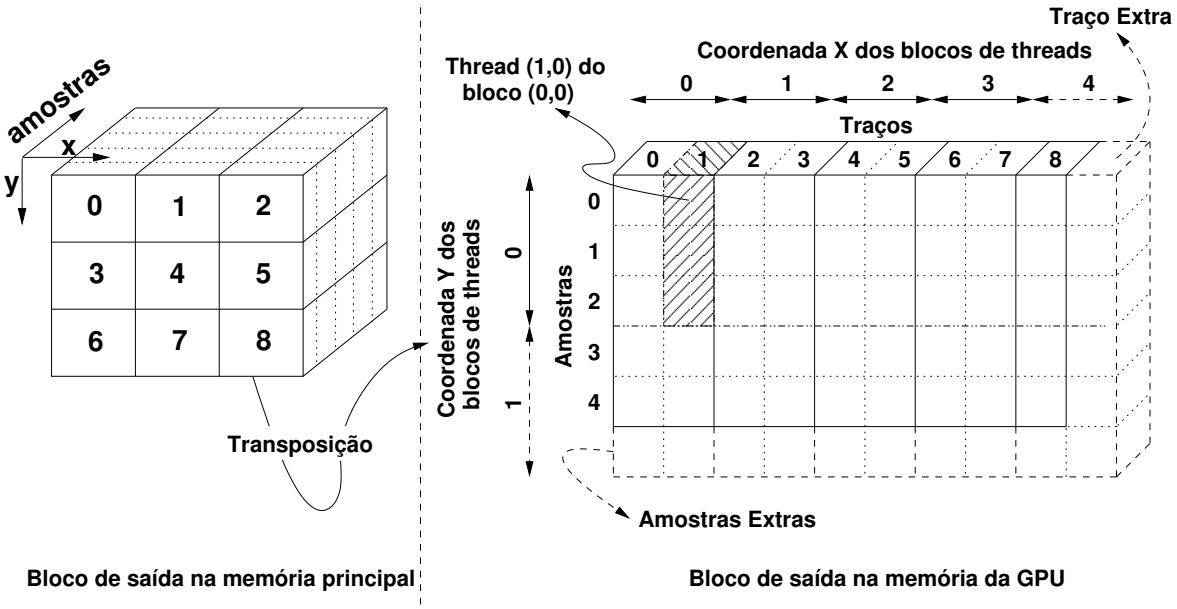


Figura 4.4: Alocação do bloco de saída na memória principal e na memória da GPU.

Na linguagem CUDA as threads são divididas em blocos de dimensões x, y e z. Nessa implementação, são utilizadas apenas as dimensões x e y e os traços sísmicos são transpostos para que as leituras e as escritas na memória global sejam feitas de maneira alinhada entre as threads de um mesmo bloco. O tamanho de cada traço de saída é ajustado para ser múltiplo do número de threads por bloco na dimensão y e evitar desvios condicionais dentro de um mesmo conjunto de threads. O bloco de saída é particionado em conjuntos de 128 traços que compõem um bloco de threads na dimensão x. O bloco de saída é preenchido com traços nulos para garantir que os blocos de threads tenham o mesmo número de traços.

O bloco de saída em três dimensões é mapeado em uma dimensão na memória principal, na qual a primeira dimensão percorrida é a das *amostras* dos traços, em seguida a coordenada x dos traços e, por fim, a coordenada y.

A figura 4.4 mostra como o bloco de saída é alocado na memória principal e na

memória da GPU. Nesse exemplo, foi utilizado um bloco de saída com oito traços e cinco amostras, que resultou em duas threads com três amostras por bloco de threads e um total de dez blocos e vinte threads. Veja que é necessário acrescentar um traço sísmico e uma linha de amostras para que todas as threads tenham a mesma quantidade de trabalho e evitar desvios condicionais.

4.2.2 Paralelismo de tarefas

Como apresentado na tabela 3.1, a filtragem e o laço de migração dominam o tempo de execução da migração Kirchhoff. A partir dos dados vemos que o laço de migração é o melhor candidato para a paralelização, que foi apresentada na seção 4.2.1.1 para CPU e na 4.2.1.2 para GPU. A estratégia selecionada foi manter a leitura e a filtragem exclusivamente na CPU enquanto a GPU executa apenas a tarefa mais intensiva em processamento.

No entanto, a filtragem, apesar de pouco significante, tende a aumentar a sua importância no tempo de execução na medida em que mais processadores (CPUs e GPUs) são utilizados pelo laço da migração.

Sejam s e q as frações relativas ao tempo total para a filtragem e para o laço de migração respectivamente. Paralelizando o laço de migração usando p processadores e mantendo a filtragem sequencial, o tempo total tende a s quando p aumenta muito. De acordo com a Lei da Amdahl [Amdahl, 1967], o speedup máximo alcançado será $Speedup_{Max} = 1/s$.

Dessa maneira, foram implementados, através do Anthill, um filtro responsável pela *filtragem* e outro filtro responsável pelo *laço de migração*. Esses dois filtros executam em paralelo promovendo, assim, um *paralelismo de tarefas* entre a filtragem e o laço de migração, sejam eles executados em qualquer unidade de processamento (CPU ou GPU). Além disso, esses filtros podem ter múltiplas cópias promovendo um paralelismo nos dados de entrada de cada uma dessas tarefas.

4.2.3 Paralelismo pelo dado de entrada

Outra direção interessante é o processamento simultâneo dos traços de entrada no laço de migração, promovendo o *paralelismo pelo dado de entrada*. Existe, no entanto, uma condição de corrida dos traços de entrada para um mesmo bloco de saída, pois cada amostra do bloco de saída só pode receber contribuição de um traço de entrada por vez. Assim, para contornar essa condição, vários traços de entrada são migrados simultaneamente em cópias diferentes do bloco de saída e, no final da execução, os

resultados são agregados para gerar o bloco de saída resultante.

Outras abordagens apresentaram um escalonamento dinâmico de blocos entre CPU e GPU de acordo com a estimativa do número de contribuições que o bloco receberia [Panetta et al., 2009]. Os blocos de maior contribuição seriam processados na GPU e os menores seriam processados na CPU. No entanto, o grande problema dessa abordagem é que o número de blocos com pouca contribuição é pequeno e a CPU ficará ociosa em grande parte da execução, desperdiçando recursos.

A abordagem implementada nesse trabalho utiliza um escalonamento dinâmico de grão mais fino, por traço de entrada. Assim, um mesmo bloco de saída é processado por várias unidades de processamento disponíveis num mesmo nó e consomem traços de entrada na velocidade de processamento que cada unidade pode alcançar. Isso faz com que as unidades utilizadas operem na máxima capacidade sem deixar recursos ociosos. Essa abordagem não oferece um escalonamento ótimo, mas como o grão é fino e a quantidade de traços tende a milhares, a penalidade pela unidade de execução mais lenta consumir o último traço é pequena.

Ao final da execução, os blocos de saída parciais contribuídos por cada unidade de processamento devem ser agregados, o que, apesar de ser uma operação extra, pode ser paralelizada e otimizada (escrita alinhada) resultando em uma parcela muito pequena no tempo de execução.

