UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
CENTRO MULTIDISCIPLINAR DE PAU DOS FERROS
DEPARTAMENTO DE CIÊNCIAS EXATAS E NATURAIS
CURSO INTERDISCIPLINAR EM TECNOLOGIA DA INFORMAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO (2025)

AJUSTE AUTOMÁTICO DO ESCALONAMENTO DINÂMICO DO OPENMP APLICADO A FWI

AUTO-TUNING FOR OPENMP DYNAMIC SCHEDULING APPLIED TO FWI

Felipe Hidequel Santos da Silva[1]
João Batista Fernandes[2]
Idalmis M. Sardina[3]
Tiago Barros[4]
Samuel Xavier de Souza[5]
Italo Augusto Souza de Assis[6]

RESUMO

A Inversão da Forma de Onda Completa (FWI, do inglês *Full Waveform Inversion*) é um método de imageamento sísmico amplamente utilizado, capaz de estimar modelos do subsolo a partir de dados sísmicos. Devido à sua elevada demanda computacional, a FWI normalmente requer sistemas de grande porte, como supercomputadores. Entretanto, seu paralelismo inerente possibilita a utilização de sistemas de memória compartilhada com OpenMP. A distribuição de tarefas no OpenMP depende dos escalonadores de laços, sendo o escalonador dinâmico particularmente eficaz em cargas de trabalho irregulares, como na FWI, pois atribui blocos de iterações a núcleos disponíveis em tempo de execução. Contudo, o impacto do tamanho desses blocos no desempenho ainda não é bem compreendido. Para lidar com esse problema, propomos o *framework* Parameter Auto-Tuning for Shared Memory Algorithms (PATSMA), que emprega o método de otimização Coupled Simulated Annealing (CSA) para determinar automaticamente o tamanho ótimo dos blocos na propagação de ondas, uma das etapas mais custosas da FWI. Em vez de testar exaustivamente todos os valores possíveis em uma execução completa da FWI, o PATSMA avalia candidatos medindo o tempo de execução da primeira iteração temporal do primeiro tiro sísmico na primeira iteração da FWI. O tamanho de bloco selecionado é então aplicado a todas as propagações subsequentes. Experimentos realizados com diferentes tamanhos de problema e em ambientes computacionais diversos, incluindo supercomputadores e instâncias em nuvem, demonstram que a auto-otimização proporciona ganhos significativos de desempenho: o tempo de execução foi reduzido em até 70,46% em comparação com os escalonadores padrão do OpenMP.

Palavras-chave: Auto-ajuste; OpenMP; Sistemas de memória compartilhada; Escalonamento dinâmico; Coupled Simulated Annealing.

[1]Bacharelando no curso Interdisciplinar em Tecnologia da Informação (UFERSA)
[2]Mestre em Engenharia de Computação (UFRN)
[3]Doutor em Engenharia de Computação (UFRN)
[4]Doutor em Engenharia Elétrica (UNICAMP)
[5]Doutor em Engenharia de Computação (UFRN)
[6]Doutor em Engenharia de Computação (UFRN)

ABSTRACT

Full Waveform Inversion (FWI) is a widely used seismic imaging method that estimates subsurface models from seismic data. Due to its massive computational demand, FWI typically requires large-scale computer systems such as supercomputers. However, its inherent parallelism allows the use of shared memory systems with OpenMP. Task distribution in OpenMP relies on loop schedulers, and the dynamic scheduler is particularly effective in handling irregular workloads, such as FWI. It assigns fixed-size chunks to idle processing cores at runtime. Yet, the impact of chunk size on performance remains unclear. To address this, we propose the Parameter Auto-Tuning for Shared Memory Algorithms (PATSMA) framework, which employs Coupled Simulated Annealing (CSA) to automatically determine the optimal chunk size for wave propagation, one of the most computationally intensive steps in FWI. Instead of exhaustively testing all chunk sizes within a complete FWI execution, which is impractical, our strategy evaluates candidates by measuring the runtime of the first time iteration of the first seismic shot in the first FWI iteration. The selected chunk size is then applied to all subsequent wave propagations. We conducted experiments with different problem sizes across diverse computational environments, including supercomputers and cloud instances. Results demonstrate that auto-tuning significantly improves performance, with runtimes reduced by up to 70.46% compared to standard OpenMP schedulers.

Key words: Auto-tuning; OpenMP; Shared memory systems; Dynamic scheduling; Coupled Simulated Annealing.

Disponível em: <https://doi.org/10.1016/j.cageo.2025.105932.>

1 INTRODUÇÃO

O levantamento sísmico é uma técnica que envolve a utilização de disparos sísmicos para emitir ondas e de sensores para capturar os dados das ondas que retornam. Esses dados são então processados para obter informações sobre a subsuperfície. A modelagem sísmica é uma parte do processo de levantamento que pode ser utilizada para simular a propagação de ondas e as propriedades da subsuperfície. Embora seja possível modelar diversos parâmetros para aumentar a precisão, isso pode resultar em cálculos matemáticos mais complexos que exigem processamento computacional intensivo e tempo. O resultado desse processo é o que se conhece como dado calculado.

A Inversão da Forma de Onda Completa (FWI, do inglês *Full Waveform Inversion*) (Tarantola, 1984) é um dos métodos mais utilizados para estimar um modelo de velocidade. O FWI ajusta iterativamente o modelo de velocidade usando uma estratégia de otimização. Seu objetivo é reduzir o dado residual, ou seja, a diferença entre os dados observados e os calculados. O FWI computa ajustes nos modelos de velocidade a partir dos dados estimados e residuais.

O FWI apresenta desafios, como a necessidade de dados de alta qualidade e a complexidade computacional do processo. Devido à sua alta demanda computacional, a execução deste algoritmo na indústria é restrita a grandes ambientes de computação, onde o uso de técnicas de programação paralela é necessário.

O OpenMP (Open Multi-Processing) (Dagum; Menon, 1998), uma biblioteca para programação paralela em sistemas de memória compartilhada, é uma ferramenta amplamente utilizada na paralelização de aplicações de grande escala, como o FWI. Ele permite que programadores escrevam código paralelo para distribuir o trabalho de uma aplicação entre múltiplas threads de

execução. Essas threads são então atribuídas aos núcleos de um sistema de memória compartilhada. Por padrão, no OpenMP, essa carga é dividida igualmente. No entanto, em problemas complexos como o FWI, pode ocorrer desbalanceamento de carga, pois algumas partes do código executam tarefas desiguais, fazendo com que algumas threads completem seu trabalho mais rapidamente que outras. As threads mais rápidas ficarão ociosas enquanto esperam as outras terminarem. O desbalanceamento de carga pode resultar na subutilização do hardware disponível e, consequentemente, na redução do desempenho geral do programa.

Neste trabalho, propomos um método para balancear a carga de trabalho entre as CPUs em um nó computacional, evitando a subutilização de recursos. Para isso, empregamos o Parameter Auto-Tuning for Shared Memory Algorithms (PATSMA) (Fernandes et al., 2024) para ajustar automaticamente o tamanho do chunk do escalonador *dynamic* do OpenMP. A principal contribuição deste trabalho é adaptar o PATSMA para o FWI e avaliar seu desempenho. Comparamos nossa proposta com os escalonadores padrão do OpenMP, mostrando que o auto-ajuste proposto reduz o tempo de execução do FWI em diferentes ambientes computacionais e testes para diversos tamanhos de entrada.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta as características da nossa aplicação-alvo, o FWI, e o método de auto-ajuste que utilizamos, o PATSMA. Ela também discute a distribuição de carga de trabalho através do escalonamento de laços suportado pelo OpenMP. Em seguida, na Seção 3, detalhamos nossa aplicação FWI e a abordagem de auto-ajuste proposta. Apresentamos os resultados do método proposto em comparação com os escalonadores padrão do OpenMP na Seção 4. A Seção 5 apresenta os trabalhos relacionados, e a Seção 6 conclui este trabalho.

2 REFERENCIAL TEÓRICO

Esta seção descreve brevemente nossa aplicação alvo, o FWI (Seção 2.1), apresenta os escalonadores de laço padrão do OpenMP que comparamos com nossa abordagem (Seção 2.2), e apresenta o método de auto-ajuste que utilizamos, o PATSMA (Seção 2.3).

2.1 FWI

A Inversão da Forma de Onda Completa (FWI, do inglês *Full Waveform Inversion*) (Tarantola, 1984) é uma técnica de inversão de dados sísmicos que visa obter um modelo preciso das propriedades geofísicas da subsuperfície. Essa técnica minimiza uma função objetivo que mede a diferença entre os dados observados e os calculados, ou seja, o dado residual. O gradiente da função objetivo indica a direção de busca para o problema de otimização do FWI (Plessix, 2006). O modelo é atualizado com base nesse gradiente. Matematicamente, a FWI pode ser descrita como um problema de otimização de mínimos quadrados cuja função objetivo é

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

onde $\|.\|$ denota a norma euclidiana; m é o vetor que contém todos os parâmetros do modelo; d são os dados observados, e G é o operador de modelagem sísmica responsável por simular a aquisição sísmica em uma área representada por m.

Este processo é repetido até que um critério de parada seja alcançado. Condições de parada comuns são a convergência do método de otimização (a norma do gradiente é menor que um limiar pré-definido) ou o número máximo de iterações. Os principais passos do FWI são:

1. Ler os dados observados;

2. Ler um modelo inicial;
3. Calcular a solução da equação da onda (dado calculado) para o modelo inicial;
4. Calcular a função objetivo (Equação 1);
5. Calcular o gradiente da função objetivo;
6. Atualizar o modelo com base no gradiente da função objetivo;
7. Repetir os passos 3 a 6 até que o critério de parada seja alcançado.

