UNIVERSIDADE FEDERAL FLUMINENSE

**RODRIGO ALVES PRADO DA SILVA**

# Análise de uma Aplicação Vetorizada de Migração Kirchhoff Pré-empilhamento em Tempo Usando Diversos Paradigmas de Paralelização e em Diferentes Ambientes de Virtualização

NITERÓI

2018

UNIVERSIDADE FEDERAL FLUMINENSE

**RODRIGO ALVES PRADO DA SILVA**

# Análise de uma Aplicação Vetorizada de Migração Kirchhoff Pré-empilhamento em Tempo Usando Diversos Paradigmas de Paralelização e em Diferentes Ambientes de Virtualização

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Sistemas de Computação

Orientadora:
Lúcia Maria de Assumpção Drummond

Coorientadora:
Cristiana Barbosa Bentes

NITERÓI

2018

Ficha catalográfica automática - SDC/BEE

S586a Silva, Rodrigo Alves Prado da
Análise de uma Aplicação Vetorizada de Migração Kirchhoff Pré-empilhamento em Tempo Usando Diversos Paradigmas de Paralelização e em Diferentes Ambientes de Virtualização / Rodrigo Alves Prado da Silva ; Lúcia Maria de Assumpção Drummond, orientador ; Cristiana Barbosa Bentes, coorientador. Niterói, 2018.
75 f.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2018.

DOI: http://dx.doi.org/10.22409/PGC.2018.m.10269375740

1. Vetorização. 2. Paralelização. 3. Ambientes Virtuais. 4. Kirchhoff. 5. Produção intelectual. I. Título II. Drummond,Lúcia Maria de Assumpção, orientador. III. Bentes, Cristiana Barbosa, coorientador. IV. Universidade Federal Fluminense. Escola de Engenharia.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

RODRIGO ALVES PRADO DA SILVA

Análise de uma Aplicação Vetorizada de Migração Kirchhoff Pré-empilhamento em Tempo Usando Diversos Paradigmas de Paralelização e em Diferentes Ambientes de Virtualização

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Sistemas de Computação

Aprovada em 9 de agosto 2018.

BANCA EXAMINADORA

Profa. Lúcia M. A. Drummond - Orientadora, UFF

Profa. Cristiana B. Bentes - Coorientadora, UERJ

Profa. Simone L. Martins, UFF

Profa. Luciana B. Arantes, UPMC

Niterói

2018

*À minha família.*

# Agradecimentos

Aos meus pais Maria de Fátima Alves Prado da Silva e Fernando Prado da Silva (in memoriam), pela criação, ensinamentos, amor e carinho, junto com meus irmãos Bruno Silva e Érico Silva.

À minha orientadora Professora Lúcia Maria de Assumpção Drummond, que me acompanhou durante todos os momentos deste trabalho, fornecendo muitos esclarecimentos científicos e conhecimentos profissionais. Além de ser uma excelente amiga e auxiliar na minha carreira acadêmica.

À minha coorientadora Professora Cristiana Barbosa Bentes, que me apoiou técnica e emocionalmente. Este trabalho não estaria concluído a tempo sem sua participação assertiva.

Aos meus amigos da vida Jânio Mororó, Vinícius Pereira, Vitor Vidal, Rodrigo Cruz, Kelly Bentes e Maísa Fonseca que estão sempre presentes e trazendo momentos maravilhosos de alegria e felicidade.

Aos meus companheiros de pós-gradução, Maicon Melo, Luan Teylo, Leonardo Pio, Mayara Carreiro e outros, que por ventura tenha esquecido, pelas boas risadas e receptividade ao longo destes anos.

Às professoras Simone L. Martins e Luciana B. Arantes pela presença na banca examinadora.

À CAPES pela bolsa de mestrado sem a qual não seria possível esse trabalho.

À UFF por acrescentar no meu crescimento profissional e pessoal.

# Resumo

Pre-stack Kirchhoff Time Migration (PKTM) é uma parte central do processo de exploração de petróleo. Como o PKTM é computacionalmente intensivo, trabalhos anteriores propuseram versões paralelas do PKTM para memória compartilhada, memória distribuída e aceleradores para reduzir o tempo de execução. Os processadores modernos, no entanto, têm capacidade de processamento de vetores que geralmente são negligenciados na otimização de desempenho. Neste trabalho, propomos uma análise detalhada do uso da vetorização em três versões paralelas do PKTM: uma implementação de memória compartilhada com OpenMP (Open Multi-Processing), uma implementação de passagem de mensagem com MPI (Message Passing Interface) e uma implementação híbrida que usa o OpenMP e o MPI. Como a computação em nuvem provou ser uma alternativa promissora para executar aplicativos HPC, também avaliamos o impacto negativo que a sobrecarga introduzida pela camada de virtualização pode ter nas execuções implementadas do PKTM. Assim, analisamos o desempenho dessas versões sequenciais/paralelas e vetorizadas do PKTM quando executadas em tecnologias de virtualização comumente usadas em nuvens: KVM (Máquina virtual baseada em *kernel*), Docker e Singularity. Nossos testes experimentais mostraram que a combinação da vetorização com a paralelização apresentou os melhores resultados, produzindo *speedups* de até 24,9 quando executado com 6 núcleos. Com relação aos *overheads* de virtualização, observamos que o KVM apresentou uma pequena degradação no desempenho enquanto que os ambientes baseados em contêiner apresentavam um desempenho quase nativo.

**Palavras-chave**: Kirchhoff, paralelismo, vetorização, virtualização.

# Abstract

Pre-stack Kirchhoff Time Migration (PKTM) is a central part of the oil exploration process. Since PKTM is computationally intensive, previous works have proposed parallel versions of PKTM for shared memory, distributed memory and accelerators to reduce the execution time. Modern processors, however, have vector processing abilities that are usually neglected in performance optimization. In this work, we propose a detailed analysis of the use of vectorization in three parallel versions of PKTM: a shared memory implementation with OpenMP (Open Multi-Processing), a message passing implementation with MPI (Message Passing Interface) and a hybrid implementation that uses both OpenMP and MPI. As cloud computing has proven to be a promising alternative to execute HPC applications, we also evaluated the negative impact that the overhead introduced by virtualization layer may have on the implemented PKTM executions. Thus, we analyzed the performance of these sequential/parallel and vectorized versions of PKTM when executed upon virtualization technologies commonly used on clouds: KVM (Kernel-based Virtual Machine), Docker and Singularity. Regarding the vectorization analysis, our experimental tests showed that the combination of the vectorization with the parallelism offered the best results, producing a speedup up to 24.9 when executing in a processor with six cores. Concerning the virtualization overheads, we observed that KVM showed a small degradation in performance while container-based environments presented near native performance.

**Keywords**: Kirchhoff, Parallelism, Vectorizing, Virtualization.

# Lista de Figuras


3.1 Seção sísmica antes (a) e após a (b) migração. . . . . 19

4.1 Virtualização baseada em *hypervisor*. . . . . 24

4.2 Virtualização baseada em contêiner . . . . . 25

6.1 Soma de dois vetores de pontos flutuantes usando AVX2 . . . . . 36

7.1 *Speedups* das versões paralelas e vetorizadas automaticamente do PKTM. . 59

7.2 *Speedups* das versões paralelas e vetorizadas manualmente do PKTM. . . . 60

7.3 Média dos tempos de execução da versão OpenMP do PKTM com 6 *threads* no Host, KVM, Docker e Singularity. . . . . 62

7.4 Média dos tempos de execução da versão MPI PKTM com 6 processos no Host, KVM, Docker e Singularity. . . . . 63

# Lista de Tabelas


6.1 Exemplos de funções *intrinsics* (AVX2) . . . . . 37

6.2 Resumo das vetorizações automáticas pelo o ICC 17. . . . . 41

6.3 Resumo das vetorizações automáticas das versões otimizadas pelo ICC 17. 43

6.4 Resumo de vetorização para o ICC 18. . . . . 43

7.1 Geometria do conjunto de dados sísmicos . . . . . 47

7.2 Parâmetros de compilação utilizados para cada versão do PKTM. . . . . 48

7.3 Versões dos softwares utilizadas nos experimentos com o compilador ICC versão 17. . . . . 51

7.4 Versões dos softwares utilizadas nos experimentos com o compilador ICC versão 18. . . . . 51

7.5 Tempo, *Speedup* da Vetorização e *Speedup* Geral obtidos. . . . . 52

7.6 Tempo de execução e *speedups* obtidos com OpenMP, MPI e implementações híbridas do PKTM utilizando uma carga leve de processamento. . . . . 55

7.7 Tempo de execução e *speedups* obtidos com OpenMP, MPI e implementações híbridas do PKTM utilizando uma carga moderada de processamento. . . . 56

7.8 Tempo de execução e *speedups* obtidos com OpenMP, MPI e implementações híbridas do PKTM utilizando uma carga pesada de processamento. . . . . 57

7.9 Tempos médios de execução de todas as versões do PKTM quando executados em uma mesma máquina física (Host) e em ambientes virtuais KVM, Docker e Singularity. . . . . 61

# Sumário

**1 Introdução** **9**

1.1 Organização do Texto . . . . . 12

**2 Trabalhos Relacionados** **13**

2.1 PKTM Paralelo e/ou Vetorizado . . . . . 13

2.2 Análise de Desempenho em Ambientes Virtualizados . . . . . 15

**3 Algoritmo da Migração Kirchhoff Pré-empilhamento em Tempo (PKTM)** **18**

**4 Ambientes Virtuais** **23**

4.1 *Hypervisor* . . . . . 24

4.1.1 KVM . . . . . 26

4.2 Contêiner . . . . . 27

4.2.1 Docker . . . . . 27

4.2.2 Singularity . . . . . 28

**5 Implementações Paralelas do PKTM** **30**

5.1 PKTM Paralelo com OpenMP . . . . . 31

5.2 PKTM Paralelo com MPI . . . . . 31

5.3 PKTM Paralelo Híbrido com OpenMP e MPI . . . . . 33

**6 Vetorização das Implementações Sequencial e Paralelas do PKTM** **35**

6.1 *Advanced Vector Extensions 2* (AVX2) . . . . . 35

6.2 Vetorização Manual do PKTM . . . . . 37

6.2.1 Detalhes de Implementação . . . . . 39

6.3 Vetorização Automática do PKTM . . . . . 40

6.3.1 Compilação com ICC Versão 17 . . . . . 40

6.3.1.1 Otimizações Para Vetorização Automática dos Códigos Sequencial e com MPI . . . . . 41

6.3.2 Compilação com ICC versão 18 . . . . . 43

**7 Resultados Experimentais 46**

7.1 Dados de Entrada . . . . . 46

7.2 Ambiente de Execução . . . . . 47

7.3 Experimentos sem Virtualização Usando o Compilador ICC Versão 17 . . . 51

7.4 Comparação entre Execuções em Ambientes Virtualizados e não Virtualizados 53

7.4.1 Análise de Desempenho da Vetorização . . . . . 54

7.4.2 Análise de Desempenho do PKTM em Ambientes Virtuais . . . . . 59

**8 Conclusão e Trabalhos Futuros 64**

8.1 Considerações Finais . . . . . 64

8.2 Trabalhos Futuros . . . . . 65

**Referências 66**

# Capítulo 1

# Introdução

O primeiro passo na exploração de petróleo e gás consiste em realizar um processamento nos dados sísmicos que foram adquiridos na área de interesse. Esse processo fornece imagens da subsuperfície do solo marítimo a fim de permitir a interpretação das estruturas geológicas presentes nessa área. A partir do conhecimento dessas estruturas, é possível identificar áreas onde o petróleo pode ser encontrado e extraído. Um dos passos executados nesse processo é a migração sísmica; considerada o passo central do processo. A migração sísmica produz uma imagem mais fiel das estruturas geológicas encontradas em subsuperfície ao colapsar as difrações hiperbólicas e mover as estruturas mergulhantes para suas reais posições.

Um dos métodos mais populares de migração sísmica é a Migração Kirchhoff Pré-empilhamento em Tempo (*Pre-stack Kirchhoff Time Migration* ou PKTM) que baseia-se no procedimento de soma de difrações. Este método, o PKTM, é amplamente utilizado por sua simplicidade, eficiência, confiabilidade e flexibilidade de E/S [73] [31]. No entanto, esta migração é computacionalmente intensiva, ou seja, mesmo em supercomputadores, sua execução pode levar semanas ou meses para ser concluída [64]. Em outras palavras, a execução do PKTM normalmente processa terabytes de dados e usa vários Gflops-mês de computação antes que as imagens geradas possam ser interpretadas por especialistas. Por esse motivo que as técnicas de sistemas distribuídos e programação paralela têm sido aplicadas na implementação do PKTM objetivando reduzir o seu tempo de execução [60].

Muitos trabalhos propõem o uso de aceleradores computacionais como as *Graphics Processing Units* (GPU) ou *Field-programable Gate Array* (FPGA) para reduzir o tempo de execução do algoritmo PKTM dado suas características que permitem o paralelismo de dados [60], a exemplos de [73], [64], [56] e [67]. Nesses casos, a CPU é usada somente para o controle das tarefas, como mover os dados da memória para esses dispositivos. Embora essa

abordagem com GPUs e FPGAs alcancem melhoras de desempenho significativas, alguns trabalhos, como o apresentado por Panetta *et al.* [56], têm mostrado que a combinação do poder computacional dos processadores de vários núcleos trabalhando em conjunto com esses aceleradores melhoram o desempenho computacional do PKTM.

A maioria dos trabalhos anteriores em reduzir o tempo de execução do PKTM, no entanto, negligenciou um recurso importante comumente disponível em processadores modernos: a unidade de processamento vetorial. Também conhecida como unidade funcional de processamento *Single Instruction Multiple Data* (SIMD), a unidade vetorial pode ser usada para explorar o paralelismo de dados de baixa granularidade. É possível processar, ao mesmo tempo, elementos vetoriais distintos usando um conjunto de instruções específicas de hardware [44]. Cada fabricante de processadores implementa seu próprio conjunto de instruções SIMD para fornecer a capacidade de processamento vetorial. Nos processadores Intel, o conjunto de instruções SIMD é o SSE (*Streaming SIMD Extensions*) e o AVX (*Advanced Vector Extensions*), nos processadores AMD é o 3DNow e, nos processadores PowerPC é o Altivec [57] [68]. Observe que o Knights Landing Intel Xeon Phi, por exemplo, está equipado com duas unidades de vetores por núcleo, onde cada uma dessas unidades permite processar, simultaneamente, um vetor de 512 bits. Desta forma, um único núcleo pode executar, ao mesmo tempo, 32 operações de ponto flutuante de precisão simples ou 16 operações de ponto flutuante de precisão dupla.

Portanto uma estratégia para melhorar o desempenho do PKTM é explorar as instruções SIMD disponíveis nos processadores [26] [75]. Uma solução que combina a capacidade de vetorização das CPUs com a capacidade de computação paralela dos processadores com múltiplos núcleos ou *clusters* de computadores pode aumentar o desempenho do PKTM. Outra vantagem no uso dessas instruções SIMD é a aceleração imediata do PKTM nos casos os quais existam CPUs convencionais nas estações de trabalho de um dado *cluster*. Então, um aumento expressivo no desempenho do PKTM pode ser alcançado sem maiores investimentos em hardware.

Esta capacidade de vetorização pode ser explorada através de um processo de vetorização automática fornecido por compiladores, tais como *Intel C/C++ Compiler* (ICC), *GNU Compiler Collection* (GCC) ou *IBM C Compiler* (XLC) [47]. Quando a opção de vetorização automática é ativada, o compilador procura estruturas de código que podem ser vetorizadas, ou seja, procura laços sem controle de fluxo ou dependência de dados entre elementos de vetor [19]. Portanto, a vetorização automática é transparente e não requer nenhum esforço adicional de programação a ser aplicado no código-fonte [50] [62] [19]. No

entanto, como os compiladores geralmente tomam decisões mais conservadoras, os códigos automaticamente vetorizados podem não usar todo o potencial fornecido pela unidade de processamento vetorial. Além da vetorização automática realizada pelo compilador, existe a possibilidade da vetorização manual a qual o programador emprega funções específicas que utilizam em sua implementação instruções específicas do conjunto de instruções SIMD do processador, a exemplo do conjunto de instruções AVX2 (*Advanced Vector Extensions 2*) [1, 32].

Embora o trabalho descrito em [5] tenha analisado o custo-benefício de adotar diferentes abordagens de vetorização para acelerar o PKTM sequencial, o mesmo desconsiderou o impacto que a paralelização do PKTM, com modelos de memória compartilhada e troca de mensagens, pode ter na vetorização do código-fonte. Assim, nesta dissertação, são investigados como o uso desses modelos de programação, utilizando as bibliotecas OpenMP e MPI, podem afetar a vetorização do código. São avaliados os efeitos da vetorização manual e automática sobre três novas versões do PKTM, uma paralelizada com OpenMP, outra com MPI e uma terceira, híbrida, que combina as duas anteriores. São apresentadas algumas técnicas de otimização que foram aplicadas na versão sequencial e paralelizada com MPI com a finalidade de aumentar a quantidade de laços automaticamente vetorizados.

Adicionalmente ao desempenho obtido com a vetorização das versões paralelas, este trabalho investigou a execução do PKTM em ambientes virtualizados. A computação em nuvem provou ser uma alternativa promissora para executar aplicações de computação de alto desempenho (HPC) [3], portanto são avaliados os impactos negativos que o *overhead* introduzido pelas camadas de virtualização, que normalmente oferecem suporte a serviços em nuvem, podem ter sobre as diferentes implementações do PKTM. Assim, foi analisado o desempenho de diferentes versões paralelas e vetorizadas do PKTM quando executadas em tecnologias de virtualização comumente usadas em nuvens: KVM (Máquina virtual baseada em *kernel*), Docker e Singularity. Os testes experimentais permitiram investigar qual tecnologia de virtualização impacta menos negativamente o desempenho dessas versões paralelas e vetorizadas do PKTM.

As principais contribuições deste trabalho são:

1. Vetorização manual e automática de três versões paralelas do PKTM. Uma usando o modelo de memória compartilhada, outra usando o modelo de troca de mensagens e uma terceira que combina ambos os modelos.

2. Identificação de alguns problemas que previnem o compilador de vetorizar certas

construções de código automaticamente.

3. Implementação de duas técnicas de otimização para ajudar o compilador a vetorizar automaticamente certos laços do algoritmo do PKTM apesar dos problemas identificados.

4. Uma análise detalhada de desempenho das abordagens de vetorização manual e automática aplicadas às versões paralelas.

5. Uma avaliação dos *overheads* introduzidos pelas tecnologias de virtualização (KVM, Docker e Singularity) na execução das versões paralelas e vetorizadas do PKTM.

Nossos resultados mostraram que, em relação às versões PTKM sequenciais, todas as tecnologias de virtualização testadas introduziram pequenos *overheads*, apresentando diferenças em tempos de execução de no máximo 2% em média, quando comparadas com a execução sem virtualização. No entanto, percebeu-se que a tecnologia baseada em *hypervisor* KVM apresentou um maior impacto negativo do que as tecnologias de virtualização baseadas em contêineres Docker e Singularity.

## 1.1 Organização do Texto

O restante desta dissertação está organizado como descrito a seguir. O Capítulo 2 mostra o estado da arte no que diz respeito a implementações paralelas do PKTM, implementações vetorizadas e sobre o impacto das camadas de virtualização.

O Capítulo 3 explica o problema da migração sísmica em mais detalhes, apresentando o algoritmo sequencial utilizado, a partir do qual, as versões paralelas e/ou vetorizadas são derivadas.

O Capítulo 4 descreve os ambientes virtuais utilizados para a análise de desempenho das execuções das aplicações do PKTM.

O Capítulo 5 apresenta as implementações paralelas do algoritmo PKTM.

O Capítulo 6 apresenta as versões automática e manualmente vetorizadas que são utilizadas para a comparação entre as versões de implementação do PKTM.

O Capítulo 7 apresenta os testes executados e seus resultados computacionais.

Finalmente, no Capítulo 8 são apresentadas as conclusões desta dissertação e os possíveis trabalhos futuros sugeridos.

# Capítulo 2

# Trabalhos Relacionados

Este capítulo apresenta o levantamento bibliográfico efetuado ao longo do trabalho de pesquisa para esta dissertação. Os artigos apresentados foram divididos em duas seções. A seção 2.1 apresenta os artigos focados em melhorar o desempenho do PKTM através de soluções paralelas e/ou vetorizadas. Já a seção 2.2 apresenta os trabalhos envolvendo avaliação de ambientes virtualizados.

