UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PABLO YURI SCHERER DE SOUZA

# Análise do Desempenho de uma Aplicação Geofísica com Paralelismo em Tarefas

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Arthur Francisco Lorenzon

Porto Alegre
2024

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Helena Lucas Pranke
Pró-Reitora de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Marcelo Walter
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

*“If I have seen farther than others,*
*it is because I stood on the shoulders of giants.”*
— Sir Isaac Newton

## RESUMO

Na busca por obter cada vez melhor desempenho computacional, o desenvolvimento de técnicas que possam tornar as aplicações mais velozes e eficientes é um ponto essencial. Na área da pesquisa geofísica, a modelagem Fletcher é uma aplicação importante na identificação de estruturas geológicas e reservatórios de petróleo e gás. Para o funcionamento prático da modelagem Fletcher são necessários resultados rápidos e precisos; sendo assim, sua implementação em *software* tira proveito do paralelismo *fork-join*. Contudo, essa abordagem pode ser pouco maleável em relação a recursos computacionais como sincronização de dados e contenção de cache. Neste sentido, este trabalho apresenta uma alternativa inovadora à implementação da modelagem Fletcher baseada em OpenMP, usando paralelismo em tarefas com OmpSs-2, comparando em seguida o desempenho do software sob diferentes entradas e diferentes configurações de balanceamento de carga. Os experimentos realizados evidenciaram uma redução significativa no tempo de execução, com ganhos que chegam a 14,55% em determinadas condições de simulação.



**Palavras-chave:** Paralelismo em tarefas. Computação de Alto Desempenho. Aplicações Geofísicas. Paralelismo em Tarefas.

## ABSTRACT

In the quest to achieve increasingly better computational performance, the development of techniques that can make applications faster and more efficient is an essential focus. In the field of geophysical research, the Modelagem Fletcher is a critical application for identifying geological structures and oil and gas reservoirs. For the practical operation of the Modelagem Fletcher, fast and accurate results are necessary; therefore, its implementation in software leverages fork-join parallelism. However, this approach may not be very flexible concerning computational resources such as data synchronization and cache contention. In this context, this work presents an innovative alternative for implementing Fletcher modeling based on OpenMP, utilizing task parallelism with OmpSs-2, and subsequently comparing software performance under different inputs and various load balancing configurations. The experiments conducted showed a significant reduction in execution time, with gains reaching 14.55% under certain simulation conditions.



**Keywords:** Parallelism. High performance computing. Geophysical Applications. Task-based Parallelism.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | Application Programming Interface |
| CPU | Central Processing Unit |
| SMP | Symmetric Multiprocessors |
| RAM | Random Access Memory |
| SMT | Simultaneous Multithreading |
| PCAD | Parque Computacional de Alto Desempenho |
| EDP | Equações Diferenciais Parciais |
| TBB | Threading Building Blocks |
| ARB | Architecture Review Board |

**SUMÁRIO**

# 1 INTRODUÇÃO

Na busca por alcançar desempenho cada vez melhor das aplicações, a ciência da computação tem evoluído constantemente ao longo dos anos, possibilitando operações com grandes volumes de dados, execução de cálculos intensivos e a simulação de fenômenos complexos. Um desses fenômenos é a propagação de ondas sísmicas, cuja simulação é base para as ferramentas de imagem sísmica. Assim, otimizar o desempenho de tais aplicações possibilita resultados mais rápidos e precisos, relevantes na exploração de recursos energéticos como o petróleo e gás, que tem custos muito elevados e alto risco (LUKAWSKI et al., 2014).

A programação paralela surge como uma estratégia eficiente para solucionar desafios complexos, devido à sua capacidade de dividir o trabalho entre diferentes processadores, possibilitando a execução de múltiplas tarefas simultaneamente e diminuindo o tempo de execução (SEVERANCE; DOWD, 2010). Tradicionalmente, é possível explorar a programação paralela usando o modelo *fork-join*, que apresenta um fluxo de execução principal, no qual, quando necessário, novas *threads* são criadas para executar partes do problema em paralelo. O fluxo principal é responsável por dividir o trabalho e iniciar novas *threads* (*fork*), que executam suas tarefas de forma paralelamente. Após a conclusão das tarefas paralelas, o fluxo principal espera que todas as *threads* terminem (join) para então consolidar os resultados e desfazer essas *threads* (MCCOOL; REINDERS; ROBISON, 2012). Todavia, essa abordagem não é a única, podendo ser bastante rígida e pouco maleável em aspectos de hardware e software, como na sincronização de dados e contenção de cache (LORENZON et al., 2022).

Assim, o paralelismo baseado em tarefas surge como uma alternativa ao modelo *fork-join*, implementado por APIs (*Application Programming Interface*) como OpenMP e TBB (*Intel Threading Building Blocks*). Diferentemente do *fork-join*, o modelo de tarefas adota uma abordagem mais flexível onde o fluxo de execução principal cria um *pool* de *threads* (BSC, 2024) uma única vez no início da execução e então as tarefas são distribuídas dinamicamente entre essas *threads* disponíveis durante a execução do programa, assim, eliminando a necessidade de criar e destruir *threads*. APIs como OmpSs-2 (BSC, 2024) adotam esse modelo, pois o seu uso pode trazer flexibilidade e a dinamismo na alocação de recursos que são essenciais para o desempenho da aplicação.

Um dos exemplos de paralelismo aplicando o modelo *fork-join* é a Modelagem Fletcher, que simula a propagação de ondas acústicas, essencial em ferramentas de imagem sísmica utilizadas pela indústria de petróleo e gás e está atualmente implementada com OpenMP para arquiteturas multicore (SERPA et al., 2019a). Essa modelagem é baseada na solução de Equa-

ções Diferenciais Parciais (EDPs) que descrevem a propagação de ondas acústicas através de diferentes camadas geológicas (FLETCHER; DU; FOWLER, 2009). Nesse sentido, a motivação para esta pesquisa surge da ausência de implementações da modelagem Fletcher que utilizem paralelismo em tarefas.

## 1.1 Objetivos

O objetivo geral do trabalho é implementar e comparar a Modelagem Fletcher usando o paralelismo em tarefas com o modelo *fork-join*. Este objetivo geral desdobra-se em três objetivos específicos:

1. Coletar os resultados relacionados à contenção de cache e desempenho da Modelagem Fletcher na sua versão em OpenMP aplicando paralelismo *fork-join* e obtendo os valores base para comparação;
2. Implementar a Modelagem Fletcher usando modelo de execução em tarefas com OmpSs-2, otimizando as diretivas do OmpSs-2 para atingir o maior balanceamento de carga;
3. Coletar os resultados relacionados à contenção de cache e desempenho da nova implementação em tarefas mais otimizada e comparar com os resultados da implementação em *fork-join*.

## 1.2 Organização do texto

Este trabalho está organizado como segue: no capítulo 2, apresenta o estado da arte de conceitos essenciais na área de melhoria de desempenho computacional por paralelismo, no capítulo 3, os artigos que motivaram e complementam esse trabalho e, no capítulo 4, apresenta a implementação da Modelagem Fletcher na perspectiva dos modelos fork-join e tarefas. O capítulo 5 apresenta a Metodologia adotada e a capítulo 6 apresenta e discute os resultados obtidos por meio da contribuição apresentada. O capítulo 7 apresenta a conclusão do trabalho e indicação de trabalhos futuros.

# 2 REFERENCIAL TEÓRICO

Esse capítulo discute os conceitos básicos para a realização desta pesquisa. Dentre estes conceitos destacam arquiteturas *multicore*, hierarquia de memória, computação paralela e Modelagem Fletcher. Inicia-se com a apresentação das arquiteturas *multicore*.

## 2.1 Arquiteturas Multicore

As arquiteturas multinúcleo (*multicore*) são arquiteturas que integram dois ou mais núcleos de processamento em um único *chip* ou unidade de processamento, permitindo a execução simultânea de várias tarefas (RAUBER; RÜNGER, 2013). Enquanto os sistemas mononúcleo enfrentam limitações de desempenho devido à estagnação na melhoria da frequência de *clock* e eficiência energética, as arquiteturas multicore oferecem uma abordagem prática para aumentar a capacidade de processamento e atender às crescentes demandas por poder computacional (CLEMENTS et al., 2017).

Os processadores multinúcleo, ao compartilharem uma memória física comum, se beneficiam da capacidade de comunicação rápida entre núcleos através de variáveis compartilhadas na memória. Contudo, a natureza dessa comunicação e a coordenação necessária entre os núcleos pode variar conforme a arquitetura do sistema multiprocessador. Nos sistemas de memória compartilhada, conhecidos como SMPs *Symmetric Multiprocessors*, os núcleos compartilham um único espaço de endereço físico, permitindo que cada processador acesse qualquer local da memória através de operações de carga e armazenamento (*loads* e *stores*) (PATTERSON; HENNESSY, 2016).

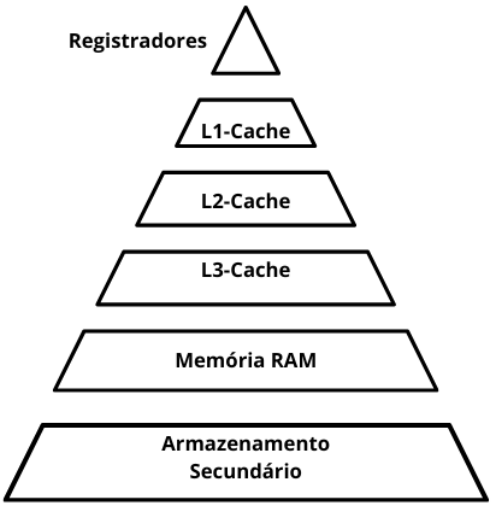
No entanto, o desenvolvimento de *software* para arquiteturas multinúcleo apresenta desafios como a sincronização e balanceamento de carga precisam ser cuidadosamente gerenciados para não reduzir os potenciais ganhos de desempenho que são introduzidos com o paralelismo e manter a corretude do resultado (PATTERSON; HENNESSY, 2016).

A sincronização garante que múltiplos processadores não acessem simultaneamente dados compartilhados de maneira desordenada, evitando, assim, condições de corrida e garantindo a consistência dos dados. Enquanto o balanceamento de carga refere-se à distribuição equitativa das tarefas entre os núcleos de processamento para evitar que alguns núcleos fiquem sobrecarregados enquanto outros permanecem subutilizados.

## 2.2 Hierarquia de Memória

A hierarquia de memória em sistemas computacionais é composta por várias camadas de armazenamento, cada uma com diferentes características de capacidade, velocidade e custo. No topo da hierarquia estão os registradores da CPU (*Central Processing Unit*) com menor capacidade de armazenamento, porém, menor latência para acessar os dados, seguidos pela memória *cache*, que é subdivida em 3 níveis (L1, L2 e L3). Abaixo da *cache* está a memória principal (RAM - *Random Access Memory*) e, por fim, o armazenamento secundário contando com dispositivos como discos rígidos que tem maiores capacidades de armazenamento, porém, maior latência no acesso aos dados (TOY; ZEE, 1986).

