UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO

PRÓ-REITORIA DE GRADUAÇÃO

DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA

CURSO BACHARELADO INTERDISCIPLINAR EM TECNOLOGIA DA INFORMAÇÃO

GUSTAVO RODRIGUES DOS REIS

DESENVOLVIMENTO E ANÁLISE DE BIBLIOTECA DE TOLERÂNCIA A FALHAS PARA APLICAÇÕES EM PYTHON EM SISTEMAS DE MEMÓRIA DISTRIBUÍDA

PAU DOS FERROS

2026

GUSTAVO RODRIGUES DOS REIS

DESENVOLVIMENTO E ANÁLISE DE BIBLIOTECA DE TOLERÂNCIA A FALHAS PARA APLICAÇÕES EM PYTHON EM SISTEMAS DE MEMÓRIA DISTRIBUÍDA

Monografia apresentada à Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Interdisciplinar em Tecnologia da Informação.

Orientador: Ítalo Augusto Souza de Assis, Prof. Dr.

PAU DOS FERROS

2026

R375d Reis, Gustavo Rodrigues dos.
Desenvolvimento e análise de biblioteca de tolerância a falhas para aplicações em Python em sistemas de memória distribuída / Gustavo Rodrigues dos Reis. - 2026.
58 f. : il.

Orientador: Ítalo Augusto Souza de Assis.
Monografia (graduação) - Universidade Federal Rural do Semi-árido, Curso de Tecnologia da Informação, 2026.

1. Tolerância a falhas. 2. Sistemas de memória distribuída. 3. Mitigação de falhas a nível de usuário. 4. Python. 5. Inversão de forma de onda completa. I. Assis, Ítalo Augusto Souza de, orient. II. Título.

Ficha catalográfica elaborada por sistema gerador automático em conformidade com AACR2 e os dados fornecidos pelo) autor(a).
Biblioteca Campus Pau dos Ferros
Bibliotecária: Erlanda Maria Lopes da Silva
CRB: 15/966

GUSTAVO RODRIGUES DOS REIS

DESENVOLVIMENTO E ANÁLISE DE BIBLIOTECA DE TOLERÂNCIA A FALHAS PARA APLICAÇÕES EM PYTHON EM SISTEMAS DE MEMÓRIA DISTRIBUÍDA

Monografia apresentada à Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Interdisciplinar em Tecnologia da Informação.

Defendida em: 01 de Julho de 2026

BANCA EXAMINADORA

Ítalo Augusto Souza de Assis, Prof. Dr. (UFERSA)
Presidente

Carla dos Santos Santana, Prof.ª Dra. (UFRN)
Membro Examinador

Gilberto Corso, Prof. Dr. (UFRN)
Membro Examinador

Samuel Xavier de Souza, Prof. Dr. (UFRN)
Membro Examinador

AGRADECIMENTOS

À minha família, pelo apoio.

Aos meus amigos, por toda ajuda. Especialmente à Eduarda e Aparecida, por estarem comigo desde o início.

Ao meu orientador, Prof. Dr. Ítalo Augusto Souza de Assis, pela paciência, suporte e ensinamentos.

Aos demais orientadores e colegas de pesquisa, Prof.ª Dra. Carla dos Santos Santana, Prof. Dr. Samuel Xavier de Souza e Angelo Marcelino Cordeiro, por participarem da construção dessa pesquisa.

EPÍGRAFE

"O mistério da vida não é um problema a ser resolvido, mas uma realidade a ser experimentada."

(Frank Herbert)

RESUMO

A distribuição de altas cargas de trabalho entre diferentes nós computacionais é uma abordagem eficiente, principalmente para aplicações científicas. Todavia, à medida que aumentamos os recursos computacionais, o risco de ocorrência de falha (de *hardware* ou *software*) em um dos nós também aumenta. Dessa forma, é preciso que as aplicações possuam mecanismos de tolerância a falhas para assegurar sua resiliência. DeLIApy é um *wrapper Python* que estende as funcionalidades de tolerância a falhas da biblioteca original DeLIA (implementada em C++) para que aplicações desenvolvidas em *Python* garantam a resiliência em ambientes de memória distribuída. O DeLIApy possuía uma versão que implementava parcialmente algumas funcionalidades do DeLIA. Neste trabalho, é realizado o desenvolvimento das funcionalidades que abordam o *User-Level Failure Mitigation* que não estavam disponibilizadas no DeLIApy. Também é feita a análise do desempenho da biblioteca em uma aplicação de Inversão de Forma de Onda Completa desenvolvida com o *framework* Devito, para analisar o custo da adição da tolerância a falhas causado pela biblioteca. Os resultados validaram a correta execução das rotinas de resiliência e demonstraram que o custo adicional da biblioteca torna-se percentualmente pequeno em problemas maiores, consolidando o DeLIApy como uma solução eficiente.

Palavras-chave: tolerância a falhas; sistemas de memória distribuída; mitigação de falhas ao nível de usuário; python; inversão de forma de onda completa

ABSTRACT

High-performance scientific applications often rely on distributing computational workloads across multiple nodes. However, as the number of computing resources increases, so does the probability of hardware or software failures affecting one of the nodes. Therefore, fault-tolerance mechanisms are essential to ensure application resilience. DeLIApy is a Python wrapper that extends the fault-tolerance capabilities of the original DeLIA library, implemented in C++, enabling Python applications to achieve resilience in distributed-memory environments. A previous version of DeLIApy only partially implemented DeLIA functionalities. In this work, we developed the missing functionalities related to User-Level Failure Mitigation (ULFM), making them available within DeLIApy. In addition, we evaluated the library's performance using a Full Waveform Inversion application developed with the Devito framework to assess the overhead introduced by fault-tolerance mechanisms. The results validated the correct execution of the resilience routines and demonstrated that the additional overhead becomes proportionally smaller as problem size increases. These findings indicate that DeLIApy provides an efficient solution for incorporating fault tolerance into Python-based distributed-memory scientific applications.

Keywords: fault tolerance; distributed memory systems; user level failure mitigation; python; full-waveform inversion

LISTA DE FIGURAS

Figura 1 – Sistema de memória compartilhada 23
Figura 2 – Sistema de memória distribuída 24
Figura 3 – Comunicação coletiva com *allReduce* 25
Figura 4 – Execução de um processo com mecanismo de tolerância a falhas baseado em *checkpointing* e recuperação por *rollback* 26
Figura 5 – Mecanismo de detecção de falhas baseado em *Heartbeat* com monitoramento centralizado. 27
Figura 6 – Aquisição sísmica marinha 33
Figura 7 – Comparação entre o modelo de velocidade observado, o modelo inicial e o modelo obtido após a aplicação da FWI 34
Figura 8 – Modelo de velocidade observado da aplicação 2D 41
Figura 9 – Modelo de velocidade observado da aplicação 3D 41
Figura 10 – Sobrecarga e RSD com e sem DeLIApy da FWI 2D variando o número de tiros em instâncias EC2 48
Figura 11 – Sobrecarga e RSD com e sem DeLIApy da FWI 2D variando o tamanho do problema em instâncias EC2 49
Figura 12 – Sobrecarga e RSD com e sem DeLIApy da FWI 3D variando o número de tiros em instâncias EC2 51
Figura 13 – Sobrecarga e RSD com e sem DeLIApy da FWI 3D variando o tamanho do problema em instâncias EC2 52
Figura 14 – Sobrecarga e RSD com e sem DeLIApy da FWI 3D variando o número de tiros em supercomputador 54
Figura 15 – Sobrecarga e RSD com e sem DeLIApy da FWI 3D variando o tamanho do problema em supercomputador 56

LISTA DE TABELAS

Tabela 1 – Distribuição de tarefas entre processos 39
Tabela 2 – Parâmetros de simulação FWI 2D (Caso 1 em instâncias EC2) 48
Tabela 3 – Parâmetros de simulação FWI 2D (Caso 2 em instâncias EC2) 49
Tabela 4 – Parâmetros de simulação FWI 3D (Caso 1 em instâncias EC2) 50
Tabela 5 – Valores de *num_shots* e *num_receivers* por dimensão nas execuções em instâncias EC2 51
Tabela 6 – Parâmetros de simulação FWI 3D (Caso 2 em instâncias EC2) 52
Tabela 7 – Parâmetros de simulação FWI 3D (Caso 1 em supercomputador) 54
Tabela 8 – Valores de *num_shots* e *num_receivers* por dimensão nas execuções em supercomputador 55
Tabela 9 – Parâmetros de simulação FWI 3D (Caso 2 em supecomputador) 55