## 2.1 PKTM Paralelo e/ou Vetorizado

Como o PKTM é uma aplicação de grande importância na área de exploração de petróleo, há uma série de esforços para reduzir seu tempo de execução. Em He *et al.* [31], é proposto o uso de um coprocessador de plataforma reconfigurável baseado na tecnologia FPGA, integrando um número de módulos aritméticos paralelos em um único chip FPGA. Neste trabalho, entretanto, a CPU é utilizada apenas para operações de E/S.

Em Dai *et al.* [18], foi introduzida uma versão PKTM paralela que executa num *cluster* com processadores com um núcleo onde cada traço de saída é processado individualmente por cada nó do *cluster*. Esta versão utiliza a biblioteca MPI para a divisão do trabalho. Cada traço de entrada é enviado para ser migrado pelos processos trabalhadores. Diferente da paralelização proposta nesta dissertação em que todos os traços de um *offset* são migrados por cada processo trabalhador. O trabalho de Dai *et al.* desconsidera também o uso de paralelismo de memória compartilhada e ignora o paralelismo das instruções vetoriais.

Em Yerneni *et al.* [75], estudou-se a demanda computacional do PKTM. Foi desenvolvida uma versão paralela projetada para executar num *cluster* de estações de

trabalho. Através da reestruturação do código, utilizando paralelismo de E/S, sobrepondo computação com comunicação, atingiu-se tanto eficiência quanto *speedups* satisfatórios.

Em Panetta *et al.* [55], são descritos experimentos em um *cluster* de processadores de vários núcleos, no supercomputador Blue Gene e em um *cluster* de Sony Playstation 3. Porém, este trabalho não detalha o algoritmo utilizado, apenas informa que a biblioteca MPI é utilizada para o paralelismo e especifica uma granularidade variável no envio dos dados de entrada, de forma que um bloco contendo uma quantidade de traços de entrada é enviado a processos trabalhadores. Possibilitando, dessa forma, o controle da quantidade de mensagens trocadas entre os processos. Logo, o *overhead* desta troca de mensagens depende do tamanho do bloco contendo os traços a serem migrados. Apesar disso, não é considerado o uso do modelo de programação com memória compartilhada, assim como não é avaliada a vetorização do código-fonte da aplicação.

Há também uma série de trabalhos que exploram o poder computacional da GPU para acelerar o processamento do PKTM. Em Shi *et al.* [64], foi descrita uma versão GPU do PKTM em que pode-se aumentar o número de *threads* do *kernel* da GPU. O algoritmo proposto usa faixas de memória para sobrepor computação na GPU e CPU. Em Panetta *et al.* [56], é apresentada uma solução que combinada o poder da GPU com o da CPU. Nesta estratégia, o poder computacional da CPU foi empregado para executar operações de leitura e filtragem dos traços de entrada, além de executar as operações de controle e E/S tradicionais. Em Sun e Shi [67], foi introduzida uma implementação do PKTM otimizada para a GPU. Mostrou-se como se obter um código competitivo utilizando OpenCL. Em Xu *et al.* [73], estudaram-se os efeitos do uso de um modelo de programação baseado em diretivas de alto nível, através da API OpenACC que trabalha utilizando aceleradores, para paralelizar o PKTM em arquiteturas utilizando a GPU. Além disso, discute-se sobre quais mecanismos podem efetivamente se aproveitar da capacidade de computação e a hierarquia de memória da GPU. Todos esses trabalhos, porém, não exploram a capacidade paralela dos vários núcleos da CPU e das suas unidades vetoriais.

Com relação a explorar a vetorização para acelerar o PKTM, há poucos trabalhos na literatura. Em Melo *et al.* [4] e [5], foram avaliadas abordagens distintas de vetorização para acelerar o PKTM. Sendo a abordagem mais significativa a que utiliza funções intrínsecas (*intrinsics*), que são funções de baixo nível para acessar as instruções vetoriais do processador. No entanto, os autores consideraram apenas a versão sequencial desse método de migração, desconsiderando dessa forma o uso eficiente de todos os núcleos do processador e desconsiderando também a possibilidade de escalar o processamento para nós

de um *cluster*. O trabalho [65], derivado desta dissertação, avalia os efeitos da vetorização em duas versões paralelas da PKTM, uma memória compartilhada e uma versão de memória distribuída e algumas técnicas de otimização que aumentam a quantidade de laços vetorizados automaticamente. Porém, não considera o uso combinado dos paradigmas de paralelização, memória compartilhada e memória distribuída.

Embora os trabalhos anteriores tenham acelerado o algoritmo do PKTM através de paralelismo, nenhum deles levou em consideração o *overhead* dessas tecnologias num ambiente virtualizado. Esta dissertação apresenta diversas versões do algoritmo do PKTM e avalia o impacto das diferentes abordagens em ambientes virtuais.

## 2.2 Análise de Desempenho em Ambientes Virtualizados

Há uma enorme quantidade de trabalhos sobre avaliação de ambientes virtualizados com o uso de diferentes *benchmarks*. O foco destes trabalhos foi avaliar o *overhead* de CPU, memória, operações de E/S e rede na execução das aplicações.

Luszczek *et al.* [46] compararam KVM com VirtualBox e VMware para aplicações de álgebra linear, STREAM e FFT. Foi concluído que o KVM é a ferramenta de virtualização mais adequada para estes tipos de aplicação. Embora a avaliação do *overhead* nos diferentes componentes de hardware mostrem alguns gargalos de execução, nenhuma aplicação real foi testada por este trabalho.

Xavier *et al.* [72] argumentaram que aplicações HPC têm evitado os ambientes virtuais devido ao grande *overhead*, mas com o advento dos ambientes virtuais baseados em contêiner e seus baixos *overheads* seria possível uma execução próxima a do sistema operacional hospedeiro.

Em Morabito *et al.* [52], foram comparados os ambientes de virtualização KVM, LXC e Docker usando os *benchmarks* Y-cruncher, NBENCH, Linpack e Bonnie++. Foram avaliados o desempenho da CPU, operações de E/S e rede. Foi observado que o desempenho do KVM tem melhorado ao longo dos anos, embora a E/S ainda seja um gargalo. Em relação aos contêineres o nível de *overhead* foi praticamente insignificante.

O trabalho de Chung *et al.* [14] compara Docker com KVM para aplicações intensivas em computação (Linpack) e intensivas em dados (Graph500) mostrando que o desempenho do Docker é muito superior para aplicações intensivas em dados.

Já Arango *et al.* [6] comparam LXC, Docker e Singularity com aplicações reais de áreas como bioinformática [8], medicina [54] e gerenciamento de fluxo de trabalho [20] que estressam aspectos diferentes como computação, largura de banda de memória, latência de memória, largura de banda de rede e largura de banda de E/S. De modo geral, foi observado que Singularity obteve melhor desempenho para aplicações HPC.

Felter *et al.* [23] analisaram o desempenho de máquinas virtuais (KVM) e compararam com contêineres (Docker). Mostraram que os *overheads* de CPU e memória foram mínimos, tanto para máquinas virtuais quanto para contêineres. Porém, essas tecnologias encontraram dificuldades quando a carga trabalho foi intensiva em E/S.

Sharma *et al.* [63] avaliaram as diferenças de desempenho causadas pelas diferentes tecnologias de virtualização em ambientes de data center nos quais várias aplicações são executadas localmente no mesmo servidor, reduzindo assim os custos de operação. Os autores mostram que as aplicações estão sujeitas a interferências de desempenho, e o grau dessa interferência é maior no caso de contêineres para certos tipos de cargas de trabalho.

Kozhirbayev e Sinnott [38] compararam Docker com Flockport, utilizando ferramentas de *benchmarks* que avaliam os *overheads* de CPU, memória, rede e disco, e demonstraram que não há praticamente nenhum *overhead* na uso da memória ou da CPU para essas tecnologias. Mas há *overheads* de E/S.

Bhimani *et al.* [10] investigaram o desempenho de diferentes aplicações do Apache Spark usando máquinas virtuais e contêineres do Docker. Esse estudo compara as diferentes estruturas de virtualização para um ambiente de nuvem empresarial de big data usando o Apache Spark. Além do *makespan* e do tempo de execução, também são analisadas diferentes utilizações de recursos (CPU, disco, memória etc) pelas aplicações Spark. Os resultados mostram que, usando o Docker, o Spark pode obter uma aceleração de mais de 10 vezes em comparação com o uso da máquina virtual. Essa aceleração foi devida ao recurso *copy-on-write* (COW) disponibilizado pelo Docker e *driver* específico para armazenamento que melhoram o desempenho de algumas aplicações Spark.

Younge *et al.* [77] utilizam os *benchmarks* HPCG e IMB para avaliar os *overheads* e a escalabilidade dos contêineres, usando Singularity, no sistema Cray XC30. Além disso, utilizaram o ambiente EC2 da Amazon para comparações. Os autores chegaram à conclusão de que os contêineres utilizando o Singularity atingem desempenho próximo ao nativo quando utilizam as bibliotecas MPI da CRAY, enquanto o EC2 da Amazon seria útil para o desenvolvimento inicial e teste de aplicações HPC.

Kon *et al.* [37] avaliaram o impacto do uso de contêiner, no caso o Docker, para o caso de muitos ambientes computacionais serem executados em uma mesma máquina física. Os autores revelaram que o contêiner poderia alcançar o desempenho semelhante ao do sistema operacional do *host*, exceto em casos de processamento de E/S intensivo.

Os trabalhos citados não esgotam todos os encontrados na literatura sobre o tema. Porém, do nosso conhecimento, nenhum tocou nos mesmos pontos do trabalho desta dissertação que buscou avaliar o impacto de diferentes bibliotecas de paralelização e métodos de vetorização nos diferentes ambientes virtualizados. Como as operações de E/S do PKTM são pequenas em comparação ao tempo de computação (uso de CPU), os resultados encontrados nesta dissertação estão de acordo com o apontado nos estudos anteriores, uma vez que os *overheads* das tecnologias baseadas em contêineres foram baixos.

# Capítulo 3

# Algoritmo da Migração Kirchhoff Pré-empilhamento em Tempo (PKTM)

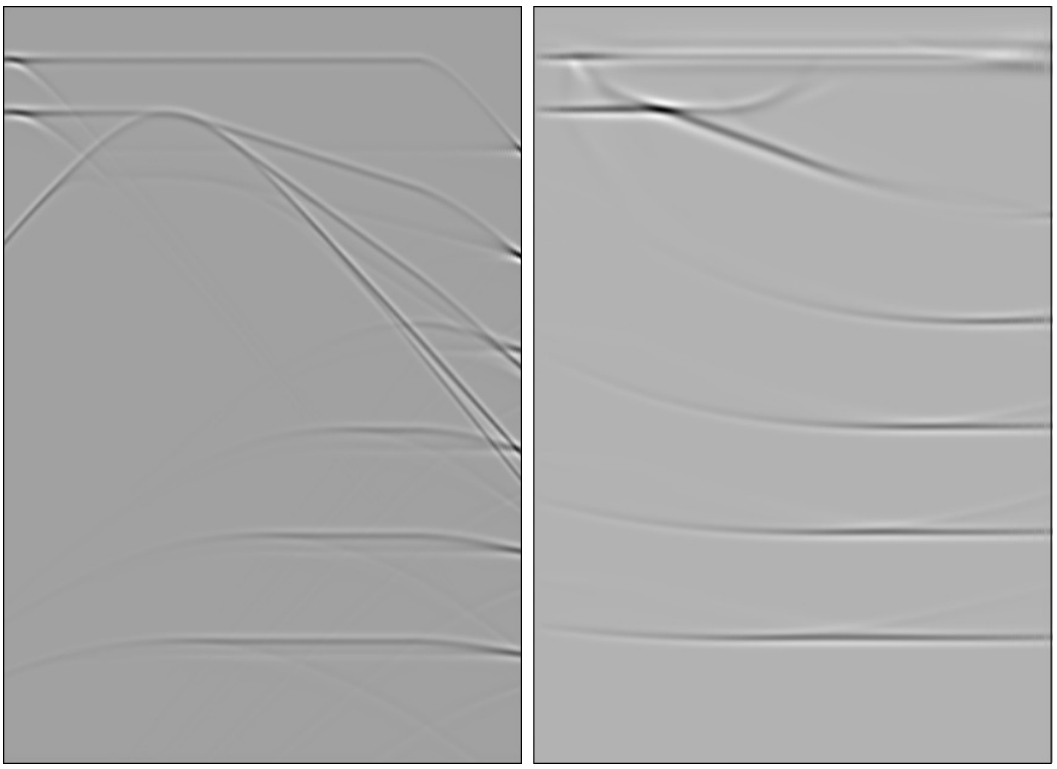
O fluxo de processamento de dados sísmicos é responsável por converter dados crus, adquiridos da subsuperfície da terra, em uma imagem que pode ser usada por especialistas para saber sobre suas características físicas e estrutura geológica. O fluxo de processamento é realizado após a coleta dos dados sísmicos.

Em uma coleta de dados sísmicos, uma fonte e um conjunto de receptores são colocados em uma área de interesse. A distância entre a fonte e cada um dos receptores é chamada *offset*. A fonte propaga uma onda através da subsuperfície da Terra que é refratada e refletida quando atinge uma camada subsuperficial de rocha. Esse processo conhecido como *aquisição de tiro comum* é feito várias vezes durante uma coleta de dados. Em cada tiro, fontes e receptores são colocados em diferentes posições da área de interesse para que peguem informações redundantes sobre um mesmo ponto da subsuperfície [56][76] [73].

Durante períodos discretos de tempo, receptores colocados próximos à superfície coletam energia refletida por camadas de rocha subsuperficiais durante todo o tempo de propagação de um tiro. Dados coletados por um receptor durante um período discreto de tempo $t$, são conhecidos como *amostra* e representam a amplitude da energia refletida por um ponto na subsuperfície. Supondo um plano paralelo, é como se esse ponto, conhecido como *Ponto Médio Comum* (ou apenas CMP do inglês *Commom MidPoint*), tivesse sido posto no meio do caminho entre a posição da fonte e dos receptores. Um conjunto de amostras coletadas por um receptor durante um tiro é chamado de *traço sísmico* e representa a energia refletida por um CMP durante um tiro inteiro de propagação. Um conjunto de traços de uma área demarcada gera um conjunto de dados chamado de *seção sísmica* que geralmente é salvo em um formato de arquivo padrão [76].

Após algum pré-processamento, a seção sísmica está pronta para ser migrada [76]. Basicamente, a migração: (i) colapsa difrações hiperbólicas, (ii) move refletores mergulhantes hiperbólicos para suas posições reais na subsuperfície e (iii) aumenta a resolução espacial. Uma seção sísmica precisa ser migrada porque pontos do refletor mergulhante na subsuperfície podem produzir uma imagem que originalmente não representa as estruturas geológicas reais. Isso ocorre porque um ponto de mergulho produz uma onda semicircular (de acordo com as leis de Huygens) que é coletada por receptores como uma hipérbole. Assim, a imagem resultante pode aparentar ter estruturas não colocadas nas suas posições reais ou estruturas que não existem de fato. Desse modo, a migração corrige essas distorções e gera uma imagem mais precisa, onde aparentemente a localização dos refletores mergulhantes são movidas até suas localizações reais [76] [16]. Um exemplo de uma seção sísmica antes e depois da migração é mostrado na Figura 3.1 onde (a) representa a imagem antes da migração e (b) representa a imagem após a migração.

Figura 3.1: Seção sísmica antes (a) e após a (b) migração.



(a) (b)

A migração Kirchhoff de pré-empilhamento em tempo recebe uma seção comum de *offsets* pré-empilhada como entrada e produz uma imagem final em coordenadas de tempo. O pressuposto básico da migração Kirchhoff é que qualquer ponto na seção sísmica pode ser considerado um refletor mergulhante e, consequentemente, cada um desses pontos está

localizado no ápice da hipérbole. Esse pressuposto indica que cada ponto deve receber de volta a energia dispersa de quando foram excitados por uma frente de onda. Em outras palavras, esse ápice deve receber a contribuição de energia de todas as amostras que compõem a hipérbole [70]. O comportamento desta hipérbole pode ser definido por uma equação de tempo de trânsito duplo que determina os pontos (amostras de entrada) que contribuíram para o seu ápice (amostras de saída) [76].

O Algoritmo 1 é uma implementação sequencial do PKTM. Para cada *offset*, o PKTM executa o *Laço de Traços de Entrada* (linhas 2 a 34) onde cada traço de entrada do *offset* atual é processado. Em seguida, o PKTM executa o *Laço de Cópia de Amostras de Traço de Entrada* (linhas 3 a 5) para poder copiar as amostras de traço de entrada para um vetor auxiliar. Depois disso, o PKTM executa o *Laço de Filtragem* (linhas 6 a 14) para conseguir versões *anti-alias* filtradas de um mesmo traço de entrada através da aplicação de um filtro *lowpass* usando *Lanczos Smoothing* [30]. Então, para cada versão, um filtro *anti-alias* é calculado (linhas 7 a 9), todas as amostras de traço de entrada são filtradas (linha 10) e salvas em um vetor de saída (linhas 11 a 13).

Após filtrar o traço de entrada, o PKTM executa o *Laço de Migração* (linhas 15 a 33) como descrito no Algoritmo 1. Para cada amostra de saída, a velocidade do CMP atual é lida (linha 16) e traços que compõem a abertura são determinados (linha 17). Uma abertura define quais traços de saída irão receber contribuições de um dado traço de entrada. Em seguida, o PKTM executa o *Laço de Contribuição* (linhas 18 a 32). No primeiro passo deste Laço, o tempo de trânsito duplo é calculado (linha 19) para identificar qual amostra de entrada irá contribuir para a amostra de saída atual. Entretanto, como esse tempo de trânsito calculado faz parte de um domínio contínuo, um conjunto de amostras para interpolação é selecionado (linha 20) para transpor-lo a um domínio discreto. Depois disso, a lentidão horizontal do operador de migração é calculada (linha 21) e o seu valor é usado como um parâmetro de entrada para determinar os filtros apropriados para o processo de *anti-alias* (linha 22).

Então, o PKTM executa o *Laço de Cópia de Amostras Selecionadas* (linhas 23 a 25) onde amostras selecionadas no passo *Seleciona_Amostras_Para_Interpolação()* são salvas em um vetor auxiliar. Em seguida, a amplitude da energia é calculada através de uma interpolação entre filtros e estas amostras selecionadas (linha 26). Esse processo de interpolação visa prover um valor aproximado de amplitude após dados os filtros escolhidos e o conjunto pré-definido de amostras. Então, (i) o fator de obliquidade, (ii) o ângulo de abertura, e (iii) o fator de espalhamento geométrico, são calculados (linhas 27, 28, e 29,

**Algoritmo 1** Algoritmo Sequencial do PKTM

```
    /* Laço de Offset */
 1: para todos Offsets faça
        /* Laço de Traços de Entrada */
 2:     para todos Traços de Entrada faça
            /* Laço de Cópia de Amostras de Traço de Entrada */
 3:         para todos Amostras de Traço de Entrada faça
 4:             Copia_Amostra_Para_Filtragem()
 5:         fim para
            /* Laço de Filtragem */
 6:         para todos Versões de Traços de Entrada faça
                /* Laço de Cálculo de Filtro Anti-alias */
 7:             para todos Frequências faça
 8:                 Calcula_Filtro_Anti-alias()
 9:             fim para
10:             Aplica_Filtro_Anti-alias()
                /* Laço de Cópia de Amostras Filtradas */
11:             para todos Amostras Filtradas faça
12:                 Salvar_Amostras_Filtradas()
13:             fim para
14:         fim para
            /* Laço de Migração */
15:         para todos Amostras de Saída faça
16:             Le_Velocidade()
17:             Determina_Traços_Dentro_da_Abertura()
                /* Laço de Contribuição */
18:             para todos Laços de Saída na Abertura faça
19:                 Calcula_Tempo_De_Transito()
20:                 Seleciona_Amostras_Para_Interpolação()
21:                 Calcula_Operador_De_Migração()
22:                 Define_Filtros()
                    /* Laço de Cópia de Amostras Selecionadas */
23:                 para todos Amostras Selecionadas faça
24:                     Copia_Amostra_Para_Interpolação()
25:                 fim para
26:                 Interpola_Amostras_Selecionadas()
27:                 Calcula_Fato_De_Obliquidade()
28:                 Calcula_Angulo_De_Abertura()
29:                 Calcula_Fator_De_Espalhamento_Geometrico()
30:                 Corrige_Amplitude()
31:                 Acumula_Contribuição_Na_Amostra_De_Saída()
32:             fim para
33:         fim para
34:     fim para
35: fim para
```

respectivamente), permitindo a correção da amplitude resultante no passo seguinte (linha 30). Finalmente, essa amplitude de energia corrigida é acumulada (linha 31) na amostra de saída atual. Uma descrição mais detalhada do PKTM pode ser encontrada em [76].

