Análise de Escalabilidade em um Código de Inversão de Forma de Onda Completa

Felipe H. Santos-da-Silva[1], **João B. Fernandes**[2],
Samuel Xavier-de-Souza[2], **Ítalo A. S. Assis**[1]

[1] Departamento de Engenharias e Tecnologia
Universidade Federal Rural do Semi-Árido (UFERSA)

[2]Departamento de Engenharia de Computação e Automação
Universidade Federal do Rio Grande do Norte (UFRN)
Natal – RN – Brasil

felipe.silva65229@alunos.ufersa.edu.br,

joao.batista.fernandes.094@ufrn.edu.br,

samuel@dca.ufrn.br, italo.assis@ufersa.edu.br

Resumo. *A Inversão de Forma de Onda Completa (FWI) é conhecida por sua alta demanda computacional. A fim de otimizar seu desempenho e o uso dos recursos computacionais, é necessária uma análise cuidadosa da sua escalabilidade. Este artigo avalia a escalabilidade de um código de FWI em um sistema de memória compartilhada. Para tal avaliação, utilizamos a ferramenta Parallel Scalability Suite (PaScal Suite) a fim de identificar gargalos. O PaScal Suite foi utilizado para automatizar testes de escalabilidade e visualizar seus resultados. Além da identificação de gargalos, o estudo inclui a otimização desses trechos no código.*

1. Introdução

A Inversão da Forma de Onda Completa (FWI, do inglês *Full Waveform Inversion*) [Tarantola 1984] é uma técnica geofísica amplamente utilizada para otimizar um modelo que representa as velocidades de propagação da onda na subsuperfície. Apesar de sua eficácia, a FWI é um método computacionalmente intensivo, exigindo grande capacidade de processamento e recursos de hardware substanciais para lidar com a complexidade e o volume de dados envolvidos.

Com o aumento da complexidade dos modelos e a quantidade de dados a serem processados, torna-se crucial garantir que o código implementado para o FWI escale eficientemente com o acréscimo de recursos computacionais. Testes de escalabilidade são fundamentais para identificar possíveis gargalos e limitações no código, permitindo ajustes e otimizações que maximizem o uso eficiente dos recursos disponíveis. Esses testes requerem a avaliação do comportamento do programa sob diferentes condições de carga e configurações de hardware, o que pode gerar uma quantidade massiva de dados a serem analisados. Ferramentas especializadas são, portanto, imprescindíveis para automatizar e simplificar esse processo.

Muitos autores têm estudado e implementado técnicas de otimização para a FWI. [da Silva et al. 2024] exploraram abordagens paralelas para autoajuste de carga visando melhorias de desempenho na FWI. [Zeng et al. 2022] integraram técnicas de redes neurais profundas à FWI para aprimorar a precisão dos modelos de velocidade e reduzir os custos computacionais. [Okita et al. 2020] propôs uma técnica paralelizada por disparos para FWI, analisando a escalabilidade e o impacto dos preços na nuvem, e desenvolveu um algoritmo para otimizar a seleção do número de nós e o tamanho da carga de trabalho, visando melhorar a eficiência e reduzir os custos. [Etienne et al. 2014] propuseram otimizações que aceleram a FWI em até 27 vezes. As técnicas incluem a substituição das bordas absorventes CPML por bordas aleatórias, o uso de vetorização via instruções AVX e a aplicação de *cache blocking* para melhorar a eficiência de memória. As técnicas de [da Silva et al. 2024], [Zeng et al. 2022] e [Etienne et al. 2014] podem ser complementares às estratégias adotadas neste trabalho, como a otimização das operações de *checkpointing* e a paralelização com OpenMP, visando minimizar os gargalos identificados no código da FWI. Diferentemente de [Okita et al. 2020], focamos em avaliar a escalabilidade e propor otimizações para a FWI em sistemas de memória compartilhada.

Neste trabalho, foi utilizado o PaScal Suite, um conjunto de ferramentas projetadas para auxiliar os desenvolvedores na realização de testes de escalabilidade. O PaScal Analyzer [da Silva et al. 2022] fornece instrumentação manual e automática, mapeamento direto de regiões paralelas e reduções de dados com preservação de precisão, além de automatizar as execuções das instâncias. Já o PaScal Viewer [da Silva et al. 2019] é uma aplicação web que facilita a visualização dos dados gerados pelo Analyzer, utilizando elementos visuais intuitivos.

Este artigo visa avaliar a escalabilidade de um código de FWI implementado em C++, utilizando o PaScal Suite para identificar possíveis gargalos em sistemas de memória compartilhada. Além disso, busca-se mitigar esses gargalos no código para garantir uma execução eficiente e robusta do algoritmo FWI em ambientes computacionais variados.