2.2 Escalonadores OpenMP

O OpenMP (Open Multi-Processing) é uma Interface de Programação de Aplicações (API) para programação paralela em sistemas multicore com memória compartilhada (Board, 2021). Ele fornece ferramentas para que programadores desenvolvam códigos paralelos em C, C++ e Fortran sem alterações significativas no código original, utilizando suas diretivas para distribuir o trabalho da aplicação entre múltiplas *threads*.

#pragma omp for é uma diretiva do OpenMP amplamente utilizada. Essa diretiva distribui as iterações de um laço (*loop*) entre as *threads* que as executam simultaneamente em diferentes núcleos, permitindo que o programa execute esses laços mais rapidamente em sistemas de memória compartilhada. A cláusula *schedule* é uma das mais importantes da diretiva *omp for*. Uma cláusula é uma instrução para alterar o comportamento padrão de uma diretiva. A cláusula *schedule* especifica como as iterações do laço devem ser distribuídas entre as *threads*. Ela fornece três escalonadores padrão: *static*, *dynamic* e *guided*. A operação de cada escalonador padrão é definida a seguir.

- *static*: Blocos (*chunks*) de iterações são distribuídos em modo *round-robin* entre as threads antes da execução do laço. O programador pode definir o tamanho dos *chunks*. Se o tamanho do *chunk* não for definido, seu valor é aproximadamente N_i/N_t, onde N_i é o número de iterações e N_t é o número de *threads*.
- *dynamic*: Durante o tempo de execução, *chunks* de iterações do laço são distribuídos entre as *threads*. Quando uma thread termina de executar seu *chunk*, ela solicita um novo *chunk* de uma fila global. Se o programador não especificar o tamanho do *chunk*, seu valor é 1.
- *guided*: Semelhante ao *dynamic*, os *chunks* de iterações são distribuídos em tempo de execução. No entanto, o tamanho do *chunk* diminui a cada nova solicitação. Essa estratégia visa lidar melhor com o desbalanceamento de carga. Quando o programador especifica o tamanho do *chunk*, nenhum *chunk* pode ser menor que esse valor, exceto o último. Se o tamanho do *chunk* não for definido, seu valor é 1.

O escalonador *static* pode ser vantajoso porque a distribuição estática garante que cada *thread* executará aproximadamente o mesmo número de iterações. Como não há sobrecarga (overhead) de redistribuição de trabalho durante a execução, essa estratégia pode levar a um melhor desempenho para hardware e cargas de trabalho de iteração homogêneas. No entanto, se a carga de trabalho das iterações ou o hardware for heterogêneo, a distribuição estática pode resultar na subutilização de recursos, pois algumas *threads* podem terminar antes de outras.

Aplicações com cargas de trabalho de iteração heterogêneas ou executando em hardware heterogêneo podem ter um desempenho melhor utilizando o escalonador *dynamic*. Como tal

escalonador distribui os *chunks* de iterações em tempo de execução, nenhuma *thread* fica ociosa até que todos os *chunks* sejam distribuídos. Contudo, o programa pode perder desempenho se o tamanho do *chunk* não for escolhido apropriadamente. Se o *chunk* for muito grande, ele pode gerar tempo ocioso. Por outro lado, se o *chunk* for muito pequeno, pode haver um número excessivo de solicitações de *chunks*, levando a um aumento significativo no tempo de execução.

O escalonador *guided* visa reduzir a ociosidade em comparação com o escalonador *static* e a sobrecarga de gerenciamento em comparação com o escalonador *dynamic*. Como os últimos *chunks* são menores, as *threads* não ficarão ociosas por muito tempo. Além disso, como os primeiros *chunks* são maiores, as *threads* ficarão ocupadas por mais tempo, reduzindo o número de novas solicitações de *chunks*. As desvantagens do escalonador *guided* são que (1) há uma sobrecarga adicional para gerenciar o tamanho do *chunk*, e (2) um método de auto-ajuste só poderia controlar o tamanho do último bloco.

Portanto, encontrar um tamanho de *chunk* ideal é essencial para alcançar o melhor desempenho possível no escalonamento de laços de uma aplicação usando OpenMP. Dessa forma, é fundamental ajustar cuidadosamente o tamanho do *chunk* para cada aplicação, dependendo do número de iterações do laço, das características da carga de trabalho e do ambiente computacional.

Assis et al. (2020) avaliaram o desempenho dos escalonadores *dynamic* e *static* usando uma variedade de simulações em um algoritmo de migração em tempo reverso (RTM) (Baysal; Kosloff; Sherwood, 1983). Os resultados mostraram que o escalonador *dynamic* com auto-ajuste do tamanho do *chunk* superou consistentemente os escalonadores padrão *static* e *guided*, com melhorias de até 33% em alguns casos. Portanto, utilizamos o escalonador *dynamic* neste trabalho.

2.3 PATSMA

O Parameter Auto-Tuning for Shared Memory Algorithms (PATSMA) é um software que visa ajustar um parâmetro de uma aplicação de memória compartilhada usando um método de otimização. O programador da aplicação pode definir tanto o parâmetro quanto o método de otimização. Embora o PATSMA seja adequado para diferentes métodos de otimização, utilizamos o Coupled Simulated Annealing (CSA) (Souza et al., 2010), um dos otimizadores nativos do PATSMA.

O CSA é uma extensão do algoritmo Simulated Annealing (SA) (Kirkpatrick; Gelatt; Vecchi, 1983) que utiliza múltiplas instâncias de SA para explorar o espaço de soluções simultaneamente. O CSA aumenta a diversidade da busca ao usar vários processos de SA (otimizadores) em diferentes subconjuntos do espaço de busca. A troca de informações entre os otimizadores ajuda a diversificar a busca, permitindo que o CSA encontre soluções globais de forma mais eficiente.

O algoritmo CSA segue os mesmos passos de geração e aceitação do SA. Ambos os algoritmos usam temperaturas para controlar seu processo. No entanto, o CSA possui um processo automático que controla e diversifica a busca ajustando a temperatura de aceitação de cada otimizador. Isso permite que alguns otimizadores foquem na busca local, enquanto outros, na busca global.

O CSA explora o espaço de soluções amplamente com diferentes pontos de partida. Ele gera uma nova solução a cada iteração, que pode ser aceita mesmo que represente uma piora. É eficaz para otimização global, mas o ajuste adequado dos parâmetros é crucial. O CSA é menos sensível à escolha de parâmetros do que o SA e produz soluções de melhor qualidade mesmo com um ajuste subótimo.

O auto-ajuste do PATSMA é um processo que envolve laços iterativos onde um parâmetro afeta o tempo de execução de uma seção de código específica dentro do laço. Para realizar o auto-ajuste, a aplicação-alvo precisa definir a seção de código onde o PATSMA atuará. No entanto, às vezes o progresso do laço pode influenciar negativamente o auto-ajuste, pois o comportamento da seção pode mudar durante sua execução. Para mitigar esse problema, uma cópia desta seção pode ser criada antes do laço onde executaremos o auto-ajuste.

Uma vez que a seção é definida, o PATSMA utiliza o CSA para gerar soluções prováveis para o uso desta seção e mede o tempo de execução resultante. Após a conclusão de todas as iterações do PATSMA, a solução final pode ser enviada para o executor do código da seção original para sua execução.

3 AUTO-AJUSTE APLICADO AO FWI

A execução da FWI pode empregar um algoritmo para resolver a equação da onda 3D através de diferenças finitas. O algoritmo envolve laços aninhados que representam o domínio espacial e é executado repetidamente durante o FWI. Esses laços estão presentes nas etapas de propagação direta, propagação reversa e no cálculo do gradiente. Eles são frequentemente paralelizados usando a API OpenMP e um escalonador de laços do OpenMP. Na Seção 2.2, demonstramos a necessidade de definir o parâmetro de tamanho do *chunk* para melhorar o desempenho do escalonador de laços. Em alguns casos, a escolha de um *chunk* pode depender de fatores como o tamanho da memória, o número de núcleos e o tamanho da entrada.

A FWI é um algoritmo complexo que envolve uma quantidade variável de cálculos, dependendo da região do espaço onde os cálculos são realizados. Esse comportamento afeta a definição do tamanho do *chunk*, o que pode influenciar o tempo de execução do FWI. Para minimizar o tempo de execução de um FWI, propomos o uso do PATSMA com o CSA para ajustar o tamanho do *chunk* para o escalonamento dinâmico do OpenMP em laços paralelos no FWI. Utilizamos o tempo de execução para resolver a equação da onda 3D como a função de custo do PATSMA e o tamanho do *chunk* para o escalonamento dinâmico do OpenMP como seu parâmetro.

Nossa implementação do FWI é baseada no trabalho de (Fernandes et al., 2025). Esta versão do FWI emprega *checkpointing* ótimo (Symes, 2007) utilizando a biblioteca proposta por (Griewank; Walther, 2000). O *checkpointing* ótimo opera armazenando apenas um subconjunto dos estados intermediários da simulação durante a etapa de propagação direta. Durante a etapa reversa, os estados que não foram armazenados são recalculados a partir dos *checkpoints* disponíveis. Este método reduz significativamente os requisitos de memória, com um aumento logarítmico no custo computacional em relação ao número total de passos. Adicionalmente, como o *checkpointing* ótimo também calcula a solução da equação da onda, ajustamos dinamicamente o tamanho do chunk para escalonar os laços de *checkpointing* de forma eficiente.