4.3 Detalhes da implementação usando o Anthill

A paralelização da migração sísmica de Kirchhoff no Anthill possui três filtros: Filtragem, Kirchhoff e Escalonador. Cada instância do filtro Kirchhoff é responsável por processar um bloco de saída do offset atual por vez. Esse bloco de saída é definido pelo filtro Escalonador. O filtro Kirchhoff calcula quais traços de entrada serão necessários para migrar o bloco recebido e os requisita, na medida em que a migração acontece, para o filtro Filtragem. Todos os filtros podem ter múltiplas instâncias permitindo a escalabilidade da aplicação. A figura 4.5 apresenta os três filtros implementados e o tipo de informação que é comunicado entre eles.

A migração sísmica se encaixa bem no modelo filtro-fluxo utilizado pelo Anthill. No entanto, o Anthill não possui controle de fluxo na comunicação entre os filtros, o que pode resultar em estouro de memória quando um filtro enviar mensagens em uma frequência maior do que o outro pode receber. Para evitar esse problema, foi utilizado uma estrátegia de requisição/resposta, que não faz parte do modelo filtro-fluxo, na qual o filtro pede eventos de maneira assíncrona na medida em que fica ocioso.

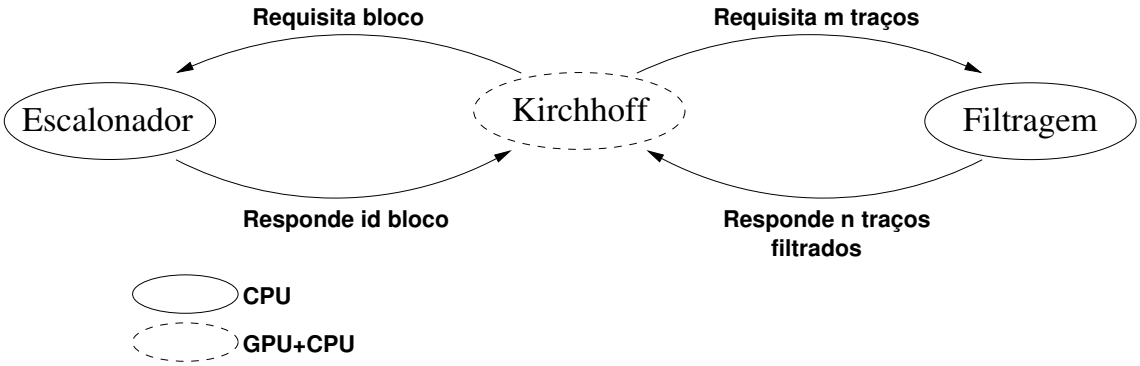


Figura 4.5: Filtros Anthill da migração sísmica de Kirchhoff.

O objetivo dessa implementação foi obter o máximo de assincronia e com isso sobrepor o processamento com a comunicação, além de implementar as três dimensões de paralelismo apresentadas. Os detalhes de cada filtro estão a seguir.

4.3.1 Filtragem

Este filtro é o responsável pela filtragem sob demanda dos traços de entrada. De acordo com as requisições do Kirchhoff, ele filtra e envia o resultado para a instância que o requisitou. Cada instância desse filtro pode acessar todos os traços de entrada e fica a cargo do filtro Kirchhoff definir como será feito o balanceamento das requisições. Como podem existir f instâncias da Filtragem para k instâncias do Kirchhoff, várias configurações são possíveis. Para aproveitar o máximo da largura de banda da rede de interconexão dos nós, as requisições e as respostas são feitas em lotes de tamanhos m e n, respectivamente.

4.3.2 Kirchhoff

O filtro Kirchhoff é o que executa a etapa de migração utilizando como entrada os traços de entrada filtrados que contribuem para o bloco de saída recebido. Ele foi implementado para permitir a execução de um mesmo bloco por todas as unidades de processamento intra-nó disponíveis. Essa abordagem permite a cooperação a toda capacidade de todos os dispositivos existentes dentro de uma mesma máquina evitando recursos ociosos.

A partir do momento que chega um novo bloco são criados os contextos relativos a esse bloco para todos os dispositivos que serão utilizados. Nessa criação estão incluídos, principalmente, alocação de memória e cópias de dados que persistirão durante toda a migração do bloco. Cada contexto possui uma cópia privada do bloco de saída e os

traços de entrada podem ser migrados em qualquer unidade de processamento.

Como apresentado na seção 4.2.1, foram feitas implementações para duas unidades de processamento distintas, CPU e GPU. A implementação da GPU conta com uma thread na CPU que é responsável por copiar os traços para a memória da GPU.

Com o objetivo de tentar evitar um desbalanceamento de carga entre as unidades de processamento e tornar o algoritmo robusto a múltiplas arquiteturas, foi implementado um mecanismo dinâmico de atribuição traços de entrada filtrados para as unidades de processamento existentes. Uma thread fica responsável por receber os eventos com os traços de entrada filtrados e alimentar uma fila de tarefas com traços prontos para serem migrados. Cada tarefa possui um conjunto de k traços. As threads da CPU e da GPU acessam essa fila na medida em que ficam ociosos para buscar novos traços. O objetivo é manter as unidades de processamento em total capacidade de funcionamento.

Listas	*Necessários*	*Aguardados*	*Significado*	*Ação*
	Vazia	Vazia	Fim Bloco	Pede novo bloco
	Não Vazia	Vazia	Acabaram os traços	Requisita mais traços
	Vazia	Não Vazia	-	-
	Não Vazia	Não Vazia	-	-

Tabela 4.1: Gerenciamento dos traços necessários pelo filtro Kirchhoff.

O gerenciamento dos traços necessários (veja a tabela 4.1) é feito utilizando o identificador único de cada traço e duas listas: a lista de traços necessários e a lista de traços aguardados. A lista de traços necessários contém inicialmente todos os identificadores dos traços de entrada que contribuem para o bloco em questão. Na medida em que esses traços são requisitados, os seus identificadores são movidos para a lista de traços aguardados, quando recebidos são retirados da lista de aguardados.

Para sobrepor a comunicação entre os filtros Kirchhoff e Filtragem com a migração dos traços, as requisições são sempre feitas de maneira adiantada para quando as unidades ficarem ociosas, já existam traços filtrados prontos para serem migrados. Quando a lista de aguardados chega ao fim uma nova requisição de m traços é feita.

Ao final, os resultados das cópias privadas do bloco de saída em todos os contextos são somados e gravados em disco.

Um novo bloco é pedido antes da última leva de traços de entrada filtrados serem migrados. Isso faz com que no momento em que o bloco atual finalizar, um novo bloco esteja disponível para ser migrado.

4.3.3 Escalonador

O filtro Escalonador é o mais simples. Ele possui uma lista de blocos do offset atual para serem processados. Os identificadores desses blocos são enviados para as instâncias do filtro Kirchhoff na medida em são requisitados por elas. O objetivo da existência desse filtro é que o escalonamento de blocos seja dinâmico para tentar evitar o desbalanceamento de carga, dado que o processamento dos blocos de saída possui cargas diferentes. Caso sejam utilizadas múltiplas instâncias desse filtro é necessário dividir a lista de blocos entre elas.

4.4 Sumário

Diversas arquiteturas de processadores e aceleradores estão disponíveis no mercado, considerando as várias maneiras de integrar múltiplos núcleos em um mesmo *chip*. Assim, o conhecimento da arquitetura utilizada é muito importante, pois permite aumentar significativamente o desempenho das aplicações.

A migração sísmica foi implementada para um ambiente heterogêneo composto de multiprocessadores e aceleradores com o objetivo de reduzir o seu tempo de execução. Foram implementadas três dimensões de paralelismo para aproveitar ao máximo esse ambiente a partir da cooperação entre os dispositivos paralelos.