# Capítulo 4

# Ambientes Virtuais

A nuvem é um conjunto de recursos compartilhados entre vários usuários [51]. Ela tem sido utilizada para inúmeras aplicações e seus usuários finais variam de clientes ingênuos a técnicos experientes. Atualmente, está sendo chamada de XaaS (*Anything-as-a-Service*), o que significa que os provedores oferecem uma ampla variedade de serviços.

Em ambientes de nuvens computacionais a tecnologia de virtualização é chave para permitir sua operação, o compartilhamento de recursos e tornar flexível o acesso aos recursos. A virtualização começou na década de 1960, como um método para dividir logicamente os recursos do sistema fornecidos pelos computadores *mainframe* entre diferentes aplicações. A partir daí o significado do termo foi ampliado [25].

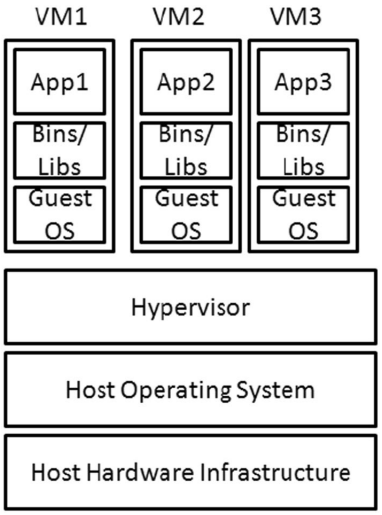
Um dos mais recentes serviços fornecidos pela nuvem são os ambientes de HPC para aplicações complexas. Usuários de HPC podem utilizar infraestruturas de nuvem virtualizadas como uma alternativa de baixo custo para executar suas aplicações [27, 13]. Os *clusters* HPC tradicionais são compostos de muitos servidores dedicados separados, denominados nós, e podem ser compartilhados entre diferentes organizações. Os requisitos de cada usuário ou organização diferem entre si, o que exige a criação de ambientes personalizados que não afetem os demais. A virtualização cria ambientes separados, virtualizados e personalizados do sistema com base nos requerimentos de cada usuário. O grande obstáculo da virtualização em ambientes HPC é seu *overhead*.

Existem duas técnicas básicas de virtualização [33]: *hypervisors* e contêineres. A virtualização baseada em *hypervisor* também chamada de Monitor de Máquina Virtual (MMV) fornece virtualização por meio de Máquinas Virtuais (MVs). O sistema operacional *guest* nas MVs cria chamadas para o *hypervisor*, ao invés de realizar uma comunicação direta com o hardware. Isso pode causar alguma redução no desempenho da aplicação,

devido ao custo de carregamento do sistema operacional, mais os binários e bibliotecas necessários para as aplicações.

Já a virtualização em contêineres permite implantar e executar aplicações sem criar MVs separadas para cada usuário. Os recursos do Linux, como *namespaces*, *chroot* e *cgroups*, fornecem a execução segura para contêineres no mesmo *kernel*. Uma vez que os contêineres não usam instâncias separadas do sistema operacional, eles requerem menos CPU, memória e armazenamento. Portanto, o mesmo *host* pode incorporar mais contêineres virtualizados. O tempo necessário para criação e implantação de contêineres é muito menor se comparado aos sistemas baseados em *hypervisor*. Contêineres carregam apenas os binários necessários para a execução [45], possuem pacotes já prontos para implantação de aplicações ou partes delas e, se necessário, algum *middleware* [15]. As figuras 4.1 e 4.2 mostram as duas arquiteturas de virtualização diferentes [7].

Figura 4.1: Virtualização baseada em *hypervisor*.

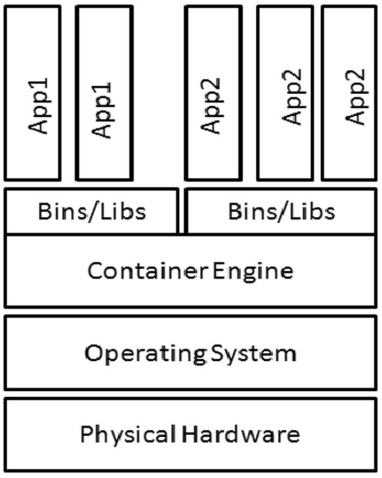



Fonte: [49]

## 4.1 *Hypervisor*

A técnica de virtualização *hypervisor*, também chamada de virtualização de hardware refere-se à criação de uma máquina virtual que funciona como um computador real com um sistema operacional. O software executado nessas máquinas virtuais é separado dos recursos de hardware subjacentes. Por exemplo, um computador que esteja executando o Microsoft Windows pode, por exemplo, hospedar uma máquina virtual com o sistema operacional do Ubuntu [71, p. 27].

Figura 4.2: Virtualização baseada em contêiner




Fonte: [49]

Na virtualização de hardware, a máquina *host* é a máquina na qual a virtualização ocorre e a máquina *guest* é a máquina virtual. As palavras *host* e *guest* são usadas para distinguir o software que é executado na máquina física do software do que é executado na máquina virtual. O software que cria uma máquina virtual no hardware *host* é chamado de *hypervisor* ou monitor de máquina virtual (MMV).

Os *hypervisors* podem ser classificados como do Tipo 1 ou do Tipo 2. Os *hypervisors* do primeiro tipo são aqueles que são executados nativamente no hardware. Já os do segundo tipo executam dentro do contexto de outro sistema operacional. Exemplos de *hypervisors* do tipo 1 incluem o *Kernel-based Virtual Machine* (KVM). Exemplos de *hypervisors* do tipo 2 incluem QEMU e WINE.

Os *hypervisors* do tipo 1 executam na camada acima da do hardware (como o VWware ESXi [1]) ou modificam o sistema operacional da máquina para permitir o acesso direto ao hardware através do processo de para-virtualização (por exemplo, Xen [2]) [22]

Por outro lado, nos *hypervisors* do Tipo 2, também conhecidos como virtualização completa, o sistema operacional virtualizado executa em cima do sistema operacional da máquina física. Essa abordagem abstrai todos os dispositivos de hardware e fornece um isolamento total do ambiente virtualizado [48]. Dessa forma, as instruções do sistema operacional virtualizado são traduzidas para a máquina física.

Na virtualização completa, o sistema operacional virtualizado executa sem modificações

[1] https://www.vmware.com/products/vsphere-hypervisor
[2] http://www.xenproject.org/

de *kernel*, por conta disso, sua execução impõe alto *overhead* no desempenho da MV. Para aplicações HPC, altos *overheads* são indesejados. Mas esse tipo de virtualização fornece um isolamento total do sistema trazendo mais segurança entre os ambientes virtuais e o *host*. As penalidades de desempenho causadas pela virtualização completa podem ser mitigadas usando a virtualização assistida por hardware, um conjunto de instruções específicas que auxiliam no processo de virtualização e está presente em quase todos os processadores e componentes de E/S [74]. Geralmente essa abordagem é utilizada em conjunto com a virtualização completa.

Na para-virtualização, os dispositivos de hardware não são emulados; eles são acessados através de *drivers* especiais, obtendo um desempenho melhor do que a virtualização completa, mesmo quando essa virtualização é assistida por hardware [53]. A principal diferença entre a virtualização completa e a para-virtualização é que o *kernel* do sistema operacional virtualizado precisa ser alterado de forma a fornecer novas chamadas de sistema. Essas modificações aumentam o desempenho do ambiente virtualizado por reduzirem o consumo da CPU, ao mesmo tempo que reduzem a segurança, confiabilidade e aumentam a dificuldade de gerenciamento.

A virtualização de hardware pode ser vista como tendência geral na area de TI corporativa que inclui computação autônoma, um cenário no qual o ambiente de TI será capaz de se gerenciar com base na atividade percebida. O objetivo comum da virtualização é centralizar as tarefas administrativas, melhorando a escalabilidade e a utilização geral de recursos de hardware. Com a virtualização, vários SOs podem ser executados em paralelo em uma CPU. Usando a virtualização, uma empresa pode gerenciar melhor as atualizações e as mudanças rápidas no sistema operacional e aplicações sem interromper o usuário. Em última análise, a virtualização melhora a disponibilidade de recursos e aplicações em uma organização. Em vez de depender do modelo antigo de uma aplicação por servidor, que permitem que os recursos sejam subutilizados, nos ambientes virtuais, os recursos são disponibilizados dinamicamente para atender às necessidades de negócios evitando desperdícios.

### 4.1.1 KVM

O KVM é um *hypervisor* de código aberto integrado ao *kernel* do Linux. Essa abordagem de virtualização aproveita o próprio desenvolvimento do *kernel* do Linux para proveito próprio [53]. No KVM, as MVs são simplesmente processos Linux instanciados pelo escalonador do sistema operacional. O gerenciamento de E/S, incluindo a interface virtual

de rede, é realizado por uma versão modificada do QEMU (*Quick Emulator*) que é uma ferramenta para virtualização de hardware. Toda requisição de E/S feita pela MV é encaminhada para o QEMU que, por sua vez, a redireciona para a máquina física. O desenvolvimento do *driver* virtio, uma estrutura de E/S para-virtualizada para KVM, tornou o KVM um forte candidato a aplicativos HPC [59]

Uma ampla variedade de SOs funciona com o KVM, incluindo Linux, BSD, Solaris, Windows, Haiku, ReactOS, OS X e outros.

## 4.2 Contêiner

Os contêineres compartilham o *kernel* com o sistema operacional do *host*. Portanto, são mais leves que as máquinas virtuais pois não há o *overhead* de execução de outra camada de abstração, porque a virtualização no nível do sistema operacional não precisa de um *hypervisor* [66]. Os contêineres são criados a partir de "imagens"que especificam seus conteúdos precisos. As imagens geralmente são criadas combinando e modificando imagens padrão baixadas dos repositórios. Seus processos e sistema de arquivo são acessíveis através do *host*.

### 4.2.1 Docker

O Docker é uma solução de contêiner de alto nível que aprimora o Linux Container (LXC) ao permitir o empacotamento de todo o ambiente em imagens de contêiner através da sobreposição do sistema de arquivos [36]. O Docker começou como um projeto de código aberto que utilizava o LXC para sua execução, posteriormente se tornou um ambiente completo de contêiner e, por fim, descartou o LXC. Com o Docker, é possível criar imagens personalizadas que podem ser usadas como base para a implantação de muitos contêineres concorrentes. Em um caso de uso de exemplo típico, um contêiner executa um servidor e uma aplicação da web, enquanto um segundo contêiner executa um servidor de banco de dados que é usado pela aplicação da web. Os contêineres são isolados uns dos outros e usam seu próprio conjunto de ferramentas e bibliotecas; eles podem se comunicar através de canais bem definidos.

O Docker foi desenvolvido pensando principalmente no Linux. Ele usa recursos de isolamento do *kernel*, como *cgroups* e *namespaces*, um sistema de arquivos que permite a união de arquivos e diretórios para permitir que contêineres independentes sejam executados dentro de uma única instância do Linux, evitando o *overhead* da criação e gerenciamento

de MVs. O suporte do *kernel* do Linux para *namespaces* isola a visão que a aplicação tem do ambiente operacional, incluindo árvores de processo, rede, IDs de usuário e sistemas de arquivos montados, enquanto o *cgroups* do *kernel* fornece recursos para limitar memória e CPU. Desde a versão 0.9, o Docker inclui a biblioteca *libcontainer* como uma maneira de usar diretamente os recursos de virtualização fornecidos pelo *kernel* do Linux, além de manter interfaces de virtualização abstratas como *libvirt*.

Além disso, é possível reduzir o *overhead* de virtualização através da criação de imagens de contêiner em vez de MVs de forma que o desempenho próximo ao nativo possa ser alcançado [78]. Os contêineres se tornaram uma importante alternativa de virtualização, porque, normalmente, são mais eficientes do que a virtualização completa.

À medida que as operações são executadas em uma imagem na base do Docker, essas camadas do sistema de arquivos que permitem união são criadas e documentadas, de modo que cada camada descreva totalmente como recriar uma operação. Essa estratégia permite que imagens de contêineres do Docker funcionem gerando pouco *overhead*, já que apenas as atualizações nessas camadas precisam ser propagadas (em comparação com VMs completas, por exemplo).

O Docker é uma ferramenta que pode empacotar um aplicativo e suas dependências em um contêiner virtual que pode ser executado em qualquer servidor Linux. Isso ajuda a permitir flexibilidade e portabilidade no local onde a aplicação pode ser executada, seja em instalações, nuvem pública, nuvem privada etc.

O uso de contêineres pode simplificar a criação de sistemas altamente distribuídos, permitindo que vários aplicativos, tarefas de trabalho e outros processos sejam executados de forma autônoma em uma única máquina física ou em várias máquinas virtuais. Isso permite que a implementação de nós seja executada à medida que os recursos se tornem disponíveis ou quando mais nós forem necessários, permitindo um estilo de implantação e dimensionamento de plataforma como um serviço (PaaS) para sistemas como Apache Cassandra, MongoDB e Riak.

### 4.2.2 Singularity

O contêiner Singularity, desenvolvido pelo Laboratório Nacional Lawrence Berkeley (LBNL), concentra-se na mobilidade. A ideia é definir, criar e manter localmente um fluxo de trabalho, garantindo que o fluxo de trabalho possa ser executado em diferentes hosts, SOs Linux ou provedores de serviços de nuvem [40]. O Singularity contém toda a pilha de

software, dos arquivos de dados à pilha da biblioteca, e permite que ela se mova de sistema para sistema de maneira confiável. A capacidade de migrar diferentes sistemas é crucial para a ciência reproduzível e para a implantação consistente e contínua de aplicativos [39]. O Singularity foi desenvolvido especificamente para solucionar problemas de segurança e atender às necessidades de computações HPC.

O contêiner Singularity usa um formato de imagem distribuída que possui todas as permissões de arquivos do Linux e pode ser copiado, compartilhado e arquivado. O Singularity também usa *namespaces* para portabilidade de aplicativos. É diferente do Docker no sentido de que o Singularity herda as permissões dos usuários do sistema operacional hospedeiro. Por causa disso, as operações de E/S fluem diretamente entre os ambientes reduzindo a sobrecarga da operação e os ambientes de execução [39].

O Singularity é um programa gratuito, que pode ser utilizado entre várias plataformas diferentes e de código aberto. Ele é capaz de suportar interconexões nativamente de alto desempenho, como InfiniBand e OPA (*Intel Omni-Path Architecture*). Além disso, fornece suporte nativo para a biblioteca Open MPI utilizando uma abordagem de contêiner MPI híbrida, na qual o OpenMPI existe dentro e fora do contêiner.

# Capítulo 5

# Implementações Paralelas do PKTM

O aumento do desempenho das aplicações executadas num computador é o propósito prático do processamento paralelo. Normalmente, existe um alto custo para obter-se alto desempenho através de paralelismo. Esse custo é devido ao desenvolvimento extra necessário para projetar e implementar um código paralelo voltado para múltiplos processadores [12].

A natureza de algumas aplicações exige que o processamento seja executado em vários processadores. A migração sísmica estudada nessa dissertação é um exemplo desse tipo de aplicação. Ela exige poder de computação que está além do que um simples processador pode fornecer, e a qualidade da imagem gerada no processo de migração é diretamente proporcional à quantidade de dados que ela pode processar.

Além disso, existem diferentes modelos de programação paralela que podem ser utilizados, como memória compartilhada e troca de mensagens. A escolha de um modelo de programação deve levar em consideração a arquitetura de hardware e a natureza da aplicação.

No modelo de programação com memória compartilhada, em que todos os processadores têm a possibilidade de acessar diretamente qualquer parte da memória, o padrão OpenMP é a forma mais utilizada para se escrever programas paralelos. De modo geral, os objetivos do OpenMP são: (i) fornecer um padrão de biblioteca portável para programas paralelos utilizando múltiplos processadores que compartilham memória e (ii) facilidade de uso. Como as diretivas disponibilizadas pela biblioteca OpenMP são identificadas e processadas pelo compilador, isso permite que o mesmo efetue otimizações [12].

No modelo de programação com troca de mensagens, a biblioteca MPI tem sido largamente utilizada ao longo de duas décadas. O modelo de troca de mensagens, entretanto,

é conhecido por ter uma programação mais complicada e, ao contrário do modelo de memória compartilhada, necessita que as estruturas de dados sejam particionadas e todo o programa precisa ser paralelizado de modo a utilizar essas estruturas particionadas.

Para comparar os efeitos da vetorização sobre versões paralelas do PKTM, foram desenvolvidas três novas versões paralelizadas do PKTM. A primeira utilizando a biblioteca OpenMP, a segunda utilizando a biblioteca MPI e a terceira, híbrida, juntando as duas abordagens anteriores.

## 5.1 PKTM Paralelo com OpenMP

Na utilização da biblioteca OpenMP, existem dois tipos de paralelização, o que Chandra *et al.*, em seu livro [12], chama de: (i) paralelismo de grão-fino ou a nível de laço; e (ii) paralelismo de grão-grosso que busca seccionar porções do código-fonte que podem ser processadas em paralelo. Analisando a execução do código-fonte do Algoritmo 1, foi verificado que o maior tempo de processamento da migração sísmica devia-se ao *Laço de Traços de Entrada* (linha 2). Dessa forma, para reduzir esse tempo de execução, utilizou-se a diretiva de compilação *omp parallel for* para que o trabalho pudesse ser distribuído entre um determinado número de *threads* (linha 2), realizando assim, uma paralelização de grão-fino. Pois cada uma dessas *threads* processa uma mesma quantidade de traços de entrada, ou seja, todas as *threads* recebem uma carga de trabalho similar. A execução dos passos internos do *Laço de Traços de Entrada* segue conforme definido pelo Algoritmo 1.

**Algoritmo 2** PKTM Paralelo com OpenMP

```
   /* Laço de Offset */
1: para todos Offsets faça
       /* Laço de Traços de Entrada */
       #pragma omp parallel for
2:     para todos Traços de Entrada faça
           /* Laço de Cópia de Amostras de Traço de Entrada */ [1]
3:     fim para
4: fim para
```

[1] Parte do pseudocódigo foi omitido para uma melhor compreensão.

## 5.2 PKTM Paralelo com MPI

A implementação MPI do PKTM explora o paralelismo de granularidade grossa. Duas abordagens foram consideradas: (i) a paralelização do trabalho realizado pelo laço mais

externo (o *Laço de Offset*), ou (ii) a paralelização do laço seguinte (o *Laço de Traços de Entrada*).

Na paralelização do trabalho realizado pelo laço mais externo, cada processo é responsável pela migração de todos os traços de um offset. Já na paralelização do *Laço de Traços de Entrada*, cada processo é responsável por migrar vários traços de entrada de um mesmo *offset*.

A abordagem de paralelização do *Laço de Offset* troca menos mensagens do que a paralelização do *Laço de Traços de Entrada*. Por este motivo, foi decidido implementar a paralelização do laço mais externo. Todas as operações de entrada (leitura de *offsets*) e saída (gravação dos traços migrados) são executadas por um processo mestre, enquanto os outros processos, chamados trabalhadores, executam efetivamente a migração dos traços de entrada.

Os Algoritmos 3 e 4 apresentam os pseudocódigos MPI dos processos mestre e trabalhadores, respectivamente. No Algoritmo 4, o processo trabalhador envia uma mensagem de solicitação de *offset* para o processo mestre (linha 2) e, ao receber a resposta (linha 3), realiza o processamento do correspondente *offset* (linha 6 a 7). Este procedimento é repetido até que o processo mestre termine a leitura de todos os *offsets*. Quando isso ocorre, o processo mestre envia ao trabalhador uma mensagem de término (linha 7) do algoritmo 4).

Essa implementação favorece o balanceamento de carga entre os processos trabalhadores. Quando um trabalhador finaliza seu processamento, ele pode solicitar outro *offset* para o mestre sem ter que esperar que os outros trabalhadores concluam seu processamento.