O Algoritmo 1 ilustra um FWI com nossa estratégia proposta. Propomos a execução do PATSMA apenas no primeiro passo de tempo ($t_i == 0$) do primeiro disparo do FWI (Linha 10). A função $autotuning()$ do PATSMA (Linha 11) otimiza o tamanho do *chunk* para os laços do propagador de ondas. O tamanho de *chunk* resultante é utilizado na propagação direta (Linha 14), na propagação reversa (Linha 23) e no recálculo do campo direto via *checkpointing* (Linha 27). Escolhemos esses três conjuntos de laços porque eles representam as etapas computacionalmente mais custosas de um FWI e implementam a solução da equação da onda, permitindo a reutilização do tamanho de *chunk* obtido pelo PATSMA.

A solução inicial (tamanho do *chunk*) de cada otimizador CSA é escolhida aleatoriamente no intervalo [50, N_i/N_t]. Desconsideramos tamanhos de *chunk* inferiores a 50 devido à alta

Algorithm 1 Inversão da Forma de Onda Completa com o auto-ajuste proposto

Iniciar medição de tempo
Ler o modelo inicial m_0
Ler o número de iterações do FWI (N_{fwi})
Ler o número de iterações de tempo (N_{s})
Inicializar outros parâmetros para checkpointing, FWI e PATSMA
for k = 0 até N_{fwi} **do**
    # Início da seção paralela OpenMP
    for todos os disparos **do**
        Ler os dados observados do disparo
        if é o primeiro disparo e k == 0 **then**
            tamanho do chunk = $autotuning()$
        end if
        for cada iteração de tempo $t_i = 0 \cdots N_{\text{s}} - 1$ **do**
            #Diretiva de laço paralelo OpenMP usando escalonador dinâmico com o tamanho de chunk ajustado
            for todos os pontos no domínio espacial **do**
                Calcular o campo de onda direto para o modelo m_k
            end for
            if t_i é um checkpoint **then**
                Salvar checkpoint
            end if
        end for
        for cada iteração de tempo $t_i = N_{\text{s}} - 1 \cdots 0$ **do**
            #Diretiva de laço paralelo OpenMP usando escalonador dinâmico com o tamanho de chunk ajustado
            for todos os pontos no domínio espacial **do**
                Calcular o campo de onda adjunto para o modelo m_k
            end for
            Recalcular o campo de onda direto em t_i a partir dos checkpoints usando escalonamento dinâmico com o tamanho de chunk ajustado
            #Laço paralelo OpenMP usando escalonador estático
            Calcular o gradiente da função objetivo
        end for
    end for
    Calcular a direção de busca
    Determinar o tamanho do passo
    Atualizar o modelo m_{k+1}
    # Fim da seção paralela OpenMP
end for
Medição de tempo final

Fonte: Autoria própria (2025).

sobrecarga (*overhead*) gerada para distribuí-los dinamicamente, como demonstrado por (Fernandes et al., 2018; Barros et al., 2018). Tamanhos de *chunk* maiores que o tamanho do *chunk* da distribuição *static* padrão (N_i/N_t) também não são considerados, pois fariam com que o número de *chunks* fosse menor ou igual ao número de *threads*, forçando assim uma distribuição estática.

Para cada iteração do CSA, cada otimizador mede apenas o tempo de execução do primeiro passo de tempo da propagação direta, utilizando seu tamanho de *chunk* atual. De acordo com (Assis et al., 2020), o tempo de execução do primeiro passo de tempo pode ser representativo do tempo de execução total da propagação. Este primeiro passo é executado duas vezes, e apenas o tempo decorrido da segunda repetição é registrado para evitar efeitos de preenchimento de cache. O CSA então utiliza essas medições de tempo como os valores da função de custo e gera o próximo conjunto de soluções.

4 RESULTADOS

Esta seção detalha os testes que executamos para medir o desempenho da estratégia proposta em comparação com o escalonamento de laços padrão do OpenMP. As Subseções 4.1 e 4.2 apresentam, respectivamente, os ambientes computacionais e os valores dos parâmetros que utilizamos em nossos testes. Na Subseção 4.3, descrevemos os testes que realizamos e apresentamos e discutimos seus resultados.

4.1 Ambientes Computacionais

Utilizamos diferentes ambientes computacionais para os testes de desempenho do autoajuste proposto, conforme descrito abaixo:

- **OPT3** - VM.Optimized3.Flex é uma máquina virtual da Oracle com um processador Intel Xeon 6354 de 18 núcleos, frequência base de 3.0 GHz, frequência turbo máxima de 3.6 GHz, 256 GB de RAM e 39 MB de cache L3.

- **SD** - representa um nó do supercomputador SDumont, localizado no Laboratório Nacional de Computação Científica (LNCC). Este nó contém 2 CPUs Intel Xeon E5-2695v2 Ivy Bridge (12c @2.4GHz), totalizando 24 núcleos, 64 GB de RAM e 30 MB de cache L3.

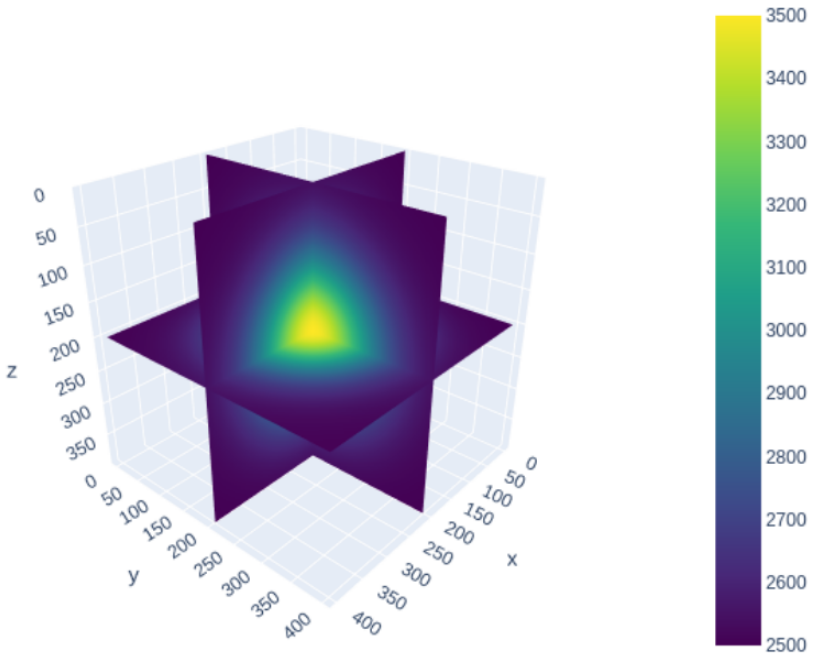
- **NPAD** - NPAD representa um nó do supercomputador NPAD, localizado na Universidade Federal do Rio Grande do Norte (UFRN). Este nó contém 2 CPUs Intel Xeon CPU E5-2683 v4 @ 2.10GHz, totalizando 32 núcleos, 512 GB de RAM e 40 MB de cache L3.

- **STD3** - VM.Standard3.Flex é uma máquina virtual da Oracle com um processador Intel Xeon Platinum 8358 de 32 núcleos, frequência base de 2.6 GHz, frequência turbo máxima de 3.4 GHz, 512 GB de RAM e 48 MB de cache L3.

- **STDE4** - VM.Standard.E4.Flex é uma máquina virtual da Oracle Cloud com um processador AMD EPYC 7J13 de 64 núcleos, frequência base de 2.55 GHz, frequência de *boost* máxima de 3.5 GHz, 1024 GB de RAM e 256 MB de cache L3.

- **DENSE** - BM.DenseIO.E4.128 é uma máquina virtual da Oracle Cloud com um processador AMD EPYC 7J13 de 128 núcleos, frequência base de 2.55 GHz, frequência de *boost* máxima de 3.5 GHz, 2048 GB de RAM e 256 MB de cache L3.

Figura 1: Modelo de velocidade com dimensões $n1 = n2 = n3 = 400$.



Fonte: Autoria própria (2025).

4.2 Parâmetros dos Testes

Em termos da parametrização do FWI, para todos os testes realizados aqui, utilizamos uma frequência de pico (f_{peak}) de $10Hz$, um intervalo de amostragem no tempo de $1ms$, um total de 2458 passos de tempo, uma resolução espacial de $\Delta x1 = \Delta x2 = \Delta x3 = 10m$ e uma espessura de borda absorvente de 25 pontos de grade em todas as direções da malha 3D.

Utilizamos três modelos de velocidade verdadeiros com dimensões $(n_1, n_2, n_3) = (100, 400, 400)$, $(200, 400, 400)$ e $(400, 400, 400)$, onde n_1, n_2 e n_3 representam o número de amostras ao longo dos eixos espaciais x_1, x_2 e x_3, respectivamente. O modelo com $n_1 = 400$, mostrado na Fig. 1, serviu como o modelo base. Ele foi construído usando uma perturbação Gaussiana esférica embutida em um fundo homogêneo. A velocidade de fundo é de 2500, m/s, e a perturbação atinge uma velocidade máxima de 3500, m/s no centro da esfera. Os dois modelos menores foram gerados extraindo subvolumes deste modelo completo, pegando apenas as primeiras n_1 fatias ao longo do eixo x_1.

Esses modelos verdadeiros foram então usados para gerar dados sísmicos sintéticos, que serviram como entrada observada para a Inversão da Forma de Onda Completa (FWI). Os modelos iniciais usados na inversão eram homogêneos com uma velocidade constante de 2500, m/s, sem quaisquer perturbações internas.