A primeira dimensão é o *paralelismo pelo dado de saída*, na qual as iterações do laço de migração são executadas simultaneamente nas unidades de processamento. A segunda dimensão é o *paralelismo de tarefas*, na qual as principais etapas do algoritmo são executadas em paralelo. Por último, a terceira dimensão é o *paralelismo pelo dado de entrada*, que permite a migração concorrente dos traços de entrada em múltiplos dispositivos diferentes.

O ambiente Anthill foi utilizado para implementar essas três dimensões de paralelismo. Foram definidos três filtros: Filtragem, Kirchhoff e Escalonador. Os dois primeiros são responsáveis pela filtragem e pelo laço de migração, respectivamente. Essas duas etapas dominam quase que totalmente o tempo de execução da migração. O filtro Escalonador é responsável por atribuir dinamicamente os blocos para as instâncias do filtro Kirchhoff. O objetivo da implementação foi sobrepor ao máximo a comunicação com processamento, em conjunto com a utilização do paralelismo de dados e do paralelismo de tarefas.

Capítulo 5

Experimentos

Os experimentos foram executados utilizando um computador com dois processadores Intel Xeon L5420 (figura 4.1b) de 2,50 GHz com 4 núcleos cada, 32 GB de memória principal e uma GPU Nvidia Tesla C1060.

Utilizamos somente um computador porque o processamento dos blocos de traços de saída é embaraçosamente paralelo e o speedup para múltiplas máquinas, com cada uma processando um bloco por vez, é linear [Panetta et al., 2009]. A configuração utilizada para uma máquina pode ser replicada para todas as outras, no caso de execução em um aglomerado.

Foram utilizados blocos de um dado sísmico gerado sinteticamente, que possui as mesmas características computacionais e representatividade com relação aos dados sísmicos reais. Cada bloco possui dimensões de 128 por 128 traços sísmicos com 1.728 amostras por traço e contempla uma área de 2,5 km^2 aproximadamente. As execuções foram limitadas à contribuição de 100 mil traços de entrada. Cada experimento foi repetido cinco vezes e os tempos de execução encontrados obtiveram um desvio padrão relativo menor que 1%.

Os experimentos, detalhados a seguir, foram feitos com o objetivo de avaliar as três dimensões de paralelismo apresentadas anteriormente.

5.1 Paralelismo pelo dado de saída

O paralelismo pelo dado de saída foi avaliado utilizando o algoritmo da migração sísmica em CPU e GPU sem o Anthill, porque essa dimensão de paralelismo ocorre independente da utilização dele.

5.1.1 Usando CPU

Como descrito na seção 4.2.1.1, o escalonamento dos traços para as threads pode ser feito estaticamente ou dinamicamente, além de variar o número de traços escalonados por vez. Foram avaliados os dois tipos de escalonamentos e de 1 a 128 traços escalonados por vez. Os resultados para 8 processadores estão na figura 5.1 e vemos que o escalonamento dinâmico foi melhor, pois amenizou o desbalanceamento de carga das contribuições dos traços de entrada no bloco de saída. Além disso, para esse caso, o escalonamento de 16 traços por vez foi o que atingiu o melhor resultado, 391 milhões de amostras contribuídas por segundo.

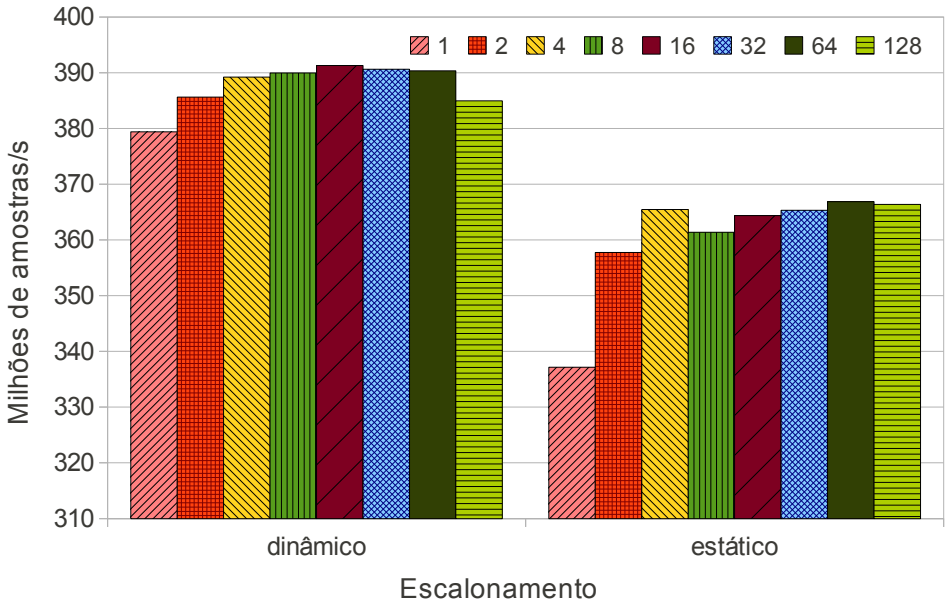


Figura 5.1: Comparação entre escalonamento dinâmico e estático de traços na CPU em função do número de traços por escalonamento.

O tempo de execução na CPU utilizando de 1 a 8 processadores está apresentado na figura 5.2. O tempo de execução reduziu de 39.868 segundos utilizando 1 processador para 7.180 segundos utilizando 8 processadores. Apesar da redução significativa esperava-se uma redução proporcional ao número de processadores utilizados, o que não aconteceu devido aos gargalos encontrados na arquitetura do processador.

Através da figura 5.3 vemos mais claramente que o desempenho apresentado não foi ótimo a partir da utilização de 4 processadores. O speedup encontrado foi de 3,95 para 5 processadores e 5,55 para 8 processadores.

A ferramenta *Intel VTune*^{TM}*Performance Analyser* foi utilizada para averiguar a causa desse resultado. A partir da análise de contadores em hardware selecionados verificamos que a causa não está relacionada à má utilização das caches L1 e L2 do

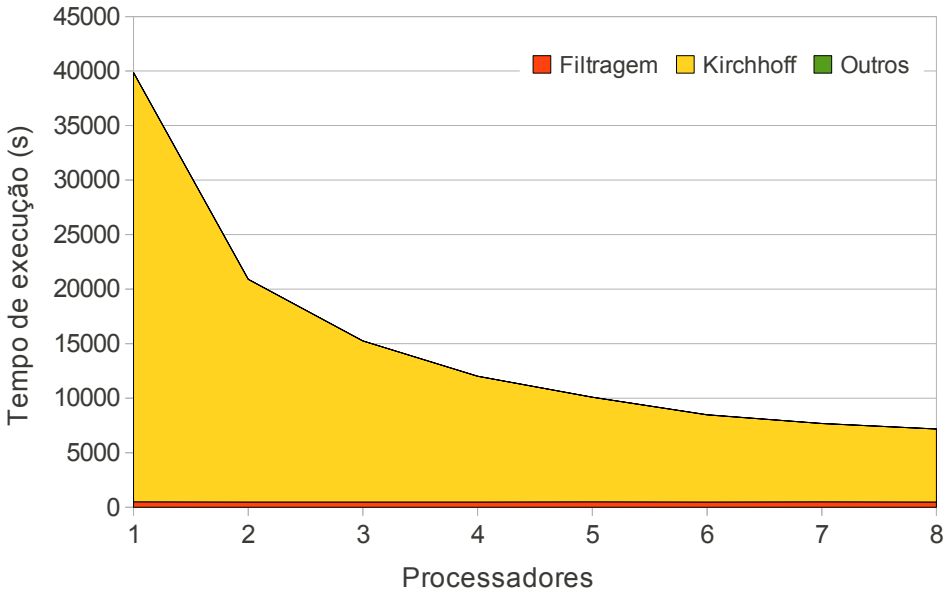


Figura 5.2: Tempo de execução da migração na CPU.