**Algoritmo 3** PKTM Paralelo com MPI - Processo Mestre

1: **enquanto** $\exists\ fim_processo[i] == FALSO,\ for\ i \in \{1 \ldots n\}$ **faça**
2:  **recebe** mensagem *PEDIDO* do processo i
3:  **Lê** *Offset*
4:  **se** não é o fim do arquivo de *Offsets* **então**
5:  **envia** mensagem *OFFSET* para o processo i
6:  **senão**
7:  **envia** mensagem *FIM* para o processo i
8:  $fim_processo[i] = VERDADEIRO$
9:  **fim se**
10: **fim enquanto**

**Algoritmo 4** PKTM Paralelo com MPI - Processo Trabalhador

1: **enquanto** não terminou **faça**
2: **Send** message *REQUEST* to master process
3: **Receive** message *msg* from master process
4: **se** *msg* == {*END*} **então**
5: terminou = VERDADEIRO
6: **senão**
/* *Offset Loop* */ [1]
7: **fim se**
8: **fim enquanto**

[1]Parte do pseudocódigo foi omitido para uma melhor compreensão.

## 5.3 PKTM Paralelo Híbrido com OpenMP e MPI

Também foi proposta uma versão paralela híbrida com OpenMP e MPI. Nesta versão, os *offsets* são distribuídos entre os processos. Em seguida, cada processo divide a migração dos traços de entrada entre suas *threads*, combinando as duas abordagens de paralelismo apresentadas em Seções 5.2 e 5.1.

A distribuição do trabalho realizado pelo laço externo (*Laço de Offset*) é feita entre os processos trabalhadores disponíveis, e o processamento da migração sísmica é dividido entre as *threads* criadas por cada processo trabalhador.

A abordagem da versão híbrida é semelhante à solução usando MPI, com um processo mestre e vários processos trabalhadores. O algoritmo do processo mestre é o mesmo que o apresentado na seção 5.2 (Algoritmo 3), mas o processo trabalhador é alterado para incluir a diretiva *#pragma omp parallel for*, como usado no algoritmo apresentado na seção 5.1, resultando no Algoritmo 5.

**Algoritmo 5** Algoritmo PKTM com MPI e OpenMP - Processo Trabalhador

```
1: enquanto not finished faça
2:     Envia mensagem REQUISIÇÃO para o processo Mestre
3:     Recebe mensagem MSG do processo Mestre
4:     se MSG == {SEM MAIS OFFSETS DE ENTRADA} então
5:         Terminado = Verdadeiro
6:     senão
           /* Laço de Offset */
7:         para todos Offsets faça
                   /* Laço de Traços de Entrada */
                   #pragma omp parallel for
8:             para todos Traços de Entrada faça
                       /* Laço de Cópia de Amostras de Traço de Entrada */ [1]
9:             fim para
10:        fim para
11:    fim se
12: fim enquanto
```

[1] Parte do pseudocódigo foi omitido para uma melhor compreensão.

# Capítulo 6

# Vetorização das Implementações Sequencial e Paralelas do PKTM

Neste capítulo, são descritos os métodos de vetorização utilizados nas diferentes implementações paralelas do PKTM. A seção 6.1 explica, brevemente, o conjunto de instruções AVX2. As seções 6.2 e 6.3 explicam, respectivamente, as versões PKTM vetorizadas manualmente e automaticamente.

## 6.1 *Advanced Vector Extensions 2* (AVX2)

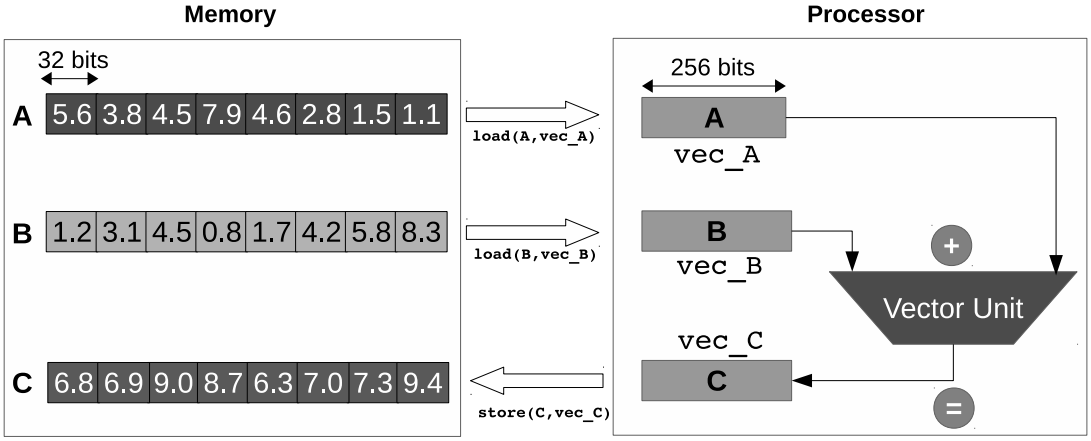
Em 2011, a Intel lançou uma nova microarquitetura chamada Sandy Bridge que trouxe consigo o conjunto de instruções vetoriais SIMD chamado AVX (*Advanced Vector Extensions*) [68]. Esse conjunto de instruções permite vetorizar, além das operações lógicas e matemáticas, tarefas como arredondamento, comparações e *shuffles* [44] [32].

Os processadores que possuem este conjunto de instruções AVX são dotados de uma unidade vetorial especial capaz de processar dois operandos vetores ao mesmo tempo. Esses processadores também possuem registradores de 256 bits (o número exato de registradores é específico de cada processador) que são usados para entrada e saída pela unidade vetorial especial. É possível processar vetores contendo oito números de pontos flutuantes de precisão simples (32 bits) ou quatro números de pontos flutuantes de precisão dupla (64 bits) simultaneamente [44] [68] [32].

Um exemplo da soma de dois vetores usando instruções vetoriais de baixo nível é mostrado na Figura 6.1. Neste exemplo, os vetores A e B, que contêm, cada um, oito números de pontos flutuantes de precisão simples são somados e o resultado é armazenado no vetor C. Primeiramente, esses dois vetores A e B são alocados em variáveis e depois

carregados para os registradores vetoriais. Em seguida, a unidade vetorial executa a soma dos dezesseis (oito de cada vetor) números simultaneamente e armazena o resultado num terceiro registrador vetorial. Finalmente, os dados desse terceiro registrador são carregados para o vetor C localizado na memória principal.

Figura 6.1: Soma de dois vetores de pontos flutuantes usando AVX2




Depois de Sandy Bridge, a Intel lançou a arquitetura Haswell dotada com o conjunto de instruções AVX2, uma extensão do conjunto de instruções anterior AVX [35] [61] [29]. Além de todos os recursos fornecidos pelo conjunto de instruções AVX, esse novo conjunto de instruções SIMD permite, também, o processamento de vetores contendo oito números inteiros (32 bits) ao mesmo tempo. Mais ainda, foram adicionadas ao AVX2 novas instruções de operações aritméticas para determinadas classes de aplicações específicas e novas instruções que simplificam a permutação dos elementos de dados [32].

Todo o potencial do conjunto de instruções AVX2 pode ser alcançado usando-se as funções intrínsecas (*intrinsics*) da biblioteca da Intel. Além dessas funções, a biblioteca da Intel (arquivo de cabeçalho `immintrin.h`) também fornece tipos específicos de dados que permitem tratar vetores declarados como tipos padrões do C (float, double, int etc). Por exemplo, um vetor de inteiros deverá estar associado a um vetor de mesmo tipo AVX2 para ser processado pela unidade vetorial. A Tabela 6.1 lista alguns exemplos de funções intrínsecas (*intrinsics*). Uma descrição detalhada das funções intrínsecas pode ser encontrada em [34].

Vale lembrar que o conjunto de instruções AVX2 é uma extensão do conjunto AVX. Logo, a versão que utiliza o conjunto de instruções AVX2 pode utilizar instruções presentes, também, no conjunto de instruções AVX. Porém a versão que utiliza o conjunto de instrução AVX pode utilizar, apenas, as instruções disponíveis neste conjunto.

Tabela 6.1: Exemplos de funções *intrinsics* (AVX2)

| **Função *instrisic*** | **Descrição** |
|---|---|
| _mm256_load_ps(vet,avx2) | Carrega um vetor de float *vet* num vetor AVX2 *avx2* |
| _mm256_store_ps(avx2,vet) | Armazena um vetor AVX2 *avx2* num vetor de float *vet* |
| _mm256_set_ps(f1,f2,f3,f4,f5,f6,f7,f8) | Retorna um vetor AVX2 com os floats *f1* até *f8* |
| _mm256_add_ps(a,b) | Soma dois vetores AVX2 de pontos flutuantes |
| _mm256_sub_ps(a,b) | Subtrai dois vetores AVX2 de pontos flutuantes |
| _mm256_div_ps(a,b) | Divide dois vetores de ponto flutuante AVX2 |
| _mm256_mul_ps(a,b) | Multiplica dois vetores de ponto flutuante AVX2 |
| _mm256_sqrt_ps(a) | Calcula a raiz quadrada do vetor de ponto flutuante AVX2 |
| _mm256_add_epi32(a,b) | Soma dois vetores de inteiros AVX2 |
| _mm256_hadd_epi32(a,b) | Horizontalmente soma dois vetores de inteiros AVX2 |

## 6.2 Vetorização Manual do PKTM

A vetorização manual do PKTM foi proposta em [4]. Nesta proposta, foi feita uma análise do código para identificar os laços que podem ser vetorizados do algoritmo sequencial e serial do PKTM (Algoritmo 1) resultando no Algoritmo 6. Todas as funções com prefixo "Vet" foram manualmente vetorizadas, isso é, o código original dessas funções foi modificado para processar oito elementos ao mesmo tempo. Como pode ser visto no Algoritmo 6, o laço de contribuição (originalmente nas linhas 18 a 32 do Algoritmo 1) foi dividido em dois novos laços chamados laço de contribuição vetorizado (linhas 18 até 26 do Algoritmo 6) e laço escalar de contribuição (linhas 27 a 34 do Algoritmo 6).

Todos os passos de vetorização foram agrupados dentro do laço de contribuição vetorizado, que foi executado através de iterações de oito em oito traços de saída. Cada iteração computa oito traços de saída ao mesmo tempo e armazena os dados resultantes em vetores, onde cada elemento corresponde a um traço de saída, para serem processados através de passos escalares.

Todos os passos remanescentes que não puderam ser vetorizados foram inseridos no laço de contribuição escalar que executa iterações de um por um dos traços de saída. No passo de interpolação (Interpola()), muitos cálculos matemáticos distintos são efetuados em apenas alguns números de ponto flutuante, o que não é adequado à vetorização, que requer que a mesma operação seja executada em vários elementos do vetor. O passo responsável pela correção das amplitudes resultantes (Corrige_Amplitude()) não foi vetorizado, porque depende de dados fornecidos pelo passo de interpolação, que também não foi vetorizado. Além disso, o passo que executa o acúmulo de contribuição na amostra

de saída (Acumula_Contribuição()) não foi vetorizado por causa da dependência dos dados entre cada uma das iterações do laço. Ainda, o laço de filtragem também não pôde ser vetorizado, porque o passo Aplica_Filtro_Anti-alias() usa números complexos que não são suportados pelo AVX2.

Como exemplo, será descrito como as instruções AVX2 são aplicadas na função usada para calcular o tempo de trânsito duplo. Descrever a vetorização de um passo simples do PKTM é o suficiente para entender toda a estratégia de vetorização utilizada nesta dissertação, uma vez que outros passos de vetorização seguem essa mesma abordagem. Então, considere a função de tempo de trânsito (executada na linha 19 do Algoritmo 1) descrita em detalhes no Algoritmo 7.

A função de Tempo_de_Trânsito() recebe como parâmetros de entrada: (i) a coordenada $x$ dos traços de entrada e saída, (ii) a amostra de tempo de saída, (iv) o *offset*, e (v) a velocidade. Como saída, o Algoritmo 7 retorna o tempo de trânsito relacionado ao tempo de amostra de saída e aos traços de saída dados. Todos os parâmetros e variáveis são representados como números de ponto flutuante de precisão simples (tipo de dado float em C).

Na linha 2 do Algoritmo 7, a distância entre os traços de entrada e saída são computados. O tempo de trânsito da fonte para o ponto médio é calculado na linha 3, enquanto o tempo de trânsito do ponto médio para o receptor é calculado na linha 4 (raiz() e potência() calculam a raiz quadrada e a potência de dois respectivamente). Finalmente, ambos os tempos de trânsito calculados são somados (linha 5) e é retornado o tempo de trânsito duplo resultante (linha 6). Essa função calcula o tempo de trânsito duplo para apenas um traço de saída por vez. O PKTM executa essa função para cada um dos traços de saída da abertura (linha 19 do Algoritmo 1).

No Algoritmo 8, a versão vetorizada do Algoritmo 6, chamada de *Vet_ Tempo_ de_ Trânsito()*, é descrita. Esta função recebe 8 traços de saída e calcula, ao mesmo tempo, os respectivos valores dos 8 tempos de trânsito usando AVX2. Os parâmetros dessa função são os mesmos da versão escalar original, exceto pelo parâmetro que recebe os traços de saída. Na versão vetorizada, a função recebe um vetor de 8 números de ponto flutuante ao invés de apenas um número de ponto flutuante como executado na versão escalar. A função *Vet_ Tempo_ de_ Trânsito()* retorna um vetor com 8 números de ponto flutuante onde o tempo de trânsito de cada traço de saída é armazenado nos elementos deste vetor. Todos os parâmetros de entrada e saída são representações de ponto flutuante de precisão simples.

As instruções AVX2 usadas pelo *Vet_ Tempo_ de_ Trânsito()* permitem o processamento simultâneo de 256 bits de dados por vez, ou seja, 8 números de ponto flutuante de precisão simples (32 bits cada). O procedimento do *Vet_ Tempo_ de_ Trânsito()* acessa o conjunto de instruções AVX2 usando as funções intrínsecas como previamente mencionado na subseção 6.1. Como essas funções intrínsecas processam somente dados específicos AVX2, todos os dados devem ser copiados para uma variável desse tipo antes de serem processados.

Primeiramente, todos os parâmetros de entrada são copiados para variáveis do tipo específico do AVX2 (variáveis com prefixos avx2) nas linhas 2 à 6. Depois disso, as distâncias dos 8 traços de entrada e saída são calculadas ao mesmo tempo na linha 7 (instrução equivalente a da linha 2 na versão escalar). Em seguida, o tempo de trânsito do ponto médio até o receptor dos oito traços de saída (correspondente a linha 3 da versão escalar) são calculados da linha 8 até a 12. Os tempos de trânsito dos pontos médios ao receptor dos oito traços de saída são calculados da linha 13 até a 17 (equivalente a linha 4 da versão escalar). Os tempos de trânsito duplo dos oito traços de saída são então computados na linha 18 (linha 5 na versão escalar). Depois, os oito tempos de trânsito duplo (armazenados no vetor AVX2) são copiados para um vetor de oito números de ponto flutuante. Então, em vez de processar apenas um traço de saída como executado pelo *Tempo_ de_ Trânsito()* a *Vet_ Tempo_ de_ Trânsito()* processa oito tempos de trânsito para oito traços de saída ao mesmo tempo.

Este mesmo processo utilizado para vetorizar manualmente o algoritmo sequencial do PKTM também foi usado para vetorizar os seguintes algoritmos paralelos do PKTM apresentados no Capítulo 5: o Algoritmo 2, o Algoritmo 4 e o Algoritmo 5.

### 6.2.1 Detalhes de Implementação

Nos processadores Intel, a aritmética de ponto flutuante sempre é executada sob a precisão estendida de 80 bits, mesmo quando estão sendo processados números de 32 bits representados com o padrão original IEEE 754 de precisão simples [11]. Como o AVX2 adota a representação original IEEE 754, existe uma pequena perda na precisão quando comparado com o código C tradicional. Entretanto, não existem diferenças significativas na imagem final de migração, uma vez que a precisão utilizada é mais do que adequada para a migração Kirchhoff [41].

Em alguns casos, não foi possível utilizar funções intrínsecas convencionais para carregar e armazenar (*_mm256_load_ps()* e *_mm256_store_ps()*, respectivamente), porque os

dados eram não alinhados em limites de 32 bytes. Assim, nesses casos, usamos as funções intrínsecas *_mm256_loadu_ps()* e *_mm256_storeu_ps()* que podem lidar com dados não alinhados. Essas operações, no entanto, são mais lentas do que as que acessam os dados alinhados [2]. Por fim, algumas funções trigonométricas usadas pelo PKTM, como seno e cosseno, não são suportadas pelo conjunto de instruções AVX2. Então, foram utilizadas funções trigonométricas disponíveis na biblioteca matemática do AVX2 [24].

## 6.3 Vetorização Automática do PKTM

Além da vetorização manual, há a vetorização automática, desempenhada pelo compilador. O compilador busca por laços que possam ser vetorizados, ou seja, laços que não possuam impeditivos quanto à vetorização como controle de fluxo, dependência de dados, operadores especiais e funções não vetorizáveis [62]. Essa busca automática feita pelo compilador ICC é realizada habilitando-se alguns *flags* do compilador ICC. Os *flags* de compilação utilizados em nossos experimentos estão descritos no Capítulo 7.

Apresentamos a seguir uma análise de como a vetorizações automática do compilador conseguiu vetorizar os laços do PKTM. Esta análise foi realizada com duas versões do compilador ICC da Intel, uma vez que as versões apresentam vetorizações diferentes. A Seção 6.3.1 especifica os laços do PKTM que foram vetorizados pela versão 17 do compilador ICC e os motivos pelos quais alguns laços não foram vetorizados. São apresentados, também, algumas otimizações para ajudar o compilador a vetorizar esses laços. A Seção 6.3.2 apresenta os laços que foram vetorizados pela versão 18 do compilador ICC. Nesta versão, nenhuma otimização foi necessária.

### 6.3.1 Compilação com ICC Versão 17

Analisando os relatórios de vetorização do ICC versão 17, foi observado que: (i) os laços vetorizados automaticamente na compilação da versão sequencial do PKTM foram: *Cópia de Amostras de Traço de Entrada*, *Cálculo de Filtro Anti-alias* e *Cópia de Amostras Filtradas*; (ii) os laços vetorizados automaticamente na compilação da versão paralelizada com OpenMP do PKTM foram: *Cópia de Amostras de Traço de Entrada*, *Cálculo de Filtro Anti-alias*, *Cópia de Amostras Filtradas* e *Cópia de Amostras Selecionadas*; (iii) os laços vetorizados automaticamente na compilação da versão paralelizada com MPI do PKTM foram: *Cópia de Amostras de Traço de Entrada* e *Cópia de Amostras Filtradas*. Um resumo desses laços e suas vetorizações automáticas é apresentado pela Tabela 6.2.

Tabela 6.2: Resumo das vetorizações automáticas pelo o ICC 17.

| **Laço** | **Paralelização (Vetorização)** | | | |
|---|---|---|---|---|
| | **Sequencial (Automática)** | **OpenMP (Automática)** | **MPI (Automática)** | ***Todos* (Manual)** |
| Offset | Não | Não | Não | Não |
| Traços de Entrada | Não | Não | Não | Não |
| Cópia de Amostras de Traço de Entrada | **Sim** | **Sim** | **Sim** | **Sim** |
| Filtragem | Não | Não | Não | Não |
| Cálculo de Filtro Anti-alias | **Sim** | **Sim** | Não | **Sim** |
| Cópia de Amostras Filtradas | **Sim** | **Sim** | **Sim** | **Sim** |
| Migração | Não | Não | Não | Não |
| Contribuição | Não | Não | Não | **Sim (Parte)** |
| Cópia de Amostras Selecionadas | Não | **Sim** | Não | **Sim** |

Comparando a eficiência da vetorização entre as versões sequenciais e paralelas do PKTM, percebeu-se que o ICC conseguiu vetorizar mais laços com OpenMP do que com MPI e também do que a versão sequencial. O compilador acusou dependência de dados nos laços que não foram vetorizados automaticamente em MPI e na versão sequencial. Porém, a verdade é que o compilador não tem como ter certeza que a dependência de dados nesses casos não existe. Portanto, adotando uma postura conservadora, ele previne a vetorização e informa o problema no relatório. Investigando mais profundamente os problemas relatados nesses relatórios de vetorização, pode-se adotar modificações nas implementações das versões sequencial e paralelizada com MPI para ajudar o compilador a alcançar a mesma eficiência de vetorização obtida com OpenMP. A Subseção 6.3.1.1 explica os problemas encontrados e apresenta as técnicas utilizadas para resolvê-los.