LISTA DE ABREVIATURAS E SIGLAS

BSP	*Bulk Synchronous Parallel Model*
CPU	*Central Processing Units*
CR	*Checkpointing* e *Rollback*
DeLIA	*Dependability Library for Iterative Applications*
DeLIAJ	DeLIA para Julia
DeLIApy	DeLIA para *Python*
FWI	*Full-Waveform Inversion*
HM	*Heartbeat Monitoring*
HPC	*High-Performance Computing*
IaC	*Infrastructure as Code*
MIMD	*Multiple Instructions Multiple Data*
MPI	*Message-Passing Interface*
RSD	Desvio Padrão Relativo
ULFM	*User-Level Failure Mitigation*

SUMÁRIO

1 INTRODUÇÃO 14

1.1 Problematização 15

1.2 Objetivos 16

1.2.1 Objetivos Específicos 16

1.3 Justificativa 17

1.4 Organização do texto 17

2 TRABALHOS RELACIONADOS 19

3 FUNDAMENTAÇÃO TEÓRICA 21

3.1 Threads e processos 22

3.2 Sistemas MIMD 22

3.2.1 Sistemas de memória compartilhada 23

3.2.2 Sistemas de memória distribuída 24

3.3 Tolerância a falhas em aplicações de memória distribuída 25

3.3.1 *Checkpointing* e *Rollback* 26

3.3.2 Monitoramento por batimentos cardíacos 26

3.3.3 *User Level Failure Mitigation* 27

3.3.4 Sobrecarga em técnicas de tolerância a falhas 28

3.4 *Bulk Synchronous Parallel Model* 28

3.5 Computação em nuvem 29

3.5.1 Infraestrutura como código 30

3.6 DeLIA 31

3.7 Portabilidade com *Cython* 32

3.8 Inversão de Forma de Onda Completa 32

4 METODOLOGIA 36

4.1 Implementação do *wrapper* 36

4.2 Teste em nuvem 37
4.2.1 Configuração da arquitetura em nuvem 38
4.3 Teste em supercomputador 38
4.4 Validação das funcioalidades ULFM implementadas 39
4.4.1 Aplicação *Python* 39
4.5 Integração da biblioteca na aplicação sísmica 40
4.6 Análise de desempenho 43
5 RESULTADOS 46
5.1 Teste de Validação 46
5.2 Execuções da FWI 2D em nuvem 47
5.2.1 Análise com base na quantidade de tarefas por nó 47
5.2.2 Análise com base no tamanho do problema 49
5.3 Execuções da FWI 3D em nuvem 50
5.3.1 Análise com base na quantidade de tarefas por nó 50
5.3.2 Análise com base no tamanho do problema 51
5.4 Execuções da FWI 3D em um supercomputador 53
5.4.0.1 Análise com base na quantidade de tarefas por nó 53
5.4.1 Análise com base no tamanho do problema 54
6 CONCLUSÕES E TRABALHOS FUTUROS 57
6.1 Trabalhos Futuros 57
REFERÊNCIAS 59

1 INTRODUÇÃO

A arquitetura das *Central Processing Units* (CPU, do inglês: Unidades Centrais de Processamento) passou por diversas mudanças ao longo das décadas, mas uma das mais importantes foi a adição de mais de um núcleo de processamento em um único *chip* (Pacheco; Malensek, 2022). Com essa mudança, a responsabilidade por melhorar o desempenho dos programas deixou de estar somente na evolução do *hardware* e passou, em grande parte, para os desenvolvedores, que agora precisam adaptar seus *softwares* para explorar o paralelismo oferecido pelos múltiplos núcleos.

Sistemas paralelos podem ser classificados de acordo com a quantidade de dados e instruções que gerenciam simultaneamente. Os sistemas *Multiple Instructions Multiple Data* (MIMD, do inglês: Múltiplas Instruções, Múltiplos Dados) podem ser divididos em duas categorias: sistemas de memória compartilhada e sistemas de memória distribuída. Num sistema de memória compartilhada, a paralelização é feita por meio de *threads* que compartilham o mesmo espaço de memória. A comunicação nesses sistemas ocorre de forma implícita por meio do acesso a estruturas de dados compartilhadas (Pacheco; Malensek, 2022). Já nos sistemas de memória distribuída, a paralelização é realizada distribuindo a aplicação em diferentes nós computacionais que possuem seu próprio bloco de memória, cada um com um ou mais processos do mesmo programa em execução. Dessa forma, a comunicação entre os processos é feita de forma explícita por meio de uma rede de interconexão que liga os nós. Essa rede de interconexão é geralmente o *Ethernet* ou fibra óptica (Pacheco; Malensek, 2022).

A computação de alto desempenho (HPC, do inglês: *High-Performance Computing*) se beneficia significativamente de ambientes de memória distribuída para reduzir o tempo de execução de suas aplicações. Programas ou aplicações classificadas como *Bulk Synchronous Parallel Model* (BSP, do inglês: Modelo Paralelo Síncrono em Massa) (Valiant, 1990), que dividem seu fluxo em superpassos, distribuídos entre os nós computacionais, fazem uso intensivo desse tipo de arquitetura para coordenar fases de computação e comunicação de maneira eficiente. A organização em superpassos com sincronização global torna o modelo especialmente adequado para aplicações iterativas em larga escala. Esse modelo é amplamente adotado em aplicações científicas iterativas de grande escala, como a *Full-Waveform Inversion* (FWI, do inglês: Inversão de Forma de Onda Completa) (Tarantola, 1984) na área da geofísica (Santana *et al.*, 2024).

Embora o modelo BSP forneça uma estrutura organizada para computação paralela, sua execução em ambientes distribuídos em larga escala torna as aplicações naturalmente suscetíveis

a falhas decorrentes da independência dos nós e da comunicação em rede, exigindo estratégias de tolerância a falhas. Logo, é necessário garantir a sincronização dos dados entre os nós, lidar com falhas de comunicação na rede e assegurar que o sistema continue funcional mesmo quando um ou mais nós se tornem indisponíveis.

Diversas técnicas são aplicadas para contornar cenários de falha em uma aplicação de memória distribuída, entre elas, destacam-se o *Checkpointing* e *Rollback* (CR, do inglês: Pontos de verificação e retrocesso) (Kalaiselvi; Rajaraman, 2000) e o *Heartbeat Monitoring* (HM, do inglês: Monitoramento por batimentos cardíacos) (Chetan; Ranganathan; Campbell, 2005). O CR salva o estado de execução da aplicação para retomar o processo a partir do último ponto salvo. Já no HM, os nós enviam mensagens periódicas a um coordenador (ou entre si) para verificar a disponibilidade e detectar falhas rapidamente.

Com o objetivo de integrar essas estratégias em um ambiente unificado e de fácil utilização, foi desenvolvida a biblioteca *Dependability Library for Iterative Applications* (DeLIA, do inglês: Biblioteca de Dependabilidade para Aplicações Iterativas) (Santana *et al.*, 2024). Implementada em C++, a DeLIA oferece mecanismos configuráveis de tolerância a falhas ao nível de aplicação em programas BSP, fornecendo funcionalidades de salvamento de estado, reinicialização a partir do estado salvo, detecção de interrupções de execução e de sinais de terminação em ambientes preemptivos. Dessa forma, a biblioteca busca simplificar o desenvolvimento de sistemas distribuídos mais confiáveis, reduzindo o impacto de falhas durante a execução de aplicações iterativas.

A DeLIA conta com interfaces que utilizam *wrappers* de funções para realizar a portabilidade para as linguagens de programação *Python* (DeLIApy) e *Julia* (DeLIAJ) (Irigoyen *et al.*, 2025), isto é, camada de *software* que permite acessar e utilizar, em *Python* e *Julia*, as funcionalidades implementadas originalmente em C++. O *wrapper Python* implementava parcialmente as funcionalidades do DeLIA. Assim, o presente trabalho tem o objetivo de concluir o desenvolvimento do DeLIApy e realizar os testes em supercomputador e ambientes em nuvem, a fim de analisar o desempenho da biblioteca nesses contextos.

1.1 Problematização

A DeLIApy implementava parcialmente as funcionalidades do DeLIA. Essas funcionalidades dizem respeito ao salvamento de estado, reinicialização a partir do estado salvo, detecção de interrupções de execução e sinais de terminação em ambientes preemptivos. Todavia, algumas

funcionalidades da biblioteca em C++ que englobam o uso de *User-Level Failure Mitigation* (ULFM, do inglês: Mitigação de Falhas ao Nível de Usuário) não foram implementadas nessa versão da interface *Python*.

