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

LUCAS BARROS DE ASSIS

Modelagem de ondas sísmicas através de 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 uma onda sísmica. Apesar de contar com uma implementação paralela utilizando *OpenMP*, esse paralelismo só é obtido dentro de cada um de seus *macro-kernels*. O trabalho aqui apresentado estuda as alterações necessárias para dividir as estruturas utilizadas pelo simulador em ladrilhos, possibilitando posteriormente uma implementação em forma de tarefas utilizando a biblioteca *StarPU*. Essa divisão em tarefas permite um controle da granularidade dos processos paralelos através do tamanho dos ladrilhos utilizados, o que leva à uma otimização dos acessos em memória. Além disso, é possível que diferentes etapas do processo sejam executadas simultaneamente graças às especificações das dependências que o modelo de programação em tarefas inclui. Dessa forma, espera-se alcançar uma execução que aproveita arquiteturas *multi-core* de forma mais vantajosa que a versão original.

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 a seismic wave. Even though it has a parallel implementation using *OpenMP*, this parallelism is only achieved inside each of its macro-kernels. 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. By splitting the code in tasks, it becomes is possible to control the granularity of the parallel processes through the tile sizes, which enables a memory access optimization. Other than that, different steps of the computation can be executed simultaneously thanks to the dependency specifications from the task-based model. With these modifications, it may be possible to achieve an execution which exploits multi-core architectures even better than the original version.

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). A aplicação *Ondes3D* realiza essa simulação através do método de diferenças finitas, utilizando a biblioteca *OpenMP* para produzir paralelismo local e o protocolo *MPI* em contextos distribuídos. No entanto, o paralelismo local somente existe dentro de cada *macro-kernel*, isto é, não existe execução paralela entre diferentes etapas do cálculo. Uma alternativa possível para acelerar a execução pode ser alcançada utilizando um modelo que permita um paralelismo ainda maior que o atual.

A biblioteca *StarPU* é uma alternativa atual que implementa a programação baseada em tarefas para obter paralelismo. Dentro dese modelo, uma tarefa consiste em uma função cuja especificação inclui, além de seus parâmetros, os seus modos de acesso: somente leitura, somente escrita ou leitura e escrita. Essas tarefas são incluídas ao longo do código e, baseando-se na ordem de inserção e nos modos de acesso de cada uma delas, a biblioteca *StarPU* constrói um grafo acíclico direcionado representando as dependências entre essas tarefas. Durante a execução do programa, esse grafo é utilizado para escalonar as tarefas conforme a disponibilidade dos recursos computacionais. Apesar de contar com uma implementação paralela, o modelo de programação utilizado na versão original da aplicação não explora o paralelismo tanto quanto seria possível devido à sua incapacidade de considerar as dependências entre os dados utilizados. Ao utilizar o paradigma de tarefas, torna-se possível considerar essas dependências para que diferentes etapas do algoritmo possam ser executadas simultaneamente, alcançando assim um paralelismo superior ao atualmente implementado.

Este trabalho tem como objetivo estudar e implementar as alterações necessárias para que o algoritmo existente seja executado na forma de tarefas, assim como avaliar o paralelismo obtido ao utilizar esse paradigma.

1.1 Contribuições

2 TRABALHOS RELACIONADOS

A aplicação *Ondes3D* já foi objeto de estudo em outros trabalhos, frequentemente visando um melhor desempenho, mas passando também pela busca de um consumo energético mais eficiente. Além disso, outras aplicações também tiveram seu tempo de execução diminuído empregando a programação baseada em tarefas. Alguns desses trabalhos serão discutidos abaixo.

Boito et al. (BOITO et al., 2017) verificaram que mais de 72% do tempo de execução do *Ondes3D* é gasto realizando 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 foram substituídos por escritas diretas no sistema de arquivos. A terceira implementação passa a utilizar diversos arquivos de saída, o que permite que diversas operações de escrita aconteçam paralelamente. Apesar dos resultados impressionantes desse trabalho, o estudo aqui proposto não tem como foco as operações de entrada e saída mas sim a paralelização dos cálculos das equações elastodinâmicas, o que permitiria em um momento futuro a combinação das duas otimizações para alcançar uma aceleração ainda mais significativa.

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, causando um desbalanceamento de cargas 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%. O presente estudo não considera a utilização de programação distribuída, mas a granularidade parametrizável alcançada utilizando o modelo proposto permite uma mitigação dos efeitos de uma distribuição desbalanceada. Quanto maior o número de tarefas disponível em um dado momento, menores são as chances de uma unidade de cálculo permanecer ociosa.