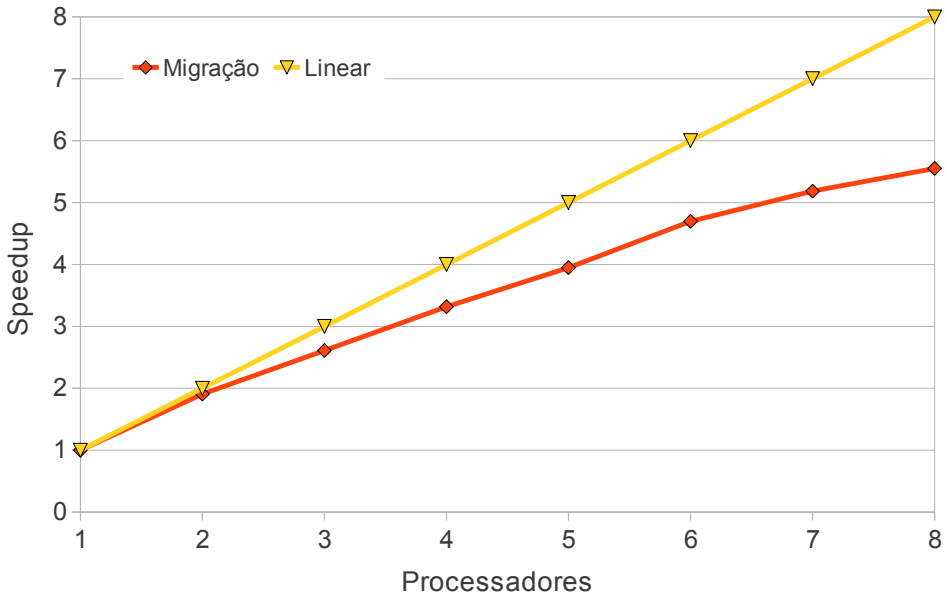


Figura 5.3: Speedup da migração na CPU.

processador. No entanto, contadores relativos à utilização do barramento mostraram que essa aplicação possui uma alta taxa de operações de memória[1], que sobrecarregam o acesso aos dados no subsistema de memória. Essa sobrecarga aumenta com a utilização de mais núcleos, pois, como apresentado na figura 4.1b, o barramento de acesso à memória é compartilhado por todos eles. Esses dados refletem a grande quantidade de dados temporários utilizados no algoritmo e que aumentam de acordo com o número

[1] *Store and load operations.*

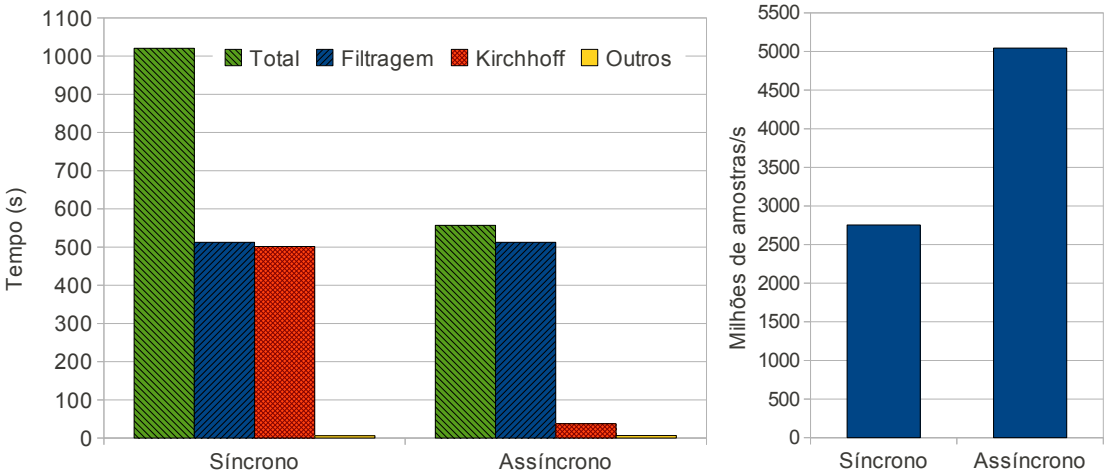
de núcleos.

Dessa maneira, a causa do speedup abaixo do ótimo é o gargalo na arquitetura do processador, na qual a utilização concorrente do subsistema de memória degrada o desempenho da aplicação. Esse problema de escalabilidade nesse processador também foi verificado em outros trabalhos, como em Williams et al. [2008] e Abdelkhalek et al. [2009].

Na figura 5.5 são apresentadas a quantidade de amostras contribuídas no tempo para esta execução com e sem o Anthill. O número de milhões de amostras contribuídas por segundo por processador cai de 70,48 quando usado 1 processador para 48,91 com 8 processadores, uma queda de 30% no desempenho por processador.

5.1.2 Usando GPU

A implementação do laço de migração na GPU alcançou um desempenho muito bom, o que reduziu bastante o tempo de execução do algoritmo.



(a) Tempo de execução (b) Desempenho

Figura 5.4: Cooperação CPU e GPU.

Na figura 5.4 são apresentados os tempos da execução na GPU. Para comparação com os resultados da CPU, foram utilizados os dados da execução síncrona, uma vez que na CPU a filtragem e o laço de migração são síncronos. Para a execução síncrona, o número de amostras contribuídas na GPU foi de 2.753 milhões por segundo, enquanto um núcleo da CPU obteve 70 milhões. A utilização da GPU reduziu o tempo total em 39 vezes, passando de 39.868 para 1.020 segundos.

5.2 Paralelismo de tarefas

Como descrito no capítulo 4, apenas a etapa mais significativa no tempo de execução, o laço de migração, foi implementada na GPU e, consequentemente, a etapa de filtragem dos traços continuou a ser feita na CPU, o que promoveu uma cooperação entre CPU e GPU. Além disso, com a implementação no Anthill, o paralelismo da filtragem pode ser aumentado a partir da utilização de mais instâncias.

Os resultados sem a utilização do Anthill estão na figura 5.4 e foram divididos em execução síncrona e assíncrona. Na execução síncrona o processamento de cada traço na GPU inicia somente após a filtragem terminar e a filtragem do traço seguinte somente inicia após o processamento do traço corrente na GPU terminar. Na execução assíncrona, as etapas de filtragem e do laço de migração acontecem simultaneamente. Além disso, a cópia dos traços filtrados para a GPU também é feita de maneira assíncrona utilizando mecanismo com duas áreas de memória temporária.