O ULFM é uma interface que oferece tolerância a falhas no *Message-Passing Interface* (MPI, do inglês: Interface de Passagem de Mensagens) (MESSAGE Passing Interface Forum, 2015) e permite que a aplicação lide com falhas no modelo *fail-stop* (em que o processo interrompe sua execução na presença de uma falha). O ULFM possibilita a detecção da falha e executa o redimensionamento dos comunicadores, removendo os processos defeituosos e permitindo a continuidade da execução (Laguna *et al.*, 2016).

Na biblioteca DeLIA, essas funcionalidades são utilizadas para a aplicação continuar executando mesmo diante de falhas em um ou mais nós computacionais, redimensionando os comunicadores e eliminando os processos defeituosos. Logo, torna-se necessário que esses recursos sejam abordados pela interface de portabilidade, a fim de garantir que a DeLIApy seja congruente à DeLIA em termos funcionais.

Portanto, é preciso mapear esses mecanismos de baixo nível para o *Python* e verificar, mediante ambientes controlados de injeção de falhas, se após falhas do tipo *fail-stop*, os comunicadores são corretamente redimensionados e a aplicação prossegue sua execução sem finalizar globalmente.

1.2 Objetivos

O objetivo geral deste trabalho é realizar a portabilidade da biblioteca DeLIA, desenvolvida em C++, para a linguagem *Python*. Isso ocorre por meio da implementação de *wrappers*, de modo a garantir a equivalência funcional entre a versão original e sua interface em *Python*, incluindo a disponibilização dos mecanismos de tolerância a falhas baseados em ULFM para aplicações iterativas executadas em ambientes de memória distribuída.

1.2.1 Objetivos Específicos

- Finalizar o *Wrapper Python* da biblioteca DeLIA em *Python*.
- Incorporar a tolerância a falhas em uma aplicação geofísica de FWI.
- Realizar testes em arquitetura de nuvem.
- Realizar testes em supercomputador.

- Analisar a sobrecarga causada pela biblioteca durante a inversão de forma de onda completa.

1.3 Justificativa

Falhas em sistemas complexos, que demandam um alto poder computacional, são inevitáveis (Rojas *et al.*, 2021). Em cenários de longa duração de um processo, torna-se mais provável que a aplicação seja interrompida por eventos adversos. Tais falhas podem estar associadas a problemas de *hardware*, como o mau funcionamento de componentes (como CPU ou memória) ou quedas de energia; a inconsistências de *software*, como falhas no sistema operacional; ou ainda à revogação de instâncias preemptivas em ambientes de nuvem.

Diante desse cenário, mecanismos de tolerância a falhas tornam-se essenciais para garantir a continuidade das aplicações. Esse desafio é particularmente relevante para aplicações científicas, que frequentemente demandam grande poder computacional, recorrendo a ambientes HPC.

Entre as linguagens mais utilizadas para o desenvolvimento dessas aplicações, destaca-se o *Python*, apontado como a linguagem de programação mais popular de 2024[1]. Sua ampla adoção deve-se à alta produtividade e ao vasto ecossistema de bibliotecas. No entanto, enquanto bibliotecas em C++ (como a DeLIA) já oferecem mecanismos de tolerância a falhas, o *Python* muitas vezes carece de interfaces que exponham essas funcionalidades de baixo nível de forma acessível.

Dessa forma, a construção deste trabalho justifica-se pela necessidade de fazer com que a interface de portabilidade DeLIApy abranja a totalidade das funcionalidades da biblioteca em C++, garantindo que a execução de aplicações desenvolvidas em *Python*, especialmente em aplicações de ciência de dados e geofísica, que comumente recorrem a ambientes HPC, continue mesmo diante de falhas.

1.4 Organização do texto

O Capítulo 2 apresenta os trabalhos relacionados, discutindo soluções existentes na literatura voltadas à computação distribuída e à tolerância a falhas. O Capítulo 3 apresenta a fundamentação teórica necessária para o entendimento das soluções propostas, abordando os

[1] https://spectrum.ieee.org/top-programming-languages-2024

conceitos e tecnologias que sustentam o desenvolvimento realizado. O Capítulo 4 descreve o processo de portabilidade e os testes conduzidos em supercomputador e em ambientes em nuvem, permitindo a avaliação prática da biblioteca em diferentes cenários de execução. O Capítulo 5 apresenta os resultados dos testes, além de realizar uma análise dos dados obtidos. Por fim, o Capítulo 6 apresenta a conclusão dos resultados obtidos, além de indicações para trabalhos futuros.

2 TRABALHOS RELACIONADOS

Este capítulo é dedicado à apresentação e análise dos principais trabalhos relacionados ao tema desta pesquisa, com foco em soluções existentes para computação distribuída e tolerância a falhas, especialmente em linguagens de alto nível como *Python*.

O suporte à tolerância a falhas em linguagens de alto nível como *Python* é pouco explorado na computação de alto desempenho. Todavia, algumas bibliotecas oferecem suporte à tolerância a falhas, como o Parsl (Chard *et al.*, 2021), uma biblioteca *Python* que possibilita a execução paralela de programas por meio de *function decorators* (do inglês: decoradores de função). Ele lida com a distribuição dos processos e garante a tolerância a falhas implementando um modelo de concorrência seguro baseado na construção de um Grafo Acíclico Dirigido (DAG) dinâmico, que assegura que as tarefas só serão executadas quando suas dependências (como dados de entrada) estiverem todas atendidas. A biblioteca monitora as execuções das tarefas. Ela detecta exceções e tenta executar as tarefas novamente quando falham. O Parls lida com diferentes tipos de falha, como falhas em tarefas e falhas em nós.

O SBN-Python (Tereshchenko; Gankevich, 2021) é uma interface *Python* do *framework Subordination*, cujo objetivo é garantir a tolerância a falhas em tarefas de processamento de dados em lote (*batch*). Toda essa capacidade de processamento distribuído e tolerante a falhas é realizada completamente na memória RAM, baseando-se em um modelo de programação reativa. O SBN-Python parte do princípio fundamental de que a saída de operação (falha) de um dos nós computacionais é uma situação normal e esperada, e não uma exceção catastrófica. A tolerância a falhas nele está ligada à sua arquitetura de objetos de controle, onde o objeto "pai"delega tarefas aos processos "filhos"por meio de uma fila de processamento, fazendo com que o sistema consiga lidar com diferentes cenários de falhas. A biblioteca isola o erro e readapta a carga de trabalho aos recursos sobreviventes.

A DeLIA (Santana *et al.*, 2024) é uma biblioteca desenvolvida em C++ que oferece mecanismos de tolerância a falhas para aplicações iterativas. A biblioteca disponibiliza funcionalidades de monitoramento por batimentos cardíacos e ponto de salvamento e retrocesso, permitindo que a aplicação recupere o progresso realizado antes da ocorrência de uma falha. Ela conta com interfaces de portabilidade para *Python* (DeLIApy) e Julia (DeLIAJ) (Irigoyen *et al.*, 2025) que expõem a biblioteca C++ para essas linguagens de alto nível. A DeLIApy conta com uma versão que implementa parcialmente as funcionalidades do DeLIA. Essa versão não conta com as funcionalidades ULFM implementadas. Essas funcionalidades permitem que a aplicação

lide com falhas no modelo *fail-stop*, permitindo que a aplicação continue executando mesmo na ocorrência de falha em um processo MPI.

A versão DeLIApy proposta nesse trabalho apresenta maior abrangência no suporte à tolerância a falhas comparado aos trabalhos citados. A DeLIApy dispõe de mais de uma abordagem para tolerância para aplicações desenvolvidas em *Python*, ou seja, o desenvolvedor pode utilizar as funcionalidades de HM e CR e/ou funcionalidades com suporte a ULFM de acordo com sua necessidade e preferência.

3 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos teóricos que fundamentam o desenvolvimento deste trabalho, estabelecendo a base necessária para a compreensão das decisões de projeto e das soluções propostas. Inicialmente, na Seção 3.1, são discutidos os conceitos de processos e *threads*, que constituem os elementos fundamentais da execução concorrente e distribuída em sistemas computacionais.