Figura 2.1 – Hierarquia de Memória




Fonte: o autor

A organização da memória em níveis hierárquicos representa importante papel na velocidade com que os dados são acessados pela CPU. A *cache*, por exemplo, é projetada para armazenar dados e instruções frequentemente acessados pelas CPUs, reduzindo, assim, o tempo de acesso à memória principal. No entanto, a gestão eficiente da *cache* em sistemas *multicore* é um desafio devido à competição por recursos entre os diferentes núcleos de processamento (TOY; ZEE, 1986).

Em arquiteturas *multicore*, um dos principais desafios é a coordenação eficiente da *cache* entre os núcleos de processamento quando vários núcleos competem pelo acesso aos recursos de memória. A contenção de memória pode resultar em gargalos de desempenho, especialmente em situações onde múltiplos núcleos estão acessando a mesma região de memória simultaneamente. Para enfrentar esses desafios, são necessárias técnicas de otimização de

localidade de memória.

### 2.2.1 Localidade de referência

A localidade de referência é um conceito computacional que descreve a tendência de um programa de acessar frequentemente um conjunto de endereços de memória em um curto período de tempo (STALLINGS, 2009). Essa tendência é baseada na observação de que os programas exibem padrões previsíveis de acesso à memória, o que pode ser explorado para melhoria de desempenho. De acordo com (DENNING, 2005) ocorrem dois tipos principais de localidade:

- Localidade Temporal: refere-se à tendência de um programa de acessar repetidamente os mesmos dados ou instruções em curtos intervalos de tempo. Isso significa que os dados que foram acessados recentemente têm maior probabilidade de serem acessados novamente em um futuro próximo. A localidade temporal é frequentemente observada em laços de repetição e estruturas de dados como pilhas e *buffers*;
- Localidade Espacial: refere-se à tendência de um programa de acessar dados que estão próximos uns dos outros em endereços de memória. Por exemplo, quando um programa acessa um determinado endereço de memória, é provável que ele também acesse endereços adjacentes em um padrão sequencial (localidade espacial por proximidade) ou em blocos de dados próximos (localidade espacial por bloco). A localidade espacial é comumente observada em *arrays*, estruturas de dados e funções que acessam memória de forma contígua.

## 2.3 Computação Paralela

A computação paralela é uma abordagem na qual várias tarefas computacionais são executadas simultaneamente e de maneira concorrente, assim aumentando a eficiência e o desempenho dos sistemas de computação. E para expressar a computação paralela a nível de algoritmos usamos a programação paralela, que é capaz de dividir o trabalho entre múltiplos processadores ou computadores separados. Ao invés de realizar uma tarefa após a outra, como na programação sequencial, a computação paralela permite que várias tarefas sejam realizadas ao mesmo tempo, acelerando assim o processamento de grandes volumes de dados e/ou alto número de instruções. Embora essa abordagem tenha benefícios claros, uma nova gama de de-

safios também é apresentada ao programador que optar por seguir essa abordagem, tais como problemas de sincronização, concorrência de recursos, divisão de tarefas e balanceamento de carga (PATTERSON; HENNESSY, 2016).

Nesse sentido, dois modelos de programação paralela são mais comuns: o modelo *fork-join* e o modelo de tarefas.

### 2.3.1 Modelo Fork-Join

O modelo fork-join é uma maneira flexível de paralelizar código. No modelo *fork-join*, a execução de uma tarefa começa com a fase de "*fork*", onde a tarefa principal é dividida em múltiplas sub-tarefas. Essas sub-tarefas podem, por sua vez, ser divididas em sub-tarefas das sub-tarefas, continuando essa divisão até que as tarefas sejam pequenas o suficiente para serem executadas diretamente e de forma eficiente. Após a divisão, as sub-tarefas são executadas em paralelo, utilizando múltiplos *threads* ou processos, dependendo da infraestrutura e das ferramentas de paralelismo empregadas e os resultados são combinados na fase de "*join*"para formar o resultado final. Esse processo de combinação geralmente começa com as sub-tarefas menores e se move em direção à tarefa principal, agregando resultados ao longo do caminho (MCCOOL; REINDERS; ROBISON, 2012).

O modelo *fork-join* oferece várias vantagens. Primeiramente, sua simplicidade conceitual facilita a compreensão e a implementação de paralelismo em problemas divisíveis recursivamente. Outra vantagem é a escalabilidade, pois o modelo pode escalar bem com o aumento do número de núcleos, desde que a tarefa possa ser suficientemente dividida. No entanto, o modelo *fork-join* também apresenta desafios, como o *overhead* de criação de tarefas (SHOSHANY, 2024). A criação e a destruição frequente de muitas sub-tarefas podem introduzir um *overhead*, reduzindo a eficiência geral.

Outro desafio é o balanceamento de carga: dividir o trabalho de forma equitativa pode ser complicado, pois algumas sub-tarefas podem ser significativamente mais complexas ou demoradas que outras, resultando em um balanceamento de carga desigual e subutilização de recursos (DURAN et al., 2008). Ainda, a fase de combinação (*join*) pode causar *threads* inativas enquanto aguardam outras sub-tarefas terminarem. Encontrar o nível correto de granularidade para dividir as tarefas é essencial, pois tarefas muito pequenas podem aumentar o *overhead* de gerenciamento, enquanto tarefas muito grandes podem não explorar adequadamente o paralelismo disponível.

Esses desafios podem limitar o desempenho do modelo *fork-join* em determinados ce-

nários e aplicações e, para que essas limitações possam ser contornadas, são necessárias estratégias de otimização. Por este motivo, na próxima sessão será explorado o modelo de tarefas, que aborda algumas dessas dificuldades de maneiras diferentes.

### 2.3.2 Modelo de Tarefas

O modelo de tarefas é uma abordagem de programação paralela que propõe alternativas a algumas das limitações do modelo *fork-join*, oferecendo maior flexibilidade e eficiência no gerenciamento de trabalho paralelo. Nesse modelo, o trabalho é dividido em pequenas unidades chamadas "tarefas". Cada tarefa representa uma unidade de trabalho discreta e autossuficiente, o que facilita a adaptação a diferentes tipos de problemas paralelos (AYGUADÉ et al., 2007).

Uma das principais características do modelo de tarefas é o uso de um *pool* (fila) de tarefas. Inicialmente, o fluxo de execução principal cria um *pool* de *threads* uma única vez no início da execução e então as tarefas são distribuídas dinamicamente entre essas *threads* disponíveis, que são chamadas de *workers*, durante a execução do programa, eliminando, assim, a necessidade de criar e destruir *threads* (DURAN et al., 2008).

As tarefas são colocadas nessa fila e retiradas pelos *workers* conforme ficam disponíveis para execução. Esse mecanismo permite um balanceamento de carga mais dinâmico e eficaz, pois os *workers* podem buscar novas tarefas assim que concluem suas tarefas atuais reutilizando, *threads* ao longo da execução do programa (DURAN et al., 2008). Isso minimiza o *overhead* associado à criação e destruição de *threads*, resultando em um uso mais eficiente dos recursos do sistema. O modelo de tarefas também oferece maior flexibilidade na definição das dependências entre as tarefas. Em muitos casos, as tarefas podem ser executadas de forma completamente independente, mas, quando há dependências, estas podem ser expressas explicitamente.

Apesar das suas vantagens, o modelo de tarefas também apresenta desafios. Um dos principais desafios é a complexidade de programação. Escrever programas paralelos usando o modelo de tarefas pode ser mais complexo do que usar modelos mais intuitivos como o *fork-join*. É necessário definir as tarefas, suas dependências e como elas serão agendadas e executadas, necessitando muitas vezes, uma compreensão completa do algoritmo.

A implementação dos modelos de programação paralela *fork-join* e de tarefas pode ser implementada de várias maneiras, sendo duas delas as APIs OpenMP (Open Multi-Processing) e OmpSs-2, ambas descritas a seguir.

## 2.4 OpenMP

O OpenMP (Open Multi-Processing), lançado em 1997, é uma API para programação paralela em memória compartilhada. O OpenMP, fornece um conjunto de diretivas de compilador e rotinas de biblioteca que estendem Fortran, C e C++ para expressar paralelismo em memória compartilhada, permitindo que desenvolvedores paralelizem código existente de maneira incremental sem a necessidade de reescrita completa (DAGUM; MENON, 1998; CHANDRA, 2001).

O padrão OpenMP é mantido pelo OpenMP ARB (Architecture Review Boards), cujo conselho de administração inclui representantes de muitos dos principais fornecedores de hardware e software. Embora tenha sido projetado para computação paralela em um computador com vários núcleos, o OpenMP também pode ser parte de uma solução de paralelismo entre nós num sistema distribuído e abranger uma grande variedade de aplicações, incluindo o desenvolvimento para computação de alto desempenho (CHANDRA, 2001; CAMERON, 2022).

Nesse sentido é possível dar destaque a duas diretivas do OpenMP (BOARD, 2018) que serão abordadas pelo trabalho:

- A diretiva `#pragma omp parallel` é usada para iniciar uma região paralela, ou seja, um bloco de código que será executado simultaneamente por múltiplas *threads*. Quando uma *thread* principal encontra essa diretiva, ela cria as threads que executam o código dentro da região paralela. A criação de *threads* e sua sincronização são gerenciadas automaticamente pelo *runtime* do OpenMP, permitindo ao programador focar na lógica paralela sem se preocupar com os detalhes de gerenciamento de *threads*.
- A diretiva `for`, por sua vez, é utilizada para paralelizar *loops*. Quando combinada com a diretiva *parallel*, como em `parallel for`, ela divide as iterações do *loop* entre as *threads* criadas pela diretiva `parallel`. Cada *thread* executa uma parte das iterações de forma independente, aproveitando o paralelismo para acelerar o processamento. A combinação de `parallel` e `for` é uma prática comum em OpenMP para otimizar o desempenho de *loops* que possuem iterações independentes, permitindo uma utilização eficiente dos recursos de hardware disponíveis.

Apesar de o OpenMP ter introduzido o conceito de *tarefas*, permitindo a criação de unidades de trabalho que podem ser executadas de forma assíncrona e dinâmica, sua execução ainda segue o modelo tradicional *fork-join*. Isso ocorre porque, no final, as *tarefas* são gerenciadas dentro de uma região paralela, onde um conjunto de *threads* é criado (*fork*) para executar

as *tarefas*, e, em seguida, essas threads são sincronizadas (*join*) antes de sair da região paralela. Portanto, embora as *tarefas* ofereçam maior flexibilidade e eficiência na divisão do trabalho, a estrutura subjacente de *fork-join* permanece intacta no OpenMP.