#### 6.3.1.1 Otimizações Para Vetorização Automática dos Códigos Sequencial e com MPI

Durante a implementação das versões paralelas do PKTM apresentadas no Capítulo 5, foram observados dois problemas. O primeiro problema encontrado foi com o uso de ponteiros e o segundo foi com a chamada de procedimentos. O primeiro problema acontece quando o compilador, ao analisar as expressões que envolvem ponteiros, cogita a possibilidade de que dois endereços de memória apontados por variáveis desse tipo possam se sobrepor, o que implica em dependência de dados. Neste caso, o compilador não vetorizava o laço. O segundo problema ocorre quando o compilador efetua o processo de *inlining* de um procedimento. Este processo, a grosso modo, copia o corpo do procedimento nos pontos em que ele foi chamado no código principal. No algoritmo sequencial do PKTM (Algoritmo 1), o *Laço de Cálculo de Filtro Anti-alias* está encapsulado dentro de um procedimento cuja finalidade é o cálculo do filtro. Por causa disso, uma das implementações do PKTM não vetoriza este laço.

O problema com o uso de ponteiros ocorre nas versões do PKTM sequencial e paralelizada com MPI. A explicação para isso é que na versão paralelizada com o OpenMP, a biblioteca cria novas variáveis para usar dentro das regiões paralelizadas. Essas novas variáveis garantem ao compilador que não há sobreposição de memória e o compilador, por causa disso, vetoriza os laços. Já o problema da chamada de procedimento ocorre na versão paralelizada com MPI. Nesta versão, foi criado um laço mais externo, como demonstrado no Algoritmo 4, para tratar as trocas de mensagens para a distribuição dos trabalhos. Essa modificação deixa o compilador confuso quanto às variáveis do tipo ponteiro presentes no corpo dos procedimentos que são copiados pelo processo de *inlining* para dentro desse laço mais externo. Com isso, o compilador acusa dependência de dados e não vetoriza os laços presentes nesse procedimento. Mas além do corpo principal do programa, todos os procedimentos e funções presentes no código-fonte também são vetorizados isoladamente. Como essas funções estão fora das estruturas de troca de mensagens criadas, elas são vetorizadas normalmente.

Para contornar o problema com o uso de ponteiros, foi utilizado o modificador de tipo *restrict* para algumas variáveis presentes no código-fonte. Esse modificador de tipo, que foi originalmente introduzido na especificação ISO C99, foi usado para informar ao compilador que uma determinada variável do tipo ponteiro seria a única a apontar para um determinado endereço de memória. Indicando ao compilador com o tipo *restrict* que os ponteiros não apontam para o mesmo endereço, ele foi capaz de vetorizar automaticamente nas versões sequencial e MPI os mesmos laços que foram vetorizados na versão OpenMP. Com esta alteração no código, o compilador sabe que não há mais o risco de haver dependência de dados nestes laços de execução.

Já para o problema com chamada de procedimento, foi necessário impedir que o compilador executasse o *inlining* do procedimento que não foi vetorizado. Ao impedir que o compilador faça o *inlining*, o mesmo mantém a chamada para o procedimento que será empilhada quando for executada. Essa chamada de procedimento empilhada acrescenta um pequeno *overhead*, mas durante os testes, este *overhead* se mostrou inferior ao ganho obtido com a vetorização. Por conta dessas otimizações, foram criadas duas novas versões do PKTM. A primeira otimiza a versão sequencial, que apresenta o problema com o uso de memória. Enquanto a segunda versão otimiza a versão paralelizada com MPI, que apresenta ambos os problemas explicados.

Analisando os relatórios de vetorização do ICC versão 17 para as novas versões otimizadas do PKTM, foi observado que: (i) os laços vetorizados automaticamente na

compilação das versões sequencial otimizada do PKTM e MPI otimizado do PKTM foram *Cópia de Amostras de Traço de Entrada*, *Cálculo de Filtro Anti-alias*, *Cópia de Amostras Filtradas* e *Cópia de Amostras Selecionadas*. Com as otimizações, todas as versões, sequencial, paralelizada com OpenMP e paralelizada com MPI obtiveram a mesma eficiência de vetorização. Esse resumo das vetorizações automáticas das versões otimizadas é apresentado pela Tabela 6.3.

Tabela 6.3: Resumo das vetorizações automáticas das versões otimizadas pelo ICC 17.

| **Laço** | **Paralelização (Vetorização)** | | | |
|---|---|---|---|---|
| | **Sequencial Otimizado (Automática)** | **OpenMP (Automática)** | **MPI Otimizado (Automática)** | ***Todos* (Manual)** |
| Offset | Não | Não | Não | Não |
| Traços de Entrada | Não | Não | Não | Não |
| Cópia de Amostras de Traço de Entrada | **Sim** | **Sim** | **Sim** | **Sim** |
| Filtragem | Não | Não | Não | Não |
| Cálculo de Filtro Anti-alias | **Sim** | **Sim** | **Sim** | **Sim** |
| Cópia de Amostras Filtradas | **Sim** | **Sim** | **Sim** | **Sim** |
| Migração | Não | Não | Não | Não |
| Contribuição | Não | Não | Não | **Sim (Parte)** |
| Cópia de Amostras Selecionadas | **Sim** | **Sim** | **Sim** | **Sim** |

### 6.3.2 Compilação com ICC versão 18

Analisando os relatórios de vetorização do ICC versão 18, foi observado que os laços vetorizados automaticamente na compilação das versões sequencial e paralelas do PKTM foram *Cópia de Amostras de Traço de Entrada*, *Cálculo de Filtro Anti-alias*, *Cópia de Amostras Filtradas* e *Cópia de Amostras Selecionadas*. Esse resumo das vetorizações automáticas é apresentado pela Tabela 6.4. Diferentemente da compilação com a versão anterior do ICC, esta versão conseguiu vetorizar automaticamente os mesmos laços nas diferentes versões do PKTM, dispensando quaisquer modificações no código-fonte.

Tabela 6.4: Resumo de vetorização para o ICC 18.

| **Laço** | **Paralelização (Vetorização)** | | | | |
|---|---|---|---|---|---|
| | **Sequencial (Automática)** | **OpenMP (Automática)** | **MPI (Automática)** | **Híbrida (Automática)** | ***Todas* (Manual)** |
| Offset | Não | Não | Não | Não | Não |
| Traços de Entrada | Não | Não | Não | Não | Não |
| Cópia de Amostras de Traço de Entrada | **Sim** | **Sim** | **Sim** | **Sim** | **Sim** |
| Filtragem | Não | Não | Não | Não | Não |
| Cálculo de Filtro Anti-alias | **Sim** | **Sim** | **Sim** | **Sim** | **Sim** |
| Cópia de Amostras Filtradas | **Sim** | **Sim** | **Sim** | **Sim** | **Sim** |
| Migração | Não | Não | Não | Não | Não |
| Contribuição | Não | Não | Não | Não | **Sim (Parte)** |
| Cópia de Amostras Selecionadas | **Sim** | **Sim** | **Sim** | **Sim** | **Sim** |

**Algoritmo 6** Algoritmo PKTM Vetorizado Manualmente

```
    /*Laço de Offset*/
 1: para todos Offsets faça
        /* Laços de Traços de Entrada */
 2:     para todos Traços de Entrada faça
            /* Laço de Cópia de Amostras de Traço de Entrada */
 3:         para todos Amostras de Traço de Entrada passo 8 faça
 4:             Vet_Copia_Amostra_Para_Filtragem()
 5:         fim para
            /* Laço de Filtragem */
 6:         para todos Versões de Traços de Entrada faça
                /* Laço de Cálculo de Filtro Anti-alias */
 7:             para todos Frequências faça
 8:                 Vet_Calcula_Filtro_Anti-alias()
 9:             fim para
10:             Aplica_Filtro_Anti-alias()
                /* Laço de Cópia de Amostras Filtradas */
11:             para todos Amostras Filtradas passo 8 faça
12:                 Vet_Salvar_Amostras_Filtradas()
13:             fim para
14:         fim para
            /* Laço de Migração */
15:         para todos Amostras de Saída faça
16:             Le_Velocidade()
17:             Determina_Traços_Dentro_da_Abertura()
                /*Laço de Contribuição Vetorizado*/
18:             para todos Laços de Saída na Abertura passo 8 faça
19:                 Vet_Calcula_Tempo_De_Transito()
20:                 Vet_Seleciona_Amostras_Para_Interpolação()
21:                 Vet_Calcula_Operador_De_Migração()
22:                 Vet_Define_Filtros()
23:                 Vet_Calcula_Fato_De_Obliquidade()
24:                 Vet_Calcula_Angulo_De_Abertura()
25:                 Vet_Calcula_Fator_De_Espalhamento_Geometrico()
26:             fim para
                /*Laço Escalar de Contribuição*/
27:             para todos Traços de Saída na Abertura passo 1 faça
                    /*Laço de Cópia de Amostras Selecionadas*/
28:                 para todos Amostras Selecionadas passo 8 faça
29:                     Vet_Copia_Amostra_Para_Interpolação()
30:                 fim para
31:                 Interpola_Amostras_Selecionadas()
32:                 Corrige_Amplitude()
33:                 Acumula_Contribuição()
34:             fim para
35:         fim para
36:     fim para
37: fim para
```

**Algoritmo 7** Função Escalar de Cálculo de Tempo de Trânsito Duplo

**ENTRADA:** *traço_de_saida_x, traço_de_entrada_x, amostra_de_tempo, offset, velocidade*
**SAIDA:** *tempo_de_trânsito*

1: **função** TEMPO_DE_TRÂNSITO()
2: d ← traço_de_saida_x - traço_de_saida_x
3: ts ← raiz(amostra_de_tempo + potência((d+offset)/velocidade,2))
4: tr ← raiz(amostra_de_tempo + potência((offset-d)/velocidade,2))
5: tempo_de_trânsito ← ts + tr
6: **Retorna** tempo_de_trânsito
7: **fim função**

**Algoritmo 8** Função Vetorizada de Cálculo de Tempo de Trânsito Duplo

**ENTRADA:** *traço_de_saida_x[8], traço_de_entrada_x[8], amostra_de_tempo[8], offset[8], velocidade[8]*
**SAIDA:** *tempo_de_transito[8]*

1: **função** VET_TEMPO_DE_TRÂNSITO
*/* Copiando parâmetros de entrada para variáveis AVX2 */*
2: avx2_traço_de_saida_x ← _mm256_load_ps(traço_de_saida_x)
3: avx2_traço_de_entrada_x ← _mm256_set1_ps(traço_de_entrada_x)
4: avx2_amostra_de_tempo ← _mm256_set1_ps(amostra_de_tempo)
5: avx2_offset ← _mm256_set1_ps(offset)
6: avx2_velocidade ← _mm256_set1_ps(velocidade)
*/* Correspondente a linha 2 do Algoritimo 7*/*
7: avx2_d ← _mm256_sub_ps(avx2_traço_de_saida_x,avx2_traço_de_entrada_x)
*/* Correspondente a linha 3 do Algoritimo 7 */*
8: avx2_aux1 ← _mm256_add_ps(avx2_d,avx2_offset)
9: avx2_aux2 ← _mm256_div_ps(avx2_aux1,avx2_velocidade)
10: avx2_aux3 ← _mm256_mul_ps(avx2_aux2,avx2_aux2)
11: avx2_aux4 ← _mm256_add_ps(avx2_amostra_de_tempo,avx2_aux3)
12: avx2_ts ← _mm256_sqrt_ps(avx2_aux4)
*/* Correspondente a linha 4 do Algoritimo 7 */*
13: avx2_aux1 ← _mm256_sub_ps(avx2_offset,avx2_d)
14: avx2_aux2 ← _mm256_div_ps(avx2_aux1,avx2_velocidade)
15: avx2_aux3 ← _mm256_mul_ps(avx2_aux2,avx2_aux2)
16: avx2_aux4 ← _mm256_add_ps(avx2_amostra_de_tempo,avx2_aux3)
17: avx2_tr ← _mm256_sqrt_ps(avx2_aux4)
*/* Correspondente a linha 5 do Algoritimo 7 */*
18: avx2_tempo_de_trânsito ← _mm256_add_ps(avx2_ts,avx2_tr)
*/* Copiando tempos de trânsito de uma variável AVX2 para uma variável float */*
19: _mm256_store_ps(tempo_de_trânsito, avx2_tempo_de_trânsito)
20: **Retorna** tempo_de_trânsito
21: **fim função**

# Capítulo 7

# Resultados Experimentais

Este capítulo apresenta os resultados experimentais das execuções das versões paralelizadas e vetorizadas do PKTM. São apresentados os ganhos de desempenhos assim como os efeitos dos ambientes virtualizados como KVM, Singularity e Docker.

No Capítulo 6, observou-se que diferentes versões do compilador ICC apresentaram diferentes eficiências de vetorização automática. A versão 17 teve problemas em vetorizar automaticamente determinados laços de algumas versões do PKTM enquanto que a versão 18 vetorizou automaticamente todas as versões da mesma maneira. Para os problemas encontrados pela versão 17 do compilador, foram apresentados otimizações que o ajudaram a lidar com esses problemas encontrados. Como essas duas versões exibiram diferentes eficiências de vetorização automática, decidiu-se apresentar os resultados experimentais de ambas as versões. A Seção 7.3 apresenta os resultados experimentais usando o compilador ICC versão 17 e a Seção 7.4 apresenta os resultados experimentais usando o compilador ICC versão 18.

## 7.1 Dados de Entrada

Experimentos foram realizados em conjuntos de dados sísmicos sintéticos. O conjunto de dados sintéticos e seu respectivo modelo de velocidade foram criados usando um sismógrafo sintético. Vale mencionar que o conjunto de dados sintéticos, como os modelos Marmousi e SEG/EAGE, são comumente usados para avaliar soluções geofísicas, técnicas e algoritmos, pois representam de forma fiel características geológicas presentes em conjuntos de dados reais [42] [43] [9] [58] [75]. A tabela 7.1 descreve a geometria dos dados de entrada para a aplicação do PKTM.

Tabela 7.1: Geometria do conjunto de dados sísmicos

| **Parâmetro** | **Sintético** |
|---|---|
| Número de *Offsets* | 63 |
| Amostras por Traço | 512 |
| Traços por Offset | 128 |
| Total de Traços | 8064 |
| Intervalo de *Offset* (m) | 20 |
| Intervalo de Amostragem (s) | 0.004 |

Foram avaliados dois tipos diferentes de parâmetros de entrada cujas variações podem influenciar significantemente o tempo de execução do PKTM. O primeiro parâmetro, FWIDTH, é usado para definir o incremento de frequência *high-end* para os filtros *low-pass* que são usados para calcular o número de frequência de cortes. Assim, o valor de FWIDTH está diretamente relacionado com o número de iterações feitas pelo laço de filtragem. O segundo parâmetro, NFC, define o número de coeficientes de Fourier usados para aplicar filtros *low-pass*. O tempo de execução para aplicar filtros *low-pass* é incrementado conforme o crescimento do NFC. Em outras palavras, quanto maior o NFC e/ou quanto menor o FWIDTH, maior será o tempo de execução do PKTM.

Esses parâmetros permitem aumentar ou diminuir a carga computacional do PKTM, dessa forma, para os testes, foram considerados diferentes parâmetros de entrada de forma a avaliar o comportamento das versões do PKTM conforme aumenta-se a carga computacional.

## 7.2 Ambiente de Execução

Todas as versões do PKTM foram compiladas com o compilador da Intel (ICC) através da ativação da opção de otimização `O3`. Este parâmetro de compilação habilita muitos tipos de otimizações, como o processo de vetorização automática realizado pelo ICC. Contudo, para a realização de uma análise justa do desempenho alcançado por cada tipo de vetorização, as soluções que envolvem vetorização manual foram compiladas com o *flag* `no-vec`, que desabilita a vetorização automática desempenhada pelo compilador [69]. Além disso, os códigos-fonte dessas versões são baseados no código-fonte original do PKTM fornecido pelo *Seismic Unix* [17], que é um pacote *open source* para processamento sísmico largamente usado pela industria de petróleo e pela academia [21].

Os experimentos foram realizados em uma máquina com processador Intel i7-5930K de 3,50 GHz, 32 GB de RAM. Este processador possui 6 núcleos físicos, 12 núcleos virtuais

devido ao *hyperthreading* e está equipado com o conjunto de instruções vetoriais *AVX2*. A seguir, a Tabela 7.2 apresenta os parâmetros de compilação utilizados para cada versão do PKTM testada.

Tabela 7.2: Parâmetros de compilação utilizados para cada versão do PKTM.

| **Código-fonte** | **Método de vetorização** | **Flags de otimização** |
|---|---|---|
| Sequencial Escalar | N/A | -O3<br>-xcore-avx2<br>-no-vec<br>-no-fma |
| Sequencial Auto | Automático | -O3<br>-xcore-avx2<br>-no-fma |
| Sequencial Auto Otimizado | Automático | -O3<br>-xcore-avx2<br>-no-fma |
| Sequencial Manual | Manual | -O3<br>-xcore-avx2<br>-no-vec<br>-no-fma |
| OpenMP Escalar | N/A | -O3<br>-xcore-avx2<br>-no-vec<br>-no-fma<br>-openmp |
| OpenMP Auto | Automático | -O3<br>-xcore-avx2<br>-no-fma<br>-openmp |
| OpenMP Manual | Manual | -O3<br>-xcore-avx2<br>-no-vec<br>-no-fma<br>-openmp |

| | | |
|---|---|---|
| MPI Escalar | N/A | -O3<br>-xcore-avx2<br>-no-vec<br>-no-fma |
| MPI Auto | Automático | -O3<br>-xcore-avx2<br>-no-fma |
| MPI Auto Otimizado | Automático | -O3<br>-xcore-avx2<br>-no-fma |
| MPI Manual | Manual | -O3<br>-xcore-avx2<br>-no-vec<br>-no-fma |
| MPI+OpenMP Escalar | N/A | -O3<br>-xcore-avx2<br>-no-vec<br>-no-fma<br>-openmp |
| MPI+OpenMP Auto | Automático | -O3<br>-xcore-avx2<br>-no-fma<br>-openmp |
| MPI+OpenMP Manual | Manual | -O3<br>-xcore-avx2<br>-no-vec<br>-no-fma<br>-openmp |

Como pode ser visto na Tabela 7.2 avaliamos as seguintes versões do PKTM: (i) o código original do *Seismic Unix* sem nenhum tipo de vetorização (`Sequencial Escalar`), (ii) automaticamente vetorizado pelo ICC com o conjunto de instruções AVX2 (`Sequencial Auto`), (iii) com otimizações no código-fonte para que seja eficientemente vetorizado pelo ICC com o conjunto de instruções AVX2 (`Sequencial Auto Otimizado`) e (iv) manualmente vetorizado pelo desenvolvedor com o conjunto de instruções AVX2 (`Sequencial`

`Manual`); (v) o código do PKTM paralelizado com OpenMP apresentado no Capítulo 5, Seção 5.1, Algoritmo 2 sem nenhum tipo de vetorização (`OpenMP Escalar`), automaticamente vetorizado pelo ICC com o conjunto de instruções AVX2 (vi) (`OpenMP Auto`) e (vii) manualmente vetorizado pelo desenvolvedor com o conjunto de instruções AVX2 (`OpenMP Manual`); o código do PKTM paralelizado com MPI apresentado no Capítulo 5, Seção 5.2, (viii) Algoritmo 3 e Algoritmo 4 sem nenhum tipo de vetorização (`MPI Escalar`), (ix) automaticamente vetorizado pelo ICC com o conjunto de instruções AVX2 (`MPI Auto`), (x) com otimizações no código-fonte para que seja eficientemente vetorizado pelo ICC com o conjunto de instruções AVX2 (`MPI Auto Otimizado`) e (xi) manualmente vetorizado pelo desenvolvedor com o conjunto de instruções AVX2 (`MPI Manual`); o código do PKTM paralelizado com OpenMP e MPI, apresentado no Capítulo 5, Seção 5.3, (xii) Algoritmo 3 e 5 sem nenhum tipo de vetorização (`MPI+OpenMP Escalar`), (xiii) automaticamente vetorizado pelo ICC com o conjunto de instruções AVX2 (`MPI+OpenMP Auto`) e (xiv) manualmente vetorizado pelo desenvolvedor com o conjunto de instruções AVX2 (`MPI+OpenMP Manual`).