Em seguida, na Seção 3.2, são abordados os sistemas paralelos segundo a classificação MIMD, com ênfase nos modelos de memória compartilhada e distribuída. Esses modelos são essenciais para compreender como aplicações podem explorar o paralelismo em diferentes arquiteturas, bem como as implicações no desenvolvimento e na execução de programas. Nesse contexto, também são apresentados mecanismos de comunicação entre processos, com destaque para o padrão MPI, amplamente utilizado em ambientes de computação de alto desempenho.

A partir dessa base, o capítulo discute, na Seção 3.3, sobre a tolerância a falhas em sistemas de memória distribuída, incluindo técnicas como *checkpointing* e *rollback*, *heartbeat monitoring*, além do ULFM, uma interface de tolerância a falhas para o MPI. Esses mecanismos são relevantes para este trabalho, uma vez que a confiabilidade da execução em ambientes de memória distribuída é um fator crítico, especialmente quando há risco de falhas nos nós de processamento. Nesta seção, também se discute o custo computacional associado à incorporação de cada um desses mecanismos em aplicações.

Na sequência, a Seção 3.4 apresenta o modelo de computação BSP, que organiza a execução paralela em etapas sincronizadas, facilitando a aplicação de estratégias de tolerância a falhas. Posteriormente, a Seção 3.5 discute conceitos de computação em nuvem, destacando a necessidade de automação da infraestrutura por meio de técnicas como *Infrastructure as Code* (IaC, do inglês: Infraestrutura como Código), fundamentais para garantir reprodutibilidade e resiliência do ambiente de execução.

Em seguida, o capítulo apresenta, na Seção 3.6, uma visão geral da biblioteca DeLIA e de suas interfaces de portabilidade, além de abordar, na Seção 3.7, o uso do *Cython* como ferramenta de integração entre linguagens de alto e baixo nível, permitindo combinar desempenho e produtividade no desenvolvimento de aplicações.

Por fim, o capítulo apresenta, na Seção 3.8, uma visão geral sobre a FWI, justificando o uso de ambientes HPC para sua execução e discorrendo sobre o uso de ferramentas como Devito para a implementação dessas aplicações.

Dessa forma, os conceitos apresentados ao longo deste capítulo se inter-relacionam ao fornecer o embasamento necessário para o desenvolvimento de uma solução paralela, distribuída e tolerante a falhas, executada em ambientes de alto desempenho.

3.1 Threads e processos

Um processo é a instância de um programa em execução, criada pelo sistema operacional quando o usuário o executa (Pacheco; Malensek, 2022). Ele é constituído por código executável em linguagem de máquina, um bloco de memória que abrange o próprio código, a pilha de execução (do inglês: *call stack*), responsável pelas chamadas de funções, a área de alocação dinâmica (do inglês: *heap*) e outras regiões de dados (Pacheco; Malensek, 2022). Além disso, o processo possui descritores de recursos fornecidos pelo sistema operacional, como arquivos abertos, informações de segurança que definem quais recursos podem ser acessados e dados sobre seu estado de execução, incluindo registradores e situação atual (em execução, pronto ou aguardando) (Pacheco; Malensek, 2022).

Uma *thread* é uma unidade de execução dentro de um processo que permite dividir um programa em tarefas parcialmente independentes, possibilitando que, quando uma *thread* esteja bloqueada (por exemplo, aguardando uma operação de entrada e saída), outra possa continuar executando. As *threads* são consideradas mais "leves" do que processos ao compartilharem o mesmo código executável, a maior parte da memória e os dispositivos de entrada e saída do processo ao qual pertencem, tornando a troca de contexto entre elas geralmente mais rápida do que entre processos distintos (Pacheco; Malensek, 2022). Apesar de compartilharem diversos recursos, cada *thread* mantém seu próprio contador de programa e sua própria pilha de execução, garantindo que possam executar de forma independente. Dessa forma, um processo é a entidade que cria e gerencia suas *threads*, as quais são iniciadas e finalizadas ao longo da execução do programa.

3.2 Sistemas MIMD

A chegada de processadores *multicore* possibilitou que sistemas, até então desenvolvidos para executar somente em processadores *single core* (chamados de programas sequenciais/seriais), fossem reescritos para aproveitar a capacidade computacional dessa mudança na arquitetura das CPUs, obtendo melhor desempenho em sua execução. Para alcançar tal objetivo, é preciso

aplicar técnicas que identifiquem e explorem oportunidades de paralelismo em um programa serial, dividindo as tarefas em partes que possam ser executadas simultaneamente por diferentes núcleos do processador. Tais técnicas constituem a computação paralela.

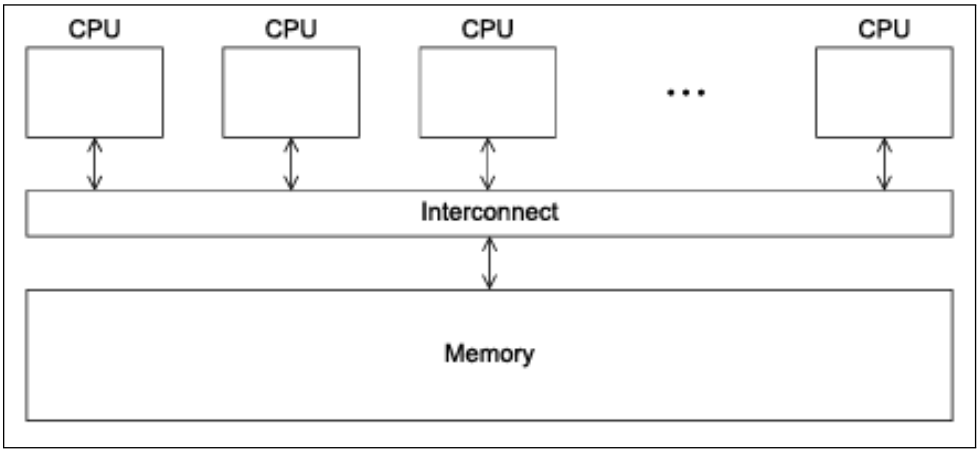
A Taxonomia de *Flynn* propõe classificações de sistemas paralelos de acordo com a quantidade de instruções e de dados que eles gerenciam simultaneamente. Uma dessas classificações é a MIMD, em que sistemas desse tipo operam com diferentes fluxos de instruções e de dados de forma assíncrona (Pacheco; Malensek, 2022).

Os sistemas MIMD podem ser divididos em duas categorias: sistemas de memória compartilhada — vários núcleos compartilham um mesmo espaço de memória — e sistemas de memória distribuída — cada processador (ou nó) tem seu próprio espaço de memória privado ou não tem acesso direto a algum endereço de memória, exigindo comunicação explícita pela rede para a troca de dados.

3.2.1 Sistemas de memória compartilhada

Em sistemas de memória compartilhada, o uso de *threads* permite que um processo possua múltiplos fluxos de controle independentes (Rauber; Runger, 2010). A Figura 1 ilustra como são organizados esses sistemas, onde as CPUs compartilham do mesmo espaço de memória. O acesso a esse espaço é realizado por meio de uma estrutura de interconexão, geralmente um barramento.

Figura 1 – Sistema de memória compartilhada



Fonte: Pacheco; Malensek, (2022)

A execução de um programa inicia com uma *thread* principal, responsável por criar novas *threads* quando necessário. Esse modelo permite a ramificação do fluxo de execução e a

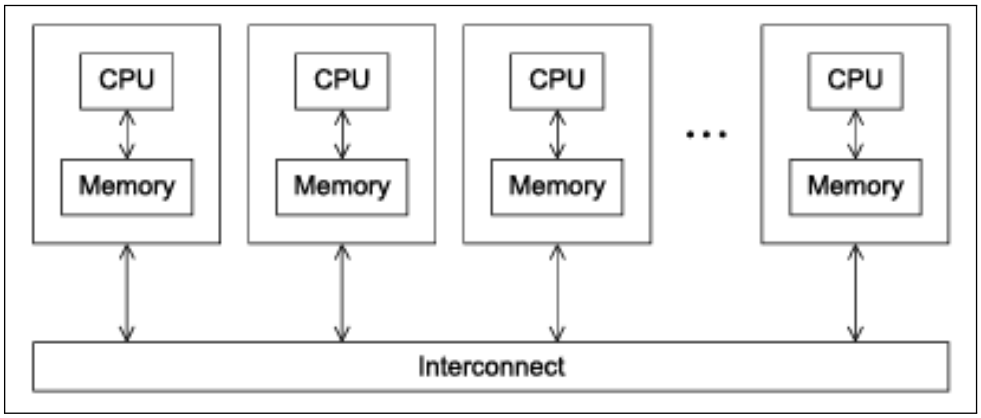
realização de tarefas concorrentemente.

