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

LUCAS BARROS DE ASSIS

Modelando uma aplicação de ondas sísmicas através do paralelismo de tarefas

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

Orientador: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
2021

RESUMO

A aplicação *Ondes3D* tem como objetivo realizar a simulação da propagação de ondas sísmicas utilizando o método de diferenças finitas. Apesar de contar com uma implementação paralela utilizando *OpenMP*, esse paralelismo só é obtido dentro de cada uma das quatro grande etapas do algoritmo utilizado. O trabalho aqui apresentado estuda as alterações necessárias para dividir as estruturas utilizadas pelo simulador em blocos, possibilitando posteriormente uma implementação em forma de tarefas utilizando a biblioteca *StarPU*, possibilitando um nível de paralelismo superior ao atual. Depois de feitas as considerações necessárias para a conversão do modelo atual para o modelo baseado em tarefas, uma implementação é proposta e parcialmente elaborada, de maneira a explorar as dificuldades envolvidas no processo e analisar, mesmo que parcialmente, os resultados obtidos. Finalmente, a partir desse projeto posto em prática, algumas sugestões de aperfeiçoamento são apresentadas, buscando aproveitar a experiência obtida para construir uma versão capaz de explorar ainda mais as diferentes arquiteturas hoje existentes.

Palavras-chave: Programação paralela. programação baseada em tarefas. Ondes3D. StarPU.

Seismic waves modelling through task-based programming

ABSTRACT

The *Ondes3D* simulator aims to simulate the propagation of seismic waves through the finite differences method. Even though it has a parallel implementation using *OpenMP*, this parallelism is only achieved inside each of its four major steps. The study presented in this document studies the modifications needed to split the structures used by *Ondes3D* in tiles, which allows a task-based implementation using the *StarPU* library in order to achieve a higher level of parallelism. From the observations made by studying the needed conversion from the current model to a task-based one, an implementation is proposed and partially executed to explore the main obstacles involved in the process and, even if partially, analyse the results obtained. Finally, taking in account the developed project, some improvements are suggested in order to use the development experience acquired to construct a version capable of exploiting even more the current available architectures.

Keywords: Parallel programming. Task-based programming. Ondes3D. StarPU.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CPU	Central Process Unit
DAG	Directed Acyclic Graph
GPU	Graphics Processing Unit
HPC	High-Performance Computing
MPI	Message Passing Interface

SUMÁRIO

1 INTRODUÇÃO

Uma ferramenta importante para a mitigação dos riscos decorrentes de terremotos é a simulação da propagação de ondas sísmicas (DUPROS et al., 2010). Esse tipo de aplicação simula fenômenos físicos com base na aproximação de derivadas por diferenças finitas, iterando sobre uma discretização do espaço analisado em três dimensões, o que resulta em tempos de execução elevados. Por essa razão, diversos trabalhos (BOITO et al., 2017; DUPROS; DO; AOCHI, 2013; MARTíNEZ et al., 2015), apresentados no capítulo 2, buscam alternativas para acelerar a sua execução, identificando as operações mais custosas e, frequentemente, explorando possibilidades de processamento paralelo e distribuído.

A aplicação *Ondes3D* realiza a simulação descrita acima através do método de diferenças finitas e utilizando bibliotecas paralelas para obter tempos de execução menores. O paralelismo local, aquele cujas linhas de execução compartilham a mesma memória, é obtido utilizando a biblioteca *OpenMP*, repartindo o cálculo das células que discretizam o domínio do cálculo entre diferentes *threads* executadas paralelamente. Além disso, o protocolo *MPI* pode ser usado para repartir o espaço estudado em diferentes regiões, sendo cada uma delas fornecida a uma unidade de processamento com seu próprio espaço de endereçamento.

Apesar de empregar técnicas que permitem que vários cálculos simultâneos sejam realizados, o paralelismo alcançado pelo *Ondes3D* existe somente dentro de cada uma de suas quatro grande etapas, sem a possibilidade de entrelaçar a computação das estruturas envolvidas em diferentes etapas. Dessa maneira, as unidades de processamento que concluírem seus cálculos antes das demais precisam aguardar ociosamente até que todas tenham finalizado a etapa atual.

O paradigma de programação paralela baseada em tarefas, que consiste em construir pequenos blocos de código que atuam diversas vezes sobre diferentes conjuntos de dados, permite a obtenção de uma granularidade mais fina, isto é, de uma quantidade maior de tarefas a serem distribuídas entre as unidades de processamento. Através dessa técnica, espera-se reduzir a possibilidade de haver unidades ociosas, resultando em um ganho de velocidade na execução das aplicações que empregam-na.

A biblioteca *StarPU* é uma alternativa atual que implementa a programação baseada em tarefas. Em sua implementação, uma tarefa consiste em uma função cuja especificação inclui, além dos dados utilizados, seus modos de acesso: somente leitura, somente

escrita ou leitura e escrita. Ao longo do algoritmo, essas tarefas são submetidas à um escalonador gerenciado pela biblioteca, que utiliza a ordem de submissão e os modos de acesso descritos por cada uma delas para deduzir as dependências existentes. Ao longo da execução, a bibliteca distribui as tarefas conforme as suas dependências e os recursos computacionais disponíveis.

Este trabalho tem como objetivo estudar e implementar as alterações necessárias para que o algoritmo da aplicação *Ondes3D* seja executado na forma de tarefas gerenciadas pela biblioteca *StarPU*. Com isso, espera-se obter uma granularidade mais fina assim como um paralelismo entre etapas graças ao gerenciamento de dependências realizado pela biblioteca. Ao final da implementação, a ferramenta *StarVZ* será utilizada para avaliar o grau de paralelismo obtido.

1.1 Contribuições

O estudo aqui realizado explora algumas das dificuldades envolvidas na implementação do método de diferenças finitas aplicado à simulação de propagação de ondas sísmicas. No contexto matemático, busca-se por alternativas para absorver as ondas quando estas atingem os limites artificiais criados pela discretização do problema. Em relação à computação, o objetivo é elaborar um algoritmo paralelo capaz de aproveitar ao máximo as unidades de processamento disponíveis.

A partir do estudo realizado, este trabalho fornece também uma tentativa de implementação do simulador *Ondes3D* utilizando o paradigma de programação baseada em tarefas implementado pela biblioteca *StarPU*. A versão implementada atualmente não é capaz de produzir os mesmos resultados numéricos que a versão original. No entanto, a estrutura geral, assim como a definição das tarefas que computam as etapas envolvidas no algoritmo e os dados por elas utilizados foram desenvolvidas e servem de base para uma versão correta eventualmente mais eficiente que a original.

Finalmente, aproveitando as experiências obtidas no desenvolvimento da versão paralela e no estudo de trabalhos similares, algumas propostas de melhorias são sugeridas para trabalhos futuros.

1.2 Estrutura do texto

O restante do trabalho é dividido da seguinte maneira: o capítulo 2 apresenta estudos realizados buscando acelerar a execução da aplicação estudada e similares; o capítulo 3 introduz os conhecimentos necessários para o entendimento da aplicação e de como deseja-se otimizá-la; o capítulo 4 apresenta a estrutura geral planejada e a implementação desenvolvida; o capítulo 5 descreve os experimentos realizados e o capítulo 6 finaliza com as conclusões sobre esse estudo.

2 TRABALHOS RELACIONADOS

A aplicação *Ondes3D* já foi objeto de estudo de outros trabalhos, frequentemente visando um tempo de execução reduzido mas passando também pela busca de um consumo energético inferior. Além disso, outras aplicações também empregaram a programação baseada em tarefas para obter uma aceleração nos cálculos realizados. Alguns desses trabalhos são discutidos abaixo e suas diferenças em relação ao trabalho apresentado são explicitadas.

Boito et al. (BOITO et al., 2017) verificaram que mais de 72% do tempo de execução do *Ondes3D* é utilizado em operações de leitura e escrita. Três otimizações diferentes foram propostas: incialmente, uma camada de *software* foi criada para centralizar as solicitações de entrada e saída da aplicação. Em um segundo momento, os passos de comunicação entre processos foram substituídos por operações de escrita efetuadas diretamente no sistema de arquivos. A terceira implementação passa a utilizar diversos arquivos de saída, o que permite que múltiplas operações de escrita aconteçam paralelamente. Apesar dos resultados impressionantes desse trabalho, o estudo aqui proposto desconsidera as operações de entrada e saída para focar-se na paralelização dos cálculos das equações elastodinâmicas. No entanto, é possível que a busca pelo paralelismo também influencie no tempo utilizado em entradas e saídas ao encontrar aquelas que podem ser executadas de maneira assíncrona.

Tesser et al. (DUPROS; DO; AOCHI, 2013), partindo de uma implementação distribuída em que o domínio é dividido em partes menores, perceberam que o tempo de cálculo das equações depende da região sendo tratada e dos valores da simulação, o que causa um desbalanceamento de trabalho entre os processadores utilizados. Ao utilizar uma implementação de *MPI* construída sobre um suporte de execução capaz de realizar um balanceamento dinâmico de tarefas, foi possível reduzir o tempo de execução em 23%, graças a capacidade da biblioteca utilizada de migrar processos entre diferentes processadores.