Este artigo está dividido da seguinte forma: primeiramente, é detalhado nosso objeto de estudo, a FWI (Seção 2). Na Seção 3, é apresentada uma descrição da ferramenta PaScal Suite. Na Seção 4, são descritos os testes de escalabilidade realizados na FWI, as modificações feitas com base na análise e discutidos os resultados obtidos. A Seção 5 conclui o trabalho.

2. Algoritmo de Inversão de Forma de Onda Completa

A Inversão da Forma de Onda Completa (FWI, do inglês: *Full Waveform Inversion*) é uma técnica geofísica avançada de inversão de dados que busca obter um modelo preciso das características da subsuperfície [Tarantola 1984]. Essa técnica reduz a diferença entre os dados sísmicos sintéticos, gerados por algoritmos de modelagem de campo de ondas, e os dados sísmicos observados em aquisições reais. O resultado esperado da FWI é um modelo de velocidades que gera dados sintéticos altamente semelhantes aos dados reais.

A FWI pode ser descrita como um problema de otimização, onde a diferença entre os dados calculados e observados é minimizada. O método utiliza o gradiente da função objetivo para determinar a direção de busca [Plessix 2006]. O modelo é atualizado com base nesse gradiente, reduzindo a diferença entre os dados observados e calculados. Esse processo é iterativo e continua até atingir uma condição de parada, como um número

máximo de iterações ou um critério de convergência (por exemplo, a norma do gradiente). Matematicamente, a FWI é representada por:

$$\min_{\mathbf{m}\in\mathbb{R}^n} \frac{1}{2}\|\mathbf{d} - \mathbf{G}(\mathbf{m})\|^2, \tag{1}$$

onde $\|.\|$ denota a norma euclidiana; $\mathbf{m}$ é o vetor que contém todos os parâmetros do modelo; $\mathbf{d}$ são os dados observados e $\mathbf{G}$ é o operador de modelagem responsável por simular computacionalmente o processo de aquisição sísmica.

Os principais passos da FWI são:

1. Receber dados observados;
2. Receber um modelo inicial;
3. Obter os dados calculados executando o operador de modelagem sísmica para o modelo inicial;
4. Comparar a solução calculada com os dados sísmicos observados;
5. Calcular o gradiente da função objetivo com base na diferença entre a solução calculada e os dados observados;
6. Atualizar o modelo com base no gradiente da função objetivo;
7. Repetir os passos 3 a 6 até atingir a condição de parada.

Neste trabalho, o objeto de estudo é a FWI do software Mamute, implementado em C++ utilizando técnicas de paralelização para otimização do código. Essa versão da FWI utiliza a estratégia de *checkpointing* ótimo [Symes 2010] com a biblioteca proposta por [Griewank and Walther 2000]. O repositório do código encontra-se disponível em: `https://gitlab.com/lappsufrn/seismic/ufrn-fwi/mamute`.

3. PaScal Suite

O PaScal Suite engloba duas ferramentas: o PaScal Analyzer [da Silva et al. 2022] e o PaScal Viewer [da Silva et al. 2019].

3.1. PaScal Analyzer

O Parallel Scalability Analyzer, ou PaScal Analyzer, é uma ferramenta útil para avaliar a escalabilidade de desempenho de programas de computação de alto desempenho em sistemas de memória compartilhada. Disponível para as linguagens C e C++, o Analyzer permite a um desenvolvedor coletar dados de múltiplas execuções do programa e automatizar o processo de obtenção de medidas comparativas entre as execuções, incluindo a variação de parâmetros de configuração definidos pelo usuário. O Analyzer permite a instrumentação automática de um código compilado com o *GCC*, além disso, a biblioteca do Analyzer tem suporte à instrumentação manual do código.

A natureza do Analyzer visa minimizar a intrusão e o *overhead* associados à coleta de dados, garantindo uma análise eficiente e precisa do desempenho do programa. A ferramenta também fornece medidas de consumo de energia, contribuindo para uma avaliação abrangente do impacto da escalabilidade no consumo de recursos computacionais. Um repositório público do Analyzer encontra-se disponível em: `https://gitlab.com/lappsufrn/pascal-suite-tutorial`.

3.2. PaScal Viewer

O Parallel Scalability Viewer (PaScal Viewer) é uma aplicação web projetada para visualizar métricas de desempenho em programas paralelos em sistemas de memória compartilhada. Ele aceita como entrada o arquivo de saída do Analyzer, fornecendo uma maneira simplificada de analisar e otimizar o código para escalabilidade paralela, introduzindo uma abordagem inovadora por meio de diagramas de cores que lembram mapas de calor. Esses diagramas são um suporte visual para identificar tendências de eficiência paralela do programa inteiro ou de partes específicas de um código paralelo. A interface do Viewer também possibilita a análise de zonas paralelas em hierarquia, permitindo o perfilamento de um código paralelo. O Viewer se encontra disponível no domínio: `https://pascalsuite.imd.ufrn.br/`.