Castro et al. (CASTRO et al., 2016) utilizaram um processador de baixa potência para propor uma implementação da aplicação *Ondes3D* com um baixo consumo energético. A partir de uma estratégia de ladrilhamento multi-nível, a versão desenvolvida para o processador *MPPA-256* 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*. Além de incluir tarefas voltadas para *GPU*s, o traço de execução resultante foi avaliado e refinado utilizando o conjunto de ferramentas *StarVZ*. Utilizando a técnica de células fantasma, explicada na seção 4.1, foi possível alcançar uma aceleração de 77x em relação à versão original.

Martínez et al. (MARTíNEZ et al., 2015) propuseram uma implementação 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 descartas as interações com as bordas do domínio devido a complexidade introduzida pelo uso delas. A melhor performance foi alcançada utilizando quatro *CPU*s e oito *GPU*s, uma aceleração de 25x em relação à versão original utilizando doze *CPU*s.

A Tabela 2.1 resume as diferenças entre os trabalho desenvolvido 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.	*Ondes3D*	Otimização de operações de entrada e saída
Tesser et al.	*Ondes3D*	Balanceamento dinâmico de carga
Martínez et al.	*Ondes3D*	Implementação baseada em tarefas em arquiteturas heterogêneas sem condições de borda
Castro et al.	*Ondes3D*	Otimização de consumo energético
Nesi et al.	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*.

3.1 Ondes3D

A aplicação discretiza o mundo real em diversos tensores tri-dimensionais, endereçados a partir de suas coordenadas. Esses tensores são parte de diferentes estruturas de dados, que podem variar de acordo com os dados experimentais utilizados na simulação.

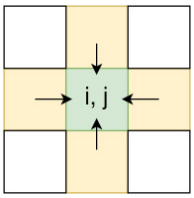
Esse trabalho toma como objeto de estudo os dados do terremoto de *Chuetsu-Oki*, que atingiu a costa japonesa em 2007. Nesse caso, a aplicação conta com as seguintes estruturas, usadas para representar os fenômenos físicos que compõem a onda:

- *STRESS*: composta por tensores que representam as tensões sofridas em diferentes planos. O tensor *xy*, por exemplo, representa 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 velocidade da onda nas direções 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 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. Essas bordas são utilizadas para limitar o domínio computacional onde o cálculo é feito gerando o mínimo de interferência nos cálculos realizados. (NATAF, 2013).

Os cálculos são realizados seguindo um padrão *stencil*, onde o valor atribuído aos tensores que descrevem as grandezas físicas são calculados usando informações das células vizinhas. A Figura 3.1 ilustra um *stencil* 2D com quatro vizinhos. 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*:

- *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 bastante restrito, 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 sob as coordenadas em X, Y e Z, a biblioteca *OpenMP*, através da diretiva citada anteriormente, cria *threads* capazes de tratar diferentes coordenadas simultaneamente.

Teoricamente, a partir do momento que a tensão, por exemplo, de uma célula fosse calculada, já seria possível iniciar o cálculo de sua velocidade. No entanto, como a diretiva descrita acima não suporta o estabelecimento de dependências entre os dados, é necessário que a tensão de todas as células seja calculada para que o algoritmo possa iniciar a fase de cálculo de velocidades. Dessa maneira, apesar da solução atual alcançar uma aceleração graças ao paralelismo introduzido, a quantidade de operações que podem ser executadas simultaneamente é menor do que seria possível. O paralelismo baseado em tarefas busca explorar as dependências entre os dados para alcançar esse nível maior de independência entre as etapas do algoritmo.

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. Essa biblioteca 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 do momento de sua execução é baseada nas dependências de dados existentes entre elas.

A utilização de tarefas é composta por dois momentos, sendo o primeiro deles a sua descrição e o segundo a sua inserção 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 de seu *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*, tipicamente valores constantes.

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 3.2.

A criação de instâncias de tarefas é feita através da função *starpu_insert_task*,

Figura 3.2 – 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

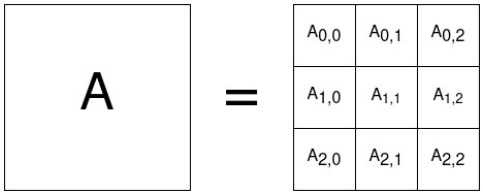
tendo como parâmetros o *codelet* correspondente, os *buffers* utilizados e os ponteiros registrados como *cl_args*.

A biblioteca segue o modelo de *sequential task flow*, que 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 (também chamado de *Directed Acyclic Graph*, ou *DAG*) 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.