Castro et al. (CASTRO et al., 2016) utilizaram o processador de baixa potência (*MPPA-256*) para propor uma implementação da aplicação *Ondes3D* com um baixo consumo energético. A partir de uma estratégia de ladrilhamento multi-nível, onde os blocos são reparticionados para tornarem-se bidimensionais, a versão desenvolvida para esse processador apresentou uma diminuição de 86% do consumo quando comparada à uma execução em um processador convencional. No entanto, o consumo energético não

é um fator considerado no estudo aqui apresentado.

Nesi et al. (NESI et al., 2020) realizaram três implementações diferentes de uma aplicação de dinâmica de fluídos utilizando programação baseada em tarefas através da biblioteca *StarPU*. Essa aplicação utiliza o método de diferenças finitas, assim como o *Ondes3D*, para simular o fluxo de um fluído Newtoniano incompressível com viscosidade constante. Além de desenvolver tarefas voltadas para *GPU*s, o traço de execução resultante foi avaliado e refinado utilizando o conjunto de ferramentas *StarVZ* (PINTO et al., 2016; GARCIA PINTO et al., 2018; NESI et al., 2019). Através do emprego da técnica de células fantasma no gerenciamento dos vizinhos, apresentada e explicada na Seção 4.1, foi possível alcançar uma aceleração de 77x em relação à versão original da aplicação estudada.

Martínez et al. (MARTíNEZ et al., 2015) propuseram uma implementação do *Ondes3D* utilizando a biblioteca *StarPU*, baseada em tarefas e considerando arquiteturas heterogêneas compostas por *CPU*s e *GPU*s. De um certo modo essa proposta vai além da pesquisa aqui apresentada, onde não existe uma implementação voltada para *GPU*. No entanto, Martínez et al. optaram por descartar as interações com as bordas do domínio devido a complexidade introduzida pelo uso delas, uma simplificação não realizada no trabalho aqui apresentado e que não foi considerada nas comparações de velocidade. O melhor desempenho foi obtido utilizando quatro *CPU*s e oito *GPU*s, apresentando uma aceleração de 25x em relação à versão original utilizando doze *CPU*s.

A Tabela 2.1 resume as diferenças entre o trabalho desenvolvido e descrito neste documento e os estudos citados acima. Os projetos desenvolvidos por Nesi et al. e Martínez et al. são os que mais se assemelham ao aqui desenvolvido, apesar do primeiro tratar de uma aplicação diferente e o segundo ignorar uma parcela da simulação.

Tabela 2.1 – Trabalhos relacionados

Trabalho	*Aplicação*	*Contexto*
(BOITO et al., 2017)	*Ondes3D*	Otimização de operações de entrada e saída
(DUPROS; DO; AOCHI, 2013)	*Ondes3D*	Balanceamento dinâmico de carga
(MARTíNEZ et al., 2015)	*Ondes3D*	Implementação baseada em tarefas em arquiteturas heterogêneas sem condições de borda
(CASTRO et al., 2016)	*Ondes3D*	Otimização de consumo energético
(NESI et al., 2020)	Dinâmica de fluídos	Implementação baseada em tarefas em arquiteturas heterogêneas
Este trabalho	*Ondes3D*	Implementação baseada em tarefas

Fonte: Autor

3 FUNDAMENTAÇÃO TEÓRICA

Para a elaboração desse estudo, primeiramente é necessário compreender o que é a aplicação utilizada. A partir desse conhecimento, é possível tratar do modelo de programação baseada em tarefas e como ela é implementada na biblioteca *StarPU*. Finalmente, é necessário explicar o método de divisão das estruturas de dados usadas, conhecido como ladrilhamento ou *tiling*, que permite que as tarefas atuem sobre dados independentes, possibilitando o paralelismo entre elas.

3.1 Ondes3D

A aplicação discretiza o mundo real em diversos tensores tridimensionais, endereçados a partir de suas coordenadas, que representam diferentes grandezas físicas envolvidas na simulação. Esses tensores são armazenados como atributos de diferentes estruturas de dados, que podem variar de acordo com os dados experimentais utilizados na simulação. No caso desse trabalho, os dados utilizados são resultado do terremoto de *Chuetsu-Oki*, que atingiu a costa japonesa em 2007. A simulação desse evento é calculada utilizando as seguintes estruturas de dados:

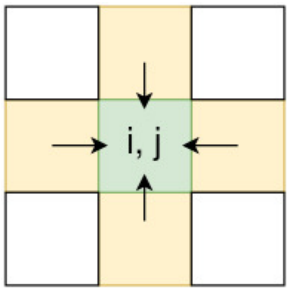
- *STRESS*: composta por tensores que representam as tensões sofridas em diferentes planos. O tensor xy, por exemplo, armazena a tensão sofrida no plano X - aquele cuja normal aponta na direção do eixo X, isto é, o plano YZ - pelas forças causadas na direção Y.
- *VELOCITY*: composta por tensores que representam a decomposição da velocidade da onda nos eixos X, Y e Z.
- *SOURCE*: composta pelos dados que descrevem a(s) fonte(s) da onda, como seu hipocentro, a longevidade da fonte, sua orientação e sua força.
- *MEDIUM*: composta pelos dados que descrevem o meio por onde a onda se propaga, como sua profundidade.
- *ABSORBING_BOUNDARY_CONDITION*: composta por vetores que representam as contribuições das bordas do domínio nos cálculos de tensão e velocidade. As contribuições em cada uma dessas grandezas são compostas por nove componentes, representando as influências de cada eixo nos três eixos possíveis (no caso da velocidade) e nos três planos possíveis (no caso da tensão). Essas bordas são utili-

zadas para limitar o domínio computacional minimizando a interferência de limites artificiais nos cálculos realizados. (NATAF, 2013).

Uma peculiaridade a ser observada nessa aplicação é que não existe um grande tensor com diversos atributos descrevendo as grandezas físicas que ali atuam. Cada grandeza é armazenada em uma estrutura a parte que possui múltiplos tensores representando as diferentes orientações possíveis. Essa característica trará implicações no gerenciamento de blocos vizinhos após a operação de ladrilhamento, explicadas em mais detalhes na Seção 4.2.2.

Os cálculos são realizados ao longo de um número pré-determinado de iterações que representam a passagem do tempo de propagação da onda. Em cada um desses passos de tempo, todas as células que compõem o espaço discretizado do problema são atualizadas uma a uma. Esse padrão de algoritmo, chamado de *stencil*, calcula os valores armazenados em cada célula a partir de valores de células vizinhas, sendo o conceito de vizinhança variável. A Figura 3.1 ilustra um *stencil* 2D com quatro vizinhos em que o valor da célula (i, j) envolve os dados das células $(i, j-1)$, $(i-1, j)$, $(i+1, j)$ e $(i, j+1)$. Apesar da aplicação estudada usar coordenadas nos eixos X, Y e Z, o mesmo conceito pode ser expandido para o espaço tridimensional.

Figura 3.1 – Representação de um *stencil* 2D com quatro vizinhos



Fonte: Autor

As estruturas descritas anteriormente são atualizadas iterativamente passando por quatro grandes etapas, também chamadas de *macro-kernels*. São elas:

- *computeSeisMoment*: etapa em que as fontes da onda são atualizadas.
- *computeIntermediates*: etapa em que as contribuições das bordas do domínio são atualizadas a partir dos tensores de velocidade.
- *computeStress*: etapa em que os tensores de tensão nos diferentes planos são atualizados a partir dos tensores de velocidade e das condições de borda.

- *computeVelocity*: etapa em que os tensores de velocidade são atualizados a partir dos tensores de velocidade e das condições de borda.

Com exceção da atualização das fontes da onda, que acontece em um domínio reduzido, os *macro-kernels* descritos acima utilizam a diretiva *omp parallel for* para obter um paralelismo interno. Já que essas etapas realizam cálculos dentro de um laço triplo, iterando sobre as coordenadas nos eixos X, Y e Z, a biblioteca *OpenMP*, através da diretiva citada anteriormente, cria *threads* capazes de tratar diferentes coordenadas simultaneamente.

Em relação ao laço principal de execução, a Figura 3.2 resume o fluxo da aplicação. O domínio do problema é dividido em cinco regiões denominadas *imodes*, sendo quatro delas as linhas e colunas limítrofes do domínio e a quinta os elementos internos. Com exceção da etapa *computeSeisMoment*, cada *macro-kernel* é executado cinco vezes, uma em cada região. Internamente, cada etapa percorre o espaço $[x_{min}, x_{max}][y_{min}, y_{max}][z_{min}, z_{max}]$.