O padrão *POSIX Threads* (ou *Pthreads*) (Electrical; Engineers, 2018) define um conjunto de funções que permitem a criação, gerenciamento e sincronização de *threads* no mesmo processo, explorando o modelo de memória compartilhada. Esse padrão garante que programas *multithreaded* desenvolvidos em conformidade com *POSIX* possam ser executados em diferentes sistemas semelhantes ao *Unix*.

3.2.2 Sistemas de memória distribuída

Como em um sistema de memória distribuída cada nó possui seu próprio espaço de memória privado, como ilustrado na Figura 2, quando um nó precisa de dados da memória local de outros nós para realizar computações locais, a troca de mensagens deve ser realizada de forma explícita através da rede de interconexão (Rauber; Runger, 2010). Geralmente, essa rede de interconexão é *Ethernet* ou fibra óptica. Essa troca de mensagens é organizada seguindo padrões de comunicação, sendo o mais utilizado o MPI.

Figura 2 – Sistema de memória distribuída



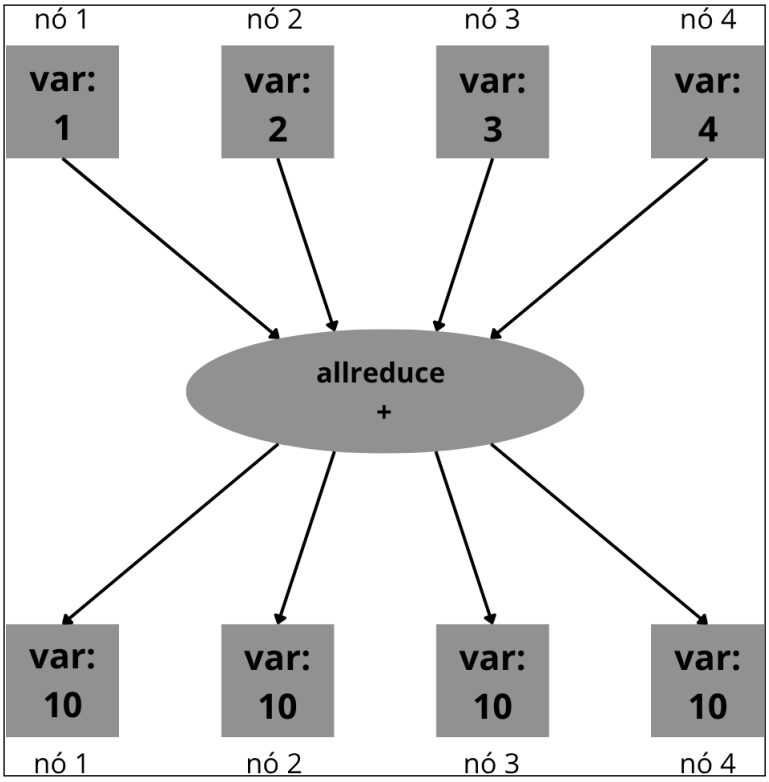
Fonte: Pacheco; Malensek, (2022)

O MPI é um padrão amplamente consolidado na área de computação paralela e distribuída, que especifica uma interface de programação para aplicações voltada à comunicação entre processos. Esse padrão define funções de comunicação ponto a ponto, como *MPI_Send*, utilizada para o envio de mensagens, e *MPI_Recv*, destinada ao recebimento, possibilitando a troca explícita de dados entre processos em execução.

Os processos que interagem por meio do MPI estão organizados em um domínio de comunicação denominado comunicador (Rauber; Runger, 2010), que delimita o escopo da comunicação. Com isso, além das operações ponto a ponto, o padrão fornece rotinas de

comunicação coletiva, que envolvem todos os processos pertencentes a um mesmo comunicador. Um exemplo é a função *MPI_Allreduce* que permite a combinação de valores de todos os nós computacionais e depois redistribui o valor para cada um deles. A Figura 3 ilustra o *Allreduce* efetuando uma operação de soma entre os nós e, em seguida, redistribuindo o valor atualizado.

Figura 3 – Comunicação coletiva com *allReduce*



Fonte: Autoria própria, (2026)

Entre as principais implementações de código aberto do padrão MPI destaca-se o *OpenMPI*[2], largamente utilizado em ambientes de computação de alto desempenho, por sua eficiência e portabilidade.

O MPI oferece suporte a linguagens de baixo nível como C, C++ e linguagens de alto nível como *Python* por meio do pacote *mpi4py*[3], que disponibiliza interfaces MPI para a linguagem.

3.3 Tolerância a falhas em aplicações de memória distribuída

Em sistemas de memória distribuída, uma falha em um dos nós pode resultar em degradação do desempenho, perda de mensagens ou até a perda total do trabalho realizado. O aumento de componentes nesses sistemas aumenta a probabilidade de falhas (Kalaiselvi; Rajaraman,

[2] https://docs.open-mpi.org/en/v5.0.x/
[3] https://mpi4py.readthedocs.io/en/stable/mpi4py.html

2000). Isso torna a tolerância a falhas um aspecto central no projeto de sistemas de memória distribuída. Uma aplicação é tolerante a falhas se ela consegue continuar sua execução mesmo na presença de uma falha (Steen; Tanenbaum, 2023).

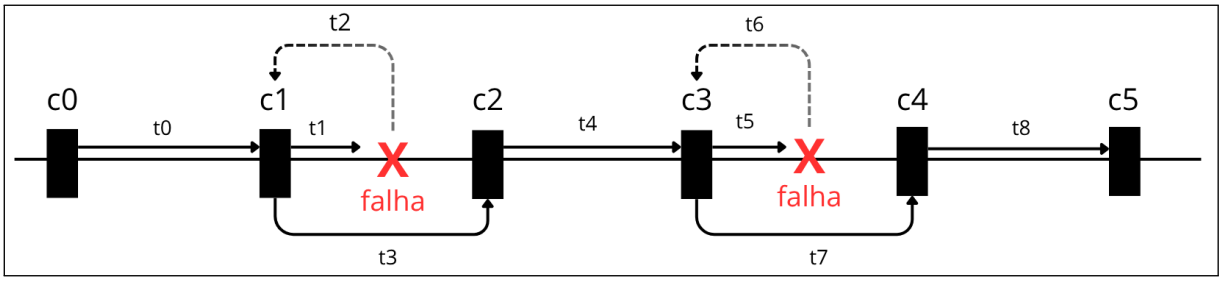
O desenvolvimento de técnicas de tolerância a falhas é uma área amplamente pesquisada na computação paralela. Dentre as técnicas mais empregadas, destacam-se o *checkpointing* e *rollback* (CR) e o *heartbeat monitoring* (HM).

3.3.1 *Checkpointing* e *Rollback*

O CR consiste em salvar o estado do programa em intervalos periódicos (*checkpoint*) para que, caso ocorra uma falha em um dos nós, a computação retorne do último estado salvo (*rollback*), evitando que o programa reinicie de seu estado inicial (Kalaiselvi; Rajaraman, 2000).

A Figura 4 ilustra, de forma esquemática, esse mecanismo de recuperação em ação sobre um processo ao longo do tempo. No diagrama, os blocos c_0 a c_5 representam os *checkpoints*, instantes em que o estado do processo é consolidado de forma segura em armazenamento estável. Os rótulos indicados de t_0 a t_8 demarcam os diferentes intervalos de processamento e as transições de estado associadas a eventos de falha e de recuperação. Os pontos t_2 e t_6 representam o processo de *rollback*, no qual a aplicação, em caso de falha, inicia sua computação a partir do último *checkpoint* salvo, neste caso, c_1 e c_3, respectivamente.