## 2.5 OmpSs-2

OmpSs-2 é uma API para programação paralela composto por um conjunto de diretivas e rotinas de biblioteca que podem ser utilizadas em conjunto com uma linguagem de programação de alto nível, a fim de desenvolver aplicações concorrentes. Foi concebida com base em dois modelos de programação: OpenMP e StarSs, pelo grupo de Modelos de Programação do departamento de Ciências da Computação do Centro de Supercomputação de Barcelona (BSC), incorporando seus princípios de *design* fundamentais. O modelo OmpSs-2 adota a abordagem do OpenMP de transformar um programa sequencial em uma versão paralela por meio de diretivas de compilador e incorpora o modelo execução de tarefas do StarSs, que proporciona paralelismo assíncrono como principal mecanismo, além de sincronizar tarefas por meio de dependências. As tarefas são a unidade elementar de trabalho que representa uma instância específica de um código executável, enquanto que as dependências são capazes de identificar tarefas predecessoras e sucessoras possibilitando o correto fluxo de dados do programa, evitando, assim - que duas tarefas causem uma corrida de dados (BSC, 2024).

Os conceitos relativos ao desempenho (programas desenvolvidos em OmpSs-2 devem ser capazes de oferecer desempenho razoável quando comparados com outros modelos de programação para as mesmas arquiteturas) e à facilidade de uso (o modelo foi concebido utilizando princípios elogiados pela eficácia) tornam o OmpSs-2 um modelo de programação produtivo. De acordo com (BSC, 2024), as características mais relevantes do modelo OmpSs-2 são:

- tempo de vida do ambiente de dados da tarefa: uma tarefa é concluída quando a última instrução do seu corpo é executada, estando completamente concluído quando as tarefas-filhas também estiverem finalizadas. Enquanto isto não ocorre, o ambiente de dados da tarefa é preservado até a finalização da mesma (observe-se que a pilha da thread que está executando a tarefa não faz parte do ambiente de dados da tarefa);
- ligação de domínios de dependência aninhados: as dependências de entrada de uma tarefa propagam-se às tarefas-filhas como se a tarefa não existisse. Quando uma tarefa termina, as suas dependências de saída são substituídas pelas geradas pelas tarefas-filhas;
- liberação antecipada de dependências: por padrão, quando uma tarefa é concluída, libera

todas as dependências que não estão incluídas em nenhuma tarefa descendente inacabada. Se a cláusula de espera for especificada na construção da tarefa, todas as suas dependências serão liberadas de uma só vez quando a tarefa estiver completamente concluída.

- dependências fracas: as cláusulas weakin/weakout especificam potenciais dependências apenas requeridas por tarefas descendentes. Estas anotações não atrasam a execução da tarefa.

Com relação à abordagem adotada neste trabalho, é possível dar destaque às seguintes diretivas do OmpSs-2 (BSC, 2024):

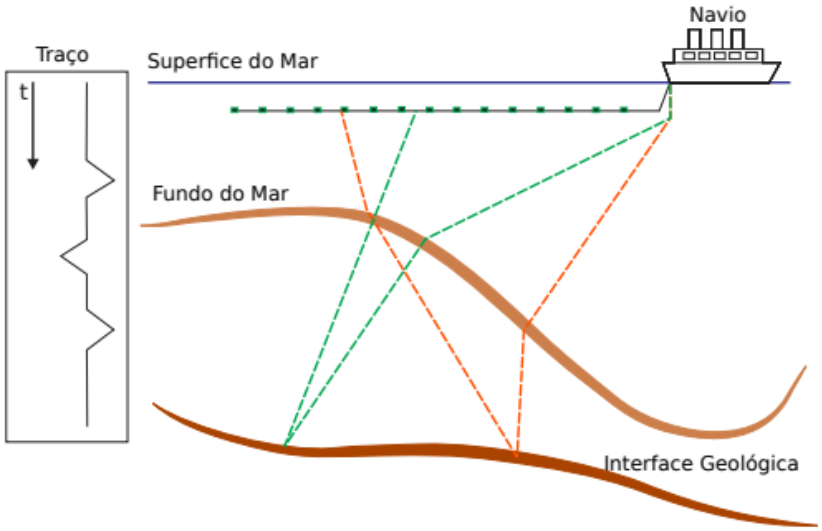
- A diretiva *#pragma oss task* que é utilizada para definir uma tarefa, uma unidade de trabalho que pode ser executada de forma assíncrona.
- A diretiva *#pragma oss taskloop* que é uma expansão da diretiva prévia, transformando cada iteração do *loop* da linha inferior a diretiva em uma tarefa. Sendo possível também, adicionar parâmetros como *grainsize* para delimitar quantas iterações cada tarefa criada irá receber, e *collapse* que adiciona a funcionalidade de juntar dois ou mais *loops* aninhados que estão imediatamente abaixo dessa diretiva.
- A diretiva *#pragma oss taskwait* que é essencial para sincronização, uma vez que a execução encontra essa diretiva, ele espera que todas outras tarefas criadas previamente estejam completas.

## 2.6 Modelagem Fletcher

O campo da geofísica busca por recursos energéticos como gás e petróleo, mas há altos custos de perfuração e baixa taxa de precisão e sucesso (LUKAWSKI et al., 2014). Sendo assim, esta área se utiliza de diversas técnicas e métodos computacionais para simular a coleta de dados em levantamentos sísmicos obtendo informações detalhadas sobre diferentes estruturas geológicas na exploração de recursos energéticos. Neste sentido, algoritmos são adotados pela indústria da área para reduzir os custos de exploração por meio do simulações de reconhecimento, sendo um deles a modelagem Fletcher (FLETCHER; DU; FOWLER, 2009).

A modelagem Fletcher simula a coleta de dado no mar em um levantamento sísmico, com equipamentos acoplados ao navio que emitem, de tempos em tempos, ondas que refletem e refratam mudanças de meio no subsolo. Quando as ondas retornam à superfície marinha, são coletadas por microfones específicos acoplados a cabos rebocados pelo navio. O conjunto

Figura 2.2 – Coleta de dados em levantamento sísmico marítimo




Fonte: (SERPA et al., 2019a)

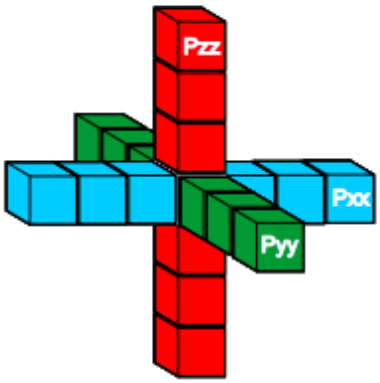
de sinais em cada fone ao longo do tempo constitui um traço sísmico e para cada emissão de ondas, gravam-se os traços sísmicos de todos os fones do cabo, enquanto o navio continua seu trajeto emitindo sinais ao longo do tempo (Figura 2.2) (SERPA et al., 2019a).

A aplicação baseia-se na solução de equações diferenciais parciais que descrevem a propagação das ondas através de diferentes camadas geológicas. Isso é efetuado pela realização de cálculos de derivadas cruzadas, que demandam menor tempo de execução por meio da redução da quantidade de operações para calcular as derivadas cruzadas (FLETCHER; DU; FOWLER, 2009). Essa abordagem tem sido amplamente utilizada na indústria de exploração de petróleo e gás, simulando a propagação de ondas acústicas ao longo do tempo para aplicações em imagens sísmicas (SCHUSSLER et al., 2024).

Nesse sentido, para se discretizar a computação equações diferenciais parciais a aplicação faz uso de *stencils* (Figura 2.3), que é uma configuração de pontos na vizinhança de um ponto central, onde cada ponto contribui para a atualização ou cálculo do valor no ponto central. O *stencil* define a forma como os pontos vizinhos são ponderados e combinados para calcular o novo valor do ponto de interesse (MARTÍNEZ et al., 2017).

O pseudo-código 1 demonstra como a Modelagem Fletcher realiza a propagação de onda utilizando computação *stencil* em um contexto temporal e espacial. Este algoritmo itera sobre pontos em uma grade tridimensional (3D) espacial ao longo de uma dimensão temporal (1D), aplicando uma função stencil $\mathcal{S}$ para calcular os novos valores das ondas em cada ponto da grade. No pseudo-código, $D_t$ representa o número de passos no tempo, enquanto $D_x$, $D_y$ e $D_z$ denotam os tamanhos dos problemas nas dimensões espaciais (YOUNT; DURAN; TOBIN,

Figura 2.3 – *Stencil* 3D de 7 pontos




Fonte: (SERPA et al., 2019a)

2019).

**Algorithm 1** – Aplicação conceitual de uma única computação stencil 1D temporal + 3D espacial

```
1: for t = 1 to D_t do
2:     for i = 1 to D_x do
3:         for j = 1 to D_y do
4:             for k = 1 to D_z do
5:                 S(t, i, j, k)
```

O *software* base da implementação da Modelagem Fletcher (SERPA et al., 2019b) está implementado C, OpenMP, OpenACC e CUDA. Os arquivos fonte são organizados em uma árvore de diretórios. Na pasta `original`, dentro da raíz do projeto, encontram-se os arquivos comuns às quatro implementações (*front-end*), escritos em C, e diretórios com arquivos específicos de cada API (*back-ends*) sendo elas OpenMP, OpenACC e CUDA.

Para a escolher o seu (*back-end*) é necessário alterar o arquivo (`Makefile`) preenchendo o campo `backend=` com o nome da API desejada.

Como valores de entrada a aplicação requer os seguintes argumentos descritos em (PANETTA, 2018):

- fNameSec (*string*): especifica o tipo de anisotropia a ser utilizada na simulação. As opções sendo: "ISO"para meios isotrópicos, "VTI"para isotropia transversal vertical e "TTI"para isotropia transversal inclinada. Este parâmetro define as propriedades do meio em que a modelagem será realizada e, portanto, influencia diretamente nas equações utilizadas e nos resultados obtidos;
- nx, ny, nz (*integer*): definem o número de pontos da grade nas direções x, y e z, respecti-

vamente. A grade é a estrutura de discretização espacial onde as equações da modelagem são resolvidas e o número de pontos em cada direção determina a resolução espacial da simulação;

- absorb (*integer*): indica o número de pontos na borda absortiva. A borda absortiva é usada para minimizar reflexões indesejadas nas bordas do domínio de simulação, proporcionando um ambiente mais realista ao simular a propagação de ondas;
- dx, dy, dz (*float*): representam a distância entre pontos consecutivos na grade nas direções x, y e z, respectivamente, em metros, definindo a resolução espacial da grade, afetando a precisão da simulação e o tempo de execução;
- dt (*float*): especifica o passo no tempo, em segundos. O passo no tempo é importante para a estabilidade e precisão da simulação. Um passo muito grande pode levar a instabilidades numéricas, enquanto um passo muito pequeno pode aumentar significativamente o tempo de execução;
- tMax (*float*): define o tempo máximo de integração, em segundos, determinando a duração total da simulação, ou seja, até quando as equações serão resolvidas ao longo do tempo;

Os arquivos *front-ends* implementam funcionalidades comuns, como inicializar a computação, percorrer o laço de propagação no tempo e finalizar a computação. As funções dos *back-ends* são invocadas conforme necessário, por exemplo, para propagar o campo de onda. Os *front-ends* conectam-se aos *back-ends* através de *drivers*, que são funções com assinatura e semântica padronizadas, e cada *back-end* implementa os *drivers* na sua linguagem, resultando nas semânticas desejadas. Os cinco *drivers* necessários são: `Initialize`, `Finalize`, `Propagate`, `UpdatePointers` e `InsertSource`. Os *drivers* lidam com funcionalidades específicas de cada arquitetura e linguagem, como alocação de memória e gerenciamento de regiões paralelas. O *driver* `Propagate` propaga o campo de ondas um passo no tempo para todos os pontos da grade (PANETTA PEDRO LOPES, 2019).