Figura 3.2 – Laço principal da aplicação original

```
for (iterations = 0; iterations < max_it; iterations++) {

  computeSeisMoment();

  for (imode = 0; imode < 5; imode++) {
    calculate_limits(imode, &x_min, &x_max, &y_min, &y_max, &z_min, &z_max);
    computeIntermediates(x_min, x_max, y_min, y_max, z_min, z_max);
  }
  for (imode = 0; imode < 5; imode++) {
    calculate_limits(imode, &x_min, &x_max, &y_min, &y_max, &z_min, &z_max);
    computeStress(x_min, x_max, y_min, y_max, z_min, z_max);
  }
  for (imode = 0; imode < 5; imode++) {
    calculate_limits(imode, &x_min, &x_max, &y_min, &y_max, &z_min, &z_max);
    computeVelocity(x_min, x_max, y_min, y_max, z_min, z_max);
  }
}
```

Fonte: Autor

Teoricamente, a partir do momento que a tensão, por exemplo, de uma célula é calculada, já é possível iniciar o cálculo de sua velocidade. No entanto, como a diretiva descrita acima é utilizada dentro da função responsável pelo cálculo de tensão de todas as células do domínio, é necessário que todas elas sejam calculadas para que o algoritmo possa iniciar a fase de cálculo de velocidades. Dessa maneira, apesar da solução atual em *OpenMP* alcançar uma aceleração graças ao paralelismo introduzido, a quantidade de operações que podem ser executadas simultaneamente é menor do que seria teoricamente possível.

3.1.1 Condições de borda

Ao simular a propagação de uma onda, é necessário considerar as reflexões criadas por elas. No entanto, o método de diferenças finitas, utilizado pelo *Ondes3D*, exige que as suas equações sejam resolvidas em um domínio discretizado restrito. Dessa maneira, ao modelar a propagação de ondas, é necessário utilizar técnicas que permitam absorver as ondas nos limites do domínio estudado de maneira que suas reflexões não interfiram nos cálculos internos.

Berenger (BERENGER, 1994) desenvolveu uma técnica voltada para absorção de ondas eletromagnéticas chamada *Perfectly Matched Layers* (*PML*). Diferentemente das outras técnicas existentes até então, o emprego de *PML*s permite a construção de uma camada de absorção que teoricamente não produz reflexão alguma independentemente da frequência e do ângulo de incidência das ondas recebidas. Experimentos realizados na época mostraram que, apesar de uma pequena quantidade de reflexões ter sido detectada, sua magnitude poderia ser reduzida adequando alguns parâmetros. Além disso, o emprego da técnica de *PML*s permite a utilização de camadas de absorção menos espessas que as alternativas da época, o que reduz a computação necessária ao simular a propagação de ondas.

Apesar da eficiência do novo técnico desenvolvido, o coeficiente de reflexão das *PML*s só é nulo antes da discretização do domínio e com ondas de incidência não rasante, isto é, com um angulo de incidência diferente de $90°$, o que torna essa técnica inadequada quando, por exemplo, existem fontes de ondas muito próximas dos limites do domínio. Roden e Gedney (RODEN; GEDNEY, 2000) propuseram um novo método chamado *Convolutional Perfect Matched Layers*, que utiliza uma convolução recursiva para atenuar a reflexão de ondas de incidência rasante.

A aplicação *Ondes3D* implementa ambas as técnicas acima descritas em diferentes experimentos. No caso do experimento de *Chuetsu-Oki*, utilizado por esse trabalho, a técnica empregada é a das *CPML*s.

3.2 Paralelismo baseado em tarefas

O paralelismo baseado em tarefas é um modelo em que se busca descrever trechos de código paralelizáveis na forma de tarefas, que são criadas em tempo de execução pela biblioteca de paralelismo utilizada. Depois de criar a tarefa, a biblioteca é responsável

por executá-la ou adicioná-la a uma fila de tarefas para que ela seja executada em um momento apropriado. A possibilidade de adiar a execução de uma tarefa é o que torna essa opção interessante ao considerarmos as dependências da aplicação aqui estudada.

3.2.1 StarPU

StarPU é uma biblioteca de programação em tarefas para arquiteturas heterogêneas. Ela foi criada visando atender à necessidade da comunidade de *HPC* de poder computacional (AUGONNET et al., 2009), criando uma ferramenta que facilita a criação de tarefas para diferentes arquiteturas e a especificação dos dados utilizados. No caso do estudo aqui descrito, essa biblioteca é especialmente interessante ao determinar o momento de execução de suas tarefas, já que a escolha desse momento é baseada nas dependências de dados existentes entre elas.

A utilização de tarefas é composta por dois instantes, sendo o primeiro deles a sua descrição e o segundo a sua inserção, ou submissão, na fila de tarefas. O paralelismo obtido é o resultado da inserção de diversas instâncias da mesma tarefa atuando sobre dados diferentes e tendo sua ordem de execução determinada pelas dependências entre elas.

A criação de uma tarefa começa pela construção um *kernel*, uma função que implementa a tarefa a ser executada e que deve seguir a seguinte interface:

```
void task_name(void *buffers[], void *cl_arg)
```

Nessa interface, os *buffers* representam os dados que serão gerenciados pela biblioteca levando em conta suas dependências e modos de leitura. Já a estrutura *cl_arg* guarda ponteiros de valores externos utilizados pelo *kernel* e são tipicamente valores constantes. O suporte à arquiteturas heterogêneas fornecido pela biblioteca vem da possibilidade de criação de diversos *kernels* para uma mesma tarefa, sendo cada um deles específico para uma unidade de cálculo distinta.

Com o *kernel* criado, utiliza-se uma estrutura chamada *codelet* para descrever a tarefa. Nessa estrutura é indicada a quantidade de *buffers* utilizados pela tarefa, seus modos de acesso e o nome dos *kernels* que implementam-na. Voltando ao exemplo da função *computeIntermediates*, um *codelet* possível é mostrado na Figura 3.3.

A biblioteca segue o modelo *STF* (*sequential task flow*), em que uma única *thread* é responsável pelo fluxo principal da aplicação, incluindo a submissão das tarefas, o que

Figura 3.3 – Definição do *codelet* da função *computeIntermediates*

```
enum starpu_data_access_mode modes_intermediates[31] =
{
 STARPU_RW,
 STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R,
 STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R,
 STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R,
 STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R,
 STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_R
};

struct starpu_codelet intermediates_cl = {
                                          .cpu_funcs = {compute_intermediates_task},
                                          .nbuffers = 31,
                                          .name = "intermediates",
                                          .dyn_modes = modes_intermediates
};
```

Fonte: Autor

permite que um fluxo paralelo seja facilmente descrito através de um algoritmo sequencial. As tarefas são submetidas sequencialmente e, a partir de sua ordem de submissão e suas dependências, cria-se um grafo acíclico direcionado que descreve a hierarquia existente entre elas. A partir desse grafo, é possível escalonar as tarefas em tempo de execução de maneira que todas as dependências sejam respeitadas e o tempo ocioso nas unidades de cálculo seja o menor possível.

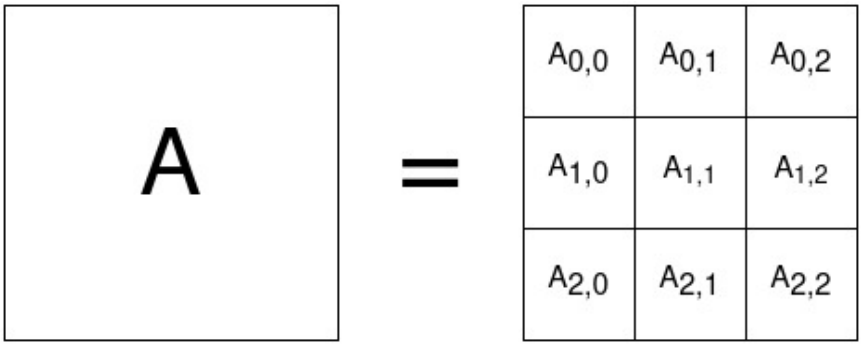
A submissão de tarefas é realizada através da função *starpu_insert_task*, tendo como parâmetros o *codelet* correspondente, os *buffers* utilizados e os ponteiros registrados como *cl_args*.

3.2.2 Matrizes *ladrilhadas*

Uma estratégia recorrente na programação de alto desempenho é o ladrilhamento de matrizes - no caso desse estudo, de tensores. Essa técnica consiste, conforme ilustra a Figura 3.4, na divisão de uma matriz em sub-matrizes (tipicamente chamadas de blocos) de menor tamanho, indexadas pela sua posição em relação à matriz original.

O ladrilhamento de matrizes exige que os algoritmos sejam re-adequados à sua estrutura, mas em contrapartida oferece numerosas vantagens. Em primeiro lugar, pode-se citar a otimização dos acessos em memória: ao utilizar um tamanho de bloco suficientemente pequeno, é possível que, durante a execução do cálculo sobre cada um dos blocos, todos os dados necessários estejam presentes simultaneamente na memória cache. Além disso, utilizando o DAG criado por ferramentas como a biblioteca *StarPU*, é possível executar simultaneamente tarefas de etapas diferentes, acelerando ainda mais a execução da

Figura 3.4 – Decomposição da matriz A em blocos

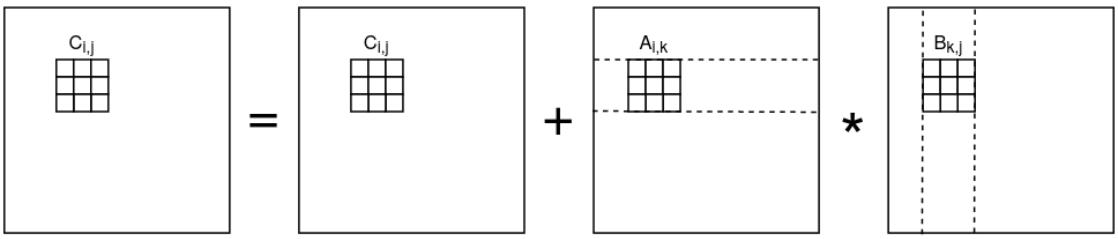


Fonte: Autor

aplicação.