4. Experimentos

Nesta seção, são apresentados os resultados da análise de escalabilidade do algoritmo de modelagem descrito na Seção 2. Os experimentos foram realizados em um nó do supercomputador NPAD, localizado na *Universidade Federal do Rio Grande do Norte* (UFRN). Este nó é equipado com 2 CPUs AMD EPYC 7713 de $2.0GHz$, totalizando 128 núcleos de processamento, e $512GB$ de RAM.

Para os experimentos, foram consideradas três dimensões espaciais ($x1$, $x2$, e $x3$), com os respectivos números de amostras $n1$, $n2$, e $n3$. Em todos os casos, os parâmetros da modelagem foram mantidos constantes, com uma frequência de pico (f_{peak}) de $10Hz$, um intervalo de amostragem temporal de $1ms$, um total de 1228 passos de tempo, resoluções espaciais uniformes de $\Delta x1 = \Delta x2 = \Delta x3 = 10m$ e uma espessura de borda de 25 pontos da grade em todas as direções da malha tridimensional. O modelo *i4* foi construído utilizando uma esfera com pertubação gaussiana, onde as camadas superiores possuem velocidades de $2500m/s$ e seu núcleo velocidade máxima de $3500m/s$. Foram utilizados 4 modelos de velocidades, denominados *i1*, *i2*, *i3* e *i4*, compostos de uma malha de $(n1, n2, n3) = (25, 200, 200)$, $(50, 200, 200)$, $(100, 200, 200)$ e $(200, 200, 200)$ pontos, respectivamente. O modelo *i4* é apresentado na Figura 1. Os demais são cortes do modelo da Figura 1 na dimensão $x1$.

Embora os parâmetros não tenham sido escolhidos para replicar cenários reais, como o pré-sal, optou-se por modelos menores para garantir a viabilidade dos testes nas limitações de tempo e memória disponíveis. Mesmo com modelos reduzidos, os resultados são considerados representativos do comportamento do código em cenários maiores.

4.1. Avaliação com o PaScal Suite

Para o perfilamento do código, foi utilizado o PaScal Analyzer (PaScal-A), avaliando a escalabilidade com combinações de 32, 16, 8, 2, e 1 núcleos de processamento com os modelos de entrada apresentados na Seção 4. Consideramos que os testes com essas quantidades de núcleos foram suficientes para identificar os gargalos de escalabilidade, o que nos levou a optar por não realizar testes com uma quantidade maior de núcleos neste momento.

O PaScal-A foi configurado para repetir 5 vezes cada combinação de número de núcleos e tamanho de problema. Além disso, utilizamos a funcionalidade de

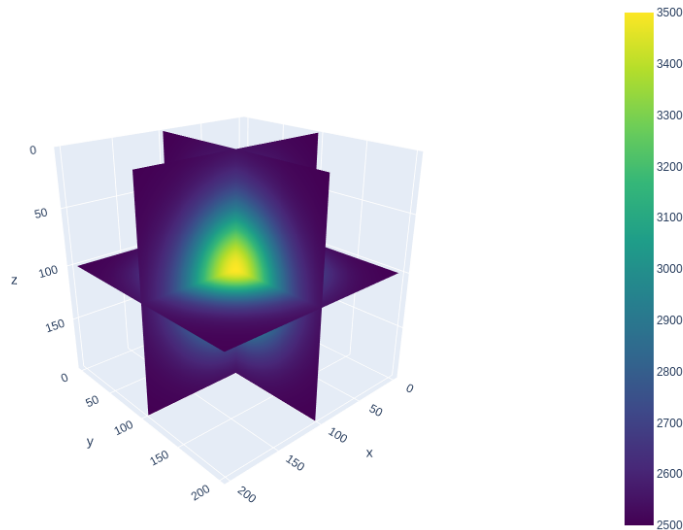


Figura 1. Modelo de velocidades com malha de $n1 = n2 = n3 = 200$ pontos. Os pontos mais próximos da borda do modelo possuem uma velocidade de 2500*m/s* enquanto o núcleo atinge uma velocidade máxima de 3500*m/s*. O modelo foi construído com resoluções espaciais de $\Delta x1 = \Delta x2 = \Delta x3 = 10$*m*, utilizando uma borda de 25 pontos da grade em todas as direções.

instrumentação manual para medir o tempo de execução em trechos específicos do algoritmo da FWI, visando identificar possíveis gargalos de desempenho. O Algoritmo 1 ilustra a instrumentação manual da FWI utilizando o PaScal-A. Cada região de código que identificamos como crítica ou potencial gargalo é envolvida por chamadas *pascal_start(id)* e *pascal_stop(id)*. Essas funções delimitam o início e o fim de cada seção medida, e o identificador único *id* permite ao PaScal-A monitorar o tempo de execução dessas regiões precisamente.