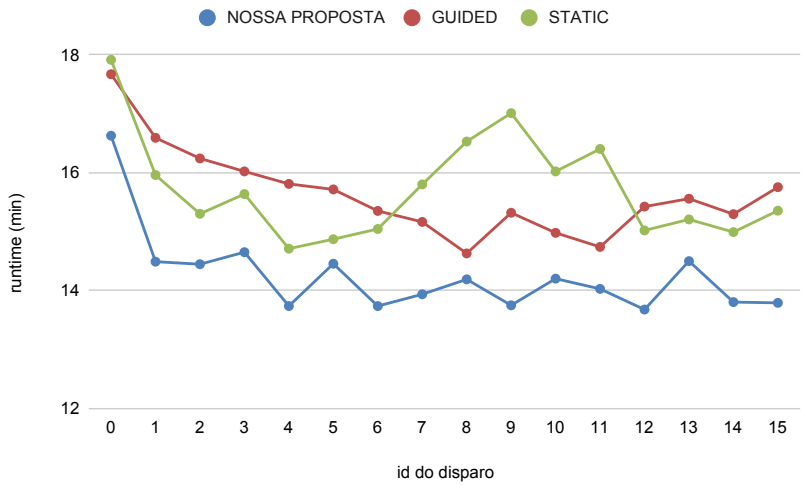
A Tabela 1 apresenta os parâmetros utilizados em nossos experimentos. A seleção do número de otimizadores do CSA, m, é analisada extensivamente em (Silva; Aloise; Souza, 2018), onde é mostrado que os valores ótimos geralmente variam entre $m = 4$ e $m = 10$ para minimizar diversas funções. Neste trabalho, definimos $m = 4$ para reduzir a sobrecarga computacional (*overhead*) do PATSMA, mantendo o desempenho do CSA. Para a temperatura de aceitação inicial, adotamos um dos valores sugeridos em (Souza et al., 2010), $T_0^{ac} = 0.9$. Além disso, (Assis et al., 2020) relata experimentos que exploraram diferentes valores para o número de iterações, N, e a geração inicial, T_0^{ac}. Com base em suas descobertas, selecionamos $N = 40$ e $T_0^{ac} = 100$, pois esses parâmetros resultaram nos menores tempos de execução em todos os

Tabela 1: Parâmetros do CSA, onde T_0^{gen} e T_0^{ac} são as temperaturas iniciais de geração e aceitação, respectivamente, N é o número total de iterações, e m é o número de otimizadores.

T_0^{gen}	T_0^{ac}	N	m
100	0.9	40	4

Fonte: Autoria própria (2025).

Figura 2: Tempo de execução por disparo para o FWI usando o auto-ajuste proposto em comparação com os escalonadores *static* e *guided* do OpenMP na máquina NPAD, com 16 disparos. Para os escalonadores OpenMP, o tamanho do *chunk* não foi definido explicitamente. O tamanho do modelo de velocidade para estes testes foi $(n1, n2, n3) = (200, 400, 400)$.



Fonte: Autoria própria (2025).

casos de teste.

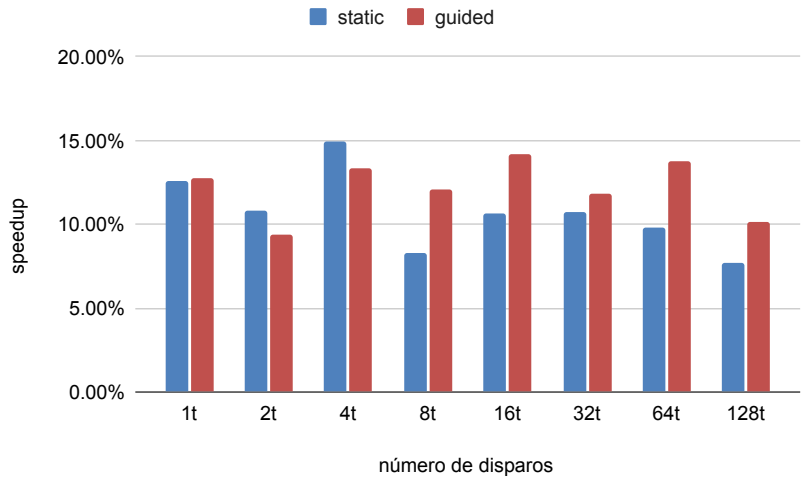
4.3 Análise de Desempenho

O primeiro conjunto de testes visa verificar se é possível reutilizar o tamanho do *chunk* ajustado para o primeiro disparo para disparos subsequentes. O tamanho do modelo escolhido foi $(n1, n2, n3) = (200, 400, 400)$. Inicialmente, medimos o tempo por disparo para execuções com 16 disparos. Os testes foram conduzidos na máquina NPAD.

A Fig. 2 mostra que nossa proposta alcança tempos de execução melhores em comparação com os escalonadores padrão *Static* e *Guided*. O primeiro disparo apresenta uma sobrecarga (*overhead*) inicial devido ao cálculo do tamanho do *chunk*; no entanto, o desempenho se estabiliza nas execuções subsequentes. O escalonador estático exibe maior variabilidade no tempo de execução entre diferentes disparos, pois é mais sensível a mudanças na distribuição computacional. Em contrapartida, o escalonador *Guided* também sofre variações, mas se beneficia de um certo grau de adaptação, mitigando flutuações extremas. Por outro lado, o escalonador dinâmico com PATSMA demonstra maior adaptabilidade, mantendo tempos de execução mais estáveis apesar das mudanças na distribuição dos disparos.

Em outro conjunto de experimentos com o objetivo de verificar o ganho no tempo de execução total por disparo, examinamos se o tamanho do *chunk* ajustado para o primeiro disparo poderia ser reutilizado em execuções subsequentes. O tamanho do modelo escolhido foi

Figura 3: *Speedup* para o FWI usando o auto-ajuste proposto em comparação com os escalonadores *static* e *guided* do OpenMP na máquina NPAD, com 1, 2, 4, 8, 16, 32, 64 e 128 disparos. Para os escalonadores OpenMP, o tamanho do *chunk* não foi definido explicitamente. O tamanho do modelo de velocidade para estes testes foi $(n1, n2, n3) = (200, 400, 400)$. Cada ponto é uma mediana de cinco execuções.



Fonte: Autoria própria (2025).

$(n1, n2, n3) = (200, 400, 400)$, e utilizamos 1, 2, 4, 8, 16, 32, 64 e 128 disparos. Estes testes foram realizados na máquina NPAD, com cada valor representando uma mediana de 5 pontos.

A Fig. 3 mostra que, em todos os casos, a estratégia proposta demonstrou melhor desempenho, com *speedups* de até 14.1% e 14.96% quando comparada aos escalonadores padrão *static* e *guided* do OpenMP, respectivamente. À medida que aumentamos o número de disparos, observamos que o desempenho da estratégia de auto-ajuste permaneceu estável, pois sua sobrecarga (*overhead*) não aumentou e permaneceu semelhante à do primeiro disparo. Neste conjunto de testes, a sobrecarga do método proposto permaneceu abaixo de 1.2% em todos os casos. Realizamos testes com diferentes números de disparos e observamos que o PATSMA superou consistentemente os escalonadores *Static* e *Guided*. Como mostrado na Fig. 2, o tempo de execução para o escalonador estático variou significativamente dependendo do disparo, provavelmente devido à influência das posições dos disparos na grade do modelo 3D, o que afeta a distribuição computacional. O escalonador *Guided* exibiu algum nível de adaptação, reduzindo flutuações extremas, mas ainda mostrando variações. Em contrapartida, o PATSMA manteve um desempenho mais estável em todos os casos testados, reforçando sua eficácia em lidar com as variações na distribuição dos disparos.

Em outro conjunto de experimentos, medimos o desempenho da estratégia proposta à medida que aumentamos o tamanho do problema e os recursos computacionais (número de núcleos e quantidade de cache L3). A Fig. 4 ilustra o desempenho do auto-ajuste proposto em comparação com os escalonadores padrão do OpenMP, *static* e *guided*, no conjunto de seis ambientes computacionais mencionados na Subseção 4.1, variando $n1 = \{100, 200, 400\}$. Para os escalonadores padrão do OpenMP, o tamanho do *chunk* não foi definido explicitamente. Cada ponto no gráfico é uma mediana de cinco amostras. A Fig. 4 mostra que o método proposto teve um desempenho superior em todos os cenários. Mostra também que, na maioria dos casos, quanto mais núcleos uma máquina possui, maior o *speedup* obtido. A razão é que, quanto maior o número de núcleos, menor o espaço de busca, implicando em menos soluções possíveis (ver

Seção 3). Dessa forma, o CSA tem maior probabilidade de encontrar uma solução próxima da ótima.

Como podemos ver, os *speedups* em relação ao escalonador *static* (Fig. 4a) são maiores do que os relacionados ao escalonador *guided* (Fig. 4b). Um desempenho ruim do escalonador *static* para problemas grandes pode explicar esse fenômeno. Quanto maior o tamanho do problema para o escalonador *static* padrão, maior o seu tamanho de *chunk* (ver Seção 2.2). Isso significa menor localidade de dados e, consequentemente, menor reuso de cache. Em comparação com o escalonador *static*, o FWI usando a estratégia proposta obteve um *speedup* de até 70.46%.