A Figura 3.5 ilustra o cálculo de multiplicação de matrizes utilizando matrizes ladrilhadas. A versão convencional do algoritmo consiste em acumular a soma dos produtos de cada linha por cada coluna das matrizes envolvidas. Apesar dos elementos consecutivos de uma mesma linha encontrarem-se tipicamente lado a lado em memória, os saltos realizados nas trocas de linha podem exigir uma reescrita completa da cache.

Figura 3.5 – Representação do algoritmo de multiplicação de matrizes por blocos



Fonte: Autor

A versão ladrilhada é composta por diversas execuções do algoritmo convencional, atuando sobre uma linha (respectivamente, coluna) de blocos de cada vez. Dessa forma, além de otimizar os acessos de memória ao realizar pequenas multiplicações bloco a bloco, é possível executar o cálculo de n blocos simultaneamente, sendo n o número de blocos que compõem a matriz resultante. Apesar dos blocos das matrizes A e B serem acessados por diversas tarefas, nenhuma operação de escrita é efetuada sobre eles, o que possibilita que um número qualquer de tarefas acessem-nos simultaneamente sem gerar incoerências. Os blocos da matriz C, onde as escritas são realizadas, são acessados cada um deles por uma única tarefa, possibilitando o paralelismo descrito anteriormente.

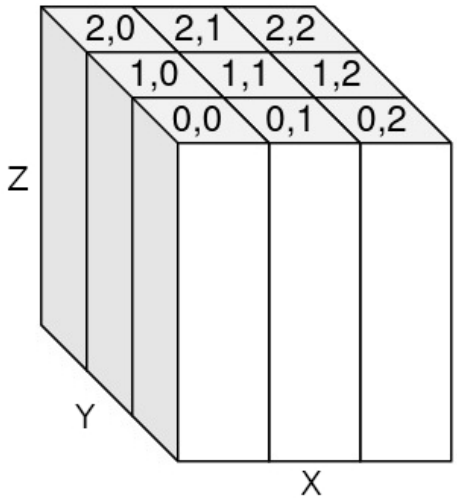
4 PROJETO E IMPLEMENTAÇÃO

Neste capítulo serão apresentadas as alterações necessárias para que a aplicação *Ondes3D* seja paralelizável na forma de tarefas gerenciadas pela biblioteca *StarPU*. Posteriormente uma descrição da implementação realizada ao longo dessa pesquisa será detalhada.

4.1 Proposta

Os diversos tensores que representam as grandezas utilizadas nas equações elastodinâmicas do problema deverão ser divididos, de forma parametrizável, em blocos de tamanho igual, que servirão de entrada para as tarefas implementadas. A Figura 4.1 ilustra o particionamento realizado pela implementação original, que parte de um arquivo de topologia para discretizar o plano XY em blocos de profundidade igual à do domínio. Apesar de manter o formato de discretização apenas no plano XY, o projeto aqui proposto tem como entradas as demais dimensões dos blocos, facilitando a execução de experimentos para adequar o particionamento à arquitetura onde a aplicação é executada.

Figura 4.1 – Representação do espaço particionado em nove blocos



Fonte: Autor

Na versão aqui proposta, cada etapa será executada tantas vezes quanto houverem blocos. Para isso, será necessário iterar sobre os blocos existentes, criando uma tarefa para atualizar cada bloco de dados, conforme mostra a Figura 4.2, dispensando a utilização dos *imodes*. Neste momento vale ressaltar que a inserção das tarefas consiste simplesmente

na criação de uma instância de tarefa com as dependências indicadas e não implica em sua execução imediata. Fica a cargo da biblioteca *StarPU* iniciar a execução de cada uma das tarefas disponíveis de maneira assíncrona quando as suas dependências estiverem satisfeitas. Internamente, cada tarefa percorrerá apenas o espaço determinado pelo bloco a ser atualizado em vez de uma região completa como na versão original.

Outro ponto a ser ressaltado é o fato de cada tarefa utilizar diversos blocos, já que as grandezas físicas são implementadas em diferentes estruturas. Por exemplo, o cálculo das tensões efetuado na função *computeStress* necessita não apenas do bloco de índice (i, j) de cada um dos nove componentes de tensão (em modo de escrita), mas também os blocos de índice (i, j) dos três componentes de velocidade, em modo de leitura.

Figura 4.2 – Laço principal da proposta

```
for (iterations = 0; iterations < max_it; iterations++) {

  starpu_insert_task(computeSeisMoment);

  for (i_block = 0; i_block < n_blocks_y; i_block++) {
    for (j_block = 0; j_block < n_blocks_x; j_block++) {
      starpu_insert_task(computeIntermediates, blocks[i][j]);
    }
  }

  for (i_block = 0; i_block < n_blocks_y; i_block++) {
    for (j_block = 0; j_block < n_blocks_x; j_block++) {
      starpu_insert_task(computeStress, blocks[i][j]);
    }
  }

  for (i_block = 0; i_block < n_blocks_y; i_block++) {
    for (j_block = 0; j_block < n_blocks_x; j_block++) {
      starpu_insert_task(computeVelocity, blocks[i][j]);
    }
  }
}
```

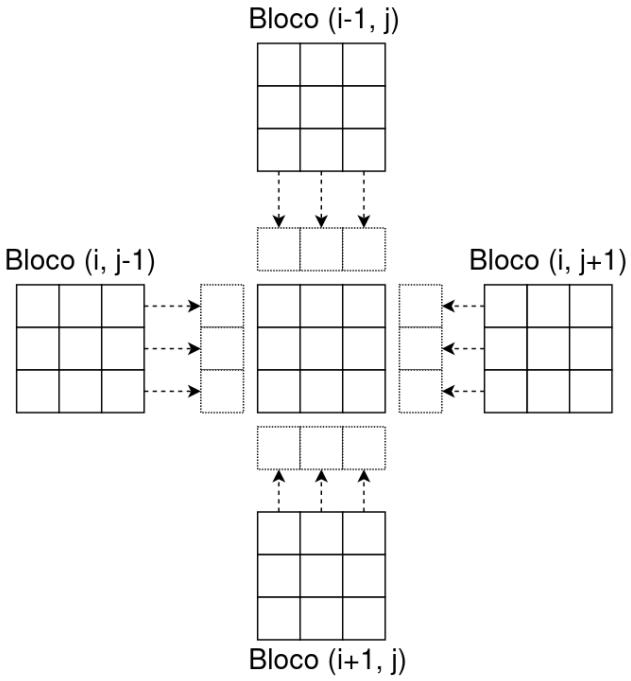
Fonte: Autor

A utilização de blocos independentes introduz, no entanto, uma dificuldade nas aplicações do tipo *stencil*: os elementos nas bordas de cada um dos blocos terão como vizinhos elementos de outros blocos. Uma primeira solução, mais simples e menos eficiente, consiste em não utilizar somente um bloco por tarefa, mas sim cinco - o bloco a ser calculado, em modo de leitura e escrita, e os seus quatro vizinhos imediatos, em modo de leitura.

Uma solução mais eficiente é a utilização das chamadas *ghost cells*. Já que é necessário ler apenas as células de uma borda de cada um dos vizinhos, em vez de acessar os quatro blocos vizinhos completos, compartilha-se apenas as bordas que serão necessárias. A Figura 4.3 mostra um exemplo de células fantasma em um *stencil* 2D, mas o conceito

pode ser expandido para três dimensões.

Figura 4.3 – Representação do uso de células fantasma



Fonte: Autor

4.2 Implementação

Com o objetivo de explorar as possibilidades da implementação em forma de tarefas e de analisar o nível de paralelismo alcançável utilizando esse paradigma, uma versão da aplicação *Ondes3D* foi desenvolvida utilizando a biblioteca *StarPU*. As subseções seguintes detalham o processo de implementação realizado.

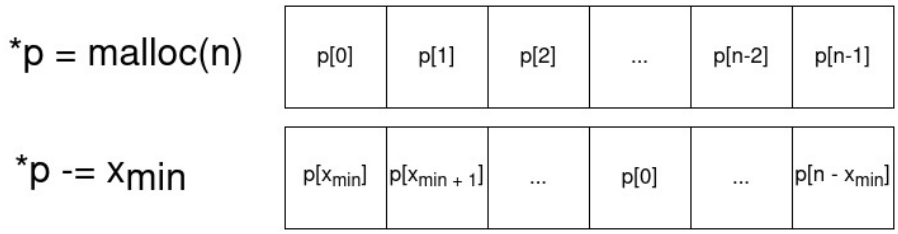
4.2.1 Inicialização

A inicialização dos dados na nova versão desenvolvida precisa considerar a necessidade de particioná-los em blocos, o que traz a tona a primeira modificação a ser realizada. A biblioteca *StarPU* fornece funções que realizam essa operação de ladrilhamento a partir do número de blocos desejado, o que garante um particionamento eficaz e funcional. No entanto, os tensores utilizados pela versão original são implementados como vetores multidimensionais, o que facilita o acesso aos dados utilizados, mas as

funções de particionamento da biblioteca *StarPU* exigem vetores unidimensionais como entrada. Portanto, o primeiro passo para adaptar o algoritmo desenvolvido de maneira a melhor aproveitar as funcionalidades oferecidas pela biblioteca foi a transformação de seus vetores multidimensionais em vetores unidimensionais de tamanho equivalente.