O fluxo principal do programa começa com o arquivo `main.c` do *frontend* que é responsável por ler os dados de entrada, definir o tipo de anisotropia (ISO, VTI, TTI), calcular o tamanho total do domínio (variáveis `sx`, `sy`, `sz`) e o número de passos no tempo. Após, a fonte é posicionada no ponto central da grade e os *arrays* para coeficientes de campos de pressão e auxiliar (variáveis pp, pq, pc) são alocados e inicializados conforme o tipo de anisotropia. O programa também calcula a condição de estabilidade e implementa a velocidade aleatória na borda absortiva (PANETTA, 2018) no final de `main.c` é chamada a função `Model`.

Em `model.c`, dentro da função `Model`, são alocados *arrays* para os coeficientes das derivadas em H1, calcula os coeficientes de H1 e H2 nas duas equações diferenciais parciais da formulação de Fletcher e iniciamos o *loop* que avança a propagação no tempo. A cada iteração desse *loop*, é inserida a fonte sísmica, chamada a função `Propagate` que propaga a onda conforme as formulações de Fletcher (FLETCHER; DU; FOWLER, 2009) e é realizada a troca os *arrays* que contém os coeficientes de campos de pressão e auxiliar, do instante atual para o futuro (entre `pp` e `pc` e entre `qp` e `qc`).

No *back-end* de OpenMP, que é o foco do trabalho, a função `Propagate` inicia com um ninho de laços para percorrer a grade e inclui diretivas para paralelismo. O interior do ninho de laços inclui o arquivo `sample.h`, que contém todos os cálculos que resolvem as equações diferenciais parciais discretas, sendo esse o trecho que o pseudo-código 1 descreve, em que $\mathcal{S}(t, i, j, k)$ se refere ao arquivo `sample.h` (PANETTA, 2018). Por fim, no último instante de tempo são escritos os *arrays* tridimensionais dos campos de pressão para dois arquivos de saída no formato RFS, um contendo os *arrays* e outro contendo informações sobre a execução.

# 3 TRABALHOS RELACIONADOS

Esse capítulo apresenta os trabalhos que motivaram e complementam o tema em questão, destacando as contribuições relevantes e identificando lacunas que justificam a abordagem proposta no presente estudo.

Uma avaliação das compensações de desempenho e energia em arquiteturas heterogêneas big.LITTLE com OpenMP e OmpSs demonstrou que o OmpSs apresenta melhorias significativas em relação ao OpenMP, especialmente se as melhores configurações do OmpSs-2 forem selecionadas (BUTKO et al., 2017). Nosso trabalho se apoia nessa ideia, buscando usar os melhores parâmetros do OmpSs-2 na aplicação sísmica Modelagem Fletcher para alcançar maiores desempenhos em relação ao OpenMP.

Em (SERPA et al., 2019a) é demonstrado que a maioria do tempo de execução da Modelagem Fletcher é concentrado na função de propagação de onda. Além disso, foram pesquisadas técnicas de otimização em CPU aplicando estratégias para usar melhor a memória cache, vetorização, balanceamento de carga e localidade de memória em (SERPA et al., 2019b). Nosso trabalho se apoia em algumas dessas estratégias e explora o paralelismo em tarefas em busca de ganhos de desempenho.

No contexto de aplicações sísmicas (PHILIPPE et al., 2016) foi realizado na simulação de propagação de ondas sísmicas utilizando os processadores *manycore* MPPA-256 e Xeon Phi. O trabalho aborda os estratégias de otimização específicas para cada arquitetura, evidenciando a importância de estratégias personalizadas para maximizar o desempenho. Nosso trabalho se relaciona com o contexto de aplicações sísmicas em sistemas *manycores*, porém buscamos medir o impacto do uso de modelos de execução como *fork-join*e *tarefas*.

Em (LORENZON et al., 2022), os autores apresentam uma implementação do kernel GEMM baseada em tarefas. Eles exploraram várias funcionalidades do OmpSs-2 para minimizar a sobrecarga em tempo de execução e melhorar o balanceamento de carga e a localidade de referência. Demonstrou-se que uma implementação otimizada baseada em tarefas oferece melhor desempenho e reduz o consumo de energia em comparação com a implementação GEMM otimizada do BLIS OpenMP *fork-join*, estratégias que nosso trabalho utiliza também, porém, explorando o modelo de tarefas na aplicação sísmica Fletcher.

(SCHUSSLER et al., 2023) compararam o desempenho do Método Fletcher em servidores locais e em nuvem, utilizando arquiteturas de hardware semelhantes. Os resultados mostraram que, apesar das limitações de latência e compartilhamento de recursos na nuvem, a execução em nuvem pode ser viável para simulações sísmicas de grande escala, especial-

mente quando são considerados os benefícios econômicos e de escalabilidade oferecidos pela nuvem. Este estudo complementa nossa investigação ao fornecer insights sobre a execução da Modelagem Fletcher em diferentes ambientes computacionais.

O estudo de (SERPA et al., 2020) abordou a melhoria do desempenho e da eficiência energética do Método Fletcher para simulação de extração de petróleo, conseguindo otimizar a aplicação para explorar o paralelismo reduzindo o tempo de execução e o consumo de energia. Os experimentos realizados em ambientes com processadores Intel Xeon e GPUs NVIDIA demonstraram que as versões otimizadas, especialmente utilizando CUDA, obtiveram os melhores resultados em termos de desempenho e eficiência energética. Este trabalho é relevante para nossa pesquisa, pois destaca a importância de otimizações matemáticas e de arquitetura na melhoria do desempenho de simulações sísmicas.

Em (CHASAPIS et al., 2015), é comparada a diferença entre fork-join e tarefas aplicando-o ao conjunto de benchmarks PARSEC. A comparação com as implementações de Pthreads/OpenMP do PARSEC com OmpSs-2 mostra que o modelo de tarefas pode obteve desempenho melhor no conjunto de benchmarks PARSEC. Este trabalho é relevante, pois demonstra o potencial do paralelismo em tarefas que iremos utilizar na Modelagem Fletcher.

# 4 EXPLORANDO PARALELISMO DE TAREFAS NA MODELAGEM FLETCHER

Para explorar o paralelismo na Modelagem Fletcher e obter melhor desempenho, é fundamental identificar e compreender quais trechos do código são responsáveis pelos maiores tempos de execução. Assim, otimizando essas partes, é possível obter maiores ganhos de desempenho proporcionais. Nesse sentido, no trabalho aqui apresentado, foi utilizada a ferramenta *gprof* (GRAHAM; KESSLER; MCKUSICK, 1982) para realizar as medições do tempo de execução para cada função que é chamada na aplicação. Antes de executar o programa, foi necessário configurar a variável de ambiente para `OMP_NUM_THREADS=1`. Após isso, configuramos o arquivo `Makefile` preenchendo o campo `backend` com OpenMP. Dessa forma, temos uma versão que executa com apenas uma única thread, de forma similar a uma execução sequencial.

Tabela 4.1 – Tempo de execução coletado com gprof

| **% tempo** | **segundos** | **chamadas** | **função** |
|---|---|---|---|
| 99.89 | 1465.91 | 100 | OPENMP_Propagate |
| 0.06 | 0.86 | 1 | Model |
| 0.05 | 0.70 | - | main |
| 0.02 | 0.28 | 1 | RandomVelocityBoundary |
| 0.00 | 0.03 | - | _init |
| 0.00 | 0.01 | 11 | DumpSliceFile |

A Tabela 4.1 demonstra que as chamadas a função `OPENMP_Propagate` consome 99.89% de todo tempo de execução da aplicação. Com esses resultados, fica evidente que a função de propagação de onda é crítica em relação ao tempo de execução da aplicação como um todo. Esta função é responsável pela propagação de ondas que se baseia na solução de equações diferenciais parciais discretas que fazem uso de *stencils*. Em geral as otimizações de *stencils* focam em aprimorar o paralelismo e explorar a localidade de referência (LI et al., 2021), sendo essa a abordagem explorada que o trabalho irá adotar.

A função de propagação de onda já conta uma solução que explora o paralelismo (SERPA et al., 2019b) *fork-join* exemplificado no pseudo-algoritmo 2, através da diretiva do OpenMP `#pragma omp for` no primeiro *loop* aninhado dentro da função de propagação de onda. O cálculo de cada ponto da propagação de onda é possível ser paralelizado, pois ele depende apenas dos pontos de iterações anteriores para seu resultado.

**Algorithm 2** – Loop em paralelo OpenMP

```
1: #pragma omp for
2: for iz ← 1 to Z do
3:     for iy ← 1 to Y do
4:         for ix ← 1 to X do
5:             S(ix,iy,iz)
```

Na implementação proposta por esse trabalho de conclusão, foi montado um novo *back-end* com base no OpenMP, porém, adotando modelo de programação de tarefas - com a API do OmpSs-2 e suas respectivas diretivas. No novo diretório `OMPSS2`, temos os mesmos arquivos da implementação OpenMP, com alteração apenas no arquivo `openmp_propagate.c`, que é responsável por grande parte do tempo de execução. Para realizar essa transição do OpenMP para OmpSs-2 foi necessário inicialmente trocar a diretiva `#pragma omp for` pela diretiva do OmpSs-2 `#pragma oss taskloop`, que é responsável por dividir cada iteração de um *loop* em uma tarefa. Além disso, foi necessário adicionar mais uma diretiva ao final da execução `#pragma oss taskwait` para sincronizar todas as tarefas, sem essa diretiva não há espera para o cálculo de toda matriz, não garantindo assim consistência nos resultados finais. Com isso, vamos ter transformado a implementação inicial em *fork-join* para uma implementação usando o modelo tarefas, como exemplificado no pseudo-código 3.