Além disso, o campo de onda é inicialmente zero para cada uma das propagações de onda do FWI. À medida que a simulação de propagação de onda avança, apenas um subconjunto do domínio possui valores de pressão de onda não nulos. Como usamos *flags* de otimização de compilação, o processador pode executar operações com valores nulos mais rapidamente. Ao usar o escalonador *static*, a cada *thread* é atribuído um bloco fixo de iterações para processar, independentemente do tempo que leva para executar cada iteração. Como a carga dos blocos pode variar, o tempo de processamento de cada bloco pode ser diferente. Se uma *thread* for mais rápida que as outras, ela pode ficar ociosa enquanto espera as outras *threads* terminarem seu trabalho. Portanto, o desempenho do escalonador *static* é dominado pela *thread* mais lenta.

Por outro lado, o escalonador *guided* reduz o tempo ocioso das *threads*. Os *chunks* são distribuídos dinamicamente e, ao final da execução, os tamanhos dos *chunks* são menores. Dessa forma, o tempo máximo de ociosidade que uma *thread* pode ter é o tempo de processamento do último *chunk*. O escalonador *guided* pode ser mais eficiente que o escalonador *dynamic* em casos onde o tamanho ideal do *chunk* é desconhecido. No entanto, o comportamento da propagação de onda pode afetar seu desempenho. Caso o primeiro *chunk* (o maior) contenha a maior parte do campo de onda não nulo, ele ditará o desempenho. Além disso, o escalonador *guided* possui uma sobrecarga (*overhead*) no gerenciamento do tamanho do *chunk*, o que não ocorre com o escalonador *dynamic*. O FWI usando a abordagem proposta teve um *speedup* de até 15.27% em comparação com o uso do *guided*.

Em nosso último conjunto de experimentos, comparamos o tempo de execução de um FWI usando o auto-ajuste proposto, os escalonadores *static* e *guided* do OpenMP com seus valores de tamanho de *chunk* padrão, e os escalonadores *dynamic*, *static* e *guided* do OpenMP com o tamanho de *chunk* definido por (Mohammed et al., 2022):

$$chunk = \left\lfloor \frac{N}{2^f \times 2P} \right\rfloor, \tag{2}$$

onde

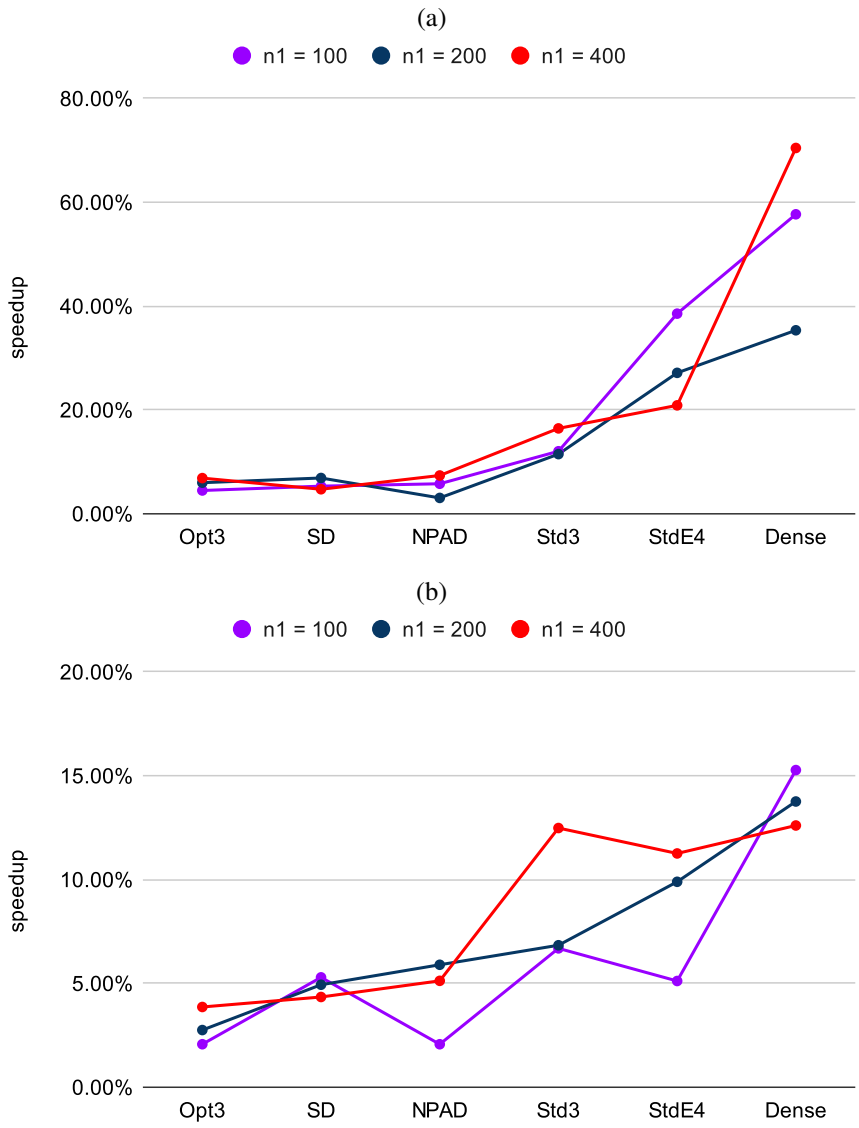
$$f = \left\lfloor log_2 \left(\frac{N}{P} \right) \times \frac{1}{\phi} \right\rfloor, \tag{3}$$

N denota o número de iterações do laço, P é o número de *threads* do OpenMP, e $\phi = 1.618$. Executamos este conjunto de experimentos nas máquinas SD e NPAD.

A Fig. 5 mostra os tempos de execução do FWI medidos na máquina SD. Para todos os tamanhos de entrada, a mediana do tempo de execução do FWI usando o escalonador *dynamic* do OpenMP com o tamanho de *chunk* definido por (Mohammed et al., 2022) foi menor do que usando os escalonadores padrão *static* e *guided* do OpenMP. Além disso, o FWI usando nosso método proposto apresentou os menores tempos de execução em todos os cenários testados neste conjunto de experimentos, sendo 3.75%, 3.88% e 3.86% mais rápido que o FWI usando o escalonador *dynamic* do OpenMP com o tamanho de *chunk* definido por (Mohammed et al., 2022) para $n1 = 100$, $n1 = 200$ e $n1 = 400$, respectivamente.

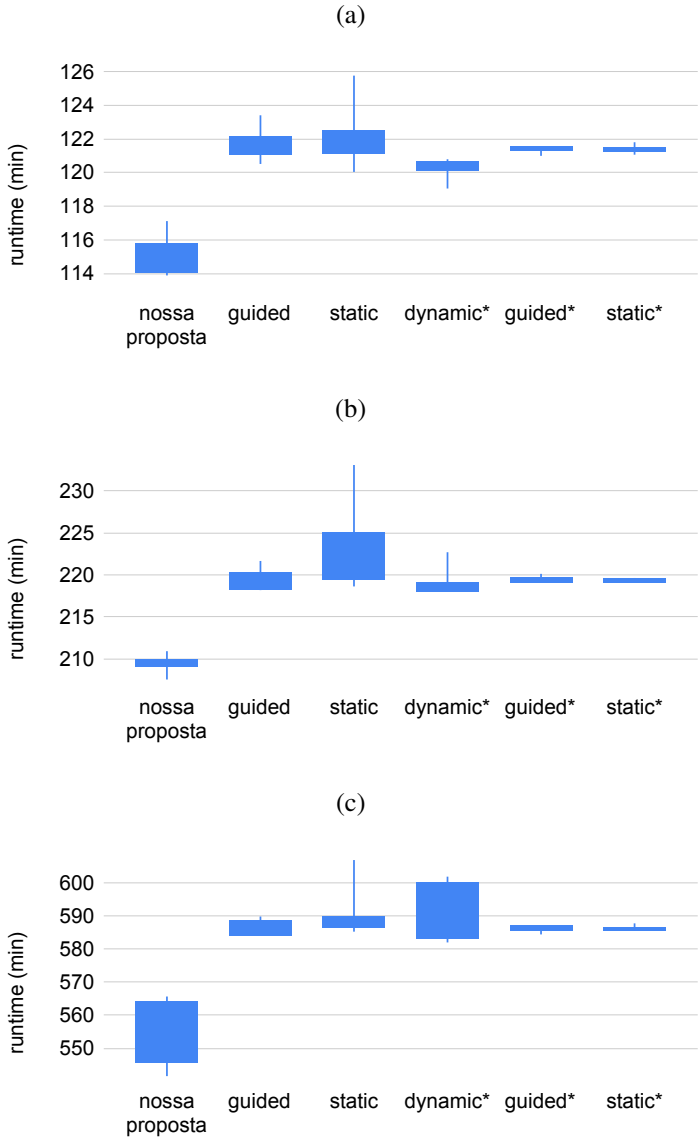
A Fig. 6 mostra os tempos de execução do FWI medidos na máquina NPAD. Para todos

Figura 4: *Speedup* do FWI com um único disparo usando o auto-ajuste proposto em comparação com os escalonadores do OpenMP (a) *static* e (b) *guided* em 5 máquinas (OPT3, SD, NPAD, STD3, STDE4 e DENSE), para três tamanhos de entrada, $(n1, n2, n3) = (100, 400, 400)$, $(200, 400, 400)$ e $(400, 400, 400)$. Para os escalonadores OpenMP, o tamanho do *chunk* não foi definido explicitamente. Cada ponto é uma mediana de pelo menos cinco execuções.



Fonte: Autoria própria (2025).