Conforme explicado no Capítulo 6, duas versões do compilador ICC foram utilizadas para gerar código vetorizado automaticamente pelo compilador. A versão 17 do ICC é mais antiga e não é capaz de vetorizar automaticamente alguns laços em todas as versões. Neste caso, foram propostas otimizações para versão automática conforme mostra a Seção 6.3.1.1. Na versão mais atual do ICC (versão 18), as otimizações da vetorização automática não são necessárias. Esta versão foi utilizada para a análise dos ambientes de virtualização. Foram realizados experimentos utilizando versões 17 e 18 do ICC. Nos experimentos com a versão 17, avaliou-se apenas o impacto de diferentes tipos de vetorizações nas versões paralelizadas (OpenMP e MPI) do PKTM. Nos experimentos com a versão 18, avaliou-se também o impacto da vetorização nas versões paralelizadas (OpenMP, MPI e Híbrido) do PKTM e o efeito dos ambientes virtualizados na execuções dessas versões paralelas. Em todos os experimentos foram utilizados o sistema operacional Ubuntu, o compilador ICC, a biblioteca OpenMP contido no próprio compilador da Intel, o OpenMPI como implementação da especificação do MPI e o Seismic Unix. Os softwares e suas versões utilizadas nos dois experimentos estão descritos nas Tabelas 7.3 e 7.4. Vale ressaltar que apenas uma única máquina física foi utilizada em todos os experimentos. Para os experimentos com ambientes virtuais, uma única máquina virtual (ou no caso de contêiner, uma imagem) foi utilizada, mesmo para a execução de diferentes processos MPI.

Tabela 7.3: Versões dos softwares utilizadas nos experimentos com o compilador ICC versão 17.

| **Software** | **Versão** |
|---|---|
| Ubuntu | 14.04 LTS |
| Intel C Compiler (ICC) | 17.0.2 |
| OpenMP | 4.5 |
| OpenMPI | 1.6.5 |
| Seismic Unix | 44R4 |

Tabela 7.4: Versões dos softwares utilizadas nos experimentos com o compilador ICC versão 18.

| **Software** | **Versão** |
|---|---|
| Ubuntu | 16.04 LTS |
| Intel C Compiler (ICC) | 18.0.1 |
| OpenMP | 4.5 |
| OpenMPI | 1.10.2 |
| Seismic Unix | 44R4 |
| KVM | 1:2.6.2+dfsg-0∼16.04∼ppa0 |
| Docker | 18.03.1-ce |
| Singularity | 2.5.1-dist |

## 7.3 Experimentos sem Virtualização Usando o Compilador ICC Versão 17

Nos primeiros experimentos, foi avaliado o impacto das diferentes estratégias de vetorização do compilador ICC versão 17 para as seguintes versões: `Sequencial Escalar`, `Sequencial Auto`, `Sequencial Auto Otimizado`,`Sequencial Manual`, `OpenMP Escalar`, `OpenMP Auto`, `OpenMP Manual`, `MPI Escalar`, `MPI Auto`, `MPI Auto Otimizado` e `MPI Manual`.

Foram considerados os valores padrões para os parâmetros de entrada FWIDTH e NFC, que são 5 e 16, respectivamente. Esses valores impõem uma carga de processamento relativamente pequena à execução algoritmo do PKTM.

Para os testes com OpenMP foram utilizados com 2, 4 e 6 *threads* e para os testes usando o MPI foram utilizados 1 processo mestre e 2, 4 e 6 processos trabalhadores.

A Tabela 7.5 apresenta (i) o número de *threads* e processos trabalhadores das versões do PKTM, (ii) o tempo médio em 10 execuções, (iii) o *speedup* da vetorização e (iv) o *speedup* geral, que é o combinado da paralelização com a vetorização. O *speedup* da

vetorização compara, para cada versão sequencial ou paralela, o tempo de execução das versões vetorizadas em relação à versão escalar. O *speedup* geral compara cada versão com a versão sequencial e escalar do PKTM. Como a variação do tempo em 10 execuções foi insignificante em todos os casos, optou-se por não apresentar intervalos de confiança para a média do tempo de execução e para os valores de *speedup*.

Tabela 7.5: Tempo, *Speedup* da Vetorização e *Speedup* Geral obtidos.

| **#*Threads* ou #Processos.** | **Versão do PKTM** | **Tempo (s)** | ***Speedup* da Vetorização** | ***Speedup* Geral** |
|---|---|---|---|---|
| 1 | Sequencial Escalar | 92,5 | 1,0 | 1,0 |
| | Sequencial Auto | 74,0 | 1,3 | 1,3 |
| | Sequencial Auto Otimizado | 69,7 | 1,3 | 1,3 |
| | Sequencial Manual | 43,6 | 2,1 | 2,1 |
| 2 | OpenMP Escalar | 46,5 | 1,0 | 2,0 |
| | OpenMP Auto | 35,1 | 1,3 | 2,6 |
| | OpenMP Manual | 23,5 | 2,0 | 3,9 |
| | MPI Escalar | 48,5 | 1,0 | 1,9 |
| | MPI Auto | 48,6 | 1,0 | 1,9 |
| | MPI Auto Otimizado | 37,1 | 1,3 | 2,5 |
| | MPI Manual | 23,6 | 2,1 | 3,9 |
| 4 | OpenMP Escalar | 24,2 | 1,0 | 3,8 |
| | OpenMP Auto | 18,2 | 1,3 | 5,1 |
| | OpenMP Manual | 12,7 | 1,9 | 7,3 |
| | MPI Escalar | 24,9 | 1,0 | 3,7 |
| | MPI Auto | 24,9 | 1,0 | 3,7 |
| | MPI Auto Otimizado | 19,2 | 1,3 | 4,8 |
| | MPI Manual | 12,8 | 2,0 | 7,3 |
| 6 | OpenMP Escalar | 16,6 | 1,0 | 5,6 |
| | OpenMP Auto | 12,9 | 1,3 | 7,1 |
| | OpenMP Manual | 9,8 | 1,7 | 9,4 |
| | MPI Escalar | 19,7 | 1,0 | 4,7 |
| | MPI Auto | 19,8 | 1,0 | 4,7 |
| | MPI Auto Otimizado | 15,6 | 1,3 | 5,9 |
| | MPI Manual | 10,1 | 1,9 | 9,1 |

Analisando a Tabela 7.5, inicialmente, pode-se observar que as otimizações propostas conseguiram melhorar o tempo de execução da versão automática do código sequencial. É possível observar também que a versão OpenMP Manual com 6 *threads* obteve o melhor desempenho, com *speedup* geral igual a 9,4, seguida da versão MPI Manual que alcançou *speedup* geral de 9,1. Esse resultado mostra que a vetorização manual foi mais eficiente do que a vetorização automática pois conseguiu vetorizar uma quantidade maior de laços de execução do PKTM. Além disso, as versões com MPI apresentaram tempos de execução

maiores do que as versões com OpenMP, o que pode ser explicado pelo tempo gasto com as operações de trocas de mensagens do MPI.

Pode-se reparar que a versão paralelizada com MPI, automaticamente vetorizada e otimizada foi capaz de alcançar o *speedup* de vetorização 1,3, o mesmo *speedup* obtido pela versão que utiliza o OpenMP. Observa-se, também, que esse *speedup* se manteve constante mesmo aumentando a quantidade de *threads* ou processos trabalhadores.

Com relação à vetorização manual, foi verificado que o *speedup* de vetorização diminui a medida que aumentou-se o número de *threads* ou processos trabalhadores. A aplicação paralelizada com OpenMP ou MPI difere da aplicação escalar por ter acrescido ao seu tempo de execução os *overheads* da troca de contexto e sincronização no caso das *threads*, e troca de mensagens no caso de processos. Esses *overheads* aumentam o tempo de execução e diminuem o impacto da vetorização manual.

Considerando o impacto da vetorização automática otimizada no MPI, verifica-se que, quando otimizado, o código com MPI foi capaz de alcançar *speedups* de vetorização iguais ao do código com OpenMP, isto é, 1,3 nas execuções com 2, 4 e 6 *threads*, no caso do OpenMP, e nas execuções com 2, 4 e 6 processos trabalhadores, no caso do MPI. É possível observar também que, nestes casos, não houve alteração no ganho de desempenho alcançado pela vetorização diante da variação do número de *threads* ou processos trabalhadores.

Considerando o impacto da vetorização manual nas versões paralelas, nota-se uma leve redução do ganho de desempenho a partir do aumento da quantidade de *threads*, no caso do OpenMP e da quantidade de processos trabalhadores, no caso do MPI. O motivo é que as bibliotecas de paralelização incluem um *overhead* escalar contribuindo para a lei de Amdahl [28]; como exemplo desses *overheads* pode-se citar a criação e sincronização das *threads* por parte do OpenMP e tempo de comunicação (troca de mensagens) por conta do MPI.

Apesar dos *overheads*, ao combinar paralelização com a vetorização, obteve-se o maior ganho de desempenho; diminuindo, assim, o tempo total de execução da aplicação PKTM.

## 7.4 Comparação entre Execuções em Ambientes Virtualizados e não Virtualizados

Na segunda bateria de experimentos, foi utilizado o compilador ICC versão 18 e as seguintes versões foram executadas: `Sequencial Escalar`, `Sequencial Auto`, `Sequencial Manual`,

`OpenMP Escalar`, `OpenMP Auto`, `OpenMP Manual`, `MPI Escalar`, `MPI Auto`, `MPI Manual`, `MPI+OpenMP Escalar`, `MPI+OpenMP Auto` e `MPI+OpenMP Manual`. Todas as versões escalares, paralelas e vetorizadas foram executadas, também, nos ambientes virtualizados KVM, Docker e Singularity.

Note que não foram listadas as versões otimizadas das versões sequencial e paralelizadas com MPI uma vez que a nova versão do compilador (ICC versão 18) foi capaz de vetorizar o mesmo número de laços em todas as versões como apresentado na Tabela 6.4 da Subseção 6.3.2 do Capítulo 6.

Para estes experimentos, foram considerados três níveis de carga de processamento, uma carga leve, representando uma instância pequena de execução, uma carga moderada, representando uma instância média de execução, e uma carga pesada, representando uma instância grande de execução. Na instância pequena, foram utilizados os parâmetros FWIDTH igual a 5 e NFC igual a 8. Na instância média, FWIDTH é igual a 3 e NFC é igual a 16. Na instância grande, FWIDTH é igual a 1 e NFC é igual a 32. Uma observação é que os valores escolhidos para os parâmetros NFC e FWIDTH obedecem um intervalo de valores possíveis de acordo com os princípios geofísicos.

Para os testes com OpenMP foram utilizados com 2, 4 e 6 *threads* e para os testes usando o MPI foram utilizados 1 processo mestre e 2, 4 e 6 processos trabalhadores.

### 7.4.1 Análise de Desempenho da Vetorização

O desempenho da vetorização é avaliado para todas as versões paralelas do PKTM, incluindo a versão híbrida. As Tabelas 7.6, 7.7 e 7.8 apresentam, considerando diferentes cargas de trabalho: (i) o número de *threads* e/ou processos trabalhadores das versões do PKTM, (ii) o tempo médio em 10 execuções, (iii) o *speedup* da vetorização e (iv) o *speedup* geral, que considera o aumento do desempenho combinado da paralelização com a vetorização.

O *speedup* de vetorização compara, para cada versão sequencial ou paralela, o tempo de execução das versões vetorizadas em relação à versão escalar. O *speedup* geral compara cada versão com a versão sequencial e escalar do PKTM. Como a variação do tempo em 10 execuções foi insignificante em todos os casos, optou-se por não apresentar intervalos de confiança para a média do tempo de execução e para os valores de *speedup*.

Assim como nos resultados obtidos com a versão 17 do ICC, as Tabelas 7.6, 7.7 e 7.8 mostram que a vetorização manual é mais eficiente que a automática em todas as

Tabela 7.6: Tempo de execução e *speedups* obtidos com OpenMP, MPI e implementações híbridas do PKTM utilizando uma carga leve de processamento.

| **#*Threads* ou #Processos** | **Versão do PKTM** | **Tempo (s)** | ***Speedup* da Vetorização** | ***Speedup* Geral** |
|---|---|---|---|---|
| 1 | Sequencial Base | 79,30 | 1,0 | 1,0 |
| | Sequencial Auto | 70,06 | 1,1 | 1,1 |
| | Sequencial Manual | 42,86 | 1,9 | 1,9 |
| 2 | OpenMP Escalar | 40,06 | 1,0 | 2,0 |
| | OpenMP Auto | 33,50 | 1,2 | 2,4 |
| | OpenMP Manual | 23,18 | 1,7 | 3,4 |
| | MPI Escalar | 40,88 | 1,0 | 1,9 |
| | MPI Auto | 35,89 | 1,1 | 2,2 |
| | MPI Manual | 21,98 | 1,9 | 3,6 |
| 4 | OpenMP Escalar | 20,55 | 1,0 | 3,9 |
| | OpenMP Auto | 17,16 | 1,2 | 4,6 |
| | OpenMP Manual | 12,59 | 1,6 | 6,3 |
| | MPI Escalar | 20,48 | 1,0 | 3,9 |
| | MPI Auto | 18,02 | 1,1 | 4,4 |
| | MPI Manual | 11,20 | 1,8 | 7,1 |
| 6 | OpenMP Escalar | 14,15 | 1,0 | 5,6 |
| | OpenMP Auto | 11,87 | 1,2 | 6,7 |
| | OpenMP Manual | 8,88 | 1,6 | 8,9 |
| | MPI Escalar | 15,05 | 1,0 | 5,3 |
| | MPI Auto | 13,35 | 1,1 | 5,9 |
| | MPI Manual | 8,56 | 1,8 | 9,3 |
| 2P x 3T | MPI+OpenMP Scalar | 18,70 | 1,0 | 4,2 |
| | MPI+OpenMP Auto | 16,43 | 1,1 | 4,8 |
| | MPI+OpenMP Manual | 11,56 | 1,6 | 6,9 |
| 3P x 2T | MPI+OpenMP Scalar | 15,61 | 1,0 | 5,1 |
| | MPI+OpenMP Auto | 13,25 | 1,2 | 6,0 |
| | MPI+OpenMP Manual | 10,09 | 1,5 | 7.9 |

versões do PKTM. Isso ocorre porque a vetorização manual é capaz de vetorizar uma parte maior do código-fonte do PKTM. A vetorização automática reduziu, em média, o tempo de execução do PKTM em cerca de 14%, 30% e 57% para as cargas leve, moderada e pesada, respectivamente. Enquanto a vetorização manual diminuiu, em média, o tempo de execução do PKTM em cerca de 41%, 56% e 78% para as cargas leve, moderada e pesada, respectivamente.

A versão híbrida apresenta desempenho inferior em comparação com as versões paralelizadas com OpenMP ou MPI. Na versão híbrida, quando são associadas *threads* a um processo, no momento em este processo estiver enviando ou recebendo mensagens,

Tabela 7.7: Tempo de execução e *speedups* obtidos com OpenMP, MPI e implementações híbridas do PKTM utilizando uma carga moderada de processamento.

| **#*Threads* ou #Processos** | **Versão do PKTM** | **Tempo (s)** | ***Speedup* da Vetorização** | ***Speedup* Geral** |
|---|---|---|---|---|
| 1 | Sequencial Escalar | 112,83 | 1,0 | 1,0 |
| | Sequencial Auto | 80,77 | 1,4 | 1,4 |
| | Sequencial Manual | 47,05 | 2,4 | 2,4 |
| 2 | OpenMP Escalar | 56,96 | 1,0 | 2,0 |
| | OpenMP Auto | 38,95 | 1,5 | 2,9 |
| | OpenMP Manual | 25,55 | 2,2 | 4,4 |
| | MPI Escalar | 57,83 | 1,0 | 2,0 |
| | MPI Auto | 41,35 | 1,4 | 2,7 |
| | MPI Manual | 24,20 | 2,4 | 4,7 |
| 4 | OpenMP Escalar | 29,16 | 1,0 | 3,9 |
| | OpenMP Auto | 19,95 | 1,5 | 5,7 |
| | OpenMP Manual | 13,59 | 2,1 | 8,3 |
| | MPI Escalar | 29,32 | 1,0 | 3,8 |
| | MPI Auto | 20,79 | 1,4 | 5,4 |
| | MPI Manual | 12,18 | 2,4 | 9,3 |
| 6 | OpenMP Escalar | 20,19 | 1,0 | 5,6 |
| | OpenMP Auto | 13,72 | 1,5 | 8,2 |
| | OpenMP Manual | 9,60 | 2,1 | 11,8 |
| | MPI Escalar | 22,10 | 1,0 | 5,1 |
| | MPI Auto | 15,44 | 1,4 | 7,3 |
| | MPI Manual | 9,36 | 2,4 | 12,1 |
| 2P x 3T | MPI+OpenMP Scalar | 24,84 | 1,0 | 4,5 |
| | MPI+OpenMP Auto | 17,40 | 1,4 | 6,5 |
| | MPI+OpenMP Manual | 11,36 | 2,2 | 9,9 |
| 3P x 2T | MPI+OpenMP Scalar | 22,92 | 1,0 | 4,9 |
| | MPI+OpenMP Auto | 15,45 | 1,5 | 7,3 |
| | MPI+OpenMP Manual | 10,45 | 2,2 | 10,8 |

Tabela 7.8: Tempo de execução e *speedups* obtidos com OpenMP, MPI e implementações híbridas do PKTM utilizando uma carga pesada de processamento.

| **#*Threads* ou #Processos** | **Versão do PKTM** | **Tempo (s)** | ***Speedup* da Vetorização** | ***Speedup* Geral** |
|---|---|---|---|---|
| 1 | Sequencial Escalar | 347,52 | 1,0 | 1,0 |
| | Sequencial Auto | 151,02 | 2,3 | 2,3 |
| | Sequencial Manual | 70,51 | 4,9 | 4,9 |
| 2 | OpenMP Escalar | 176,30 | 1,0 | 2,0 |
| | OpenMP Auto | 74,20 | 2,4 | 4,7 |
| | OpenMP Manual | 39,84 | 4,4 | 8,7 |
| | MPI Escalar | 179,39 | 1,0 | 1,9 |
| | MPI Auto | 77,28 | 2,3 | 4,5 |
| | MPI Manual | 36,55 | 4,9 | 9,5 |
| 4 | OpenMP Escalar | 89,39 | 1,0 | 3,9 |
| | OpenMP Auto | 38,15 | 2,3 | 9,1 |
| | OpenMP Manual | 20,61 | 4,3 | 16,9 |
| | MPI Escalar | 90,78 | 1,0 | 3,8 |
| | MPI Auto | 38,73 | 2,3 | 9,0 |
| | MPI Manual | 18,41 | 4,9 | 18,9 |
| 6 | OpenMP Escalar | 61,22 | 1,0 | 5,7 |
| | OpenMP Auto | 26,02 | 2,4 | 13,4 |
| | OpenMP Manual | 14,36 | 4,3 | 24,2 |
| | MPI Escalar | 66,98 | 1,0 | 5,2 |
| | MPI Auto | 28,99 | 2,3 | 12,0 |
| | MPI Manual | 13,95 | 4,8 | 24,9 |
| 2P x 3T | MPI+OpenMP Scalar | 73,99 | 1,0 | 4,7 |
| | MPI+OpenMP Auto | 32,97 | 2,2 | 10,5 |
| | MPI+OpenMP Manual | 16,43 | 4,5 | 21,2 |
| 3P x 2T | MPI+OpenMP Scalar | 69,87 | 1,0 | 5,0 |
| | MPI+OpenMP Auto | 29,17 | 2,4 | 11,9 |
| | MPI+OpenMP Manual | 15,19 | 4,6 | 22,9 |

todas as *threads*, excetuando a principal, estarão ociosas. Por este motivo, alocar duas *threads* para três processos teve desempenho superior ao esquema de alocar 3 *threads* a dois processos. Além disso, o uso de OpenMP+MPI pode fazer com que o compilador tenha dificuldade de realizar algumas otimizações [28]. Se, por exemplo, o compilador não puder aplicar a otimização chamada *loop peeling* então o acesso aos dados pode se dar de forma não alinhada durante a execução do laço vetorizado, o que pode levar a uma degradação do desempenho. Porém, pode-se perceber que a vetorização dessas versões híbridas apresentam *speedups* de vetorização próximos às outras versões paralelas.