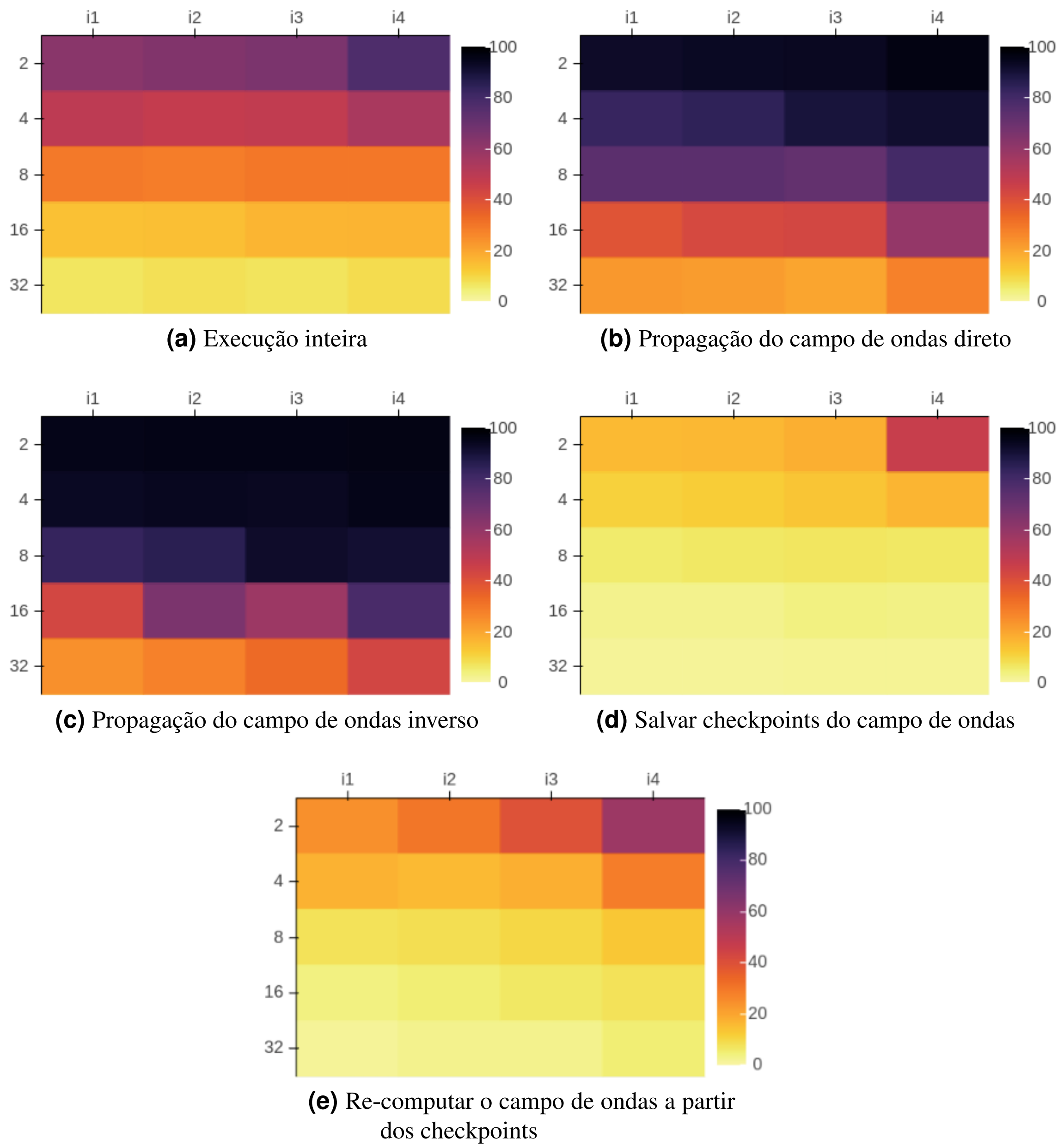
Ao analisar a execução com 32 núcleos e a entrada *i4* no PaScal-A, observamos que o cálculo do campo de onda direto corresponde a $13,91\%$ do tempo total, enquanto o cálculo do campo de onda inverso ocupa 10%. O processo de salvar *checkpoints* representa $24,96\%$ do tempo total, e a recuperação desses *checkpoints* equivale a $24,26\%$.