Figura 5: Tempo de execução do FWI com um único disparo usando o auto-ajuste proposto em comparação com os escalonadores padrão *static* e *guided* do OpenMP, e os escalonadores *dynamic*, *static* e *guided* do OpenMP usando o tamanho de *chunk* proposto por (Mohammed et al., 2022) (marcado com *). Este conjunto de experimentos foi realizado na máquina SD para três tamanhos de entrada, $(n1, n2, n3) =$ (a) $(100, 400, 400)$, (b) $(200, 400, 400)$ e (c) $(400, 400, 400)$. Cada ponto é uma mediana de cinco execuções.



Fonte: Autoria própria (2025).

os tamanhos de entrada, a mediana do tempo de execução do FWI com o tamanho de *chunk* definido por (Mohammed et al., 2022) foi maior do que usando os escalonadores padrão *static* e *guided* do OpenMP. Além disso, o FWI usando nosso método proposto apresentou os menores tempos de execução em todos os cenários testados neste conjunto de experimentos, sendo 2.07%, 3.05% e 5.12% mais rápido que o FWI mais veloz usando os escalonadores padrão *dynamic* ou *static* do OpenMP para $n1 = 100$, $n1 = 200$ e $n1 = 400$, respectivamente.

5 TRABALHOS RELACIONADOS

O auto-ajuste (*auto-tuning*) tem sido amplamente estudado. Alguns autores apresentam métodos para ajustar fluxos de trabalho (*workflows*) automaticamente. (HoseinyFarahabady et al., 2020) propuseram um método de auto-ajuste para melhorar o fluxo de dados de aplicações de processamento de fluxo (*stream*) com base em informações coletadas continuamente em tempo de execução (por exemplo, recursos disponíveis e taxa de tráfego de entrada). (Shu et al., 2021) propuseram o *Component-based Ensemble Active Learning* (CEAL). Este método de auto-ajuste combina técnicas de aprendizado de máquina com o conhecimento da estrutura do fluxo de trabalho *in situ* para permitir a configuração automatizada do fluxo de trabalho com medições de desempenho limitadas. Nossa proposta visa auto-ajustar aplicações, diferentemente de (HoseinyFarahabady et al., 2020; Shu et al., 2021).

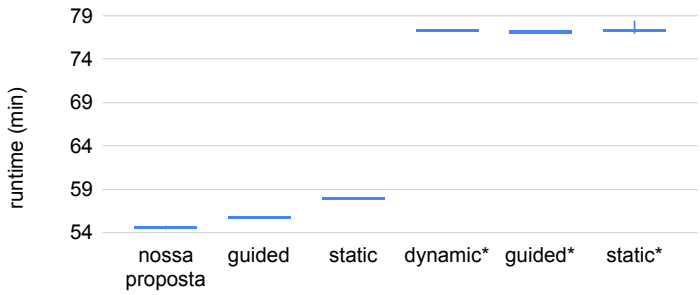
O auto-ajuste de aplicações tem sido explorado sob diversas perspectivas. (Padoin et al., 2014; Padoin et al., 2017) empregam técnicas de balanceamento de carga para controlar a frequência do processador, melhorando a eficiência energética de aplicações paralelas em sistemas multicore. Ambos os trabalhos utilizam técnicas de ajuste de frequência do processador para desacelerar os núcleos menos carregados, economizando energia. (Rocha et al., 2020) apresenta o AtTune, um *framework* baseado em heurísticas para otimizar a execução paralela de aplicações, ajustando o número de processos/*threads* e os níveis de frequência da CPU. (Bez, 2021) propõe o uso de técnicas de aprendizado de máquina para detectar automaticamente padrões de acesso de E/S (*I/O*) em tempo de execução para aplicações em plataformas de Computação de Alto Desempenho (HPC). Com base nesses padrões, é possível ajustar dinamicamente os parâmetros de E/S e alocar recursos de forma mais eficiente para melhorar o desempenho geral do sistema. (Bababa et al., 2021) apresentaram uma abordagem de auto-ajuste baseada no monitoramento de E/S e em modelagem preditiva. Essa abordagem pode encontrar um conjunto de valores de parâmetros de E/S para um determinado sistema e caso de uso da aplicação. Este auto-ajuste é aplicado a pilhas de E/S paralelas em sistemas distribuídos usando a biblioteca MPI-IO ROMIO. (Robert; Zertal; Couvee, 2020) desenvolveram uma ferramenta de otimização chamada SHAMAN, que fornece uma aplicação web pronta para uso para realizar o auto-ajuste de caixa-preta (*black-box*) de componentes computacionais personalizados em um sistema distribuído para uma aplicação específica executada através do gerenciador de cargas de trabalho Slurm. Diferentemente de (Padoin et al., 2014; Padoin et al., 2017; Rocha et al., 2020; Bez, 2021; Bababa et al., 2021; Robert; Zertal; Couvee, 2020), nosso auto-ajuste atua na otimização em nível de código.

Vários trabalhos propuseram abordagens para o auto-ajuste de otimização em nível de código. (Kale; Randles; Gropp, 2014) propuseram melhorar o escalonamento combinando abordagens estáticas e dinâmicas em máquinas de multiprocessamento simétrico (SMP), enfatizando a localidade espacial. Diferentemente de (Kale; Randles; Gropp, 2014), propomos um auto-ajuste para qualquer aplicação paralela que utilize OpenMP.

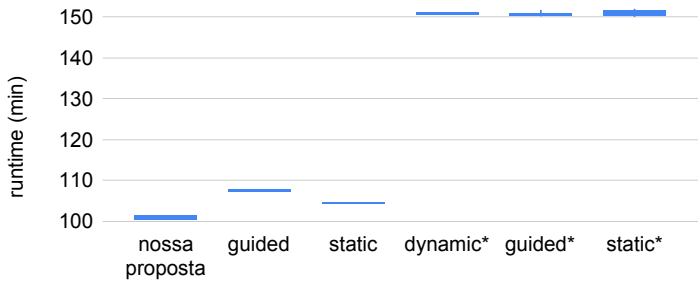
Katagiri, Ohshima e Matsumoto (2014), Katagiri, Ohshima e Matsumoto (2015) apresentaram o ppOpen-AT, um *framework* para otimização de código orientada por diretivas, como

Figura 6: Tempo de execução do FWI com um único disparo usando o auto-ajuste proposto em comparação com os escalonadores padrão *static* e *guided* do OpenMP, e os escalonadores *dynamic*, *static* e *guided* do OpenMP usando o tamanho de *chunk* proposto por (Mohammed et al., 2022) (marcado com *). Este conjunto de experimentos foi realizado na máquina SD para três tamanhos de entrada, $(n1, n2, n3) =$ (a) $(100, 400, 400)$, (b) $(200, 400, 400)$ e (c) $(400, 400, 400)$. Cada ponto é uma mediana de cinco execuções.

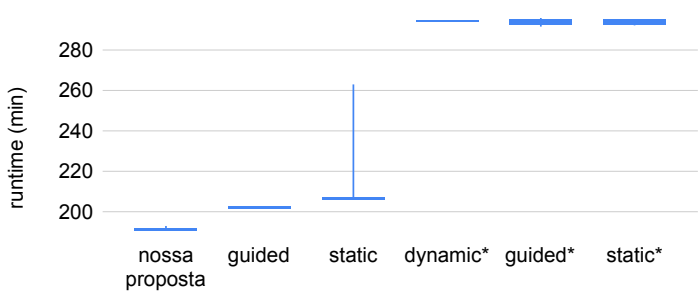
(a)



(b)



(c)



Fonte: Autoria própria (2025).

divisão de laço (*loop split*) e fusão de laço (*loop fusion*). (Menon; Bhatele; Gamblin, 2020) apresentaram o HiPerBOt, um *framework* de seleção de configuração baseado em otimização Bayesiana, para identificar parâmetros em nível de aplicação e de plataforma que resultem em configurações de alto desempenho. (Liu et al., 2021) apresentam o GPTune, um auto-ajuste baseado em aprendizado multitarefa e otimização Bayesiana, adequado para ajustar códigos de aplicações em exaescala. (Kimovski et al., 2021) desenvolveram um *framework* baseado em aprendizado de máquina para otimizar aplicações em exaescala. O AutoTuner pode ajustar os parâmetros da aplicação para melhorar o desempenho ao detectar uma anomalia. (Roy et al., 2021) apresentaram o Bliss, um algoritmo que utiliza otimização Bayesiana para encontrar a configuração de parâmetros próxima da ótima. O Bliss precisa executar a aplicação algumas vezes para determinar a função de custo. (Milani, 2020) propuseram políticas para escolher automaticamente a melhor entre versões de código definidas pelo usuário. Em comum, (Katagiri; Ohshima; Matsumoto, 2014; Katagiri; Ohshima; Matsumoto, 2015; Menon; Bhatele; Gamblin, 2020; Liu et al., 2021; Kimovski et al., 2021; Roy et al., 2021; Milani, 2020) precisam executar a aplicação sob uma variedade de configurações diferentes para aprender a correlação entre os parâmetros de uma aplicação e seu desempenho, o que pode ser proibitivo para aplicações computacionalmente custosas como o FWI. Propomos usar o tempo para computar a primeira iteração do laplaciano como uma estimativa do tempo do FWI.