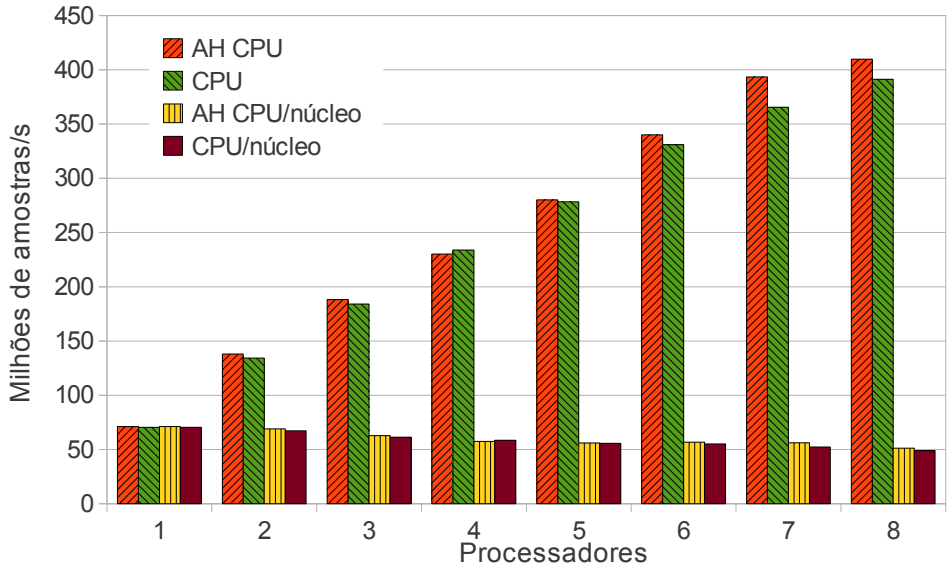


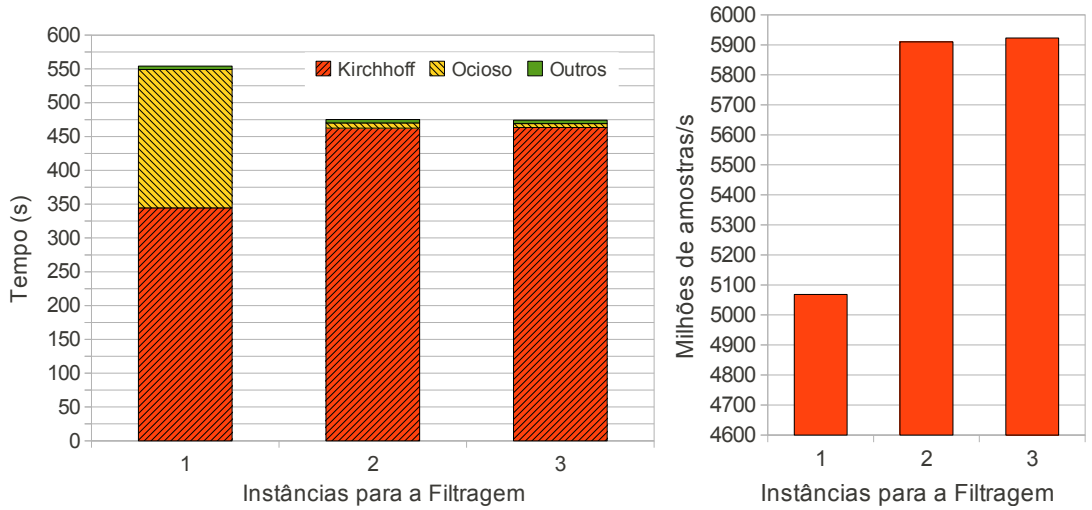
Figura 5.5: Milhões de amostras contribuídas para execuções com e sem o Anthill.

A execução síncrona terminou em 1.020 segundos, enquanto a assíncrona fez o mesmo em 556 segundos. Uma redução de 464 segundos no tempo total de execução. O número de amostras contribuídas no tempo chega a 5.045 milhões por segundo na execução assíncrona. Uma melhoria de 71 vezes com relação a um núcleo da CPU.

A utilização conjunta da CPU e da GPU resultou em aumento de 83% na execução com relação aos resultados da execução síncrona (2.753 para 5.045 milhões de amostras por segundo). Através da figura 5.4a, nota-se nos resultados da execução assíncrona que a sobreposição entre as tarefas é significativa, mas não é completa.

A execução da implementação utilizando o Anthill também foi avaliada para o número de instâncias da filtragem. O objetivo é alterar o número de instâncias para estabelecer a configuração que fornece o melhor desempenho.

Ao contrário dos resultados apresentados pela GPU, a sobreposição das etapas para o laço de migração executado na CPU não apresentou grandes melhorias. Isso ocorreu porque o tempo da filtragem com relação ao tempo do laço na CPU é muito pequeno.



(a) Tempo de execução

(b) Desempenho em milhões de amostras/s

Figura 5.6: Implementação no Anthill utilizando somente GPUs.

A figura 5.5 apresenta o desempenho comparativo entre as velocidades das implementações com e sem o Anthill utilizando apenas uma instância para a filtragem (um processador) e o filtro Kirchhoff utilizando múltiplas CPUs (um a sete). A diferença no número de amostras contribuídas por tempo entre as implementações somente é notada a partir da utilização de 6 processadores. Isso acontece porque o tempo destinado à filtragem se torna mais significativo na medida em que o laço de migração se torna mais rápido devido a paralelização. Essa diferença chega a 7,7% quando o número de processadores utilizado é 7, os valores são 56,21 milhões de amostras por segundo por processador no Anthill contra 52,21 milhões de amostras para quando a filtragem não está em paralelo. A implementação no Anthill não gerou nenhum processamento extra, pois o desempenho ficou no mínimo igual à configuração que não utilizou o Anthill.

No caso do filtro Kirchhoff utilizando somente GPUs o desempenho melhorou com o aumento do número de instâncias para a filtragem. Como mostrado anteriormente, a sobreposição entre as etapas não foi completa, o que deu margem para melhorias

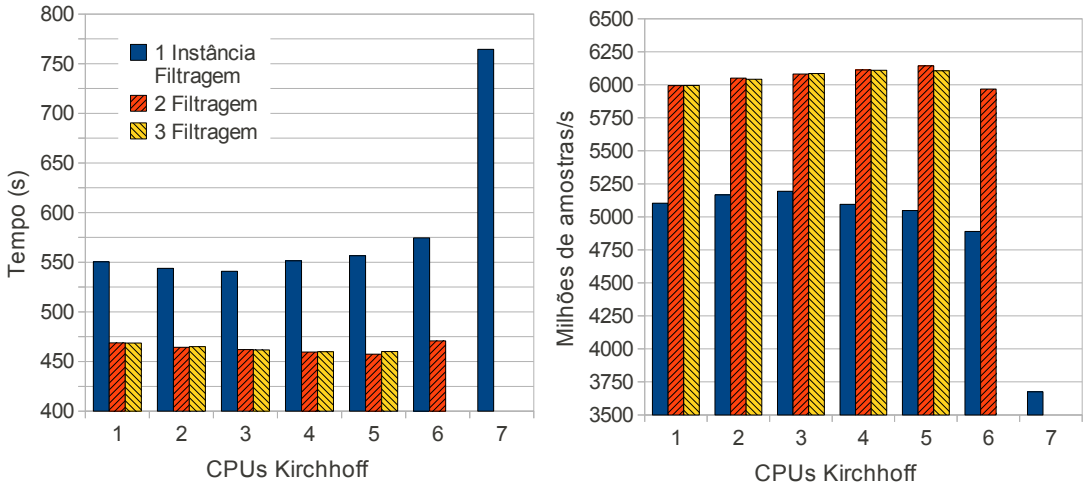
quando o número instâncias para a filtragem aumentou de 1 para 2.