Utilizamos o PaScal Viewer (PaScal-V) para visualizar as saídas geradas pelo PaScal-A. Neste trabalho, vamos nos concentrar na análise dos gráficos de eficiência produzidos pela ferramenta. A Figura 2a apresenta a eficiência paralela da FWI na totalidade, onde podemos observar uma queda na eficiência à medida que o tamanho do problema e a quantidade de núcleos aumentam na mesma proporção, indicando que essa implantação do FWI não escala com o número de núcleos.

As Figuras 2b e 2c mostram a eficiência das seções de cálculo do campo de onda direto e cálculo de onda inverso, que utilizam estruturas de repetição paralelizadas com OpenMP. Essas figuras demonstram que essas regiões possuem valores de eficiência maiores do que a aplicação completa. A Figura 2d mostra a eficiência do processo de sal-

Algorithm 1 Inversão da Forma de Onda Completa com instrumentação automática.

```
1: Incluir biblioteca pascal
2: Receber o modelo inicial m_0
3: Receber o número de iterações da FWI N_fwi
4: Inicializar variáveis de checkpointing
5: Inicialização dos parâmetros da FWI
6: for de k = 0 até N_fwi do
7:     Início da seção paralela OpenMP
8:     for todos os tiros do
9:         Ler o sismograma do tiro
10:        for t_i = 0 até n_s − 1 do
11:            pascal_start(1)
12:            Diretiva de laço paralelo OpenMP
13:            pascal_start(2)
14:            for todos os pontos do domínio do espaço do
15:                Calcular o campo de onda direto para o modelo m_k
16:            end for
17:            pascal_stop(2)
18:            if t_i é um checkpoint then
19:                pascal_start(3)
20:                Salvar checkpoint
21:                pascal_stop(3)
22:            end if
23:            pascal_stop(1)
24:        end for
25:        for t_i = n_s − 1 to 0 do
26:            pascal_start(4)
27:            Laço paralelo OpenMP
28:            for Todos os pontos do domínio do espaço do
29:                Calcular o campo de onda inverso para o modelo m_k
30:            end for
31:            pascal_stop(4)
32:            pascal_start(5)
33:            Re-computar o campo de onda direto em t_i a partir dos checkpoints
34:            pascal_stop(5)
35:            Calcular o gradiente da função objetivo
36:        end for
37:    end for
38:    Calcular a direção de busca
39:    Determinar o tamanho do passo
40:    Atualizar o modelo m_{k+1}
41:    Fim da seção paralela OpenMP
42: end for
```



(a) Execução inteira

(b) Propagação do campo de ondas direto

(c) Propagação do campo de ondas inverso

(d) Salvar checkpoints do campo de ondas

(e) Re-computar o campo de ondas a partir dos checkpoints

Figura 2. Diagrama de eficiência da FWI sem modificações. O número de núcleos varia entre 1, 2, 4, 8, 16 e 32 no eixo vertical. O tamanho do problema varia conforme os modelos de i1 a 14 horizontal. As cores no diagrama são utilizadas para representar a eficiência em porcentagem. Cada célula no diagrama é uma média de 5 amostras.

var *checkpoints* durante a propagação do campo de onda direto. A Figura 2e ilustra a eficiência na recuperação desse campo de onda a partir dos *buffers* de *checkpoint*. Em ambas as figuras é possível observar uma baixa eficiência em todos os pontos, o que nos leva acredita que esses trechos representam gargalos na eficiência do objeto de estudo.

Observamos que esses trechos de código utilizam a função *memcpy* para salvar os pontos na malha do gradiente de propagação direta nos vetores de *buffer* e, posteriormente, para recuperar esses pontos a partir dos *buffers*. No Mamute essas operações são delimitadas pela diretiva *omp single*, o que indica serem executadas sequencialmente. Identificamos esse método como um possível gargalo para a eficiência da FWI, já que a execução sequencial nesse ponto pode limitar o desempenho geral do algoritmo.

4.2. Otimizações Implementadas

Para mitigar esse problema e melhorar a escalabilidade do algoritmo, foi implementada uma distribuição estática das operações de *memcpy*. Com essa abordagem, cada *thread* fica responsável por copiar uma porção específica do *buffer* de *checkpoint*, distribuindo a carga de trabalho de maneira mais equilibrada. Ao permitir que múltiplas *threads* realizem a cópia de dados simultaneamente, aspiramos aproveitar melhor o potencial de paralelismo do sistema, o que deve resultar em um desempenho melhorado para a FWI, especialmente quando executada com mais recursos computacionais.

O Algoritmo 2 ilustra a cópia de uma amostra do campo de ondas e um *buffer* do *checkpointing* durante a propagação direta. A variável *tempo* corresponde ao instante de tempo atual, `checkpoint_atual` é o índice do próximo *checkpoint*, e *tamanho* define a quantidade de dados a serem copiados. O uso de *memcpy* para copiar o campo de ondas para o *buffer* ocorre em uma seção *omp single*, garantindo que a operação seja executada por uma única *thread*.