Ainda tratando-se de vetores, a aplicação original realizava um acesso a partir de suas coordenadas, mesmo se elas possuíssem valores negativos. A Figura 4.4 mostra a técnica utilizada para alcançar esse efeito. Os ponteiros que endereçavam essas estruturas apontavam não para a primeira posição alocada, como normalmente é feito, mas para o n-ésimo elemento, onde n é a coordenada de valor mais negativo. Devido à complexidade em manter essa implementação utilizando vetores unidimensionais, optou-se por empregar um endereçamento convencional.

Figura 4.4 – Representação de um vetor que permite índices negativos



Fonte: Autor

Devido às mudanças de dimensionalidade e limites dos vetores, o acesso à essas variáveis se torna mais complexo. Para contornar esse problema, foram utilizadas três *macros*, mostradas na Figura 4.5, que calculam o índice buscado em função das dimensões do vetor dado. Logo, todos os acessos à vetores foram substituídos por chamadas às novas *macros*.

4.2.1.1 Inicialização das CPML

A estrutura das condições de absorção das bordas não representa o domínio completo do problema, já que elas tratam unicamente das bordas, onde a propagação da onda deve ser absorvida sem produzir reflexões. Como elas são responsáveis por registrar contribuições tanto nos tensores de tensão quanto nos de velocidade, ambos construídos com nove componentes, armazená-las em tensores de tamanho igual ao do domínio exigiria um espaço de memória enorme e esparso, já que elas são calculadas somente em regiões muito limitadas do domínio.

A fim de não utilizar memória de maneira desnecessária, as contribuições físicas

Figura 4.5 – *Macros* utilizadas para acessar vetores.

```
#define i3_access(p, nrl, nrh, ncl, nch, ndl, ndh, i, j, k)
                  p[(i-(nrl))*((nch)-(ncl)+1)*((ndh)-(ndl)+1)+
                   (j-(ncl))*((ndh)-(ndl)+1)+
                   (k-(ndl))]

#define i2_access(m, nrl, nrh, ncl, nch, i, j)
                  m[(i-(nrl))*((nch)-(ncl)+1)+
                    (j-(ncl))]

#define i_access(p, nl, nh, i)
                 p[(i) - (nl) + NR_END]
```

Fonte: Autor

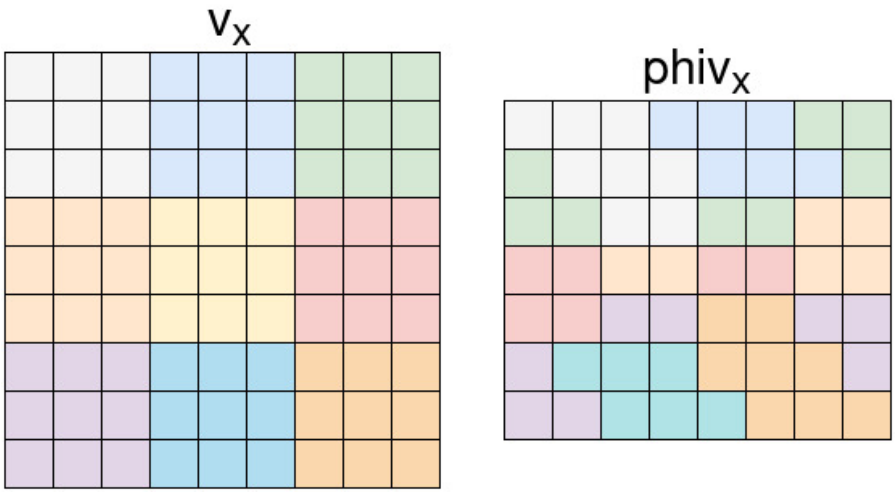
das *CPML*s são registradas em vetores unidimensionais de tamanho igual ao número de elementos nas mesmas. Um tensor chamado *ipml* armazena, para cada coordenada do domínio, o índice onde se encontram suas contribuições nos vetores das condições de absorção das bordas. No caso de coordenadas internas, esse índice tem valor -1, indicando que a célula em questão não precisa considerar contribuição alguma.

Ao dividir o problema em tarefas, no entanto, essas estruturas adicionam novas complexidades. Primeiramente, não é possível aplicar o particionamento em blocos da mesma maneira nesses vetores, já que eles possuem dimensões reduzidas, iguais ao número de elementos nas bordas. Além disso, faz-se necessário encontrar uma maneira de, mesmo após particioná-las, associar corretamente as coordenadas das células aos seus índices no vetor $ipml$.

Outra dificuldade encontrada no particionamento dessas estruturas é mostrada na Figura 4.6, que ilustra uma matriz de velocidades v_x e a matriz $phiv_x$, que armazena as contribuições das bordas, com espessura de duas células, em v_x. Considerando que as cores da figura codificam o bloco ao qual cada célula pertence, pode-se constatar que não é possível particionar os vetores de maneira convencional, pois as contribuições de um mesmo bloco não são necessariamente sequenciais. Naturalmente, ao considerarmos o espaço tridimensional no qual os cálculos são feitos, esses fatores se comportam de maneira similar.

Apesar das dificuldades acima apresentadas, é necessário encontrar uma alternativa de particionamento para esses vetores. O *macro-kernel computeIntermediates*, por exemplo, acessa as contribuições das bordas em modo de escrita. Portanto, caso nenhum particionamento seja feito, mesmo que várias tarefas sejam submetidas para tratar

Figura 4.6 – Representação de uma matriz de velocidades e um vetor de contribuições das bordas



Fonte: Autor

diferentes regiões do problema, elas não poderão ser executadas em paralelo devido à necessidade de escrever em um mesmo vetor. Por esse motivo, é necessário encontrar uma alternativa de particionamento para esses vetores.

A solução implementada para este estudo começa pela transformação da estrutura de dados das condições de borda, conforme mostra a Figura 4.7. Os diversos vetores que compunham as contribuições no cálculo de velocidade, por exemplo, foram encapsulados em uma nova estrutura *phiv_s* (respectivamente *phit_s* no caso da tensão).

Figura 4.7 – À esquerda, parte da implementação da estrutura de condições de borda e, à direita, parte da nova implementação proposta.

```
struct ABSORBING_BOUNDARY_CONDITION {
    int npmlv;          /* numbers of cell in ABC */
    int npmlt;
    int ***ipml;        /* index in the PML */

    double *phivxx;
    double *phivyy;
    double *phivzz;
    double *phivxy;
    double *phivyx;
    double *phivxz;
    double *phivzx;
    double *phivyz;
    double *phivzy;

    double *phitxxx;
    double *phitxyy;
    double *phitxzz;
    double *phitxyx;
    double *phityyy;
    double *phityzz;
    double *phitxzx;
    double *phityzy;
    double *phitzzz;
```

```
struct phiv_s{
  double* base_ptr;
  int size;
  int offset;
  double *xx;
  double *yy;
  double *zz;
  double *xy;
  double *yx;
  double *xz;
  double *zx;
  double *yz;
  double *zy;
};
struct ABSORBING_BOUNDARY_CONDITION {
  int npmlv;    /* numbers of cell in ABC */
  int npmlt;
  int *ipml;    /* index in the PML */

  struct phiv_s *phiv;    //List of phiv blocks
  struct phit_s *phit;    //List of phit blocks
```

Fonte: Autor

Essa nova estrutura armazena todos os vetores de maneira contígua em um único ponteiro e utiliza seus tamanhos para acessá-los individualmente. A Figura 4.8 mostra uma *macro* desenvolvida para simplificar o acesso aos vetores utilizando a estrutura descrita. Multiplicando o índice do vetor desejado pelo tamanho, comum a todas as componentes, tem-se o deslocamento necessário para encontrá-lo.

Figura 4.8 – *Macro* desenvolvida para construir a nova estrutura proposta.

```
#define COMPUTE_ADDRESS_PHIV_S(phiv) phiv.xx = phiv.base_ptr;\
                                     phiv.yy = phiv.base_ptr + phiv.offset;\
                                     phiv.zz = phiv.base_ptr + 2*phiv.offset;\
                                     phiv.xy = phiv.base_ptr + 3*phiv.offset;\
                                     phiv.yx = phiv.base_ptr + 4*phiv.offset;\
                                     phiv.xz = phiv.base_ptr + 5*phiv.offset;\
                                     phiv.zx = phiv.base_ptr + 6*phiv.offset;\
                                     phiv.yz = phiv.base_ptr + 7*phiv.offset;\
                                     phiv.zy = phiv.base_ptr + 8*phiv.offset;
```

Fonte: Autor

Finalmente, na estrutura *ABSORBING_BOUNDARY_CONDITION*, os conjuntos de vetores $phiv$ e $phit$ foram substituídos por vetores dos tipos *phiv_t* e *phit_t*, com tamanho igual ao número de blocos do problema. Dessa forma, sendo lx a quantidade de blocos de cada linha do domínio, cada bloco (i, j) encontra em $phiv\_t[lx * i + j]$ o conjunto de vetores associados às contribuições das bordas em (i, j).