A figura 5.6 apresenta os resultados para a execução utilizando de 1 a 3 instâncias para a filtragem. A partir do aumento de 1 para 2 instâncias houve uma melhoria no desempenho de 5.068 milhões de amostras por segundo para 5.910 milhões (aumento de 16,60%), e de 2 para 3 instâncias houve uma melhoria de apenas 0,22% (13 milhões).

O filtro Kirchhoff fica ocioso uma boa parte do tempo quando somente 1 instância é utilizada, como mostrado na figura 5.6a. A partir do aumento do número de instâncias o tempo ocioso se aproxima de zero.

5.3 Paralelismo pelo dado de entrada

O paralelismo pelo dado de entrada, descrito na seção 4.2.3, permite o processamento concorrente de múltiplos traços de entrada em diferentes unidades de processamento. Essa dimensão promoveu a cooperação eficiente entre CPU e GPU no laço de migração.



(a) Tempo de execução (b) Desempenho em milhões de amostras/s

Figura 5.7: Implementação no Anthill utilizando GPUs e CPUs no filtro Kirchhoff.

O resultado do desempenho alcançado pela cooperação entre CPU e GPU no filtro Kirchhoff está na figura 5.7. Cada nó possui 8 processadores, assim, foram executados testes variando o número de instâncias para a filtragem e o número de processadores utilizados no filtro Kirchhoff para a versão CPU. Foram utilizados de 1 a 7 processadores para o Kirchhoff e de 1 a 3 para a filtragem, lembrando que existe um processo responsável pela transferência de dados para a GPU.

O pior resultado apresentado foi utilizando 7 CPUs Kirchhoff (isto é, 7 processadores dedicados para a versão CPU do filtro Kirchhoff) e uma instância para a

filtragem, pois apenas uma instância não consegue atender à demanda da GPU e da CPU no laço de migração. A utilização de 6 CPUs Kirchhoff obteve resultados melhores que os de 7 CPUs Kirchhoff, ambos com uma instância para filtragem. Porque o processo responsável pela comunicação com a GPU disputa recursos computacionais com os outros processos, piorando o tempo de execução. Por esse mesmo motivo, não foram apresentados os resultados de 7 processadores Kirchhoff com 2 e 3 instâncias para a filtragem e 6 processadores Kirchhoff com 3 instâncias para a filtragem.

O melhor resultado encontrado foi utilizando 5 processadores para o Kirchhoff CPU e 2 instâncias para a filtragem, alcançando 6.143 milhões de amostras contribuídas por segundo. Esse resultado foi bem próximo do esperado, cerca de 99,24%, que era de 6.190 milhões de amostras por segundo (5.910 da GPU e 280 da CPU com 5 processadores). Portanto, a melhor configuração utilizou o número de processos (ou threads) igual à quantidade de processadores disponíveis, ou seja, 2 para a filtragem, 5 para o Kirchhoff CPU e 1 para a transferência de dados com a GPU, totalizando 8 processos para 8 núcleos. A utilização de 1 processador para a filtragem não foi suficiente para atender a demanda do filtro Kirchhoff e a utilização de 3 processadores não melhorou em nada o desempenho do algoritmo.

5.4 Análise comparativa

Nesta seção é feita uma análise comparativa das três dimensões de paralelismo avaliadas anteriormente. O objetivo dessa análise é avaliar a contribuição de cada dimensão para o desempenho total alcançado.

A análise comparativa utilizou como metodologia de avaliação a taxa de amostras contribuídas por segundo. Foram utilizadas seis configurações diferentes:

1. sequencial na CPU (CPU);
2. laço de migração paralelizado na CPU (8 CPUs);
3. execução síncrona do laço de migração na GPU (GPU sínc);
4. execução assíncrona do laço de migração na GPU (GPU assínc);
5. execução assíncrona do laço de migração na GPU com duas instâncias para a filtragem no ambiente Anthill (AH GPU);
6. execução assíncrona do laço de migração na GPU e na CPU com duas instâncias para a filtragem no ambiente Anthill (AH GPU+CPU).

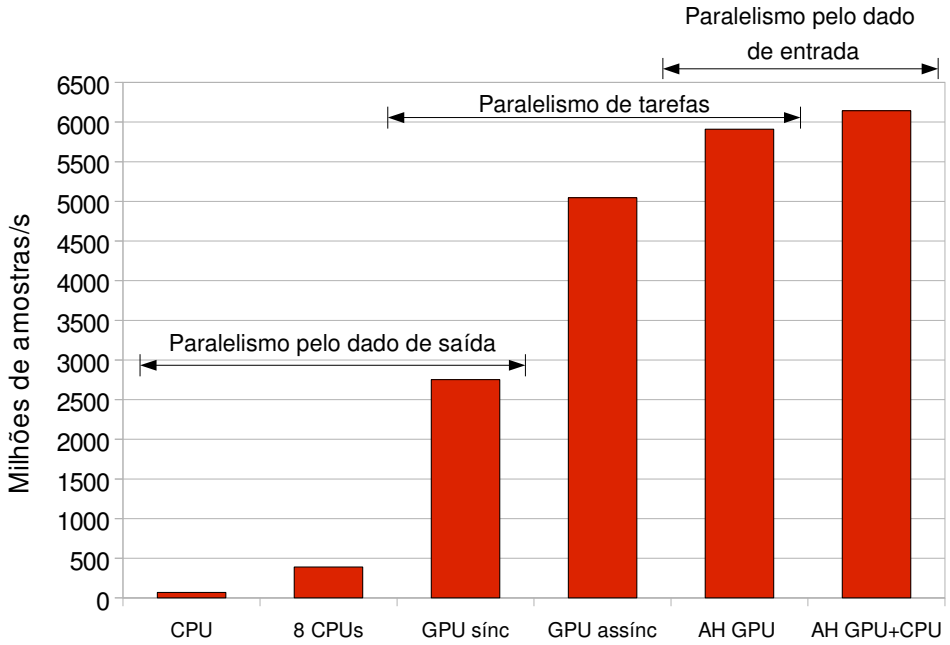


Figura 5.8: Desempenho alcançado a partir da utilização das três dimensões de paralelismo.

O paralelismo pelo dado de saída foi avaliado utilizando as configurações 1, 2 e 3. O paralelismo de tarefas utilizou as configurações 3, 4 e 5. E o paralelismo pelo dado de entrada utilizou as configurações 5 e 6. Algumas configurações foram utilizadas por mais de uma dimensão de paralelismo.

A figura 5.8 apresenta uma síntese do desempenho alcançado a partir de cada dimensão. A contribuição do paralelismo pelo dado de saída foi a aceleração de 5,55 vezes para 8 CPUs e de 39 vezes para a GPU com relação à execução sequencial na CPU.

O paralelismo de tarefas ocorre tanto para o laço de migração na CPU quanto na GPU, no entanto, a contribuição foi significativa somente para o caso da GPU. A assincronia de execução entre as tarefas aumentou de 2.753 para 5.045 milhões a taxa de amostras contribuídas por segundo. O aumento no paralelismo da etapa de filtragem, através do aumento do número de instâncias no ambiente Anthill, elevou esse ganho para 5.910 milhões.

Por último, a utilização conjunta entre CPU e GPU no filtro Kirchhoff alcançou 6.143 milhões de amostras por segundo, através da utilização do paralelismo pelo dado de entrada.