Algorithm 2 Trecho de cópia do campo de ondas para o buffer do checkpointing.

```
#ifdef _OPENMP
#pragma omp single
#endif
{
if (tempo == checkpoint_atual) {
    memcpy(buffer[checkpoint_atual]->estado_atual, onda->
       estado_atual, tamanho * sizeof(double));
    memcpy(buffer[checkpoint_atual]->nova_posicao, onda->
       nova_posicao, tamanho * sizeof(double));
    checkpoint_atual++;
    }
}
```

O Algoritmo 3 exemplifica a paralelização da etapa de realizar o *checkpoint* utilizando uma distribuição estática. Isso é feito determinando o número total de *threads* (`total_threads`) e o identificador de cada *thread* (`thread_id`) para, em seguida, distribuir o trabalho entre todas as *threads*. A variável `sobra` é utilizada para calcular o

restante da divisão do tamanho total do *buffer* pelo número de *threads*, permitindo que as primeiras *threads* recebam uma porção ligeiramente maior de trabalho, caso o tamanho total não seja divisível igualmente. Dessa forma, o tamanho do pedaço de *buffer* que cada *thread* irá processar (`tamanho_pedaco`) é ajustado para garantir que todas as partes do *buffer* sejam copiadas. Cada *thread* começa sua operação de *memcpy* no ponto apropriado (`inicio`) dentro do *buffer*, evitando sobreposição ou omissão de dados. Por fim, um `omp barrier` é utilizado para sincronizar as *threads*, assegurando que todas tenham completado suas cópias antes que o índice do *checkpoint* (`checkpoint_atual`) seja incrementado em uma seção `omp single`. A mesma abordagem foi aplicada ao método que recupera esses *buffers* para acelerar a propagação do campo de ondas inverso.

Algorithm 3 Trecho de código de processo de checkpoint de forma paralela.

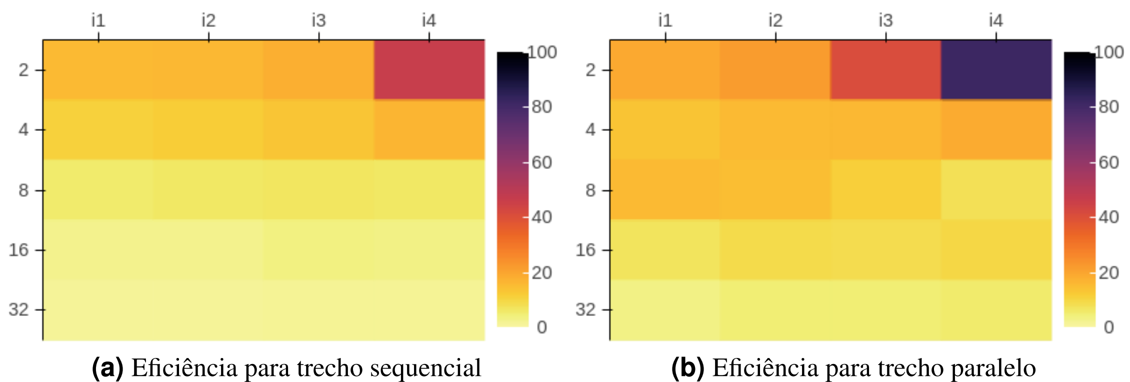
```
if (tempo == checkpoint_atual) {
    int total_threads = omp_get_num_threads();
    int thread_id = omp_get_thread_num();
    int sobra = tamanho % total_threads;
    int tamanho_pedaco = (tamanho / total_threads) + (
        thread_id < sobra);
    int inicio = thread_id * tamanho_pedaco + std::min(
        thread_id, sobra);

    memcpy(buffer[checkpoint_atual]->estado_atual + inicio,
        onda->estado_atual + inicio,
        tamanho_pedaco * sizeof(double));
    memcpy(buffer[checkpoint_atual]->nova_posicao + inicio,
            onda->nova_posicao + inicio,
            tamanho_pedaco * sizeof(double));
#pragma omp barrier
#pragma omp single
{checkpoint_atual++;}
}
```

4.3. Resultados

A Figura 3 compara os diagramas de eficiência das abordagens sequencial e paralela da etapa de salvar *checkpoints* do campo de ondas. Observa-se que, com a abordagem paralela, a eficiência desse trecho aumentou em todos os pontos, com o pico de eficiência subindo de 46% para 81%. Este resultado já era esperado, uma vez que a paralelização distribui a carga de trabalho entre as *threads*, reduzindo o tempo de execução e melhorando o uso dos recursos disponíveis. No entanto, a Figura 3b ainda revela um trecho de código ineficiente, pois a eficiência diminui à medida que o tamanho do problema cresce na mesma proporção da quantidade de recursos. Isso sugere que, apesar da melhora significativa, há ainda uma limitação na escalabilidade da abordagem paralela.