**Algorithm 3** – Loop em paralelo usando OmpSs-2

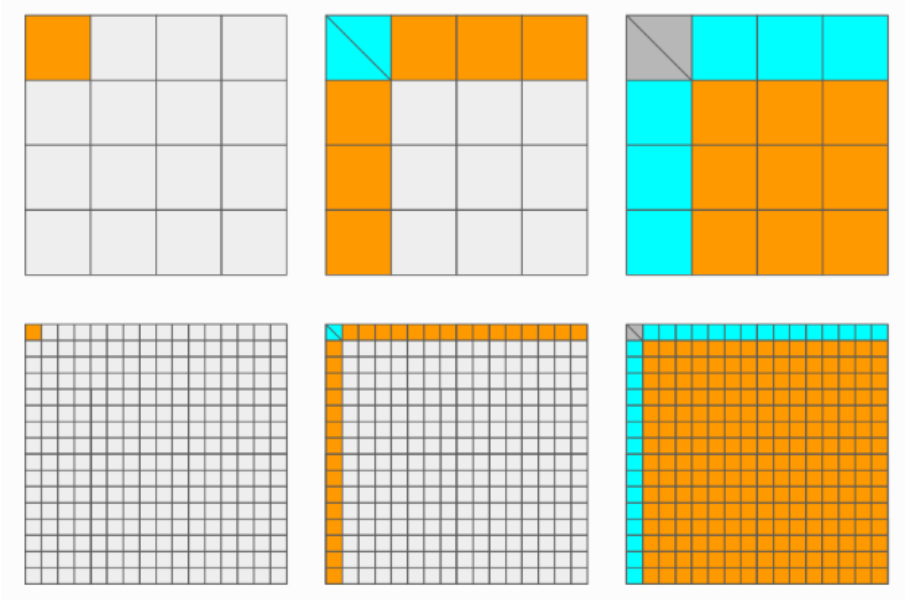
```
1: #pragma oss taskloop
2: for iz ← 1 to Z do
3:     for iy ← 1 to Y do
4:         for ix ← 1 to X do
5:             ResolveEquações()
6: #pragma oss taskwait
```

Em (SERPA et al., 2019b) técnicas de balanceamento de carga são utilizadas para distribuir melhor o trabalho realizado por cada *thread* no cálculo da propagação de onda. Essas técnicas podem ser portadas para o OmpSs-2, através das diretivas `grainsize` e `collapse`.

A granularidade de cada tarefa é determinada pela diretiva `grainsize` e refere-se ao tamanho do trabalho atribuído a cada tarefa individual. No caso do Fletcher, cada iteração é se refere ao cálculo dos novos valores das ondas em cada ponto da grade.

Uma granularidade fina pode levar a um grande número de tarefas pequenas, o que pode aumentar o *overhead* de gerenciamento de tarefas, mas melhorar a distribuição de carga entre as *threads*. Por outro lado, uma granularidade grossa pode reduzir a sobrecarga de gerenciamento, mas pode resultar em um desequilíbrio de carga, onde algumas threads ficam ociosas enquanto

Figura 4.1 – Exemplo granularidade



Fonte: (HPC2N, 2021)

outras ainda estão processando suas tarefas. A Figura 4.1 ilustra a divisão de tarefas em uma matriz 2D, sendo a granularidade grossa na primeira linha com uma carga de trabalho maior por tarefa e fina na segunda linha com uma carga menor por tarefa porém maior sobrecarga de gerenciamento.

Para exemplificar, considere no pseudo-código 4 a diretiva `taskloop grainsize (4)` determina que o *loop*, que possui 16 iterações, será dividido em tarefas, onde cada tarefa será responsável por processar 4 iterações do *loop*. A tabela 4.2 exemplifica a divisão de iterações para cada tarefa criada.

**Algorithm 4** – Exemplo de uso de `taskloop` com `grainsize`

```
#pragma taskloop grainsize (4)
for i = 0 to 15 do
    process(A[i])
```

Tabela 4.2 – Iterações por Tarefa

| **Tarefa** | **Iterações** |
|---|---|
| Tarefa 1 | 0, 1, 2, 3 |
| Tarefa 2 | 4, 5, 6, 7 |
| Tarefa 3 | 8, 9, 10, 11 |
| Tarefa 4 | 12, 13, 14, 15 |

Nesse sentido, há possibilidade que a iteração 12, por exemplo, leve um tempo muito maior que as outras, nesse caso, as iterações 13, 14 e 15 serão forçadas a esperar a iteração 12 acabar, para que assim a tarefa 4 acabe. A melhoria nesse caso, seria reduzir o *grainsize*,

deixando a granularidade de cada tarefa mais fina. Dessa forma, uma tarefa que demore mais tempo para computar não influencia no desempenho de outras. Contrariamente, se as iterações levarem tempos semelhantes, é vantajoso um *grainsize* mais grosso, para que não se tenha um *overhead* de gerenciamento de tarefas entre as *threads*.

O parâmetro `collapse` permite a combinação de *loops* aninhados em um único *loop*, aumentando as possibilidades de distribuição de tarefas. Essa técnica é útil em casos em que a paralelização de *loops* aninhados quando se deseja explorar mais níveis de paralelismo de forma mais granular. O pseudo-código 5 exemplifica essa técnica, nesse caso usando a diretiva `grainsize(4)` não seria possível adicionar 4 iterações de um laço de 2 iterações em uma única tarefa, todavia com `collapse(2)` ele indica que está colapsando os 2 *loops* aninhados formando apenas um loop de 16 iterações, resultando novamente na tabela 4.2.

**Algorithm 5** – Exemplo de uso de `taskloop` com `collapse`

```
#pragma taskloop grainsize (4) collapse(2)
for i = 0 to 2 do
    for j = 0 to 8 do
        S(i, j)
```

Nesse sentido, utilizando dessas estratégias de otimização, foi aplicado no trabalho de conclusão as diretivas `grainsize` e `collapse` em conjunto com o `taskloop` representando no pseudo-código 6, para descobrir a melhor configuração desse parâmetros no trabalho aqui apresentado, na perspectiva do modelo de tarefas, foram necessários efetuar diversas mudanças de parâmetros, que resultou em benchmarks diversos, apresentados e explorados na próxima seção.

**Algorithm 6** – Loop em paralelo usando OmpSs-2 otimizado

```
1: #pragma oss taskloop grainsize(20) collapse(2)
2: for iz ← 1 to Z do
3:     for iy ← 1 to Y do
4:         for ix ← 1 to X do
5:             ResolveEquações(ix,iy,iz)
6: #pragma oss taskwait
```

# 5 METODOLOGIA

Os experimentos foram realizados no ambiente do Parque Computacional de Alto Desempenho (PCAD), devido a disponibilidade de recursos usamos a partição *hype1* que conta com uma arquitetura Haswell com 2 nós computacionais, onde cada nó é composto por um processador Intel Xeon E5-2640 v2 de 10 núcleos. Cada núcleo suporta SMT (*Simultaneous multithreading*) de 2 vias e possui *caches* L1 e L2 privadas, enquanto a cache L3 é compartilhado entre todos os núcleos do processador, conforme a Tabela 5.1 demonstra.

Tabela 5.1 – Arquitetura Utilizada

| **Parâmetro** | **Valor** |
|---|---|
| Processador | 2 × Intel Xeon E5-2650 v3, 10 cores, 2 SMT-cores |
| Microarquitetura | Haswell |
| Caches/proc. | 10 × 32 KByte L1d, 10 × 32 KByte L1i, 10 × 256 KByte L2, 25 MByte L3 |
| Memória principal | 128 GByte DDR4-2133 |

Para compilar a aplicação foi utilizado o Clang versão 18.0.0 e os testes realizados utilizaram OmpSs-2 (MONITORING, 2023) e OpenMP na versão 5.1 (BOARD, 2018). Além disso, a execução é acompanhada do comando `taskset -c 0-19` que é responsável por associar as *threads* criadas aos primeiros 20 cores físicos do processador.

Durante as execuções foram realizadas coletas das seguintes métricas, usando a ferramenta *perf* e o seguinte em comando: `perf stat -a -e context-switches,cache-references,cache-misses`

- Tempo de execução: em programa ou tarefa é o tempo total decorrido desde o início até a conclusão da execução do programa. O tempo de execução é a métrica fundamental de desempenho, pois mede diretamente a eficiência e rapidez com que um programa pode ser executado. Sendo assim, reduzir essa métrica um dos principais objetivos da nossa implementação da Modelagem Fletcher;
- Trocas de contexto: ocorrem quando o escalonador interrompe a execução de um processo ou thread e começa a executar outro. Essas trocas com uma frequência excessiva pode levar à degradação do desempenho. Primeiramente, as trocas de contexto adicionam uma sobrecarga ao sistema, pois o estado de uma thread deve ser salvo e o estado de uma nova thread deve ser restaurado, consumindo tempo da CPU que poderia ser usado para computação efetiva. Além disso, elas podem perturbar a localidade de cache, já que o novo processo pode expulsar dados do processo anterior da cache;

- *Misses* de cache: acontecem quando o processador tenta ler ou escrever dados em um nível de cache, mas os dados não estão presentes naquele nível de cache, forçando uma busca em um nível de cache mais baixo ou na memória principal. Essa busca conforme o seu nível na hierarquia de memória custará mais.

Realizamos a coleta das métricas uma vez para cada execução, o que infelizmente impede de medir o erro das medidas. Além disso, durante todas as execuções foram mantidos alguns valores de entradas fixos que já estão presentes por padrão na Modelagem Fletcher (PANETTA PEDRO LOPES, 2019) como: a anisotropia utilizada na simulação (`fNameSec`) para isotropia transversal inclinada, a quantidade de pontos de absorção (`absorb`) para 16, a distância entre os pontos consecutivos (`nx, ny, nz`) na grade para 12.5 em todos eixos e o tamanho do passos no tempo (`dt`) para 0.001. Com o objetivo comparar como o tempo de execução da aplicação varia conforme o tempo máximo de da simulação (`tMax`) e o número de pontos da grade em cada eixo (`dx, dy, dz`).

Inicialmente, realizamos execuções da Modelagem Fletcher com a versão já implementada em OpenMP para obtermos os valores base de uma implementação em `fork-join`. Sendo assim, para entradas com tamanhos de grade 120x120x120, 248x248x248 e 378x378x378 foram coletados os tempos de execução variando o tempo da simulação máximo (`tMax`) de 0.05 até 0.80 em intervalos de 0.05. Ademais, foram executadas coletas com valor de entrada da grade de 248x248x248 e `tMax` de 0,45 com as otimizações variando os valores de `grainsize` e `collapse` na nova implementação em OmpSs-2 da Modelagem Fletcher a fim de achar os valores que resultam no melhor tempo de execução. Por fim, após a análise de qual `grainsize` e `collapse` resultam na implementação que tem melhor desempenho, repetiremos as mesmas execuções da versão OpenMP variando os 3 tamanhos da grade e o tempo de propagação máximo, porém com a implementação em OmpSs-2 otimizada para comparar as implementações usando *fork-join* e tarefas.

# 6 RESULTADOS

Nesse capítulo iremos demonstrar os resultados obtidos através da metodologia proposta em duas seções. Na primeira seção, serão analisados as métricas relativas às execuções com diferentes *grainsizes* e seus respectivos tempos de execução e na segunda seção, iremos comparar os resultados da Modelagem Fletcher usando o modelo de execução *fork-join* e o modelo de execução de tarefas.

## 6.1 Escolhendo a melhor granularidade para implementação em OmpSs-2

O balanceamento de carga é muito importante no desempenho do modelo de execução de tarefas porque garante a utilização eficiente dos recursos disponíveis de um sistema. A distribuição inadequada pode levar a uma subutilização de núcleos de processamento, com *threads* ociosas aguardando a finalização de outras tarefas mais demoradas.