A implementação eficiente na GPU do paralelismo pelo dado de saída em conjunto com o paralelismo de tarefas elevaram significativamente o desempenho do algoritmo. O paralelismo pelo dado de entrada forneceu o desempenho bem próximo do esper-

ado, mas a discrepância de desempenho entre a CPU e a GPU não possibilitou uma contribuição maior por parte da CPU. Assim, a utilização dos núcleos da CPU em cooperação com a GPU mostrou-se bastante eficiente e foi a principal causa para o expressivo desempenho alcançado.

Capítulo 6

Conclusões e Trabalhos Futuros

O petróleo se tornou um recurso bastante importante e estratégico para todas as nações. A demanda por óleo não para de crescer desde o século XIX e sua indústria se tornou a mais importante e lucrativa do mundo, investindo bilhões de dólares por ano na produção e na procura por novas reservas.

Para localizar novas reservas são utilizados métodos geofísicos, sendo a migração sísmica de Kirchhoff um dos métodos mais comuns. A migração consome enormes recursos computacionais e processa terabytes de dados, o que motiva sempre testes de novas arquiteturas para acelerar o seu desempenho.

A utilização de aglomerados de computadores para o processamento sísmico já se tornou comum e, no momento, os trabalhos estão se concentrando no uso de aceleradores, como a GPU. As arquiteturas atuais estão se tornando cada vez mais heterogêneas com a utilização de vários processadores e aceleradores diferentes compostos de vários núcleos. É interessante notar que esses processadores e aceleradores por si só podem constituir arquiteturas heterogêneas. Esse ambiente altamente paralelo apresenta grandes desafios para o programador ao implementar aplicações que devam ser robustas à heterogeneidade para aproveitar ao máximo dos recursos computacionais existentes.

Nesse trabalho discutimos a paralelização da migração sísmica de Kirchhoff para ambientes heterogêneos distribuídos. Esse algoritmo foi implementado utilizando o modelo de programação filtro-fluxo identificado. Para atingir o objetivo de aproveitar todos os recursos presentes, foram implementadas três dimensões de paralelismo: pelo dado de saída, de tarefas e pelo dado de entrada. A cooperação entre CPU e GPU foi priorizada ao invés de acelerar o processamento com toda a execução na GPU. Assim, ela ficou responsável pela tarefa que melhor encaixou no seu modelo de programação, enquanto as outras tarefas da migração utilizaram apenas a CPU.

Na primeira dimensão, que corresponde ao *paralelismo pelo dado de saída*, as iterações do laço de migração são executadas simultaneamente nas unidades de processamento. No *paralelismo de tarefas*, correspondente à segunda dimensão, as principais etapas do algoritmo são executadas em paralelo. Já a terceira dimensão, através do *paralelismo pelo dado de entrada*, permite a migração concorrente dos traços de entrada em múltiplos dispositivos diferentes.

O ambiente Anthill foi utilizado para implementar essas três dimensões de paralelismo. Três filtros foram definidos: Filtragem, Kirchhoff e Escalonador. Os dois primeiros são responsáveis pelas etapas que dominam o tempo de execução da migração, a filtragem e o laço de migração, respectivamente. O filtro Escalonador é responsável por atribuir dinamicamente os blocos para as instâncias do filtro Kirchhoff.

Foi adotado um escalonamento de tarefas de grão fino permitindo a divisão do processamento de um mesmo bloco entre múltiplas unidades de execução heterogêneas. Essa abordagem permitiu evitar um desbalanceamento de carga entre as unidades e a operação delas a toda capacidade. O escalonamento não apresenta um resultado perfeito, mas como as tarefas possuem grão mais fino a penalidade pelo erro é pequena.

Os resultados mostraram uma redução enorme no tempo de execução com a utilização da GPU para a tarefa de maior demanda computacional. Além disso, a cooperação entre GPU e CPU tanto na dimensão de tarefas quanto na dimensão dos dados de entrada contribuíram ainda mais para melhorar o desempenho da aplicação. A assincronia utilizada na implementação em conjunto com a proposta pelo modelo filtro-fluxo também contribuíram muito para o desempenho alcançado. A implementação utilizando as três dimensões propostas alcançou 6.143 milhões de amostras contribuídas por segundo, uma aceleração de até 87 vezes com relação à execução sequencial na CPU.

Como trabalhos futuros vislumbramos a aplicação dessas dimensões de paralelismo e da cooperação entre as múltiplas unidades de execução em outras aplicações. Seria interessante avaliar essa cooperação para casos em que a aceleração da GPU com relação à CPU seja menor. Assim, a CPU teria muito mais utilidade e novas dimensões de paralelismo podem ser exploradas. Várias aplicações de processamento sísmico se encaixam nesse perfil, como: RTM [Abdelkhalek et al., 2009], Kirchhoff em profundidade e migração Omega-xy.

O escalonamento de traços de entrada filtrados entre CPU e GPU pode ser melhor estudado, utilizando ideias semelhantes às apresentadas em Teodoro et al. [2009b]. Como cada traço contribui de maneira diferente para cada bloco de traços de saída, a identificação de quais devem ser processados na CPU ou na GPU pode resultar em melhora no desempenho.

Além disso, a utilização de GPUs mais modernas, que apresentam recursos como

a multiprogramação entre as tarefas, pode demandar a criação de novas dimensões de paralelismo para reduzir ainda mais o tempo de execução. O uso de outros aceleradores, por exemplo a FPGA, possibilitam o estudo de outras otimizações, como a utilização de diferentes representações numéricas, que podem acelerar mais o algoritmo da migração.

Referências Bibliográficas

Abdelkhalek, R.; Calendra, H.; Coulaud, O.; Roman, J. & Latu, G. (2009). Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster. In *The 2009 High Performance Computing & Simulation - HPCS'09*, Leipzig Germany. Total.

Almasi, G. S.; McLuckie, T.; Bell, J.; Gordon, A. & Hale, D. (1992). Parallel distributed seismic migration. *Future Gener. Comput. Syst.*, 8(1-3):9--26.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483--485, New York, NY, USA. ACM.

Araujo, R.; Trielli, G.; Orair, G.; Meira Jr., W.; Ferreira, R. & Guedes, D. (2006). Partricluster: A scalable parallel algorithm for gene expression analysis. In *SBAC-PAD '06: Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing*, pp. 3–10, Washington, DC, USA. IEEE Computer Society.

Chapman, B.; Jost, G. & Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

Claerbout, J. F. (1985). *Imaging the earth's interior*. Blackwell Scientific Publications, Inc., Cambridge, MA, USA.

Dai, H. (2005). Parallel processing of prestack kirchhoff time migration on a pc cluster. *Computers & Geosciences*, 31(7):891--899.

Danek, T. (2009). Parallel and distributed seismic wave field modeling with combined linux clusters and graphics processing units. In *Geoscience and Remote Sensing Symposium,2009 IEEE International,IGARSS 2009*, volume 4, pp. IV–208 –IV–211.

Datta, K.; Murphy, M.; Volkov, V.; Williams, S.; Carter, J.; Oliker, L.; Patterson, D.; Shalf, J. & Yelick, K. (2008). Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the*

2008 ACM/IEEE conference on Supercomputing, pp. 1--12, Piscataway, NJ, USA. IEEE Press.

Dean, J. & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107--113.

Deschamps, J.-P.; Bioul, G. J. A. & Sutter, G. D. (2006). *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*. Wiley-Interscience.

Falola, T. & Genova, A. (2005). *The Politics of the Global Oil Industry: An Introduction*. Praeger Publishers.