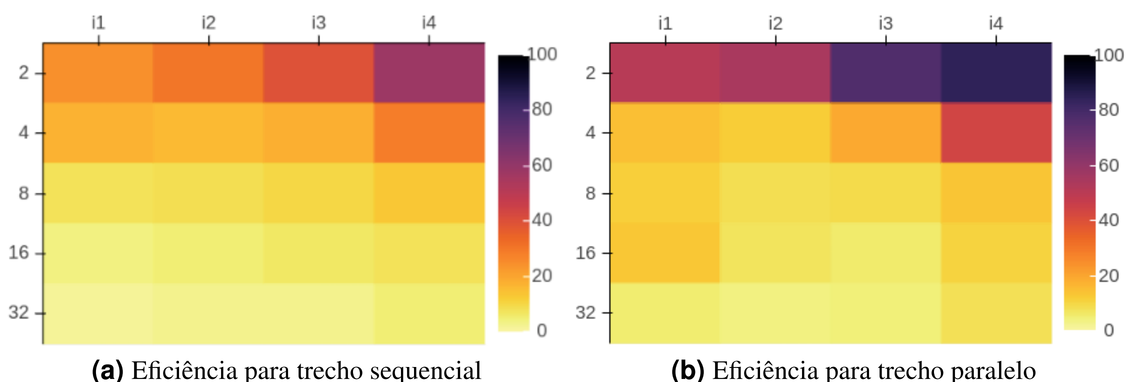
Na Figura 4, podemos ver um resultado parecido com a análise anterior, com a abordagem de distribuição estática, o trecho de recuperação dos buffers de checkpoint teve um aumento significativo de eficiência, com o pico de eficiência saindo de 57% para



(a) Eficiência para trecho sequencial **(b)** Eficiência para trecho paralelo

Figura 3. Comparativo dos diagramas de eficiências ao Salvar checkpoints do campo de ondas com abordagem sequencial e abordagem paralela. O número de núcleos varia de acordo com 1, 2, 4, 8, 16 e 32 no eixo vertical. O tamanho do problema, i1 a 14, varia de acordo com $(n1, n2, n3) = (25, 200, 200)$, $(50, 200, 200)$, $(100, 200, 200)$ e $(200, 200, 200)$ no eixo horizontal. As cores no diagrama são utilizadas para representar a porcentagem de eficiência. Cada célula no diagrama é uma média de 5 amostras.

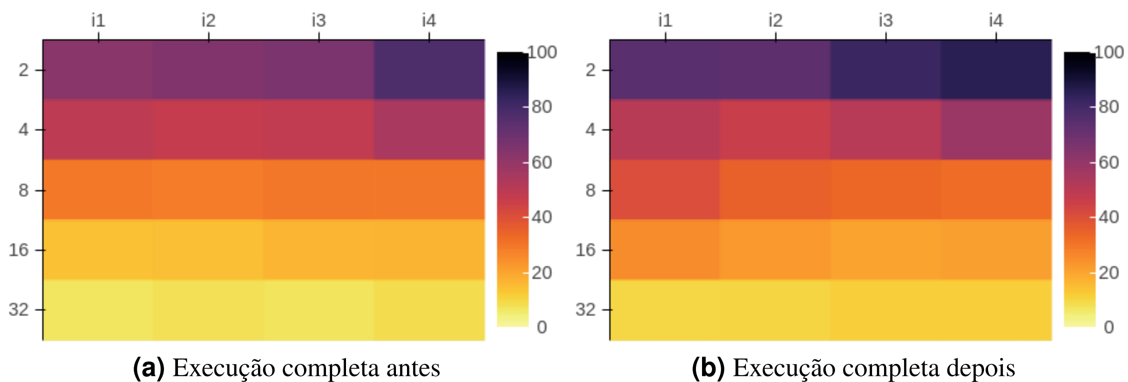
84%, porém a medida em que a quantidade de núcleos e tamanho de problema escalam, essa eficiência cai.



(a) Eficiência para trecho sequencial **(b)** Eficiência para trecho paralelo

Figura 4. Comparativo das eficiências ao Recuperar checkpoints do campo de ondas com abordagem sequencial e abordagem paralela, variando a quantidade de núcleos em 32, 16, 8, 2 e 1 no eixo vertical para 4 tamanhos de entrada, variando os modelos i1, i2, i3 e i4 no eixo horizontal. As cores no diagrama são utilizadas para representar a porcentagem de eficiência. Cada célula no diagrama é uma média de 5 amostras.