Figura 4 – Execução de um processo com mecanismo de tolerância a falhas baseado em *checkpointing* e recuperação por *rollback*



Fonte: Autoria própria, (2026)

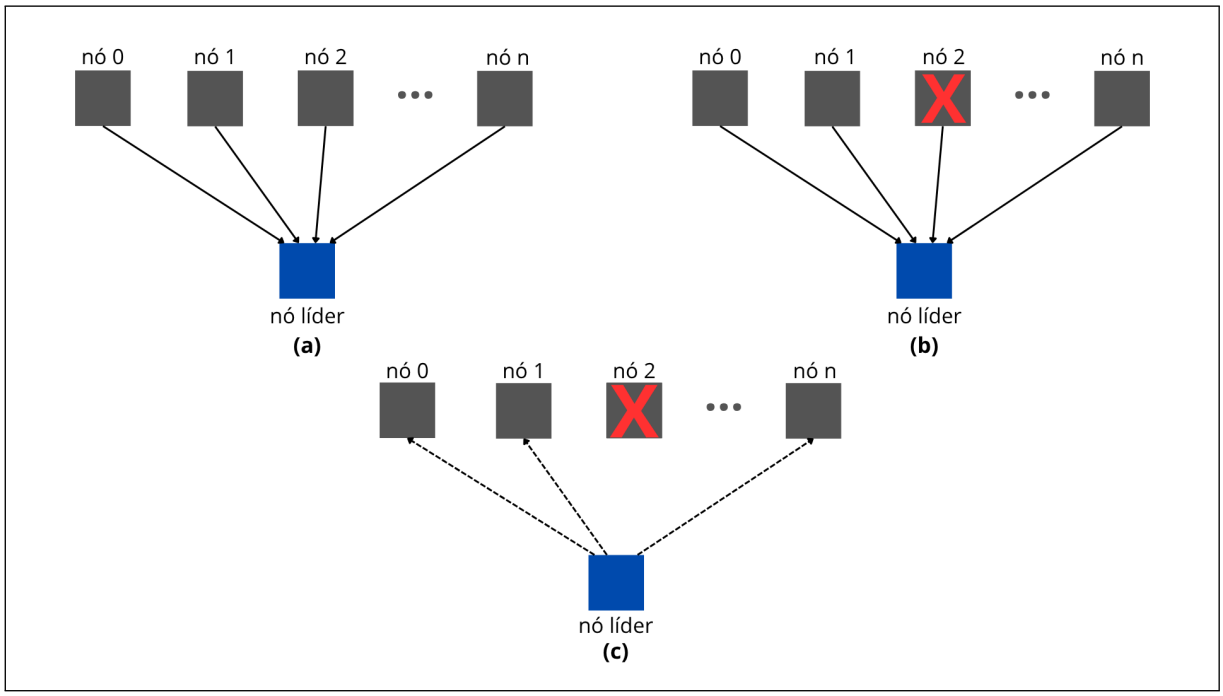
3.3.2 Monitoramento por batimentos cardíacos

O HM corresponde ao envio periódico de sinais entre os nós para detectar rapidamente uma falha ou indisponibilidade de algum componente. A ausência de sinais dentro de um intervalo de tempo pré-estabelecido indica uma possível falha (Chetan; Ranganathan; Campbell,

2005), permitindo que o sistema reaja de forma ágil, redirecionando tarefas, ativando réplicas ou acionando mecanismos de recuperação. Essa abordagem pode ser implementada de duas maneiras, conforme a arquitetura escolhida. Na estrutura centralizada, um nó principal supervisiona os demais nós. Já na forma distribuída, os nós monitoram uns aos outros.

A Figura 5 ilustra, de forma esquemática, o funcionamento do monitoramento centralizado por *heartbeat*, dividido em três estágios temporais. A Figura 5(a) ilustra a operação normal, na qual os nós trabalhadores (do nó 0 ao nó *n*) enviam mensagens periódicas de vitalidade ao nó líder. O estágio da Figura 5(b) representa a interrupção do recebimento do sinal proveniente do nó 2, caracterizando uma suspeita de falha por parte do líder após o esgotamento do tempo limite de espera (*timeout*). Por fim, na Figura 5(c), o nó líder declara formalmente o nó 2 como inativo e emite mensagens de reconfiguração aos nós remanescentes, indicadas pelas setas tracejadas, permitindo que o sistema atualize sua visão global e reaja a essa falha.

Figura 5 – Mecanismo de detecção de falhas baseado em *Heartbeat* com monitoramento centralizado.



Fonte: Autoria própria, (2026)

3.3.3 *User Level Failure Mitigation*

Além das abordagens discutidas, o MPI pode oferecer suporte à tolerância a falhas por meio da interface ULFM, uma interface que permite à própria aplicação lidar com falhas de

processos. O ULFM lida com o modelo de falha do tipo *fail-stop*, no qual um processo interrompe sua execução (por exemplo, devido a falhas de *hardware* ou ao encerramento inesperado do sistema), sem apresentar comportamento arbitrário nem enviar dados incorretos.

Diferentemente do modelo tradicional do MPI, que encerra toda a aplicação quando um processo falha, o ULFM permite que a aplicação detecte a falha por meio de códigos de erro retornados pelas operações MPI e reconfigure a comunicação. Para efetuar a mitigação local, realiza-se o redimensionamento dos comunicadores, eliminando os processos defeituosos (Laguna *et al.*, 2016).

3.3.4 Sobrecarga em técnicas de tolerância a falhas

O tempo de execução paralelo decorrente de qualquer trabalho adicional que não seja realizado pelo programa serial é chamado de sobrecarga (do inglês: *overhead*) (Pacheco; Malensek, 2022). A adição de técnicas de tolerância a falhas, como CR e HM, pode acrescentar uma sobrecarga à execução dos programas de memória distribuída.

No CR, a quantidade de pontos de verificação deve ser definida de modo a minimizar o custo associado à perda de informações em caso de falhas, sem que a sobrecarga decorrente da criação desses pontos se torne significativa (Kalaiselvi; Rajaraman, 2000).

Já no HM, a sobrecarga está principalmente associada ao envio periódico de mensagens adicionais pela rede e ao processamento necessário para o monitoramento dos sinais. Intervalos de envio muito curtos aumentam o tráfego de comunicação e o custo computacional, enquanto intervalos mais longos podem retardar a detecção de falhas.

No ULFM, o custo associado ao redimensionamento dos comunicadores e à redistribuição das tarefas pode ser elevado, aumentando a sobrecarga da aplicação e exige algoritmos de balanceamento de carga mais complexos em tempo de execução (Laguna *et al.*, 2016).

3.4 *Bulk Synchronous Parallel Model*

Aplicações que utilizam o modelo iterativo BSP são muito comuns em ambientes HPC (Laguna *et al.*, 2016). O fluxo de uma aplicação BSP é dividido em uma sequência de *supersteps* (do inglês: super-passo), em que, em cada novo *superstep*, uma tarefa é alocada a um nó computacional, que realizará passos locais de computação, bem como a transmissão e o recebimento de mensagens de outros nós (Valiant, 1990).

Esse modelo facilita o uso de estratégias de tolerância a falhas, como *checkpoint* e *rollback* (Kalaiselvi; Rajaraman, 2000), uma vez que a aplicação pode ser facilmente sincronizada ao fim de cada *superstep*, permitindo aplicar essa estratégia nesse momento de sincronização para a recuperação de dados globais em caso de falha posterior.

3.5 Computação em nuvem

A computação em nuvem consiste na disponibilização de recursos de tecnologia da informação sob demanda, por meio da internet, com cobrança baseada no uso. Em vez de adquirir, possuir e manter *data centers* e servidores físicos (*on-premises*), é possível acessar serviços tecnológicos, tais como capacidade de processamento, armazenamento e bases de dados, de acordo com as necessidades computacionais, por meio de um provedor de computação em nuvem. A computação em nuvem é uma opção comumente utilizada em ambientes HPC justamente por eliminar a necessidade de gerenciar um ambiente *on-premises*.

O custo das instâncias de máquinas virtuais pode variar conforme o modelo de contratação adotado. No modelo sob demanda, os recursos computacionais são provisionados conforme a necessidade, sem compromisso de longo prazo, e cobrados por unidade de tempo de uso, geralmente por segundo ou por hora, a um preço fixo previamente estabelecido pelo provedor. Esse modelo oferece maior previsibilidade e estabilidade operacional, uma vez que as instâncias permanecem ativas até serem explicitamente encerradas pelo usuário.

Por outro lado, no modelo preemptivo, as instâncias possuem um preço mais baixo que as instâncias sob demanda. Essas instâncias podem ser reivindicadas a qualquer momento pelo provedor. Quando isso ocorre, é enviado, cerca de dois minutos antes, um sinal de interrupção indicando que a instância será encerrada, interrompida ou hibernada[4]. Para lidar com essa característica, as aplicações precisam adotar estratégias de tolerância a falhas, como o salvamento periódico do estado, a replicação de dados e o tratamento de sinais de interrupção, garantindo que a execução possa ser retomada com impacto mínimo. Diversos provedores de nuvem oferecem ambientes preemptivos, como *Google Preemptible VM Instances*[5], *AWS EC2 Spot* [4] e *Azure Spot Virtual Machines*[6].