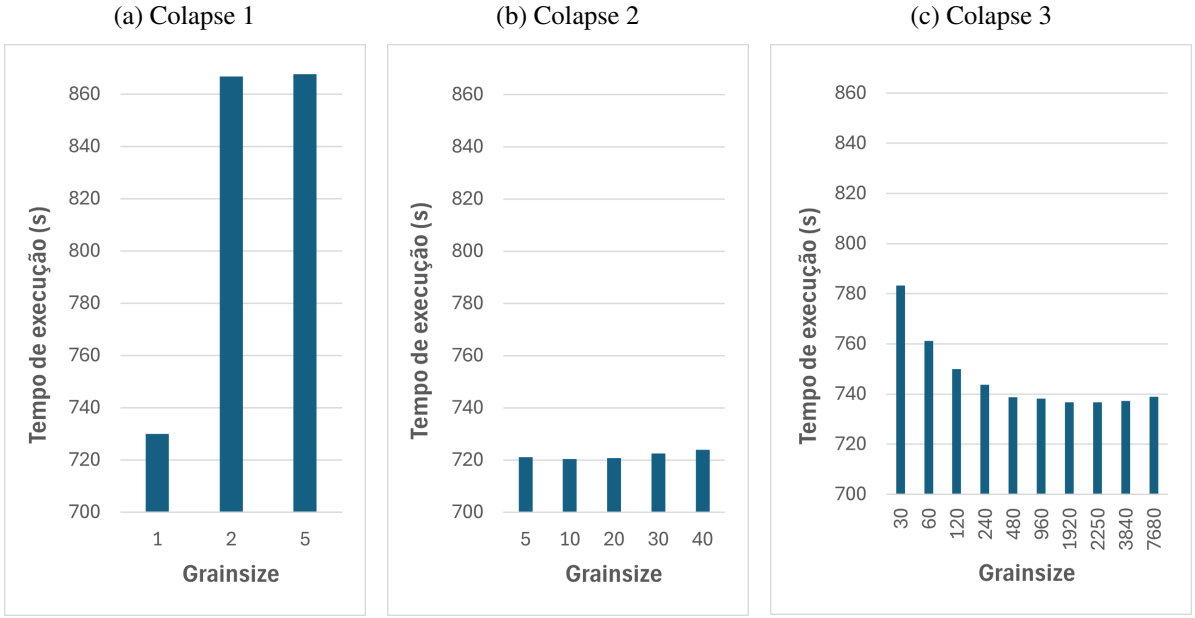
A diretiva `taskloop` é responsável por dividir a carga de trabalho em sub-tarefas, enquanto as diretivas que `grainsize` e `collapse` são responsáveis, respectivamente, por definir quantas sub-tarefas compõem uma tarefa e quão grande será a divisão da carga de trabalho por sub-tarefa. Nesse sentido, ao aumentarmos o valor de `collapse` teremos mais possibilidades de tamanhos de tarefas, pois a carga de trabalho será divida em mais sub-tarefas e ao escolhermos o valor de `grainsize` definiremos o tamanho da tarefa.

Nessa seção, comparamos os tempos de execução de coletas com valor de entrada da grade de 248x248x248 e `tMax` de 0,45, variando as diretivas do OmpSs-2 *grainsize* e *collapse* para determinar a granularidade que resulta no melhor desempenho e, portanto, a versão com melhor balanceamento de carga do OmpSs-2 para Modelagem Fletcher.

A figura 6.1, representa os tempos de execução para os valores de 1 a 3 da diretiva `collapse`. Em cada uma de suas subfiguras temos no eixo X o valor da diretiva `grainsize` utilizada na execução, e no eixo Y o resultado em tempo de execução.

Os resultados da figura 6.1(a) representam as execuções com a diretiva `collapse(2)`, uma granularidade mais grossa, pois apenas o *loop* mais externo é paralelizado. Isso gera menos tarefas, cada uma contendo mais trabalho, resultando em uma granularidade mais grossa. Na figura, a redução da diretiva `grainsize` resulta em menores tempos de execução, o melhor valor é de 729,96 segundos com a diretiva `grainsize(1)`, sendo essa a melhor granularidade para a diretiva `collapse(1)`. Esse resultado indica que possivelmente uma granularidade mais fina pode ser benéfica para o balanceamento, visto que não foi possível reduzir

Figura 6.1 – Tempos de execução para diferentes valores de `collapse` e `grainsize`




Fonte: O autor

ainda mais.

Na figura 6.1(b), que representa as execuções com a diretiva `collapse(2)`, uma granularidade intermediária, pois paraleliza os 2 primeiros *loops* aninhados, obtemos que o `grainsize` de tamanho 10 tem o maior desempenho com tempo de execução de 720,44 segundos, enquanto que aumentando ou diminuindo a granularidade pela diretiva `grainsize` apresenta piora no balanceamento de carga. O resultado confirma que com uma granularidade mais fina na aplicação melhora o desempenho por aumentar a distribuição de carga entre as *threads*.

E por último, na figura 6.1(c), com uma granularidade ainda mais fina devido a diretiva `collapse(3)` paralelizar todos os `loops` aninhados gerando mais tarefas, o melhor desempenho se observa com um `grainsize` de tamanho 2250 resultando em um tempo de execução de 736,64 segundos. Isso mostra que uma embora uma granularidade fina possa aumentar a distribuição de carga, se essa granularidade for fina demais o *overhead* de gerenciamento de tarefas supera os ganhos.

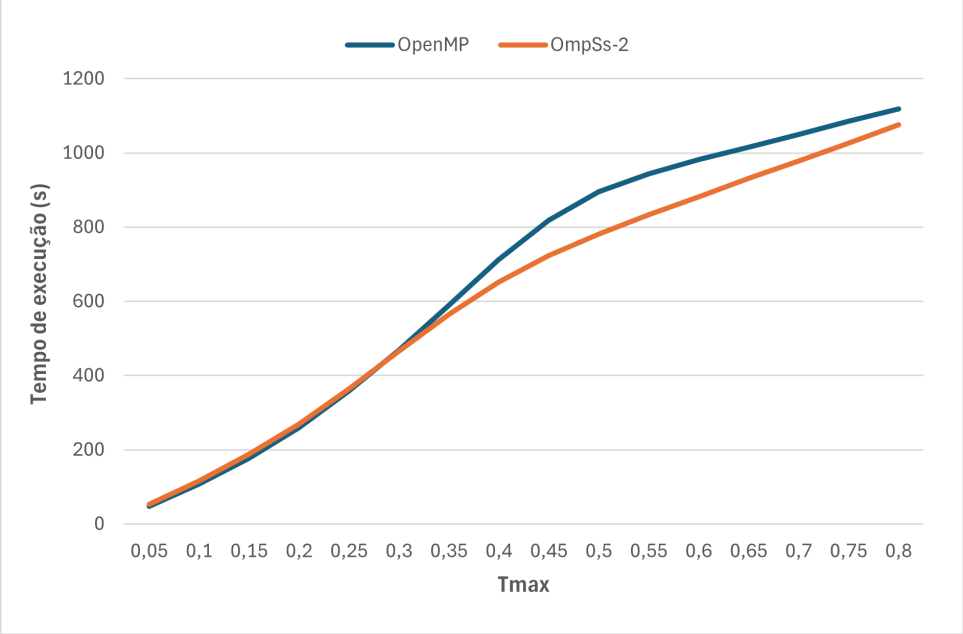
Assim, com base no resultados de todas execuções é possível concluir que maior desempenho acontece com um *grainsize* de tamanho 10 e `collapse(2)`, então, na próxima seção, será utilizado a implementação com essas diretivas e valores para comparar com as métricas da versão *fork-join* (OpenMP) que coletamos inicialmente.

## 6.2 Comparando fork-join e tarefas

Inicialmente, foram feitas medições de tempo de execução variando tempo máximo de propagação (*tMax*), para versão *fork-join* (OpenMP) da Modelagem Fletcher. O resultado é um gráfico que mostra o tempo de execução no Eixo Y e o tempo máximo de propagação no Eixo X. As curvas demonstram como o tempo de execução da aplicação varia conforme o aumento do tempo de máximo da simulação entre as versões usando *fork-join* e tarefas.

Após coletar os resultados da Modelagem Fletcher na versão OpenMP aplicando paralelismo *fork-join* e em posse da implementação da Modelagem Fletcher com melhor *grainsize*, é possível comparar os resultados da versão OpenMP com a nova implementação em OmpSs-2 que possui o melhor *grainsize*.

Figura 6.2 – Tempo de Execução na grade de 248x248x248

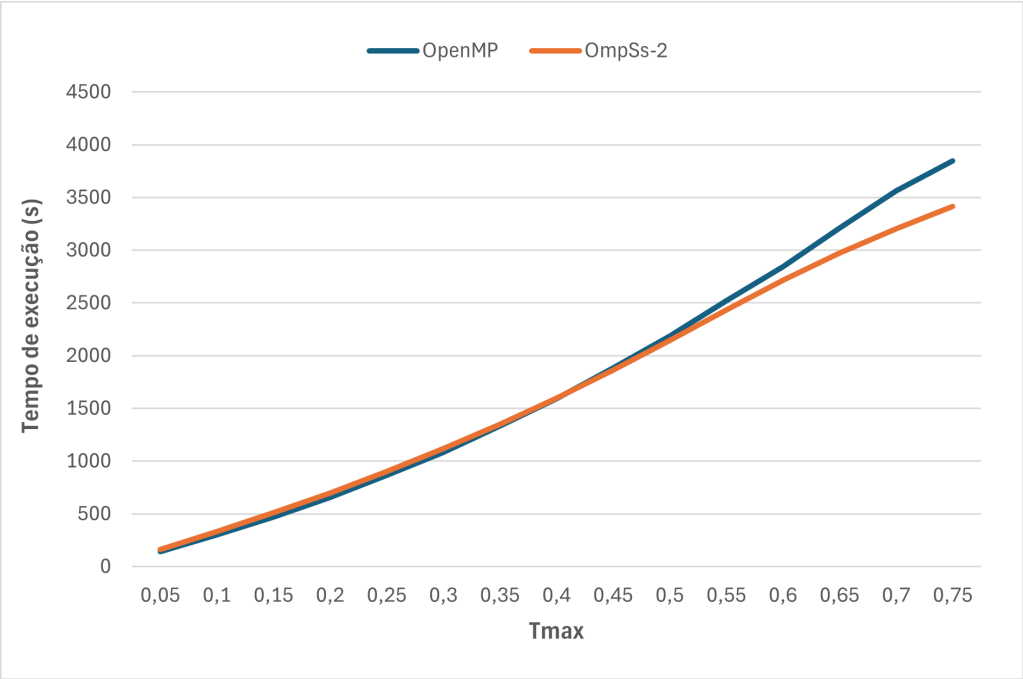



Fonte: O autor

Na figura 6.2, que possui como entrada uma grade tridimensional de 248 x 248 x 248, é possível observar uma redução no tempo de execução na versão do OmpSs-2 em relação ao OpenMP, no intervalo de `tMax` de 0.30 até um `tMax` de 0.8. A redução no tempo de execução ocorre de forma gradual até o seu pico `tMax` em 0.45, apresentando redução de até 14,55% em relação ao OpenMP. Após isso, há um decaimento da redução até o final do intervalo. Nesse sentido, é possível concluir que temos um melhor desempenho na versão do OmpSs-2 até o valor de `tMax` 0.8.