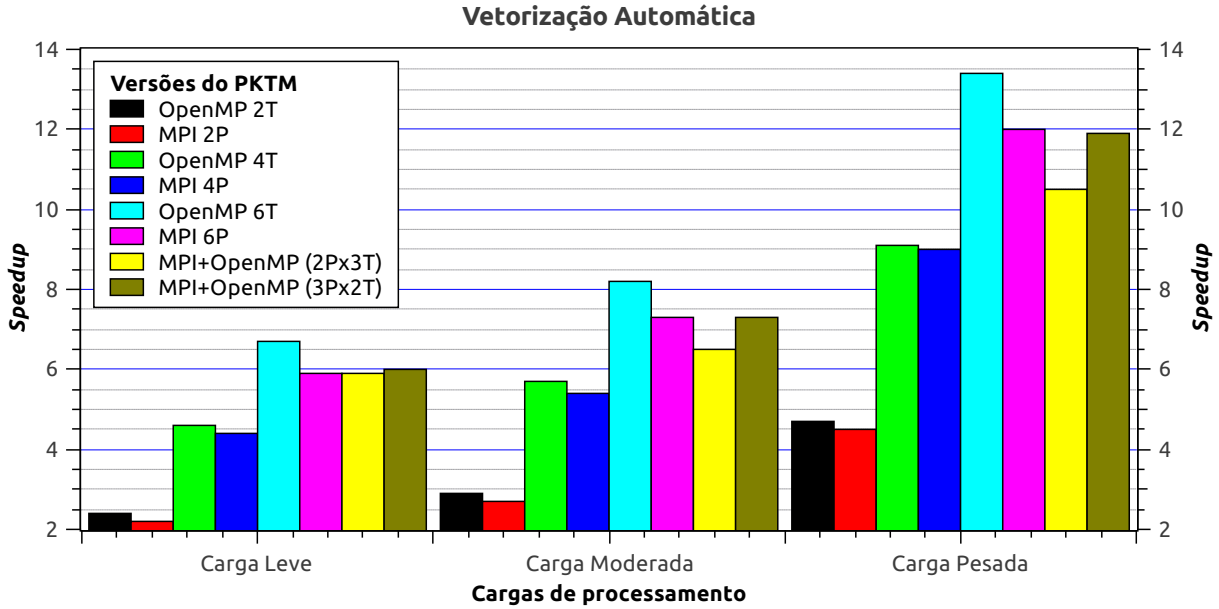
A diferença na versão híbrida com vetorização manual quando comparada às versões manuais de vetorização OpenMP e MPI é aparentemente devido ao *overhead* de chamar o sistema operacional para criar e finalizar *threads* e/ou processos. Na versão híbrida, o sistema operacional é chamado primeiro para criar os processos MPI e, em seguida, para criar as *threads* OpenMP.

Analisando as duas diferentes execuções das versões híbridas (2Px3T e 3Px2T), podemos notar também que a execução com 3 processos, cada um com 2 threads, superou a execução com 2 processos e 3 threads. Uma possível explicação é que, por causa do Hyper Threading, dois threads podem executar em um mesmo núcleo físico e, em seguida, se beneficiar da localidade da cache local.

As Figuras 7.1 e 7.2 mostram os *speedups* das execuções das versões OpenMP, MPI e Híbrida usando diferentes cargas de processamento. A Figura 7.1 considera a vetorização automática e a Figura 7.2 considera a vetorização manual. Observando essas figuras, podemos verificar que os *speedups* são maiores à medida que aumentamos a carga de processamento. Isso se deve ao aumento da caga computacional, que leva a uma maior eficiência do código paralelizado e vetorizado. Este resultado indica que o paralelismo e a vetorização podem ser mais eficazes quando aplicados a problemas maiores da PKTM.

Finalmente, vale a pena notar que o ganho de desempenho obtido ao combinar a paralelização com a vetorização foi sempre maior que o desempenho obtido usando apenas uma das duas abordagens. Isso pode ser verificado observando que os *speedups* aumentaram de 5,6, 5,6 e 5,7 para 8,9, 11,8 e 24,2 ao vetorizar manualmente as versões escalares do OpenMP (6 *threads*), considerando as cargas leve, moderada e pesada, respectivamente. Já a vetorização da versão escalar do MPI (6 processos) aumentou os *speedups* de 5,3, 5,1 e 5,2 para 9,3, 12,1 e 24,9. Enquanto que só a vetorização da versão sequencial apresentou os *speedups* de 1,9, 2,4 e 4,9 para as respectivas cargas. Portanto, podemos concluir que, no caso da PKTM, o uso combinado dessas duas técnicas reduz significativamente o tempo

Figura 7.1: *Speedups* das versões paralelas e vetorizadas automaticamente do PKTM.




total de execução da aplicação.

### 7.4.2 Análise de Desempenho do PKTM em Ambientes Virtuais

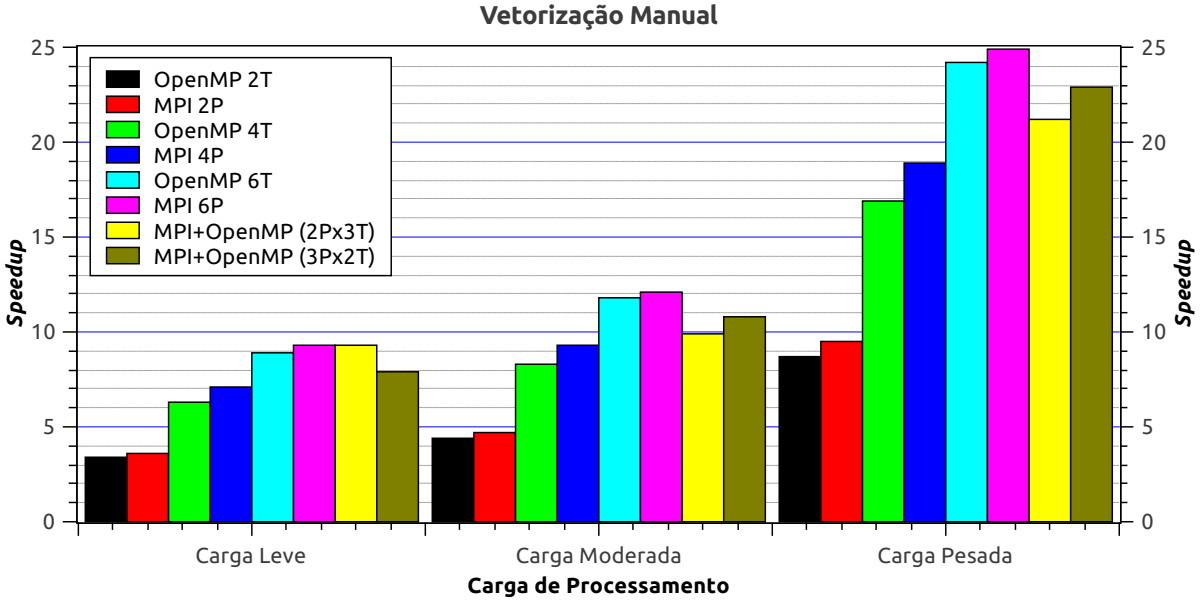
Na análise do desempenho do PKTM em ambientes virtuais, escolheu-se executar as versões do PKTM com a carga pesada por julgar ser esta carga a mais adequada para avaliar o impacto que esses ambientes impõem nos tempos de execução.

A Tabela 7.9 apresenta: (i) o número de *threads* ou processos de trabalho para cada versão do PKTM; (ii) as versões paralelas do PKTM; (iii) o tempo médio em 10 execuções em uma máquina física (Host); (iv) o tempo médio em 10 execuções no KVM e sua diferença percentual; (v) o tempo médio em 10 execuções no Docker e sua diferença percentual; e (vi) o tempo médio em 10 execuções do Singularity e sua diferença percentual. Essas diferenças percentuais são as variações entre os tempos de execução da máquina física e das máquinas virtuais.

Como pode ser visto na Tabela 7.9, as execuções nos ambientes baseados em contêineres (Docker e Singularity) apresentaram maior eficiência quando comparadas com as execuções no *hypervisor* (KVM). Isso se deve a um maior *overhead* causado pela emulação de hardware. Docker e Singularity apresentaram as menores diferenças percentuais. Embora os *drivers* de E/S possam ser para-virtualizados com o KVM, eles não são muito usados no PKTM.

Note, também, que, em testes usando a versão sequencial do PKTM, as execuções

Figura 7.2: *Speedups* das versões paralelas e vetorizadas manualmente do PKTM.




do PKTM em todos os ambientes virtualizados tiveram desempenhos próximos aos das execuções no ambiente sem virtualização.

Os tempos de execução de todas as versões, em 10 execuções, praticamente não variaram. Desta forma, apresentaram coeficientes médios de variação iguais a 1%, 1%, 2%, 2%, para ambientes Host, KVM, Docker e Singularity respectivamente.

Os tempos médios de execução aplicação do PKTM nos ambientes virtualizados foram próximos aos tempos de execução da máquina nativa, como mostram a Figuras 7.3 e a Figura 7.4. A explicação reside na própria característica do PKTM, que é intensivo em CPU [64] e flexível em E/S [31]. Observando os trabalhos relacionados da literatura, observamos que todos ambientes avaliados neste trabalho apresentam desempenho próximo ao nativo em se tratando de aplicações intensivas em CPU [59]. Logo, os resultados encontrados estão de acordo com esperado.

Tabela 7.9: Tempos médios de execução de todas as versões do PKTM quando executados em uma mesma máquina física (Host) e em ambientes virtuais KVM, Docker e Singularity.

| #*Threads* ou #Processos | Versão do PKTM | Tempo do Host (s) | Tempo do KVM (s) | Dif. % | Tempo do Docker (s) | Dif. % | Tempo do Singularity (s) | Dif. % |
|---|---|---|---|---|---|---|---|---|
| 1 | Sequencial Escalar | 347,52 | 351,70 | 1,20% | 350,19 | **0,77%** | 350,26 | 0,79% |
| | Sequencial Auto | 151,02 | 152,13 | 0,74% | 151,53 | **0,34%** | 152,06 | 0,69% |
| | Sequencial Manual | 70,51 | 71,19 | 0,96% | 71,17 | **0,94%** | 71,52 | 1,43% |
| 2 | OpenMP Escalar | 176,30 | 178,53 | 1,26% | 176,43 | **0,07%** | 176,90 | 0,34% |
| | OpenMP Auto | 74,20 | 75,31 | 1,50% | 74,51 | **0,42%** | 74,72 | 0,70% |
| | OpenMP Manual | 39,84 | 40,09 | 0,63% | 40,06 | 0,55% | 40,05 | **0,53%** |
| | MPI Escalar | 179,39 | 181,02 | 0,91% | 179,46 | **0,04%** | 179,58 | 0,11% |
| | MPI Auto | 77,28 | 77,65 | 0,48% | 77,13 | **-0,19%** | 77,55 | 0,35% |
| | MPI Manual | 36,55 | 36,80 | 0,68% | 36,39 | **-0,44%** | 36,29 | -0,71% |
| 4 | OpenMP Escalar | 89,39 | 90,43 | 1,16% | 89,30 | **-0,10%** | 89,70 | 0,35% |
| | OpenMP Auto | 38,15 | 38,35 | 0,52% | 37,92 | **-0,60%** | 37,96 | -0,50% |
| | OpenMP Manual | 20,61 | 20,72 | 0,53% | 20,57 | **-0,19%** | 20,61 | 0,00% |
| | MPI Escalar | 90,78 | 92,35 | 1,73% | 90,29 | **-0,54%** | 90,55 | -0,25% |
| | MPI Auto | 38,73 | 39,15 | 1,08% | 38,81 | **0,21%** | 38,94 | 0,54% |
| | MPI Manual | 18,41 | 18,77 | 1,96% | 18,52 | 0,60% | 18,43 | **0,11%** |
| 6 | OpenMP Escalar | 61,22 | 62,22 | 1,63% | 61,33 | **0,18%** | 61,46 | 0,39% |
| | OpenMP Auto | 26,02 | 26,56 | 2,08% | 26,12 | **0,38%** | 26,26 | 0,92% |
| | OpenMP Manual | 14,36 | 14,61 | 1,74% | 14,47 | 0,77% | 14,46 | **0,70%** |
| | MPI Escalar | 66,98 | 69,47 | 3,72% | 68,79 | 2,70% | 68,25 | **1,90%** |
| | MPI Auto | 28,99 | 29,54 | 1,90% | 29,45 | 1,59% | 29,28 | **1,00%** |
| | MPI Manual | 13,95 | 14,16 | 1,51% | 13,99 | 0,29% | 13,64 | **-2,22%** |
| 2P x 3T | MPI+OpenMP Escalar | 73,99 | 76,51 | 3,41% | 75,40 | **1,91%** | 76,48 | 3,37% |
| | MPI+OpenMP Auto | 32,97 | 33,16 | 0,58% | 32,61 | -1,09% | 32,22 | **-2,27%** |
| | MPI+OpenMP Manual | 16,43 | 17,32 | 5,42% | 16,00 | **-2,62%** | 16,68 | 1,52% |
| 3P x 2T | MPI+OpenMP Escalar | 69,87 | 71,81 | 2,78% | 71,22 | **1,93%** | 71,43 | 2,23% |
| | MPI+OpenMP Auto | 29,17 | 30,88 | 5,86% | 29,53 | **1,23%** | 29,77 | 2,06% |
| | MPI+OpenMP Manual | 15,19 | 15,30 | 0,72% | 14,99 | -1,32% | 14,89 | **-1,97%** |

Figura 7.3: Média dos tempos de execução da versão OpenMP do PKTM com 6 *threads* no Host, KVM, Docker e Singularity.

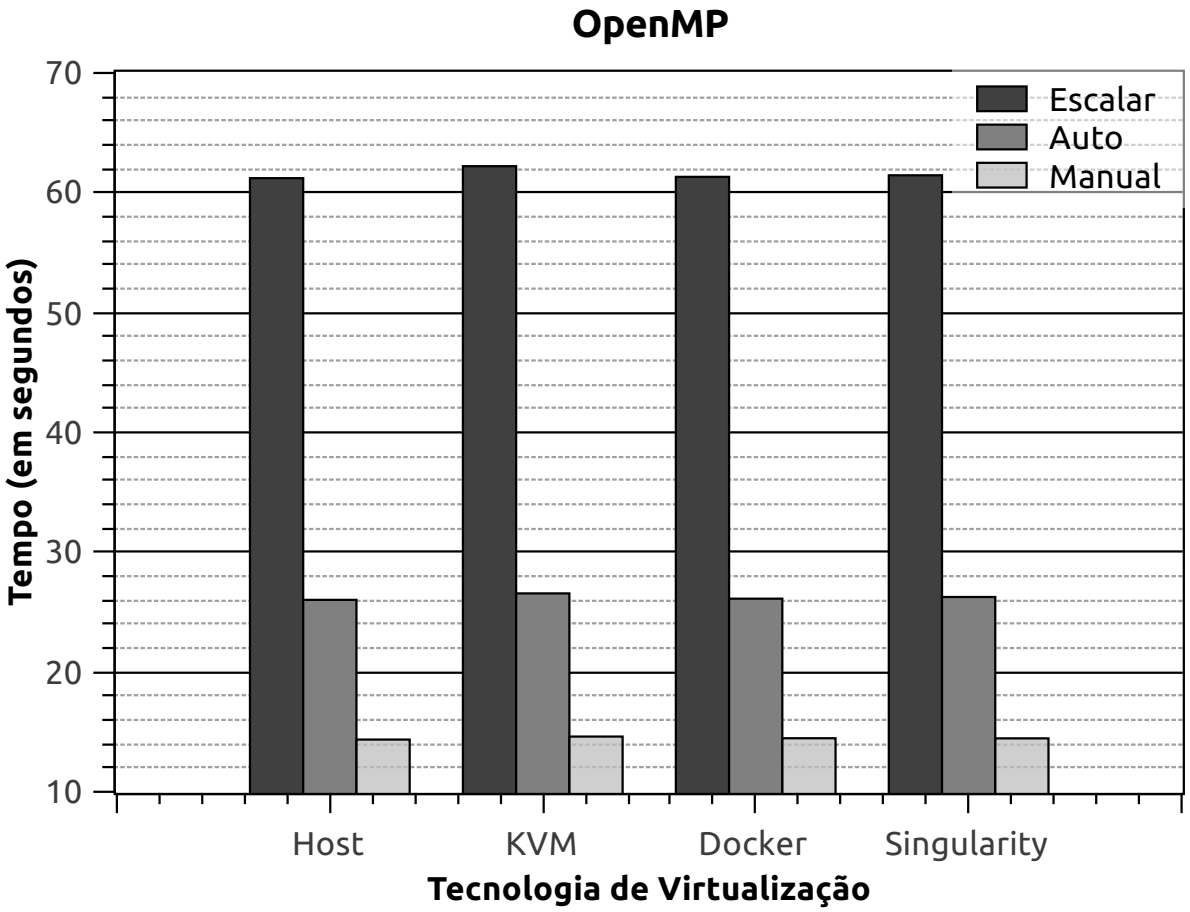
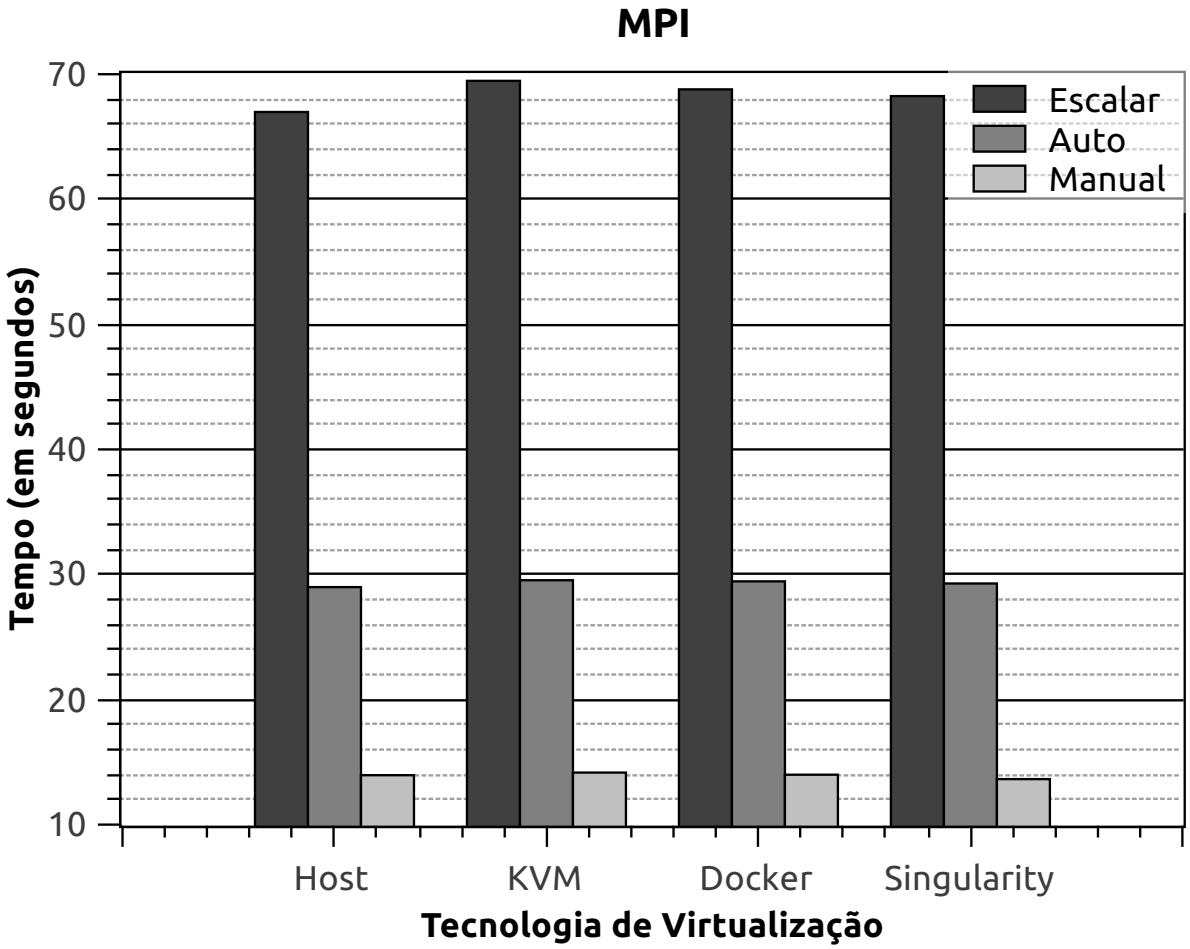


Figura 7.4: Média dos tempos de execução da versão MPI PKTM com 6 processos no Host, KVM, Docker e Singularity.