Esse particionamento realizado introduziu a necessidade de adaptar o vetor $ipml$, responsável por associar cada coordenada à sua posição nos vetores $phiv$ e $phit$. Anteriormente, o valor armazenado era um índice global, utilizado em vetores que representavam o domínio completo do problema. Com a nova estrutura, é necessário que esses índices sejam locais, associados aos vetores que pertencem cada um a um bloco.

Implementando todas as modificações descritas acima, o particionamento dos vetores das condições de borda proposto permite que cada tarefa acesse apenas as posições que lhe são necessárias e o paralelismo entre blocos se torna possível.

4.2.2 Criação de *data handles*

Para que os dados sejam acessados pelas tarefas submetidas na forma de *buffers*, foi necessário encapsulá-los em *data handles* implementados pela biblioteca *StarPU* e responsáveis por gerenciar réplicas dos dados envolvidos. Durante a execução desse projeto foram utilizadas as funções *starpu_data_vector_register*, *starpu_data_matrix_register*

e *starpu_data_block_register* para encapsular vetores unidimensionais, matrizes e tensores tridimensionais, respectivamente. Essas funções têm como argumentos o endereço de início e as dimensões dos dados, valores que posteriormente são acessíveis dentro das tarefas.

A biblioteca *StarPU* oferece funções para realizar o ladrilhamento dos dados contidos em seus *data handles*, para acessar os blocos particionados e para reconstruir a estrutura original dos dados. A Figura 4.9 resume o ciclo de vida de um dos *data handles* utilizados ao longo da implementação desenvolvida.

Figura 4.9 – Ciclo de vida de um *data handle*.

```
starpu_data_handle_t t0_xx_handle;

struct starpu_data_filter x_filter = {.filter_func = starpu_block_filter_block,
                                      .nchildren = n_blocks_x};
struct starpu_data_filter y_filter = {.filter_func = starpu_block_filter_vertical_block,
                                      .nchildren = n_blocks_y};

starpu_block_data_register(&t0_xx_handle, STARPU_MAIN_RAM, (uintptr_t)t0->xx,
                           ncols, depth, nrows, ncols, depth, sizeof(t0->xx[0]));
starpu_data_map_filters(t0_xx_handle, 2, &x_filter, &y_filter);

for (iterations = 0; iterations < max_it; iterations++) {
  for (i_block = 0; i_block < n_blocks_y; i_block++) {
    for (j_block = 0; j_block < n_blocks_x; j_block++) {
      starpu_insert_task(..., starpu_data_get_sub_data(t0_xx_handle, 2, i_block, j_block), ...);
    }
  }
}
starpu_data_unregister(t0_xx_handle);
```

Fonte: Autor

A operação de ladrilhamento dos dados é implementada pela aplicação de de filtros chamados *starpu_data_filter*, que têm como parâmetros o número de blocos a serem criados e o ponteiro para uma função que implementa o particionamento desejado.

No caso desse estudo, os blocos são formados no plano XY, mantendo cada um deles com uma altura igual à do domínio. Portanto, foi necessário a aplicação de duas funções de particionamento, uma responsável pelo eixo X e outra pelo eixo Y. Através da função *starpu_data_map_filters*, foi possível executar o ladrilhamento de um conjunto de dados utilizando a composição de diferentes funções de particionamento.

Depois de aplicar o filtro nos dados desejados, o acesso aos blocos é realizado através da função *starpu_data_get_sub_data*, que tem como argumentos o *data handle* original, o número de dimensões no qual ele foi particionado e os índices do bloco em cada uma dessas dimensões. Dessa maneira é possível fornecer para cada tarefa apenas os bloco que lhe são necessários.

Finalizados os cálculos, os dados dos *data handles* podem ser recuperados em

suas estruturas originais após uma chamada à função *starpu_data_unregister*.

Acesso aos vizinhos

Sendo o *Ondes3D* um *stencil*, a aplicação de um ladrilhamento implica em uma dificuldade introduzida na gestão de seus vizinhos, pois o cálculo dos elementos posicionados nas bordas de um bloco necessita de dados presentes em outros blocos. Para solucionar esse problema mantendo o particionamento, é necessário que cada tarefa receba não só o bloco do tensor a ser atualizado mas também blocos dos tensores lidos nas posições vizinhas. É importante ressaltar que os vizinhos são acessados em modo de leitura, permitindo que o paralelismo entre as tarefas de uma mesma etapa seja mantido.

No caso da implementação proposta, onde o ladrilhamento ocorre apenas no plano XY, um bloco pode possuir de dois a quatro vizinhos em função de sua posição no domínio: blocos posicionados em um dos quatro cantos do plano XY possuem apenas dois vizinhos, enquanto os demais blocos posicionados nas bordas possuem três e os internos possuem quatro. Por isso, o laço de inserção de tarefas precisa avaliar a posição tratada para construir um vetor de blocos contendo os vizinhos que serão utilizados em cada chamada.

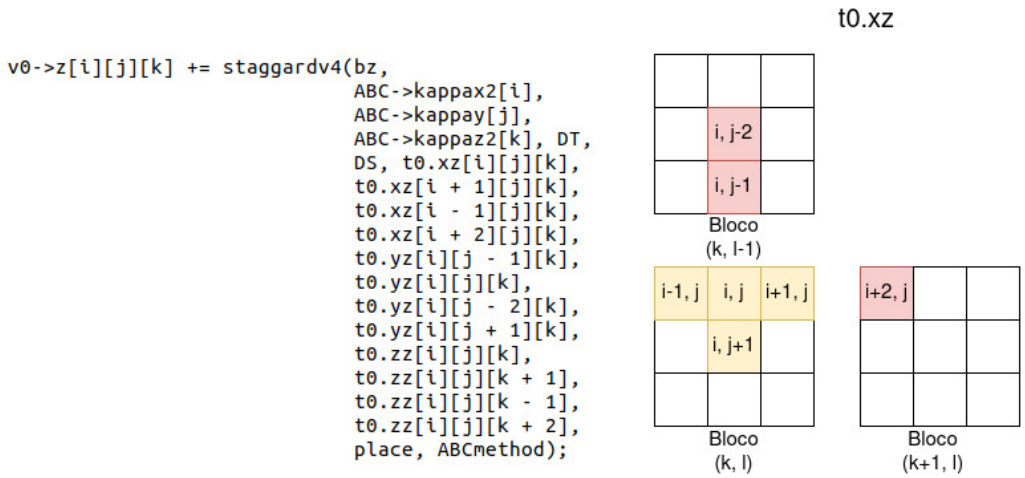
No entanto, a implementação das tarefas tornou-se muito complexa ao utilizar essa solução. O cálculo realizado em cada célula (i, j, k) não depende apenas de seus quatro vizinhos diretos, mas também dos quatro vizinhos posicionados a duas células de distância. Desde que o tamanho do bloco seja maior que dois, essa propriedade não interfere na gestão dos vizinhos em si. Porém, dentro de cada tarefa, torna-se necessário tratar uma quantidade significativa de casos distintos. São eles:

- Células na primeira ou última linha do bloco: índices do tipo $i-1$ e $i-2$ acessam os blocos vizinhos enquanto índices do tipo i acessam o bloco principal
- Células na primeira ou última coluna do bloco: índices do tipo $j-1$ e $j-2$ acessam os blocos vizinhos enquanto índices do tipo j acessam o bloco principal
- Células na segunda ou penúltima linha do bloco: índices do tipo $i-2$ acessam os blocos vizinhos enquanto índices do tipo i e $i-1$ acessam o bloco principal
- Células na segunda ou penúltima coluna do bloco: índices do tipo $j-2$ acessam os blocos vizinhos enquanto índices do tipo j e $j-1$ acessam o bloco principal
- Demais células: todos os índices acessam o bloco principal

Além das possibilidades listadas, as operações realizadas são implementadas em

funções que acessam mais de uma célula, criando assim ainda mais situações possíveis a serem consideradas. A Figura 4.10 ilustra os acessos necessários para parte do cálculo da célula $(i,\ j)$ do bloco $(l,\ k)$ do tensor de velocidade na direção do eixo z na etapa *computeVelocity*.

Figura 4.10 – Parte do cálculo de $v0 -> z[i][j][k]$.