Todavia, aumentando o valor de entrada da grade para 378 pontos em cada dimensão,

Figura 6.3 – Tempo de Execução na grade de 378x378x378




Fonte: O autor

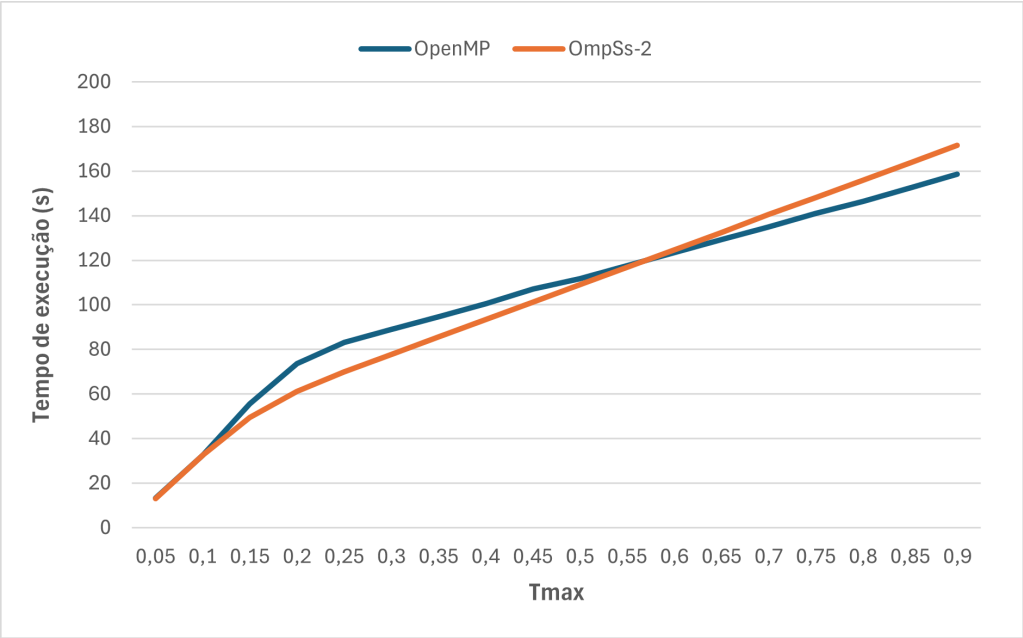
como demonstra a figura 6.3, temos um gráfico que demonstra um comportamento semelhante ao início do comportamento da figura 6.2, com uma redução gradual no tempo de execução do OmpSs-2 em relação ao OpenMP. Além disso, reduzindo o valor da grade para 120 pontos em cada dimensão, demonstrado na figura 6.4, se observa o mesmo comportamento do gráfico; porém por se tratar de menos pontos, é possível ver uma ilustração mais completa da variação do tempo de execução das duas versões em relação à `tMax`.

Nesse sentido, é possível observar que, durante um intervalo de `tMax` e conforme o tamanho, a versão com OmpSs-2 possui menor tempo de execução e, após esse intervalo, a versão com OpenMP apresenta melhores tempos de execução. A explicação por trás desse comportamento do gráfico nas 3 versões se mostrou aparente após a análise de todas as métricas obtidas com o comando *perf* sendo os principais indicadores: as trocas de contexto e os *misses* de cache.

Observando a Figura 6.5 que representa a porcentagem de *cache-misses* nas execuções com grade de dimensões 120, 248 e 378, respectivamente, é possível observar que, em todos os gráficos, a versão OmpSs-2 se encontra com uma porcentagem menor de *cache-misses*. Essa redução acontece devido ao escalonador de tarefas do OmpSs-2 explorar a localidade memória do sistema, otimizando o reuso da cache.

Esse resultado, aumenta o desempenho da aplicação, uma vez que um *cache-miss* im-

Figura 6.4 – Tempo de Execução na grade de 120x120x120




Fonte: O autor

plica em um atraso significativo na recuperação dos dados necessários para o processamento. Quando ocorre um *cache-miss*, o processador precisa buscar os dados na memória principal, implicando na maior latência no acesso aos dados.

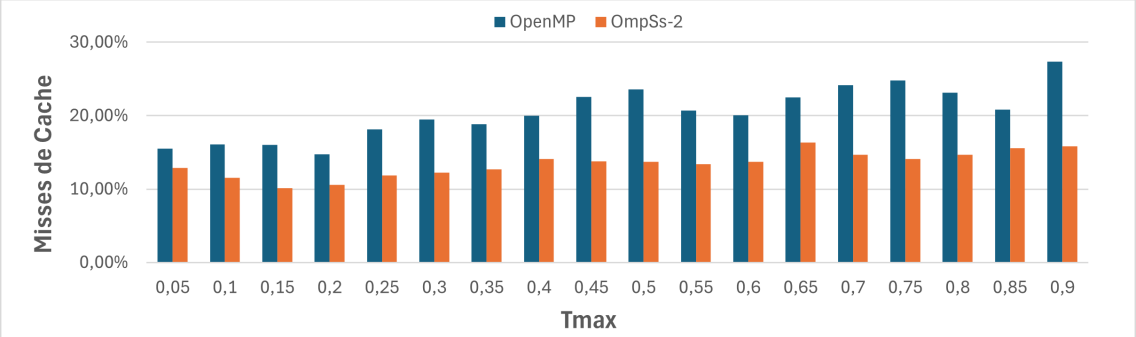
Todavia, analisando as medidas relacionadas as trocas de contexto na Figura 6.6 que representa as execuções com grade de dimensões 120, 248 e 378, respectivamente, é possível observar que as trocas de contexto da versão OmpSs-2 aumentam de forma constante em relação ao OpenMP. As trocas de contexto adicionam sobrecarga ao sistema e são responsáveis por redução do desempenho da aplicação, embora pequenas quantidades podem degradar em pouco o desempenho geral, quando os valores se começam a crescer há uma piora significativa.

Sendo assim, considerando os ganhos de desempenho devido a redução de *cache-misses* na versão do OmpSs-2 é possível explicar a redução do tempo de execução inicial que observamos nas figuras 6.4, 6.2 e 6.3, contudo conforme aumenta o número de trocas de contexto na versão OmpSs-2 em relação ao OpenMP, a degradação de desempenho devido as trocas de contexto novamente aproxima os tempos de execução das duas versões até que a versão com OpenMP supera a versão OmpSs-2 para maiores valores de `tMax`.

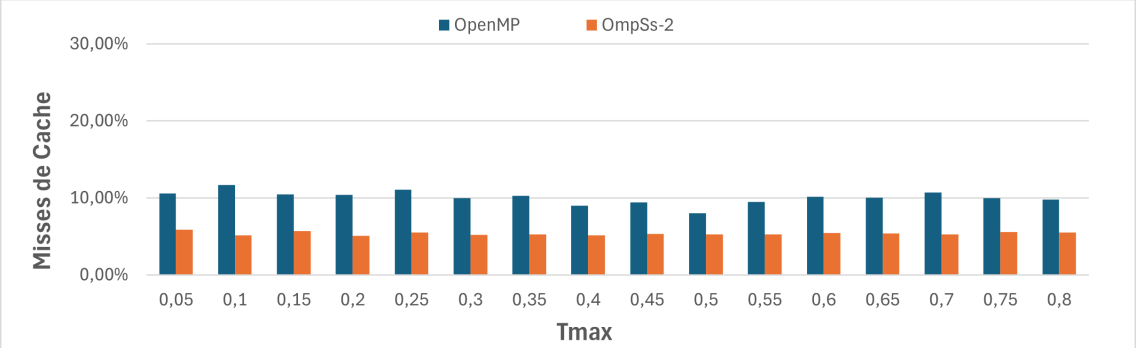
Dessa forma, os resultados demonstram que é possível ter um ganho de desempenho em relação ao *fork-join*, utilizando o modelo de tarefas. Todavia, os benefícios são limitados a um intervalo em que a redução de *cache-misses* tem maior impacto ao desempenho do que as trocas de contexto.

Figura 6.5 – *Misses* de Cache % em grades de tamanho:

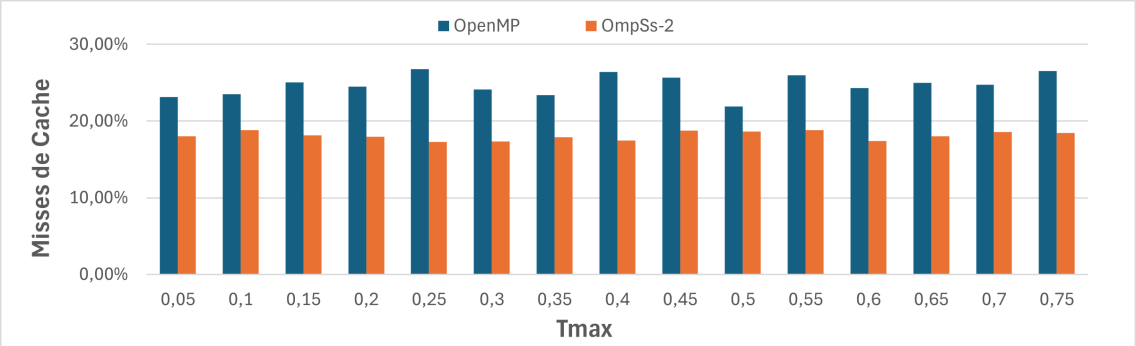
(a) 120x120x120




(b) 248x248x248



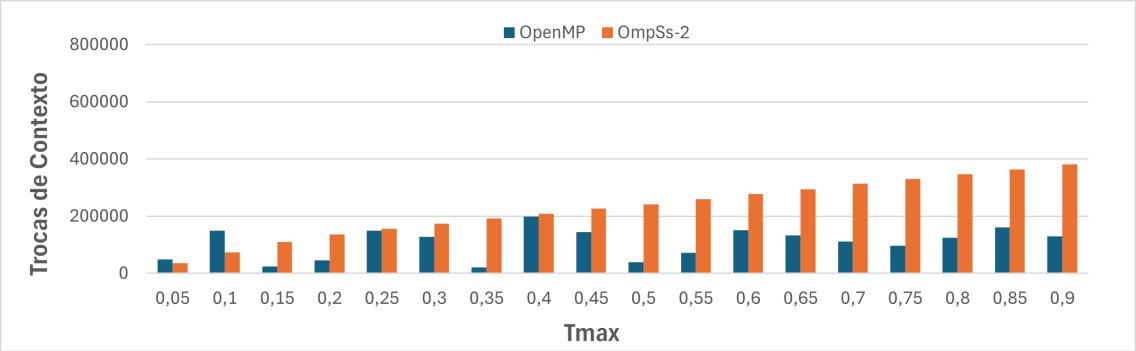

(c) 378x378x378



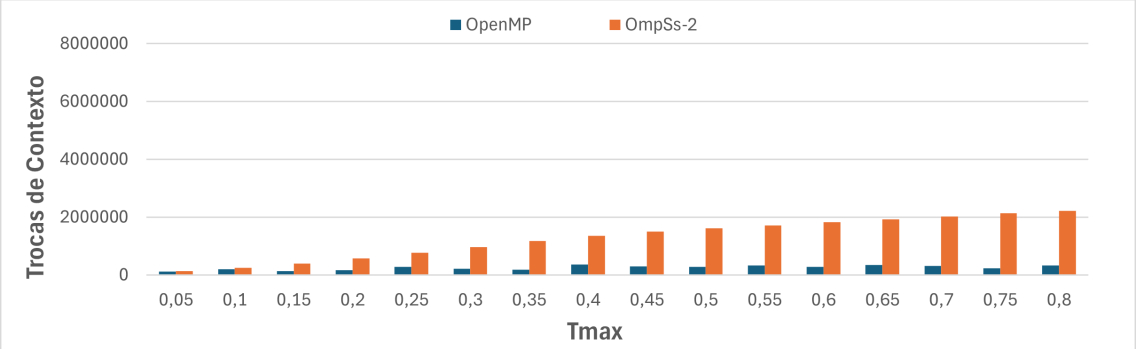

Fonte: O autor

Figura 6.6 – Trocas de contexto em grades de tamanho:

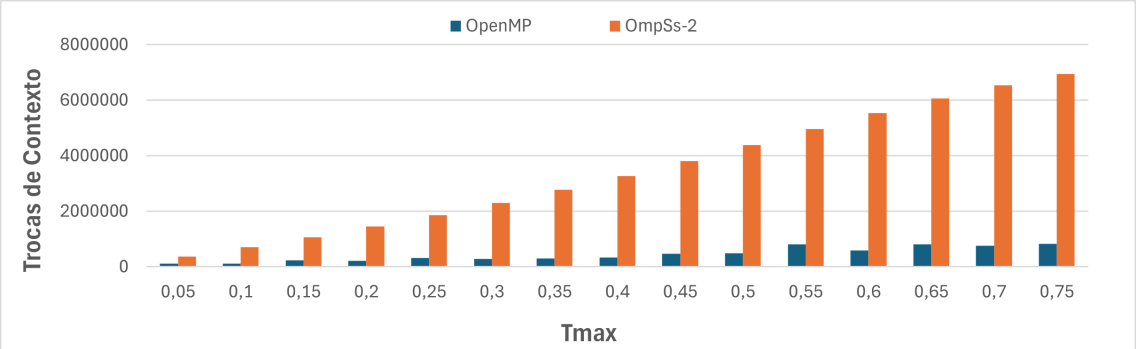
(a) 120x120x120




(b) 248x248x248




(c) 378x378x378




Fonte: O autor

# 7 CONCLUSÃO

Neste trabalho, foi investigada a viabilidade e eficácia da utilização do paralelismo em tarefas, por meio da implementação da Modelagem Fletcher com o OmpSs-2, em comparação com a abordagem já implementada baseada em *fork-join* que utiliza OpenMP. A análise dos resultados permitiu observar que, ao otimizar as diretivas de `grainsize` e `collapse`, a versão em OmpSs-2 apresentou um desempenho superior em termos de tempo de execução, dependendo da duração total da simulação e do tamanho da grade como valor de entrada.

Os experimentos realizados evidenciaram uma redução no tempo de execução da implementação em OmpSs-2 em relação ao OpenMP, com ganhos que chegam a 14,55% em determinadas condições de simulação. Além disso, foi possível demonstrar que a utilização do modelo de execução em tarefas trouxe maior flexibilidade e desempenho no gerenciamento dos recursos computacionais, minimizando o impacto de problemas como a contenção de cache e otimizando o balanceamento de carga.

E, embora, a implementação em tarefas não utilizasse todos recursos da API OmpSs-2, foi possível observar desempenhos semelhantes e superiores, demonstrando a eficácia do modelo de execução em tarefas. Além disso, esse trabalho demonstra a lição importante de realizar uma análise profunda da aplicação, pois para se obter as melhores otimizações é necessário se ter uma compreensão profunda dos gargalos de desempenho da aplicação.

Dessa forma, conclui-se que o OmpSs-2 se apresenta como uma alternativa promissora para a implementação de aplicações que demandam alto desempenho em ambientes paralelos, oferecendo uma solução mais adaptável e eficiente para a execução de tarefas complexas, como a Modelagem Fletcher utilizada na pesquisa geofísica. A continuidade deste trabalho poderia explorar a mais da versão em OpenMP usando *fork-join* para assim ter uma comparação de duas versões plenamente otimizadas tanto no modelo de execução de tarefas como no modelo de execução *fork-join*. Além disso, seria possível utilizar o OmpSs-2 em outras aplicações, em especial de aplicações que possam se beneficiar do uso de tarefas com dependências, pois o OmpSs-2 oferece recursos para otimizar essas situações evitando a necessidade de sincronizações.

# REFERÊNCIAS

AYGUADÉ, E. et al. A proposal for task parallelism in openmp. In: SPRINGER. **International Workshop on OpenMP**. [S.l.], 2007. p. 1–12.

BOARD, O. A. R. Openmp application programming interface. 2018. Accessed: 2024-05-25.

BSC, P. M. G. **Specification of OmpSs-2**. 2024. <https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf>. Accessed: 2024-05-25.

BUTKO, A. et al. Efficient programming for multicore processor heterogeneity: Openmp versus ompss. In: **Open Source Supercomputing Workshop. Springer's Lecture Notes in Computer Science (LNCS). Frankfurt, Germany**. [S.l.: s.n.], 2017. v. 22.

CAMERON, S. L. C. **Cornell Center for Advanced Computing**. 2022. <https://cvw.cac.cornell.edu/OpenMP/>. Accessed: 2024-06-20.

CHANDRA, R. **Parallel programming in OpenMP**. [S.l.]: Morgan kaufmann, 2001.

CHASAPIS, D. et al. Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 12, n. 4, p. 1–22, 2015.

CLEMENTS, A. T. et al. The scalable commutativity rule: designing scalable software for multicore processors. **Communications of the ACM**, ACM New York, NY, USA, v. 60, n. 8, p. 83–90, 2017.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. **IEEE computational science and engineering**, IEEE, v. 5, n. 1, p. 46–55, 1998.

DENNING, P. J. The locality principle. **Communications of the ACM**, ACM New York, NY, USA, v. 48, n. 7, p. 19–24, 2005.

DURAN, A. et al. Extending the openmp tasking model to allow dependent tasks. In: SPRINGER. **OpenMP in a New Era of Parallelism: 4th International Workshop, IWOMP 2008 West Lafayette, IN, USA, May 12-14, 2008 Proceedings 4**. [S.l.], 2008. p. 111–122.

FLETCHER, R. P.; DU, X.; FOWLER, P. J. Reverse time migration in tilted transversely isotropic (tti) media. **Geophysics**, Society of Exploration Geophysicists, v. 74, n. 6, p. WCA179–WCA187, 2009.

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. Gprof: A call graph execution profiler. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 17, n. 6, p. 120–126, 1982.

HPC2N. **Task-based Parallelism: Motivation**. 2021. Accessed: 2024-06-29. Available from Internet: <https://hpc2n.github.io/Task-based-parallelism/branch/spring2021/motivation/>.

LI, M. et al. Automatic code generation and optimization of large-scale stencil computation on many-core processors. In: **Proceedings of the 50th International Conference on Parallel Processing**. [S.l.: s.n.], 2021. p. 1–12.

LORENZON, A. F. et al. Seamless optimization of the gemm kernel for task-based programming models. In: **Proceedings of the 36th ACM International Conference on Supercomputing**. [S.l.: s.n.], 2022. p. 1–11.

LUKAWSKI, M. Z. et al. Cost analysis of oil, gas, and geothermal well drilling. **Journal of Petroleum Science and Engineering**, Elsevier, v. 118, p. 1–14, 2014.

MARTÍNEZ, V. et al. Performance improvement of stencil computations for multi-core architectures based on machine learning. **Procedia Computer Science**, Elsevier, v. 108, p. 305–314, 2017.

MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured parallel programming: patterns for efficient computation**. [S.l.]: Elsevier, 2012.

MONITORING, B. P. **LLVM Release 2023.11**. 2023. <https://github.com/bsc-pm/llvm/releases/tag/github-release-2023.11>. Accessed: 2024-07-06.

PANETTA, J. Sobre a modelagem fletcher. **Available at GitHUb in https://github.com/msserpa/fletcher-petrobras/blob/master/doc/DocumentaV0.pdf**, 2018.

PANETTA PEDRO LOPES, R. L. M. J. Descrição a modelagem fletcher 3.0. **Available at GitHUb in https://github.com/msserpa/fletcher-3-0-petrobras/blob/master/README.pdf**, 2019.

PATTERSON, D. A.; HENNESSY, J. L. **Computer organization and design ARM edition: the hardware software interface**. [S.l.]: Morgan kaufmann, 2016.

PHILIPPE, O. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. Elsevier Science BV, 2016.

RAUBER, T.; RÜNGER, G. **Parallel programming**. [S.l.]: Springer, 2013.

SCHUSSLER, B. S. et al. Otimização de configurações de threads por bloco para kernels gpu na exploração geofísica. In: SBC. **Anais da XXIV Escola Regional de Alto Desempenho da Região Sul**. [S.l.], 2024. p. 57–60.

SCHUSSLER, B. S. et al. Comparando o desempenho entre computaçao em nuvem e servidor local na execuçao do método fletcher. In: SBC. **Anais da XXIII Escola Regional de Alto Desempenho da Região Sul**. [S.l.], 2023. p. 33–36.

SERPA, M. et al. Melhorando o desempenho e a eficiência energética do método fletcher para simulaçao de extraçao de petróleo. In: SBC. **Escola Regional de Alto Desempenho da Região Sul (ERAD-RS)**. [S.l.], 2020. p. 141–142.

SERPA, M. et al. Portabilidade e eficiência do método fletcher de aplicações sísmicas em arquiteturas multicore e gpu. In: SBC. **Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho**. [S.l.], 2019. p. 169–180.

SERPA, M. S. et al. Optimization strategies for geophysics models on manycore systems. **The International Journal of High Performance Computing Applications**, SAGE Publications Sage UK: London, England, v. 33, n. 3, p. 473–486, 2019.

SEVERANCE, C.; DOWD, K. **High performance computing**. [S.l.]: OpenStax CNX, 2010.

SHOSHANY, B. A c++ 17 thread pool for high-performance scientific computing. **SoftwareX**, Elsevier, v. 26, p. 101687, 2024.

STALLINGS, W. Computer organization and architecture, edition, eighth. Pearson, 2009.

TOY, W. N.; ZEE, B. **Computer Hardware-Software Architecture**. [S.l.]: Prentice Hall Professional Technical Reference, 1986.

YOUNT, C.; DURAN, A.; TOBIN, J. Multi-level spatial and temporal tiling for efficient hpc stencil computation on many-core processors with large shared caches. **Future Generation Computer Systems**, Elsevier, v. 92, p. 903–919, 2019.