Nesse contexto, especialmente em cenários que utilizam instâncias preemptivas, torna-se fundamental que o ambiente computacional possa ser reconstruído de forma automática e consis-

[4] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html
[5] https://cloud.google.com/compute/docs/instances/spot?hl=pt-br
[6] https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms

tente sempre que uma nova máquina virtual é provisionada. Como essas instâncias podem ser interrompidas e recriadas dinamicamente, depender de configurações manuais comprometeria a reprodutibilidade e aumentaria o tempo de recuperação após falhas. Dessa forma, mecanismos de inicialização automatizada tornam-se essenciais para garantir que cada instância seja configurada de forma padronizada e determinística.

Na AWS, o mecanismo de *User Data* consiste em um *script* ou conjunto de instruções fornecido no momento da criação da instância, executado automaticamente durante a primeira fase de inicialização do sistema operacional. O *User Data* permite automatizar a configuração inicial da máquina virtual, incluindo a instalação de dependências, a configuração de serviços, a definição de variáveis de ambiente e a preparação do ambiente de execução. Dessa forma, elimina-se a necessidade de configuração manual pós-provisionamento. Esse mecanismo é relevante no processo de automação de arquiteturas por meio de ferramentas de IaC.

3.5.1 Infraestrutura como código

É possível automatizar arquiteturas em nuvem por meio de ferramentas de IaC, como o *Terraform*[7], onde são manifestados à ferramenta a configuração e os serviços desejados, e ela lida com todo o processo de provisionamento dos recursos.

O *Terraform* utiliza uma linguagem declarativa denominada *HashiCorp Configuration Language* (HCL), na qual o estado desejado da infraestrutura é descrito por meio de blocos como *resources*, *variables*, *outputs* e *locals*. A ferramenta então calcula um plano de execução e aplica somente as modificações necessárias para atingir o estado especificado.

No contexto do *Terraform*, o bloco *locals* permite definir variáveis locais calculadas ou compostas a partir de outras variáveis e expressões. Diferentemente das variáveis de entrada, parametrizadas externamente, os valores definidos em *locals* são utilizados internamente ao módulo, contribuindo para maior organização, legibilidade e reutilização do código.

O uso de *locals* é particularmente útil para evitar duplicação de trechos complexos de configuração, centralizando definições que podem ser referenciadas múltiplas vezes ao longo do código. Essa prática favorece a manutenibilidade e reduz a probabilidade de inconsistências durante alterações na infraestrutura.

Por exemplo, o bloco *locals* pode ser utilizado para armazenar o *script* de inicialização das instâncias, que posteriormente é injetado no mecanismo de *User Data*. Essa abordagem

[7] https://developer.hashicorp.com/terraform

separa claramente a definição do *script* da definição do recurso computacional, tornando o código mais modular e compreensível.

O Código-fonte 1 exemplifica o uso do *Terraform* para definir e provisionar uma infraestrutura básica na nuvem. Neste exemplo, é criada uma instância EC2 na AWS.

Código-fonte 1 – Exemplo de uso do *Terraform*

```
# main.tf
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "exemplo_servidor" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "Servidor-Terraform"
  }
}
```

3.6 DeLIA

A DeLIA é uma biblioteca desenvolvida em C++ para aplicações BSP que oferece mecanismos configuráveis de tolerância a falhas ao nível de aplicação, incluindo funcionalidades de salvamento de estado, reinicialização a partir do estado salvo, detecção de interrupções de execução e de sinais de terminação em ambientes preemptivos (Santana *et al.*, 2024). A biblioteca oferece suporte ao ULFM, abstraindo a complexidade dos algoritmos de redimensionamento e de redistribuição de tarefas entre os nós. Além disso, são disponibilizadas funcionalidades de replicação, que permitem que os processos compartilhem seus dados locais com os seus processos vizinhos. Assim, no momento do *checkpoint* local, o processo salva os seus dados e os dados do vizinho (Santana, 2024).

A DeLIA possui interfaces de portabilidade para linguagens de alto nível. A DeLIAJ (DeLIA para Julia) e a DeLIApy (DeLIA para *Python*) encapsulam o código em C++ utilizando *wrappers* que servem de ponte para o código C++. Dessa forma, é possível combinar a produtividade de *Julia* e *Python* com a eficiência computacional da implementação original em

C++.

3.7 Portabilidade com *Cython*

O *Cython* é uma linguagem de programação e um compilador que funciona como um superconjunto de *Python*. Ele permite escrever código com sintaxe semelhante à do *Python* (Dalcin *et al.*, 2011), adicionando anotações de tipos estáticos que possibilitam a tradução automática para código C ou C++. Esse código gerado é então compilado como um módulo de extensão, podendo ser importado diretamente em programas *Python*.

Essa abordagem permite combinar a produtividade do *Python* com o desempenho de linguagens compiladas, além de facilitar a integração com bibliotecas escritas em C ou C++. Dessa forma, é possível reutilizar implementações de alto desempenho já existentes nessas linguagens em aplicações desenvolvidas em *Python*.

O *Cython* pode ser utilizado para criar *wrappers* que expõem bibliotecas escritas em C ou C++ para programas *Python*. Esse mecanismo permite desenvolver interfaces que conectam código de alto desempenho, implementado em linguagens compiladas, a aplicações de nível mais alto escritas em *Python*.

O Código-fonte 2 apresenta um exemplo de como o *Cython* atua como uma interface para bibliotecas escritas em C ou C++. Nesse exemplo, o "*cdef extern from*"(Linha 3) é utilizado para importar a assinatura de uma função definida em um arquivo de cabeçalho C externo (*biblioteca.h*). Em seguida, uma função *Python* é declarada (Linha 7) para encapsular e realizar a chamada à função nativa (*calcular_distancia_c*). Durante a compilação, o *Cython* gera automaticamente o código de integração em C, permitindo que a função compilada seja invocada diretamente por *scripts Python* enquanto mantém o desempenho da implementação nativa.

3.8 Inversão de Forma de Onda Completa

A exploração sísmica conta com diversas técnicas para a descoberta de características da subsuperfície terrestre, uma delas é a análise da velocidade de reflexão das ondas sísmicas na subsuperfície. Essa técnica é amplamente adotada pela indústria de óleo e gás. Todo o processo pode ser dividido em três etapas: aquisição, processamento e interpretação.

Na etapa de aquisição, fontes sísmicas, como explosivos (para exploração terrestre) ou canhões de ar (para exploração marítima), são utilizadas para gerar ondas na subsuperfície a

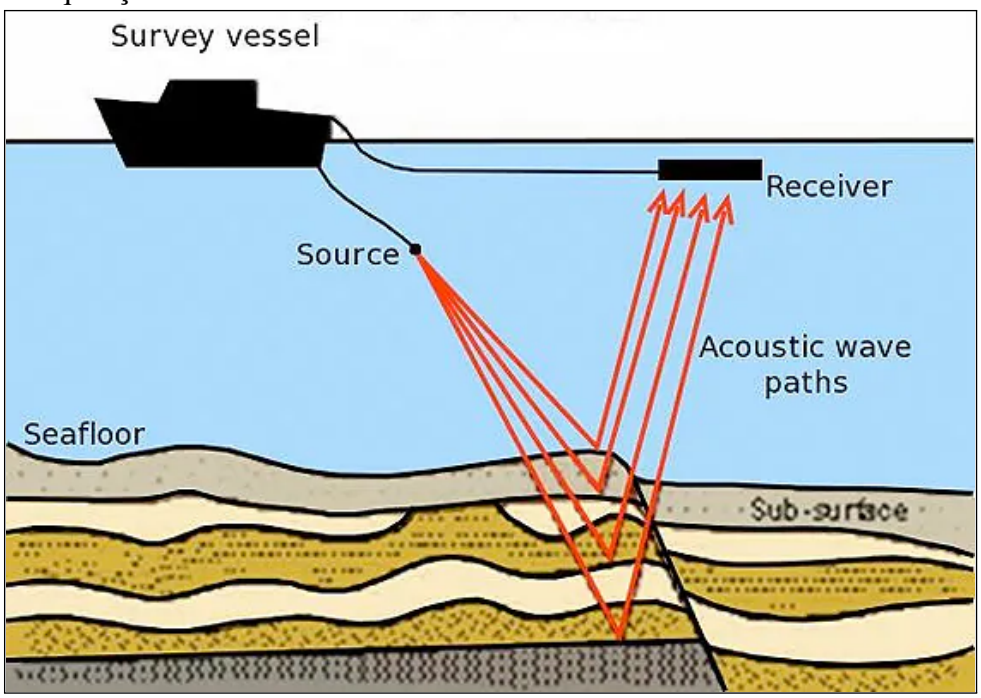
Código-fonte 2 – Exemplo de código Cython

```
# wrapper.pyx
# Importar assinatura da funcao de um arquivo de cabecalho C
cdef extern from "biblioteca.h":
    double calcular_distancia_c(double x1, double y1, double x2, double y2)

# Cria a interface acessivel nativamente pelo Python
def calcular_distancia(double x1, double y1, double x2, double y2):
    return calcular_distancia_c(x1, y1, x2, y2)
```

ser estudada. Ao passarem por um meio onde a impedância acústica difere do meio em que se encontravam, essas ondas são refletidas e capturadas por aparelhos receptores como geofones ou hidrofones (Assis, 2015) para obtenção dos sismogramas. A Figura 6 ilustra como se dá essa etapa na exploração marítima.