3.2.2 Matrizes *ladrilhadas*

Uma estratégia recorrente na programação de alta performance é o ladrilhamento de matrizes - no caso desse estudo, de tensores. Essa técnica consiste, conforme ilustra a Figura 3.3, na divisão de uma matriz em sub-matrizes (também chamadas de blocos) de menor tamanho, indexadas pela sua posição em relação à matriz original.

Figura 3.3 – Decomposição da matriz A em blocos



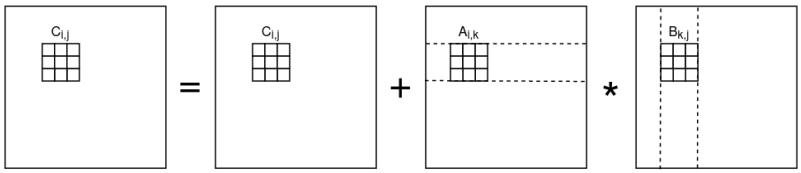
Fonte: Autor

O ladrilhamento de matrizes exige que os algoritmos sejam re-adequados à sua

estrutura, mas em contrapartida oferecem numerosas vantagens. Em primeiro lugar, podemos 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 aplicação.

A Figura 3.4 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 3.4 – 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 simultaneamente o cálculo de todos os bloco da matriz resultante. Apesar dos blocos das matrizes A e B serem acessados por diversas tarefas, eles não sofrem nenhuma escrita. Os blocos da matriz C, onde as escritas são realizadas, são acessados cada um deles por uma única tarefa, possibilitando tal paralelismo.

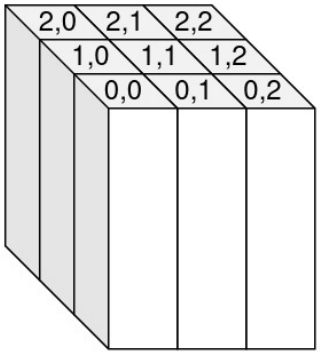
4 PROJETO E IMPLEMENTAÇÃO

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

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. A Figura 4.1 ilustra o particionamento descrito utilizado pela implementação original, que parte de um arquivo de topologia para discretizar o plano XY em blocos de profundidade igual à do domínio.

Figura 4.1 – Representação do espaço particionado em nove blocos



Fonte: Autor

Em relação ao laço principal de execução, a Figura 4.2 resume o fluxo da aplicação original. O domínio do problema é dividido em cinco regiões denominadas *imodes*, sendo quatro delas as linhas e colunas externas 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 $[x_{min}, x_{max}][y_{min}, y_{max}][z_{min}, z_{max}]$.

Na versão aqui proposta, cada etapa será executada tantas vezes quanto houverem blocos. Para isso, será necessário iterar sobre os blocos disponíveis, criando uma tarefa para cada bloco de dados, conforme mostra a Figura 4.3, dispensando a utilização dos

Figura 4.2 – 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

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.

Figura 4.3 – 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]);
    }
  }
}
```

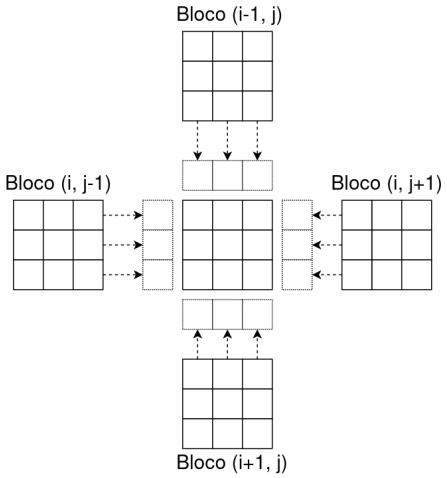
Fonte: Autor

Internamente, cada tarefa percorrerá apenas o espaço determinado pelo tamanho de bloco utilizado em vez de uma região completa como na versão original.

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 efi-

ciente, 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 leitura. Essa alternativa, dependendo do contexto em que ela é utilizada, pode representar um custo de memória mais elevado do que o necessário. 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 utilizar os quatro vizinhos completos, compartilha-se apenas as bordas que serão necessárias. A Figura 4.4 mostra um exemplo de células fantasma em um *stencil* 2D, mas o conceito pode ser expandido para três dimensões.

Figura 4.4 – Representação do uso de células fantasma



Fonte: Autor

4.2 *Tarefização* dos *macro-kernels*

5 AVALIAÇÃO EXPERIMENTAL

5.1 Ambiente de testes

5.2 Resultados

6 CONCLUSÃO

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.

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.

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>.