A Figura 5 ilustra que, após a implementação da paralelização, a execução total da aplicação apresentou um aumento na eficiência, com seu pico passando de $82,06\%$ para $85,75\%$. Contudo, apesar desses avanços, o software do Mamute ainda revela uma eficiência limitada à medida que tanto o tamanho do problema quanto o número de núcleos aumentam.



(a) Execução completa antes

(b) Execução completa depois

Figura 5. Diagramas de eficiência para a FWI após abordagem paralela no Salvamento e Recuperação de checkpointing. O número de núcleos varia de acordo com 1, 2, 4, 8, 16 e 32 no eixo vertical. O tamanho do problema varia conforme os modelos de i1 a 14 no eixo horizontal. As cores no diagrama são utilizadas para representar a eficiência em porcentagem. Cada célula no diagrama é uma média de 5 amostras.

5. Conclusão

Neste trabalho, analisamos a eficiência em uma versão de código da Inversão do Campo de Onda Completo (FWI). Mediante testes e perfilamentos realizados com o PaScal Suite, identificamos que, embora o código seja altamente paralelizado, alguns trechos críticos, como o processo de salvar e recuperar checkpoints, apresentavam gargalos significativos devido à execução sequencial.

Implementamos uma abordagem paralela para o processo de cópia de memória entre os campos de ondas e os buffers de *checkpointing* utilizando uma distribuição estática das operações de cópia de memória entre as threads. Essa modificação resultou em um aumento substancial na eficiência. O pico de eficiência da etapa de salvar checkpoints subiu de 46% para 81% e de 56% para 84% na etapa de recuperação de checkpoints, refletindo a eficácia da paralelização proposta. Além disso, observamos ganhos de eficiência em outros trechos do código, indicando uma melhoria no tempo total de execução da FWI com o pico de eficiência subindo de $82,06\%$ para $85,75\%$.

Entretanto, apesar das melhorias alcançadas, o código ainda não atingiu um estado de escalabilidade ideal, indicando que ainda existem barreiras a serem superadas.

Esses resultados apontam para direções futuras de trabalho, que incluem a continuidade das otimizações no código, com foco em reduzir as dependências de dados e melhorar a escalabilidade geral. A implementação de novas estratégias de paralelização como a de *loop blocking* e outras formas de escalabilidade, além disso, a investigação de outras possíveis fontes de ineficiência serão fundamentais para avançar na busca por um desempenho mais robusto e escalável na FWI.

Referências

da Silva, A. B. N., Cunha, D. A. M., Silva, V. R. G., de A. Furtunato, A. F., and Xavier-de Souza, S. (2019). Pascal viewer: A tool for the visualization of parallel scalability trends. In Bhatele, A., Boehme, D., Levine, J. A., Malony, A. D., and Schulz, M.,

editors, *Programming and Performance Visualization Tools*, pages 250–264, Cham. Springer International Publishing.

da Silva, F. H., Fernandes, J. B., Sardina, I. M., Barros, T., Xavier-de Souza, S., and Assis, I. A. (2024). Auto tuning for openmp dynamic scheduling applied to fwi. *arXiv preprint arXiv:2402.16728*.

da Silva, V. R. G., da Silva, A. B. N., Valderrama, C., Manneback, P., and Xavier-de Souza, S. (2022). A minimally intrusive approach for automatic assessment of parallel performance scalability of shared-memory hpc applications. *Electronics*, 11(5).

Etienne, V., Tonellot, T., Thierry, P., Berthoumieux, V., and Andreolli, C. (2014). Speeding-up fwi by one order of magnitude. In *EAGE Workshop on High Performance Computing for Upstream*, pages cp–426. European Association of Geoscientists & Engineers.

Griewank, A. and Walther, A. (2000). Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45.

Okita, N., Camargo, A., Ribeiro, J., Benedicto, C., Coimbra, T., Faccipieri, J., and Tygel, M. (2020). Highly scalable full-waveform inversion on the cloud using graphics processing units. *EarthDoc*, 2020(1):1–5.

Plessix, R. E. (2006). A review of the adjoint–state method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International*, 167(2):495–503.

Symes, W. W. (2010). Reverse time migration with optimal checkpointing. *Geophysics*, 75(5):S49–S60.

Tarantola, A. (1984). Inversion of seismic reflection data in the acoustic approximation. *Geophysics*, 49(8):1259–1266.

Zeng, Q., Feng, S., Wohlberg, B., and Lin, Y. (2022). Inversionnet3d: Efficient and scalable learning for 3-d full-waveform inversion. *IEEE Transactions on Geoscience and Remote Sensing*, 60:1–16.