Sakurai et al. (2020) propuseram duas funções de auto-ajuste para arquiteturas de computação heterogêneas. A primeira função pode realizar transformações de laço para ajustar automaticamente a posição das diretivas em laços OpenMP. A segunda função altera dinamicamente o número de *threads*. (Bak et al., 2018) introduziram um método de balanceamento de carga baseado na integração de Charm++ e OpenMP para distribuir dinamicamente tarefas criadas pelo usuário. (Kruse; Finkel; Wu, 2020) abordaram a otimização de desempenho de compiladores através do auto-ajuste de transformações de laço. Sua proposta é baseada em árvores de busca para explorar o espaço de transformações de laço e utiliza pragmas inseridos no código-fonte para aplicar essas transformações. Nossa proposta visa otimizar o escalonamento de laços paralelos, diferentemente de (Sakurai et al., 2020; Bak et al., 2018; Kruse; Finkel; Wu, 2020).

Muitos autores propuseram métodos de auto-ajuste adequados para otimizar o escalonamento de laços paralelos. (Dutta et al., 2022) apresentaram um trabalho preliminar sobre o uso de aprendizado de máquina profundo para identificar padrões de laço e ajustar parâmetros de laço com OpenMP. Como (Dutta et al., 2022) utilizam aprendizado de máquina, é necessário treiná-lo antes da execução. (Wood et al., 2021) introduziram o Artemis, um *framework* online que ajusta dinamicamente a execução de regiões paralelas através do treinamento de modelos de otimização. Nosso método proposto não requer treinamento, diferentemente de (Dutta et al., 2022; Wood et al., 2021).

Seiferth, Korch e Rauber (2020) introduziram uma abordagem que auto-ajusta aplicações classificando abordagens de otimização com base em um modelo analítico de desempenho. (Andreolli et al., 2014; Andreolli et al., 2015) trouxeram uma abordagem para ajustar automaticamente aplicações sísmicas compilando e executando cada conjunto de parâmetros escolhido por um algoritmo genético, incluindo o tamanho do *chunk* de carga e as *flags* de compilação. Os métodos de (Seiferth; Korch; Rauber, 2020; Andreolli et al., 2014; Andreolli et al., 2015) realizam o auto-ajuste estaticamente. Dessa forma, eles podem encontrar um estado de sistema diferente do de tempo de execução, pois aspectos como disponibilidade de memória, uso e acesso a recursos, carga de trabalho e a carga dos núcleos podem variar ao longo do tempo. Por outro lado, nosso método realiza o auto-ajuste dinamicamente em tempo de execução, permitindo que ele ajuste a aplicação para o estado atual do sistema.

Booth e Lane (2022) apresentaram o iCh, um método de auto-ajuste para os escalonadores de laço do OpenMP (*guided* e *dynamic*) que utiliza heurísticas de baixo custo para reduzir o desbalanceamento de carga. O iCh utiliza *work-stealing* sob demanda como seu principal mecanismo de balanceamento de carga e um sistema de distribuição com filas locais para as *threads*. Ao usar *work-stealing*, o iCh pode ter uma sobrecarga (*overhead*) paralela extra, pois precisa gerenciar filas de tarefas e a sincronização de *threads*. Pelo contrário, nossa proposta utiliza o escalonador dinâmico do OpenMP.

Rasch et al. (2021) apresentaram o ATF (*Automatic Tuning Framework*), uma abordagem geral para o ajuste automático de programas com parâmetros de ajuste interdependentes. Ele foca em otimizar a eficiência do ajuste automático, fornecendo mecanismos otimizados para gerar, armazenar e explorar o espaço de busca de parâmetros. O ATF observa restrições passadas para gerar um espaço de busca, enquanto em nossa proposta, o campo de busca é definido antecipadamente, sem a necessidade da etapa de geração. O ATF utiliza uma estratégia de busca multidimensional, enquanto utilizamos o CSA, que, como uma meta-heurística, busca reduzir o espaço de busca.

Mohammed et al. (2022) introduziram três métodos para seleção automatizada de escalonador e um ajuste automático de tamanho de *chunk* para o escalonador *auto* do OpenMP. Eles proporcionam melhor desempenho adaptando-se a variações imprevisíveis na aplicação e no sistema durante a execução. (Mohammed et al., 2022) propõem uma equação para calcular o tamanho do *chunk* com base no número de *threads* e no tamanho do problema. Fazer isso pode desconsiderar outros fatores essenciais, como a arquitetura de *hardware* e o comportamento do *software* (por exemplo, padrão de acesso à memória). Usando o PATSMA com o CSA para otimizar o tamanho do *chunk*, nossa proposta tem maior probabilidade de considerar todos os fatores relevantes. De fato, para os experimentos mostrados na Fig. 5, um FWI teve melhor desempenho usando um tamanho de *chunk* obtido com nosso método do que o tamanho de *chunk* proposto por (Mohammed et al., 2022).

Assis et al. (2020) propuseram o uso do PATSMA com o CSA para otimizar o tamanho do *chunk* de laços paralelos de uma migração em tempo reverso (RTM). Aqui, expandimos a abordagem de (Assis et al., 2020) para torná-la adequada para um FWI. O FWI impõe um desafio extra aos métodos de auto-ajuste, pois possui maior complexidade algorítmica e uma entrada potencialmente diferente para cada iteração. Este artigo também aumenta a reprodutibilidade, pois os testes também foram realizados em instâncias de nuvem. Finalmente, apresentamos testes comparativos com um método de auto-ajuste do estado da arte.

6 CONCLUSÃO

Este trabalho propôs uma estratégia de auto-ajuste baseada no *Parameter Auto-Tuning for Shared Memory Algorithms* (PATSMA) para encontrar um tamanho de *chunk* ideal para o escalonador *dynamic* do OpenMP. Esta abordagem foi projetada para funcionar em diferentes ambientes computacionais, independentemente de parâmetros como número de *threads*, processadores e quantidade de memória. Aplicamos o mecanismo proposto a um algoritmo de Inversão da Forma de Onda Completa (FWI), reduzindo seu tempo de execução.

Experimentos variando o tamanho da entrada e em diferentes ambientes computacionais mostraram que o auto-ajuste proposto supera os escalonadores padrão *static* e *guided* do OpenMP em todos os cenários testados. Para esses experimentos, a estratégia proposta alcançou *speedups* de até 70.46%. Esses resultados demonstram que nosso método se adapta bem a diferentes tamanhos de problema e ambientes computacionais, superando os escalonadores padrão do OpenMP.

Outro conjunto de experimentos aplicado a um FWI mostrou o desempenho do método proposto em comparação com os escalonadores *static* e *guided* do OpenMP para diferentes números de disparos sísmicos. Neste conjunto de testes, o auto-ajuste proposto também superou os escalonadores OpenMP nos cenários testados, alcançando até 14.96% de *speedup*. Neste conjunto de testes, foi observado que o *speedup* permaneceu estável porque a sobrecarga (*overhead*) não aumentou com o aumento do número de disparos. É possível concluir que o tamanho de *chunk* ajustado para o primeiro disparo pode ser reutilizado para os disparos e iterações seguintes do FWI. Em todos os casos, a sobrecarga foi inferior a 1.2%. Os resultados também mostram que o método proposto é escalável, apresentando *speedups* mais altos para testes com maiores tamanhos de entrada e em arquiteturas com mais núcleos.

Finalmente, comparamos os tempos de execução de um FWI usando nosso auto-ajuste proposto, os escalonadores padrão *static* e *guided* do OpenMP, e os escalonadores *static*, *dynamic* e *guided* do OpenMP com o tamanho de *chunk* determinado pela fórmula proposta por (Mohammed et al., 2022). Esses testes foram realizados em duas máquinas para três tamanhos de entrada. Nosso método superou os escalonadores testados em todos os cenários, alcançando *speedups* de 2.07% a 34.95%.

DISPONIBILIDADE DO CÓDIGO COMPUTACIONAL

O código essencial para a análise neste artigo está disponível no GitLab nos seguintes repositórios públicos: (1) <https://gitlab.com/lappsufrn/mamute> código da Inversão da Forma de Onda Completa em um repositório chamado Mamute e (2) <https://gitlab.com/lappsufrn/patsma> código do PATSMA. Todos os códigos utilizados neste artigo foram escritos em linguagem C++ e são de código aberto sob a Licença MIT.

7 AGRADECIMENTOS

Os autores agradecem o apoio da Shell Brasil através do projeto "*Novas metodologias computacionalmente escaláveis para sísmica 4D orientado ao alvo em reservatórios do pré-sal*" na Universidade Federal do Rio Grande do Norte (UFRN) e a importância estratégica do apoio concedido pela ANP através da cláusula de investimento em P&D. Este trabalho foi parcialmente apoiado por créditos da Oracle Cloud e recursos relacionados fornecidos pelo programa Oracle for Research. Os autores também agradecem ao Laboratório Nacional de Computação Científica (LNCC/MCTI, Brasil) e ao Núcleo de Processamento de Alto Desempenho da UFRN (NPAD/UFRN) por fornecerem os recursos de HPC dos supercomputadores SDumont e NPAD, que contribuíram para os resultados de pesquisa relatados neste artigo. Felipe Silva foi pesquisador de iniciação científica através dos programas PIVIC e PIVIC-Af na UFERSA.

REFERÊNCIAS

ANDREOLLI, C. et al. Genetic Algorithm Based Auto-Tuning of Seismic Applications on Multi and Manycore Computers. In: **EAGE Workshop on High Performance Computing for Upstream**. [S.l.: s.n.], 2014. p. 0. ISBN 9781634391672.

ANDREOLLI, C. et al. Characterization and Optimization Methodology Applied to Stencil Computations. In: JEFFERS, J.; REINDERS, J. (Ed.). **High Performance Parallelism Pearls**. Elsevier, 2015. cap. 23, p. 377–396. ISBN 9780128021996. Disponível em: <http://www.sciencedirect.com/science/article/pii/B9780128021187000236>.