Fonte: Autor

Mesmo sendo mais eficiente em memória, o emprego da técnica de células fantasma, explicado na Seção 4.1, não diminui a complexidade envolvida nesses casos. No entanto, constatou-se que os tensores cujos cálculos podem necessitar de blocos vizinhos não são aqueles que estão sendo atualizados. No exemplo mostrado acima, são os valores de tensão que são consultados para calcular a velocidade em uma dada coordenada. Sendo assim, optou-se por fornecer para cada tarefa um bloco, no caso dos dados que serão escritos, e o tensor completo, no caso dos dados que serão lidos. Com isso, o paralelismo pode ser mantido, já que as escritas acontecem em dados independentes, sem a necessidade de gerenciar leituras em blocos vizinhos.

Em contrapartida, conforme descrito anteriormente, um *data handle* particionado não pode mais ser acessado da maneira original. Para isso, é necessário utilizar a função *starpu_data_unpartition*, que desfaz a operação de ladrilhamento anteriormente realizada. Essa operação exige a criação de um ponto de sincronismo entre as tarefas para garantir que o *data handle* foi reconstruído a partir de seus blocos atualizados na iteração atual, resultando na criação de uma dependência que impossibilita que tarefas de duas etapas entre as quais exista essa reconstrução sejam executadas simultaneamente.

4.2.3 Transformação em tarefas

Cada um dos quatro *macro-kernels* foi escrito seguindo a interface de tarefas *StarPU*. Inicialmente serão abordados os aspectos gerais dessa transformação, comuns a todas as etapas do algoritmo. Em seguida, a implementação da tarefa responsável pelo cálculo da velocidade será vista em detalhes devido à particularidades nela encontradas.

A implementação de uma tarefa *StarPU* começa pela leitura dos blocos utilizados, fornecidos na forma de *buffers*, através das *macros STARPU_VECTOR_GET_PTR*, *STARPU_MATRIX_GET_PTR* e *STARPU_BLOCK_GET_PTR*. Além disso, é necessário consultar as dimensões desses *buffers*; apesar do ladrilhamento separar o espaço em blocos de tamanho igual, é possível que as extremidades possuam blocos de tamanho menor caso não seja possível realizar uma divisão inteira.

No caso da etapa *computeSeisMoment*, como a função se concentra em uma área muito limitada do problema, nenhum particionamento é utilizado e as alterações a serem feitas se resumem à adequação dos acessos aos vetores, que agora passam a utilizar as *macros* citadas anteriormente.

Os laços internos das etapas *computeIntermediates*, *computeStress* e *computeVelocity*, que originalmente percorriam uma região especificada na chamada do *macro-kernel*, passam a percorrer o tamanho de um bloco. No entanto, ao acessar as estruturas fornecidas apenas para leitura, é necessário realizar uma conversão das coordenadas locais em coordenadas globais, já que elas não estarão particionadas. A relação entre as coordenadas globais e locais é dada pelas seguintes equações, onde $block\_size$ é o tamanho do bloco (aqui considerado quadrado), i_{block} é o índice do bloco no eixo x e j_{block} é o índice do bloco no eixo y:

$$i_{global} = block\_size * i_{block} + i_{local}$$

$$j_{global} = block\_size * j_{block} + j_{local}$$

Cálculo de velocidades

Apesar da Seção 4.2.2 mostrar que cada uma das etapas só precisa acessar blocos vizinhos de estruturas em modo somente leitura, existe uma exceção no cálculo de velocidade. Quando o índice k é igual a 1 ou 2, certas operações adicionais são necessárias e estas eventualmente envolvem blocos vizinhos onde outras tarefas precisam atualizar

dados. No caso em que k é igual a um, lê-se dados do bloco seguinte no eixo X e do bloco anterior no eixo Y, enquanto no caso em que k é igual a dois, os dados externos pertencem ao bloco anterior no eixo X e posterior no eixo Y.

Além da gestão dos vizinhos, a necessidade de consultar dados da mesma estrutura em que se está escrevendo adiciona também novas dependências relativas à ordem dos cálculos. Se na aplicação original os vizinhos lidos são sempre aqueles cujas células já foram atualizadas, é preciso manter essa mesma ordem na implementação paralela, quando cada bloco efetua seus cálculos independentemente.

O primeiro ponto demonstrado sugere que uma implementação de tarefas incluindo os possíveis vizinhos dos tensores de velocidade seria capaz de resolver a questão, mas o segundo ponto mostra que só isso não é suficiente para garantir que o mesmo cálculo será realizado. Por isso, a solução encontrada foi particionar a etapa de computação de velocidades em três *kernels* diferentes, sendo o primeiro deles responsável pelas células de índice k igual a um, o segundo pelas células de índice k igual a dois e o terceiro pelos demais índices.

Em relação à vizinhança, em cada um dos casos especiais apenas um par de tensores de velocidade precisa de seus vizinhos, que conforme descrito anteriormente, são no máximo dois blocos. Considerando essas particularidades, foi possível implementar essas tarefas de maneira que os *kernels* responsáveis pelos casos especiais tenham, cada um deles, apenas dois blocos vizinhos, que são fornecidos apenas para leitura. A Figura 4.11 apresenta um resumo da estrutura dessa etapa, onde a ordem de submissão das três tarefas garante a ordem de execução necessária.

Figura 4.11 – Resumo das chamadas de computação de velocidade.

```
for (i_block = 0; i_block < n_blocks_y; i_block++) {
  for (j_block = 0; j_block < n_blocks_x; j_block++) {

    block_v0_x = starpu_data_get_sub_data(v0_x_handle, 2, i_block, j_block);
    block_v0_y = starpu_data_get_sub_data(v0_y_handle, 2, i_block, j_block);
    block_v0_z = starpu_data_get_sub_data(v0_z_handle, 2, i_block, j_block);

    prev_i = (i_block != 0) ? starpu_data_get_sub_data(v0_z_handle, 2, i_block-1, j_block) : NULL;
    prev_j = (j_block != 0) ? starpu_data_get_sub_data(v0_y_handle, 2, i_block, j_block-1) : NULL;
    next_i = (i_block != n_blocks_y-1) ? starpu_data_get_sub_data(v0_x_handle, 2, i_block+1, j_block) : NULL;
    next_j = (j_block != n_blocks_x-1) ? starpu_data_get_sub_data(v0_z_handle, 2, i_block, j_block+1) : NULL;

    starpu_task_insert(&velo_cl, STARPU_RW, block_v0_x, STARPU_RW, block_v0_y, STARPU_RW, block_v0_z,
                       ...);
    starpu_task_insert(&velo_k1_cl, STARPU_RW, block_v0_x, STARPU_RW, block_v0_y,
                       STARPU_RW, block_v0_z, STARPU_R, next_i, STARPU_R, prev_j,
                       ...);
    starpu_task_insert(&velo_k2_cl, STARPU_RW, block_v0_x, STARPU_RW, block_v0_y,
                       STARPU_RW, block_v0_z, STARPU_R, prev_i, STARPU_R, next_j,
                       ...);
  }
}
```

Fonte: Autor

5 AVALIAÇÃO EXPERIMENTAL

A nova versão desenvolvida foi executada no Parque Computacional de Alto Desempenho (PCAD) da Universidade Federal do Rio Grande do Sul com o intuito de comparar seu desempenho em relação à implementação original.

5.1 Ambiente de testes

Para a execução dos testes, a máquina *tupi2* do PCAD foi utilizada, contando com um processador *Intel Xeon E5-2620* com 8 núcleos, 80GB de memória RAM DDR4.

5.2 Resultados

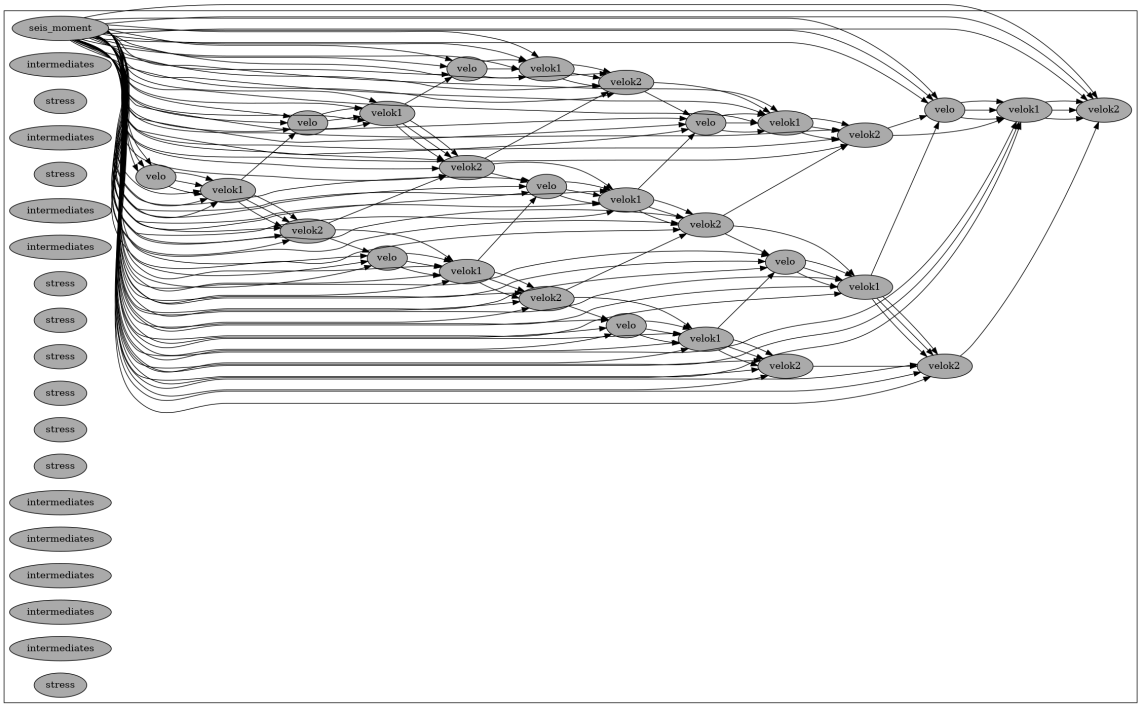
Apesar do objetivo original ser a comparação da nova implementação com a antiga, não foi possível concluir o desenvolvimento da nova versão utilizando a programação baseada em tarefas. Todas as tarefas foram construídas e o algoritmo é executado do início ao fim sem produzir erros de execução. No entanto, ao comparar o resultado final obtido, não encontra-se o valor fornecido pela aplicação original.