Fang, W.; He, B.; Luo, Q. & Govindaraju, N. (2010). Mars: Accelerating mapreduce with graphics processors. In *Parallel and Distributed Systems, IEEE Transactions on*, volume PP, p. 1.

Ferreira, R. A.; Meira Jr., W.; Guedes, D.; Drummond, L. M. A.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araujo, R. & Ferreira, G. T. (2005). Anthill: A scalable run-time environment for data mining applications. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pp. 159--167, Washington, DC, USA. IEEE Computer Society.

Fu, H.; Osborne, W.; Clapp, R. G.; Mencer, O. & Luk, W. (2009). Accelerating seismic computations using customized number representations on fpgas. *EURASIP Journal on Embedded Systems*, 2009:1--13.

Garland, M.; Le Grand, S.; Nickolls, J.; Anderson, J.; Hardwick, J.; Morton, S.; Phillips, E.; Zhang, Y. & Volkov, V. (2008). Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13 –27.

Harris, M. (2008). Many-core gpu computing with nvidia cuda. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, p. 1, New York, NY, USA. ACM.

He, C.; Lu, M. & Sun, C. (2004). Accelerating seismic migration using fpga-based coprocessor platform. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 207 – 216.

Horowitz, M. & Dally, W. (2004). How scaling will change processor architecture. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pp. 132 – 133 Vol.1.

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A. & Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pp. 59--72, New York, NY, USA. ACM.

Kahle, J. A.; Day, M. N.; Hofstee, H. P.; Johns, C. R.; Maeurer, T. R. & Shippy, D. (2005). Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604.

Knight, W. (2005). Two heads are better than one [dual-core processors]. *IEEE Review*, 51(9):32 – 35.

Kruijf, M. d. & Sankaralingam, K. (2009). Mapreduce for the cell broadband engine architecture. *IBM Journal of Research and Development*, 53(5):10:1 –10:12.

Leon, A. & Sheahan, D. (2006). The u1trasparc t1: A power-efficient high-throughput 32-thread sparc processor. In *Solid-State Circuits Conference, 2006. ASSCC 2006. IEEE Asian*, pp. 27 –30.

Liu, L.; Li, Z. & Sameh, A. H. (2008). Analyzing memory access intensity in parallel programs on multicore. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pp. 359--367, New York, NY, USA. ACM.

Madisetti, V. & Messerschmitt, D. (1991). Seismic migration algorithms on parallel computers. *Signal Processing, IEEE Transactions on*, 39(7):1642 –1654.

Marathe, J.; Mueller, F. & de Supinski, B. R. (2006). Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques. *ACM Transactions on Architecture and Code Optimization (TACO)*, 3(4):390--423.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):114--117.

Moreira, J.; Salapura, V.; Almasi, G.; Archer, C.; Bellofatto, R.; Bergner, P. & Bickford, R. (2007). The blue gene/l supercomputer: A hardware and software story. *International Journal of Parallel Programming*, 35:181–206.

Nichols, B.; Buttlar, D. & Farrell, J. P. (1996). *Pthreads programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Owens, J.; Houston, M.; Luebke, D.; Green, S.; Stone, J. & Phillips, J. (2008). Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899.

Panetta, J.; de Souza; da Cunha; Roxo; Sinedino; Pedrosa; Romanelli; Monnerat, L. R.; Carneiro, L. T.; de Albrecht, C. H. & A10 (2007). Computational characteristics of production seismic migration and its performance on novel processor architectures. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pp. 11--18.

Panetta, J.; Teixeira, T.; de Souza Filho, P. R. P.; da Cunha Finho, C. A.; Sotelo, D.; da Motta, F. M. R.; Pinheiro, S. S.; Junior, I. P.; Rosa, A. L. R.; Monnerat, L. R.; Carneiro, L. T. & de Albrecht, C. H. B. (2009). Accelerating kirchhoff migration by cpu and gpu cooperation. In *Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '09, pp. 26–32, Washington, DC, USA. IEEE Computer Society.

Patterson, D. A. & Hennessy, J. L. (2008). *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Peng, L.; Peir, J.-K.; Prakash, T.; Chen, Y.-K. & Koppelman, D. (2007). Memory performance and scalability of intel's and amd's dual-core processors: A case study. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*, pp. 55 –64.

Ranger, C.; Raghuraman, R.; Penmetsa, A.; Bradski, G. & Kozyrakis, C. (2007). Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 13 –24.

Robinson, E. (1977). Geophysical exploration for petroleum and natural gas. *Geoscience Electronics, IEEE Transactions on*, 15(1):3 –11.

Santos, W.; Teixeira, T.; Machado, C.; Meira, W.; Da Silva, A.; Ferreira, D. & Guedes, D. (2007). A scalable parallel deduplication algorithm. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pp. 79–86.

Savage, J. E. & Zubair, M. (2008). A unified model for multicore architectures. In *IFMT '08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, pp. 1--12, New York, NY, USA. ACM.

Teodoro, G.; Fireman, D.; Guedes, D.; Jr., W. M. & Ferreira, R. (2008). Achieving multi-level parallelism in the filter-labeled stream programming model. In *ICPP*

'08: Proceedings of the 2008 37th International Conference on Parallel Processing, pp. 287–294, Washington, DC, USA. IEEE Computer Society.

Teodoro, G.; Hartley, T. D. R.; Catalyurek, U. & Ferreira, R. (2010). Run-time optimizations for replicated dataflows on heterogeneous environments. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 13--24, New York, NY, USA. ACM.

Teodoro, G.; Sachetto, R.; Fireman, D.; Guedes, D. & Ferreira, R. (2009a). Exploiting computational resources in distributed heterogeneous platforms. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pp. 83 –90.

Teodoro, G.; Sachetto, R.; Sertel, O.; Gurcan, M.; Meira, W.; Catalyurek, U. & Ferreira, R. (2009b). Coordinating the use of gpu and cpu for improving performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pp. 1 –10.

Teodoro, G.; Tavares, T.; Ferreira, R.; Kurc, T.; Meira Jr., W.; Guedes, D.; Pan, T. & Saltz, J. (2006). A run-time system for efficient execution of scientific workflows on distributed environments. In *SBAC-PAD '06: Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing*, pp. 81–90, Washington, DC, USA. IEEE Computer Society.

Tian, C.; Zhou, H.; He, Y. & Zha, L. (2009). A dynamic mapreduce scheduler for heterogeneous workloads. In *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*, pp. 218 –224.

Veloso, A.; Meira, Jr., W.; Ferreira, R.; Neto, D. G. & Parthasarathy, S. (2004). Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. In *PKDD '04: Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pp. 422--433, New York, NY, USA. Springer-Verlag New York, Inc.

Wilkes, M. V. (2001). The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News*, 29(1):2--7.

Williams, S.; Carter, J.; Oliker, L.; Shalf, J. & Yelick, K. (2008). Lattice boltzmann simulation optimization on leading multicore platforms. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1 –14.

Wulf, W. & McKee, S. A. (1994). Hitting the memory wall: Implications of the obvious. Technical Report CS-94-48, Department of Computer Science, University of Virginia, Charlottesville, VA, USA.

Yilmaz, O. (1987). *Seismic Data Processing*, volume 2 of *Investigations in Geophysics*. Society of Exploration Geophysicists, Tulsa.

Zhao, C.; Yan, H.; Shi, X. & Wang, L. (2008). Decf: A coarse-grained data-parallel programming framework for seismic processing. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 3, pp. 454 –460.