# Capítulo 8

# Conclusão e Trabalhos Futuros

Este capítulo descreve os principais resultados e contribuições deste trabalho. Na Seção 8.1, são destacadas as principais conclusões desta dissertação, enquanto na Seção 8.2 são apontados alguns direcionamentos para trabalhos futuros.

## 8.1 Considerações Finais

Neste trabalho, avaliamos a paralelização e vetorização do PKTM, utilizando dois modelos de programação paralela e duas técnicas de vetorização. Analisamos também os impactos dos diferentes ambientes virtuais sobre as diferentes versões paralelas e vetorizadas do PKTM.

Os experimentos foram realizados com duas versões diferentes do compilador ICC, versão 17 e versão 18. Na versão 17 do ICC, em geral, os programas OpenMP apresentaram melhores tempos de execução dentre as implementações paralelas. Percebeu-se, também, que o OpenMP facilita o trabalho de vetorização desempenhado pelo compilador nesta versão, enquanto o MPI dificulta. Por isso, otimizações foram implementadas no código-fonte para ajudar o compilador no processo de vetorização do código. Já na versão 18 do ICC, o OpenMP se saiu melhor nas execuções das versões automáticas, mas o MPI se saiu um pouco melhor nas versões manuais. Porém, nesta versão, o compilador foi capaz de vetorizar automaticamente a mesma quantidade de laços nas versões sequencial e paralelas, dispensando a necessidade das otimizações de código aplicadas para a versão anterior do ICC. As execuções das versões manualmente vetorizadas do PKTM obtiveram os melhores tempos de execução em todos os experimentos. Essas observações demonstram que diferentes versões de software e diferentes tecnologias podem impactar na eficiência da vetorização e, consequentemente, no desempenho da aplicação paralela.

A vetorização das versões paralelas se mostraram vantajosas uma vez que alcançaram *speedups* de até 25 enquanto que usando só paralelismo foi possível atingir um *speedup* de até 6 aproximadamente e um *speedup* de até aproximadamente 5 usando só a vetorização. Além disso, o uso de cargas de processamento maiores não comprometeu nem a paralelização nem a vetorização. Pelo contrário, fez com que fossem mais eficientes, indicando a escalabilidade da solução.

Com relação à análise dos ambientes virtualizados, as virtualizações baseadas em contêineres tiveram um desempenho melhor que a gerenciada por *hypervisor*. Porém, de modo geral, todos os ambientes tiveram um desempenho próximo ao nativo com uma diferença percentual de no máximo 6% aproximadamente. O PKTM é computacionalmente intensivo em CPU e flexível em E/S. Isso favorece os ambientes virtualizados uma vez que eles estão maduros o suficiente para usar a CPU eficientemente bem próximo ao desempenho nativo.

## 8.2 Trabalhos Futuros

Os pontos abordados nesta dissertação como paralelismo, vetorização e virtualização estão sujeitos a melhorias. Em termos das estratégias de paralelismo, pretendemos estudar outras formas de divisão do trabalho entre os processos MPI e as *threads* OpenMP e comparar o desempenho entre a divisão por traços e divisão por *offsets*. Pretendemos também estudar paralelizações do PKTM para aceleradores como GPU e Xeon Phi e seus impactos em ambientes virtualizados. Em termos das estratégias de vetorização, pretendemos estudar a possibilidade de vetorização manual de outros laços dentro do código e o desempenho com vetores de 512 bits do processador Xeon Phi. Finalmente, também pretendemos investigar o desempenho dos aplicações PKTM em um cenário realmente distribuído e seu impacto levando em consideração, também, os ambientes virtualizados.

# Referências


[1] Agulleiro, J.-I.; Fernandez, J.-J. Tomo3d 2.0 - exploitation of advanced vector extensions (avx) for 3d reconstruction. *Journal of Structural Biology 189*, 2 (2015), 147–152.

[2] Alvarez, M.; Salamí, E.; Ramirez, A.; Valero, M. Performance impact of unaligned memory operations in simd extensions for video codec applications. In *International Symposium on Performance Analysis of Systems & Software (ISPASS)* (2007), IEEE, pp. 62–71.

[3] Alves, M. M.; Drummond, L. M. A. A multivariate and quantitative model for predicting cross-application interference in virtual environments. *Journal of Systems and Software 128* (2017), 150–163.

[4] Alves, M. M.; Pestana, R. C.; Drummond, L. M. A. Accelerating pre-stack kirchhoff time migration by using simd vector instructions. In *Simpósio em Sistemas Computacionais de Alto Desempenho* (2015).

[5] Alves, M. M.; Pestana, R. C.; Silva, R. A. P.; Drummond, L. M. A. Accelerating Pre-stack Kirchhoff Time Migration by Manual Vectorization. *Concurrency and Computation: Practice and Experience 29*, 22 (2016), e3935.

[6] Arango, C.; Dernat, R.; Sanabria, J. Performance evaluation of container-based virtualization for high performance computing environments. *arXiv preprint arXiv:1709.10140* (2017).

[7] Babu, A.; Hareesh, M. J.; Martin, J. P.; Cherian, S.; Sastri, Y. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *Advances in Computing and Communications (ICACC), 2014 Fourth International Conference on* (2014), IEEE, pp. 247–250.

[8] Belmann, P.; Dröge, J.; Bremges, A.; McHardy, A. C.; Sczyrba, A.; Barton, M. D. Bioboxes: standardised containers for interchangeable bioinformatics software. *Gigascience 4*, 1 (2015), 47.

[9] Bhardwaj, D.; Phadke, S.; Yerneni, S. On improving performance of migration algorithms using mpi and mpi-io. In *Expanded Abstracts, Society of Exploration Geophysicists* (2000).

[10] Bhimani, J.; Yang, Z.; Leeser, M.; Mi, N. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE* (2017), IEEE, pp. 1–7.

[11] Bryant, R. E.; O'Hallaron, D. R. *Computer systems: a programmer's perspective*, vol. 2. Prentice Hall Upper Saddle River, 2003.

[12] Chandra, R.; Dagum, L.; Kohr, D.; Maydan, D.; McDonald, J.; Menon, R. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[13] Chen, L.; Patel, S.; Shen, H.; Zhou, Z. Profiling and understanding virtualization overhead in cloud. In *Parallel Processing (ICPP), 2015 44th International Conference on* (2015), IEEE, pp. 31–40.

[14] Chung, M. T.; Quang-Hung, N.; Nguyen, M.-T.; Thoai, N. Using docker in high performance computing applications. In *Communications and Electronics (ICCE), 2016 IEEE Sixth International Conference on* (2016), IEEE, pp. 52–57.

[15] Ciuffoletti, A. Automated deployment of a microservice-based monitoring infrastructure. *Procedia Computer Science 68* (2015), 163–172.

[16] Claerbout, J. F. *Fundamentals of Geophysical Data Processing*. Pennwell Books, 1985. ISBN 0-8654-2305-9.

[17] Cohen, J. K.; Stockwell, J. J. W. Cwp/su: Seismic unix release no. 41: An open source software package for seismic research and processing. Tech. rep., Center for Wave Phenomena, Colorado School of Mines, 2008.

[18] Dai, H. Parallel Processing of Prestack Kirchhoff Time Migration on a PC cluster. *Computers & Geosciences 31*, 7 (2005), 891–899.

[19] Deilmann, M. A guide to vectorization with intel® c++ compilers. Tech. rep., Intel Corporation, 2012.

[20] Di Tommaso, P.; Chatzou, M.; Floden, E. W.; Barja, P. P.; Palumbo, E.; Notredame, C. Nextflow enables reproducible computational workflows. *Nature Biotechnology 35*, 4 (2017), 316–319.

[21] Farfour, M.; Yoon, W. J. Borehole seismic data processing and interpretation: New free software. *Journal of Applied Geophysics 123* (2015), 18–29.

[22] Fayyad-Kazan, H.; Perneel, L.; Timmerman, M. Full and para-virtualization with xen: a performance comparison. *Journal of Emerging Trends in Computing and Information Sciences 4*, 9 (2013).

[23] Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On* (2015), IEEE, pp. 171–172.

[24] Garberoglio, G. AVX-optimized sin(), cos(), exp() and log() functions. Disponível em: `http://software-lisc.fbk.eu/avx_mathfun/`, 2012. [Online; Acesso em: julho. 2018].

[25] Graziano, C. D. A performance analysis of xen and kvm hypervisors for hosting the xen worlds project [dissertação]. *Ames, USA: Iowa State University* (2011).

[26] Gueron, S.; Krasnov, V. Software implementation of modular exponentiation, using advanced vector instructions architectures. In *International Workshop on the Arithmetic of Finite Fields* (2012), vol. 12, Springer, pp. 119–135.

[27] Gupta, A.; Faraboschi, P.; Gioachin, F.; Kale, L. V.; Kaufmann, R.; Lee, B.-S.; March, V.; Milojicic, D.; Suen, C. H. Evaluating and improving the performance and scheduling of hpc applications in cloud. *IEEE Transactions on Cloud Computing 4*, 3 (2016), 307–321.

[28] Hager, G.; Jost, G.; Rabenseifner, R. Communication characteristics and hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. *Proceedings of Cray User Group Conference 4*, 500 (2009), 5455.

[29] Hammarlund, P.; Kumar, R.; Osborne, R. B.; Rajwar, R.; Singhal, R.; D'Sa, R.; Chappell, R.; Kaushik, S.; Chennupaty, S.; Jourdan, S., et al. Haswell: The fourth-generation intel core processor. *Micro 34*, 2 (2014), 6–20.

[30] Hamming, R. W. *Digital filters*. Courier Corporation, 1989.

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

[32] Hofmann, J.; Treibig, J.; Hager, G.; Wellein, G. Comparing the performance of different x86 simd instruction sets for a medical imaging application on modern multi and manycore chips. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing* (2014), ACM, pp. 57–64.

[33] Huh, J.-H.; Seo, K. Design and test bed experiments of server operation system using virtualization technology. *Human-centric Computing and Information Sciences 6*, 1 (2016), 1.

[34] Intel. The intel intrinsics guide. Disponível em: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. [Online; Acesso em: Mai. 2016].

[35] Jain, T.; Agrawal, T. The haswell microarchitecture–4th generation processor. *International Journal of Computer Science and Information Technologies 4*, 3 (2013), 477–480.

[36] Joy, A. M. Performance comparison between linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in* (2015), IEEE, pp. 342–346.

[37] Kon, J.; Mizusawa, N.; Umezawa, A.; Yamaguchi, S.; Tao, J. Highly consolidated servers with container-based virtualization. In *Big Data (Big Data), 2017 IEEE International Conference on* (2017), IEEE, pp. 2472–2479.

[38] Kozhirbayev, Z.; Sinnott, R. O. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems 68* (2017), 175–182.

[39] Kurtzer, G. M.; Sochat, V.; Bauer, M. W. Singularity: Scientific containers for mobility of compute. *PloS ONE 12*, 5 (2017), e0177459.

[40] Layton, J. A container for hpc, 2018. http://www.admin-magazine.com/HPC/Articles/Singularity-A-Container-for-HPC. Accessed March 27, 2018.

[41] Levin, S. A. Numerical precision in 3d prestack kirchhoff migration. In *SEG Technical Program Expanded Abstracts 2004*. Society of Exploration Geophysicists, 2004, pp. 1119–1122.

[42] Liu, H.; Li, B.; Liu, H.; Tong, X.; Liu, Q.; Wang, X.; Liu, W. The issues of prestack reverse time migration and solutions with graphic processing unit implementation. *Geophysical Prospecting 60*, 5 (2012), 906–918.

[43] Liu, W.; Nemeth, T.; Loddoch, A.; Stefani, J.; Ergas, R.; Zhuo, L.; Volz, B.; Pell, O.; Huggett, J. Anisotropic reverse-time migration using co-processors. In *SEG Annual Meeting* (2009), Society of Exploration Geophysicists.

[44] Lomont, C. Introduction to Intel Advanced Vector Extensions. *Intel White Paper* (2011).

[45] Louati, T.; Abbes, H.; Cérin, C.; Jemni, M. Lxcloud-cr: towards linux containers distributed hash table based checkpoint-restart. *Journal of Parallel and Distributed Computing 111* (2018), 187–205.

[46] Luszczek, P.; Meek, E.; Moore, S.; Terpstra, D.; Weaver, V. M.; Dongarra, J. Evaluation of the hpc challenge benchmarks in virtualized environments. In *European Conference on Parallel Processing* (2011), Springer, pp. 436–445.

[47] Maleki, S.; Gao, Y.; Garzar, M. J.; Wong, T.; Padua, D. A. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (2011), pp. 372–382.

[48] Manohar, N. A survey of virtualization techniques in cloud computing. In *Proceedings of international conference on vlsi, communication, advanced devices, signals & systems and networking (vcasan-2013)* (2013), Springer, pp. 461–470.

[49] Martin, J. P.; Kandasamy, A.; Chandrasekaran, K. Exploring the support for high performance applications in the container runtime environment. *Human-centric Computing and Information Sciences 8*, 1 (2018), 1.

[50] Mitra, G.; Johnston, B.; Rendell, A. P.; McCreath, E.; Zhou, J. Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-powered ARM and Intel Platforms. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013), IEEE, pp. 1107–1116.

[51] Moon, Y.; Yu, H.; Gil, J.-M.; Lim, J. A slave ants based ant colony optimization algorithm for task scheduling in cloud computing environments. *Human-centric Computing and Information Sciences 7*, 1 (2017), 28.

[52] Morabito, R.; Kjällman, J.; Komu, M. Hypervisors vs. lightweight virtualization: a performance comparison. In *Cloud Engineering* (2015), pp. 386–393.

[53] Nussbaum, L.; Anhalt, F.; Mornard, O.; Gelas, J.-P. Linux-based virtualization for hpc clusters. In *Montreal Linux Symposium* (2009).

[54] O'Connor, B. D.; Yuen, D.; Chung, V.; Duncan, A. G.; Liu, X. K.; Patricia, J.; Paten, B.; Stein, L.; Ferretti, V. The dockstore: enabling modular, community-focused sharing of docker-based genomics tools and workflows. *F1000Research 6* (2017).

[55] Panetta, J.; de Souza Filho, P. R.; da Cunha Filho, C. A.; Da Motta, F. M. R.; Pinheiro, S. S.; Pedrosa Jr, I.; Rosa, A. L. R.; Monnerat, L. R.; Carneiro, L. T.; De Albrecht, C. H. Computational characteristics of production seismic migration and its performance on novel processor architectures. In *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (2007), IEEE, pp. 11–18.

[56] Panetta, J.; Teixeira, T.; de Souza Filho, P. R. P.; da Cunha Filho, C. A.; Sotelo, D.; da Motta, F. M. R.; Pinheiro, S. S.; Rosa, A. L. R.; Monnerat, L. R.; Carneiro, L. T.; de Albrecht, C. H. B. Accelerating Time and Depth Seismic Migration by CPU and GPU Cooperation. *International Journal of Parallel Programming 40*, 3 (2012), 290–312.

[57] Rahmad, M. H.; Meng, S. S.; Karuppiah, E. K.; Ong, H. Comparison of cpu and gpu implementation of computing absolute difference. In *International Conference on Control System, Computing and Engineering (ICCSCE)* (2011), IEEE, pp. 132–137.

[58] Rastogi, R.; Phadke, S. Optimal aperture width selection and parallel implementation of kirchhoff migration algorithm. In *Fourth International Conference and Exposition of the Society of Petroleum Geophysicists* (2002), Citeseer, pp. 7–9.

[59] Regola, N.; Ducom, J.-C. Recommendations for virtualization technologies in high performance computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on* (2010), IEEE, pp. 409–416.

[60] Rizvandi, N. B.; Boloori, A. J.; Kamyabpour, N.; Zomaya, A. Y. MapReduce Implementation of Prestack Kirchhoff Time Migration (PKTM) on Seismic Data. In *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)* (2011), pp. 86–91.

[61] Rother, D. Using avx2 for the optimization of a software-based real-time ldpc decoder. In *8th Karlsruhe Workshop on Software Radios* (2014), pp. 1–8.

[62] Schuchart, J.; Waurich, V.; Flehmig, M.; Walther, M.; Nagel, W. E.; Gubsch, I. Exploiting Repeated Structures and Vectorization in Modelica. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015* (2015), no. 118 in Linköping Electronic Conference Proceedings, Linköping University Electronic Press, Linköping University Electronic Press, Linköpings universitet, pp. 265–272.

[63] Sharma, P.; Chaufournier, L.; Shenoy, P.; Tay, Y. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference* (2016), ACM, p. 1.

[64] Shi, X.; Li, C.; Wang, S.; Wang, X. Computing prestack Kirchhoff time migration on general purpose GPU. *Computers & Geosciences 37*, 10 (2011), 1702–1710.

[65] Silva, R. A. P.; Alves, M. M.; Bentes, C. B.; Drummond, L. M. A. Vetorização e análise de algoritmos paralelos para a migração kirchhoff pré-empilhamento em tempo. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD) 18*, 1/2017 (2017).

[66] Soltesz, S.; Pötzl, H.; Fiuczynski, M. E.; Bavier, A.; Peterson, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Operating Systems Review 41*, 3 (Mar. 2007), 275–287.

[67] Sun, P.; Shi, X. An OpenCL Approach of Prestack Kirchhoff Time Migration Algorithm on General Purpose GPU. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on* (2012), IEEE, pp. 179–183.

[68] Tanikawa, A.; Yoshikawa, K.; Okamoto, T.; Nitadori, K. N-body simulation for self-gravitating collisional systems with a new simd instruction set extension to the x86 architecture, advanced vector extensions. *New Astronomy 17*, 2 (2012), 82–92.

[69] Tavarageri, S.; Hartono, A.; Baskaran, M.; Pouchet, L.-N.; Ramanujam, J.; Sadayappan, P. Parametric tiling of affine loop nests. In *Proceedings of 15th Workshop on Compilers for Parallel Computers* (2010).

[70] Teixeira, D.; Yeh, A.; Gajawada, S. Implementation of Kirchhoff Prestack Depth Migration on GPU. *SEG Technical Program Expanded Abstracts 3683* (2013), 3686.

[71] Turban, E.; King, D.; Lee, J.; Viehland, D. Chapter 19: Building e-commerce applications and infrastructure. *Electronic Commerce A Managerial Perspective 27* (2008).

[72] Xavier, M. G.; Neves, M. V.; Rossi, F. D.; Ferreto, T. C.; Lange, T.; De Rose, C. A. F. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on* (2013), IEEE, pp. 233–240.

[73] Xu, R.; Hugues, M.; Calandra, H.; Chandrasekaran, S.; Chapman, B. Accelerating Kirchhoff Migration on GPU using Directives. In *Accelerator Programming using Directives (WACCPD), 2014 First Workshop on* (2014), IEEE, pp. 37–46.

[74] Ye, K.; Jiang, X.; Chen, S.; Huang, D.; Wang, B. Analyzing and modeling the performance in xen-based virtual cluster environment. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on* (2010), IEEE, pp. 273–280.

[75] Yerneni, S.; Phadke, S.; Bhardwaj, D.; Chakraborty, S.; Rastogi, R. Imaging subsurface geology with seismic migration on a computing cluster. *Current Science 88*, 3 (2005), 468–474.

[76] Yilmaz, O.; Doherty, S. M. *Seismic Data Processing*, vol. 2 of *Investigations in Geophysics*. Society of Exploration, 1987. ISBN 0-9318-3040-0.

[77] Younge, A. J.; Pedretti, K.; Grant, R. E.; Brightwell, R. A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds. In *Cloud Computing Technology and Science (CloudCom), 2017 IEEE International Conference on* (2017), IEEE, pp. 74–81.

[78] Zhang, J.; Lu, X.; Panda, D. K. Performance characterization of hypervisor-and container-based virtualization for hpc on sr-iov enabled infiniband clusters. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International* (2016), IEEE, pp. 1777–1784.