ASSIS, I. A. S. et al. Auto-tuning of dynamic scheduling applied to 3d reverse time migration on multicore systems. **IEEE Access**, v. 8, p. 145115–145127, 2020.

BABABA, A. et al. Improving the i/o performance of applications with predictive modeling based auto-tuning. In: **2021 International Conference on Engineering and Emerging Technologies (ICEET)**. [S.l.: s.n.], 2021. p. 1–6.

BAK, S. et al. Multi-level load balancing with an integrated runtime approach. In: **Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**. IEEE Press, 2018. (CCGrid '18), p. 31–40. ISBN 9781538658154. Disponível em: <https://doi.org/10.1109/CCGRID.2018.00018>.

BARROS, T. et al. Auto-tuning of 3d acoustic wave propagation in shared memory environments. In: **Proceedings of the 80th EAGE Conference and Exhibition 2018**. European Association of Geoscientists & Engineers, 2018. p. 1–5. ISSN 2214-4609. Disponível em: <https://www.earthdoc.org/content/papers/10.3997/2214-4609.201803072>.

BAYSAL, E.; KOSLOFF, D. D.; SHERWOOD, J. W. C. **Reverse time migration**. **Geophysics**, v. 48, p. 1514–1524, 1983.

BEZ, J. L. **Dynamic Tuning and Reconfiguration of the I/O Forwarding Layer in HPC Platforms**. Tese (Doutorado) — 'Universidade Federal do Rio Grande do Sul', 05 2021.

BOARD, O. A. R. **OpenMP Specifications**. 2021. Version 5.1. Disponível em: <https://www.openmp.org/specifications/>.

BOOTH, J. D.; LANE, P. A. **An adaptive self-scheduling loop scheduler**. **Concurrency and Computation: Practice and Experience**, v. 34, n. 6, p. e6750, 2022. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6750>.

DAGUM, L.; MENON, R. OpenMP: an industry standard API for shared-memory programming. **IEEE Computational Science and Engineering**, v. 5, n. 1, p. 46–55, 1998. ISSN 1070-9924.

DUTTA, A. et al. Pattern-based autotuning of openmp loops using graph neural networks. In: **2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)**. [S.l.: s.n.], 2022. p. 26–31.

FERNANDES, J. B. et al. Automatic scheduler for 3d seismic modeling by finite differences. In: **Proc. Rio Oil Gas Expo Conf.** [S.l.: s.n.], 2018. p. 1–10.

FERNANDES, J. B. et al. **Mamute: high-performance computing for geophysical methods**. 2025. Disponível em: <https://arxiv.org/abs/2502.12350>.

FERNANDES, J. B. et al. **PATSMA: Parameter Auto-tuning for Shared Memory Algorithms**. 2024.

GRIEWANK, A.; WALTHER, A. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. **ACM Trans. Math. Softw.**, Association for Computing Machinery, New York, NY, USA, v. 26, n. 1, p. 19–45, mar 2000. ISSN 0098-3500. Disponível em: <https://doi.org/10.1145/347837.347846>.

HOSEINYFARAHABADY, M. R. et al. Auto-tuning of large-scale iterative operations on modern streaming platforms. In: **Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2020. (CoNEXT '20), p. 554–555. ISBN 9781450379489. Disponível em: <https://doi.org/10.1145/3386367.3431680>.

KALE, V.; RANDLES, A.; GROPP, W. D. Locality-optimized mixed static/dynamic scheduling for improving load balancing on SMPs. In: **Proceedings of the 21st European MPI Users' Group Meeting**. [S.l.: s.n.], 2014. p. 115–116.

KATAGIRI, T.; OHSHIMA, S.; MATSUMOTO, M. Auto-tuning of computation kernels from an FDM code with ppOpen-AT. In: **Proceedings - 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, MCSoC 2014**. [S.l.: s.n.], 2014. p. 0. ISBN 9781479943050.

KATAGIRI, T.; OHSHIMA, S.; MATSUMOTO, M. Directive-Based Auto-Tuning for the Finite Difference Method on the Xeon Phi. In: **Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2015**. [S.l.: s.n.], 2015. p. 0. ISBN 0769555101.

KIMOVSKI, D. et al. Autotuning of exascale applications with anomalies detection. **Frontiers in Big Data**, Frontiers Media SA, v. 4, p. 657218, 2021.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. **Science**, v. 220, n. 4598, p. 671–680, 1983.

KRUSE, M.; FINKEL, H.; WU, X. Autotuning search space for loop transformations. **CoRR**, abs/2010.06521, 2020. Disponível em: <https://arxiv.org/abs/2010.06521>.

LIU, Y. et al. Gptune: Multitask learning for autotuning exascale applications. In: **GPTune: Multitask Learning for Autotuning Exascale Applications**. New York, NY, USA: Association for Computing Machinery, 2021. (PPoPP '21), p. 234–246. ISBN 9781450382946. Disponível em: <https://doi.org/10.1145/3437801.3441621>.

MENON, H.; BHATELE, A.; GAMBLIN, T. Auto-tuning parameter choices in hpc applications using bayesian optimization. In: **2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2020. p. 831–840.

MILANI, L. **Autotuning with machine learning of OpenMP task applications**. Tese (Doutorado) — 'Université Grenoble Alpes', 07 2020. Disponível em: <https://theses.hal.science/tel-03227414>.

MOHAMMED, A. et al. Automated scheduling algorithm selection and chunk parameter calculation in openmp. **IEEE Transactions on Parallel and Distributed Systems**, v. 33, n. 12, p. 4383–4394, 2022.

PADOIN, E. L. et al. Saving energy by exploiting residual imbalances on iterative applications. In: **2014 21st International Conference on High Performance Computing (HiPC)**. [S.l.: s.n.], 2014. p. 1–10. ISSN 1094-7256.

PADOIN, E. L. et al. Exploration of load balancing thresholds to save energy on iterative applications. In: **Communications in Computer and Information Science**. [S.l.: s.n.], 2017. p. 0. ISBN 9783319579719. ISSN 18650929.

PLESSIX, R.-E. A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. **Geophysical Journal International**, v. 167, n. 2, p. 495–503, 2006. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-246X.2006.02978.x>.

RASCH, A. et al. **Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)**. **Association for Computing Machinery**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 1, jan 2021. ISSN 1544-3566. Disponível em: <https://doi.org/10.1145/3427093>.

ROBERT, S.; ZERTAL, S.; COUVEE, P. Shaman: A flexible framework for auto-tuning hpc systems. In: CALZAROSSA, M. C. et al. (Ed.). **Modelling, Analysis, and Simulation of Computer and Telecommunication Systems - 28th International Symposium, MASCOTS 2020, Nice, France, November 17-19, 2020, Revised Selected Papers**. Springer, 2020. (Lecture Notes in Computer Science, v. 12527), p. 147–158. ISBN 978-3-030-68110-4. Disponível em: <https://doi.org/10.1007/978-3-030-68110-4_10>.

ROCHA, H. et al. Attune: A heuristic based framework for parallel applications autotuning. In: **Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. Porto Alegre, RS, Brasil: SBC, 2020. p. 151–156. ISSN 2763-9002. Disponível em: <https://sol.sbc.org.br/index.php/sbesc_estendido/article/view/13105>.

ROY, R. B. et al. Bliss: Auto-tuning complex applications using a pool of diverse lightweight learning models. In: **Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 2021. (PLDI 2021), p. 1280–1295. ISBN 9781450383912. Disponível em: <https://doi.org/10.1145/3453483.3454109>.

SAKURAI, T. et al. Autotuning by changing directives and number of threads in openmp using ppopen-at. **"**, Unpublished, 2020. Disponível em: <http://rgdoi.net/10.13140/RG.2.2.26988.80005>.

SEIFERTH, J.; KORCH, M.; RAUBER, T. Offsite autotuning approach: Performance model driven autotuning applied to parallel explicit ode methods. In: **High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings**. Berlin, Heidelberg: Springer-Verlag, 2020. p. 370–390. ISBN 978-3-030-50742-8. Disponível em: <https://doi.org/10.1007/978-3-030-50743-5_19>.

SHU, T. et al. Bootstrapping in-situ workflow auto-tuning via combining performance models of component applications. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. ACM, 2021. p. 1–15. ISBN 9781450384421. ISSN 21674337. Disponível em: <https://dl.acm.org/doi/10.1145/3458817.3476197>.

SILVA, K. Gonçalves-e; ALOISE, D.; SOUZA, S. Xavier-de. **Parallel synchronous and asynchronous coupled simulated annealing**. **The Journal of Supercomputing**, Springer, v. 74, p. 2841–2869, 2018.

SOUZA, S. Xavier-de et al. Coupled simulated annealing. **IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)**, v. 40, n. 2, p. 320–335, 2010.

SYMES, W. W. Reverse time migration with optimal checkpointing. **GEOPHYSICS**, v. 72, n. 5, p. SM213–SM221, 2007. Disponível em: <https://doi.org/10.1190/1.2742686>.

TARANTOLA, A. Inversion of seismic reflection data in the acoustic approximation. **Geophysics**, Society of Exploration Geophysicists, v. 49, n. 8, p. 1259–1266, 1984.

WOOD, C. et al. Artemis: Automatic runtime tuning of parallel execution parameters using machine learning. In: **High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings**. Berlin, Heidelberg: Springer-Verlag, 2021. p. 453–472. ISBN 978-3-030-78712-7. Disponível em: <https://doi.org/10.1007/978-3-030-78713-4_24>.