Diversas investigações foram realizadas e indicam que os erros matemáticos encontram-se no cálculo das contribuições das condições de borda, etapa previamente identificada como mais complexa. Devido ao tempo disponível, optou-se por apresentar o estudo desse desenvolvimento independentemente da corretude numérica dos resultados devido ao valor científico das ponderações realizadas ao longo da implementação e também da possibilidade de realizar uma análise sobre o paralelismo alcançado com as tarefas da maneira proposta.

Em relação ao paralelismo alcançado, a Figura 5.1 mostra o DAG construído pela biblioteca *StarPU* na execução de uma iteração do algoritmo, utilizando blocos de dimensões 100x100 no domínio $[-150, 150]$,$[-150, 150]$,$[-150, 1]$, dividido portanto em nove blocos. Os nodos do grafo mostrado representam as tarefas, enquanto as setas indicam as dependências entre elas. Analisando a figura, percebe-se que, apesar das dependências entre a tarefa *computeSeisMoment* e as três responsáveis pelos cálculos de velocidade estarem corretas, a computação das contribuições das condições de borda e da tensão estão ambas desconexas das demais. Dessa maneira, existe um paralelismo indevido entre

essas tarefas, que passam a serem consideradas executáveis simultaneamente, resultando em uma execução errônea.

Figura 5.1 – Grafo de dependências entre as tarefas.



Fonte: Autor

A falha na definição das dependências das tarefas tornou dispensável a execução de mais testes, já que não seria possível comparar os resultados obtidos com a aplicação original. Acredita-se na possibilidade das operações de reconstrução de tensores já ladrilhados ter alguma influência na construção indevida do DAG, porém o ocorrido exige investigações mais profundas.

6 CONCLUSÃO

A complexidade da simulação da propagação de ondas sísmicas através do método de diferenças finitas motiva a elaboração de diversos trabalhos que buscam otimizar sua execução de diferentes maneiras. Uma alternativa identificada foi a exploração das dependências entre os dados envolvidos para introduzir um nível de paralelismo superior ao da aplicação original. Essa exploração é possível graças ao emprego do paradigma de programação baseado em tarefas, implementado pela biblioteca *StarPU*.

O trabalho aqui descrito realiza um estudo aprofundado das estruturas envolvidas no algoritmo original e das técnicas necessárias para alcançar o modelo de tarefas. A partir da operação de ladrilhamento, responsável pela criação de blocos de dados independentes, foi apresentado um projeto com o objetivo de implementar tarefas que calculam os dados contidos em cada um desses blocos. A implementação proposta permite a parametrização do tamanho dos blocos, de maneira que a divisão deles seja adequada à arquitetura onde a aplicação será executada e considera as condições de absorção das bordas, um aspecto não abordado por outros trabalhos similares.

Apesar da conclusão de uma implementação geral, os resultados obtidos não foram os esperados. Erros numéricos foram encontrados em uma das etapas do cálculo e, analisando a aplicação finalizada, foram detectadas inconsistências entre as dependências. No entanto, acredita-se que os estudos elaborados e uma parcela significativa da implementação realizada possam ser aproveitados para a construção de uma versão mais eficiente do *Ondes3D*.

O gerenciamento de vizinhos e o particionamento das contribuições fornecidas pelas condições de borda foram os maiores obstáculos na elaboração desse estudo. Acredita-se que as propostas do projeto apresentem soluções satisfatórias para esses problemas, mas o desenvolvimento efetivo das possibilidades mais promissoras exige uma reestruturação maior do código, alterando dados internos das estruturas de dados utilizadas e dividindo os *macro-kernels* em tarefas menores e mais específicas.

A sequência natural do trabalho apresentado é a correção dos erros numéricos introduzidos na mudança de modelo de programação realizada. Assim que esses resultados forem alcançados, diversas melhorias podem ser postas em prática de maneira a acelerar ainda mais a execução da aplicação. Algumas delas são listadas abaixo:

- Implementação visando arquiteturas heterogêneas: assim como o trabalho desenvolvido por Martínez et al. (MARTíNEZ et al., 2015), é possível implementar

kernels que permitem a execução de tarefas em *GPU*s.

- Implementação incluindo otimizações de operações de entrada e saída: baseando-se nas alterações propostas por Boito et al. (BOITO et al., 2017), é possível construir uma versão paralela com uma quantidade reduzida de operações de entrada e saída, combinando os ganhos em eficiência de ambas as propostas.
- Implementação de blocos de formato diferente: em vez de limitar o ladrilhamento ao plano XY, a opção de particionar o espaço em todos os eixos permite o emprego de mais parâmetros que, refinados, podem gerar melhores resultados.
- Implementação de tarefas de granularidade menor: as tarefas implementadas são compostas por diversas etapas menores, frequentemente associadas às diferentes regiões do domínio calculado. A transformação dessas etapas menores em tarefas resulta numa granularidade mais fina, o que pode contribuir para o equilíbrio de carga entre as unidades de execução.
- Refatoração do problema: atualmente, a aplicação conta com uma função para cada grande etapa, resultando em longas implementações que tratam os mais diversos casos. Uma refatoração do código separando, por exemplo, o tratamento de cada região diferente, poderia não só facilitar o entendimento do algoritmo mas também permitir uma granularidade ainda mais fina na elaboração das tarefas.

tupi pequena cei media blaise grande

REFERÊNCIAS

AUGONNET, C. et al. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In: . [S.l.: s.n.], 2009. v. 23. ISBN 978-3-642-03868-6.

BERENGER, J.-P. A perfectly matched layer for the absorption of electromagnetic waves. **Journal of Computational Physics**, v. 114, n. 2, p. 185–200, 1994. ISSN 0021-9991. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0021999184711594>.

BOITO, F. Z. et al. High performance i/o for seismic wave propagation simulations. In: **2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)**. [S.l.: s.n.], 2017. p. 31–38.

CASTRO, M. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. **Parallel Computing**, v. 54, p. 108–120, 2016. ISSN 0167-8191. 26th International Symposium on Computer Architecture and High Performance Computing. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0167819116000417>.

DUPROS, F.; DO, H.-T.; AOCHI, H. On scalability issues of the elastodynamics equations on multicore platforms. In: **ICCS 2013 : International conference on computational cience**. Barcelone, Spain: Elsever, 2013. (Procedia Computer Science), p. 9 p. Available from Internet: <https://hal-brgm.archives-ouvertes.fr/hal-00797682>.

DUPROS, F. et al. High-performance finite-element simulations of seismic wave propagation in three-dimensional nonlinear inelastic geological media. **Parallel Comput.**, v. 36, p. 308–325, 2010.

GARCIA PINTO, V. et al. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. **Concurrency and Computation: Practice and Experience**, v. 30, n. 18, p. e4472, 2018. E4472 cpe.4472. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4472>.

MARTíNEZ, V. et al. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In: **2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [S.l.: s.n.], 2015. p. 1–8.

NATAF, F. Absorbing boundary conditions and perfectly matched layers in wave propagation problems. **Radon Series on Computational and Applied Mathematics**, v. 11, 01 2013.

NESI, L. L. et al. Task-based parallel strategies for computational fluid dynamic application in heterogeneous cpu/gpu resources. **Concurrency and Computation: Practice and Experience**, v. 32, n. 20, p. e5772, 2020. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5772>.

NESI, L. L. et al. Visual performance analysis of memory behavior in a task-based runtime on hybrid platforms. In: **2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)**. [S.l.: s.n.], 2019. p. 142–151.

PINTO, V. G. et al. Analyzing dynamic task-based applications on hybrid platforms: An agile scripting approach. In: **2016 Third Workshop on Visual Performance Analysis (VPA)**. [S.l.: s.n.], 2016. p. 17–24.

RODEN, J. A.; GEDNEY, S. D. Convolution pml (cpml): An efficient fdtd implementation of the cfs–pml for arbitrary media. **Microwave and Optical Technology Letters**, v. 27, n. 5, p. 334–339, 2000. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/1098-2760%2820001205%2927%3A5%3C334%3A%3AAID-MOP14%3E3.0.CO%3B2-A>.