Figura 6 – Aquisição sísmica marinha



Fonte: Disponível em: https://zigzag.co.za/seismic-survey-sparks-fear-for-eastern-cape-marine-life-pid12307/

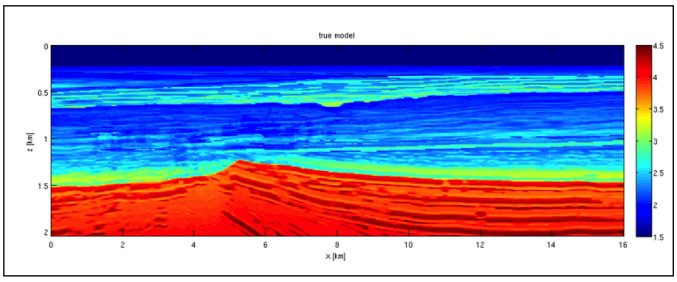
Na etapa de processamento, existem diversos métodos de imageamento da superfície, como as técnicas de migração e de inversão. Dentre elas, a FWI destaca-se por sua capacidade de estimar modelos de velocidades com maior resolução, possibilitando imagens mais fiéis da

subsuperfície (Santos, 2013).

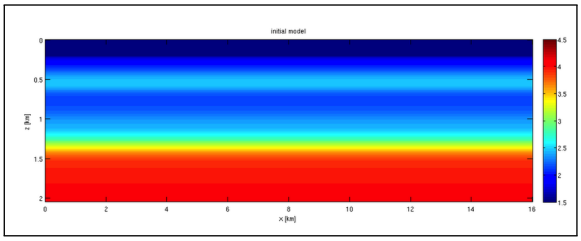
A FWI realiza a modelagem sísmica com o modelo de velocidade utilizando a equação completa da onda, gerando, assim, um dado calculado. Após isso, a FWI tenta minimizar a diferença entre o dado calculado e o dado observado utilizando algum método de otimização, como o método do gradiente. Esse processo ocorre iterativamente; a cada passo, calcula-se a diferença (resíduo) entre o dado modelado (sintético) e o dado observado, gerando um gradiente da função objetivo. Esse gradiente indica a direção na qual o modelo de velocidades deve realizar um autoajuste na iteração seguinte.

A Figura 7 mostra um exemplo de dados de entrada e saída da FWI. A inversão recebe de entrada um modelo de velocidade observado (a) e um modelo de velocidade inicial (b), é nele onde as iterações FWI ocorrerão. Ao final das iterações, é obtido o modelo de velocidade resultante (c).

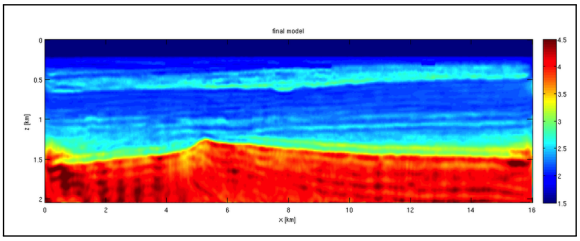
Figura 7 – Comparação entre o modelo de velocidade observado, o modelo inicial e o modelo obtido após a aplicação da FWI



(a) Modelo de velocidade observado



(b) Modelo de velocidade inicial



(c) Modelo de velocidade resultante

Fonte: Adaptado de: https://slim.gatech.edu/research/full-waveform-inversion

Todavia, a necessidade de resolver equações diferenciais em domínios espaciais complexos a cada iteração acarreta um custo computacional elevado. Por essa razão, a viabilidade prática do FWI depende intrinsecamente de sua execução em ambientes de computação de alto desempenho (HPC), exigindo uma paralelização eficiente para a distribuição do processamento.

A implementação de algoritmos de FWI é complexa e exige elevado poder de proces-

samento. Para mitigar esse desafio, ferramentas como o Devito (Louboutin *et al.*, 2019) foram desenvolvidas para otimizar o desempenho dessas rotinas, mantendo a facilidade de uso das linguagens de alto nível. O Devito é um *framework Python* que atua como uma Linguagem de Domínio Específico (DSL) para a resolução de equações diferenciais parciais. A partir de definições simbólicas, ele gera automaticamente código C/C++ otimizado, abstraindo a complexidade estrutural e aplicando paralelismo intrínseco. Isso permite a execução de problemas sísmicos em larga escala, como o FWI, extraindo desempenho melhor do hardware.

4 METODOLOGIA

Este capítulo descreve a metodologia adotada para o desenvolvimento e avaliação da biblioteca proposta. Inicialmente, a Seção 4.1 apresenta a implementação do *wrapper*, responsável por disponibilizar, em *Python*, as funcionalidades da biblioteca DeLIA. Em seguida, nas Seções 4.2 e 4.3, são descritos, respectivamente, os ambientes experimentais utilizados nos testes em nuvem e em supercomputador. Na sequência, a Seção 4.4 detalha como é realizada a validação das funcionalidades implementadas e a Seção 4.5 descreve como ocorre a integração da tolerância a falhas à aplicação FWI. Por fim, na Seção 4.6, são descritos os procedimentos utilizados na análise de desempenho, considerando tanto a capacidade de tolerância a falhas quanto o impacto da biblioteca sobre o tempo de execução das aplicações.

A metodologia adotada possui caráter aplicado e quantitativo. Ela é aplicada, por envolver o desenvolvimento de um *wrapper* para implementar funcionalidades ainda não suportadas. Também é quantitativa, pois será realizada a análise de desempenho do *wrapper*. Entre as métricas avaliadas está a sobrecarga introduzida pela biblioteca. Os experimentos serão realizados em ambientes controlados de injeção de falhas.

4.1 Implementação do *wrapper*

Para o desenvolvimento do *wrapper*, foi utilizado o *Cython*, o mesmo recurso de portabilidade do C++ para *Python* que as funcionalidades implementadas anteriormente utilizam. Nesse processo, foi realizada uma análise da versão anterior do *wrapper* DeLIApy com o objetivo de identificar quais recursos da DeLIA ainda não estavam implementados, permitindo o mapeamento adequado e garantindo a compatibilidade entre os tipos de dados em *Python* e C++.

Por exemplo, no caso de rotinas em C++ que recebem como parâmetro um ponteiro para um endereço de comunicador "*MPI_Comm*", como em "*void DeLIA_setMPIComm(MPI_Comm *comm)*", o *wrapper* teve que adaptar o endereço do comunicador definido em *Python* para um tipo compatível em C++.

Para alcançar tal objetivo, é preciso contornar o fato de que o *Python* não expõe diretamente os endereços de memória das variáveis que o usuário manipula. Em vez disso, cada variável é uma referência a um objeto interno do interpretador, representado por uma estrutura do tipo *PyObject*, que encapsula tanto os dados quanto as informações de controle, como o tipo e a contagem de referências. Esse modelo de abstração impõe a necessidade de adaptação

no *wrapper*, de modo que estruturas internas, como comunicadores MPI, sejam corretamente extraídas ou mapeadas para tipos compatíveis com C++, respeitando o modelo de memória do *Python*. Uma solução possível é utilizar a função "*py2f()*" do *mpi4py*, que converte o endereço *Python* de um comunicador em um endereço do tipo "*Fortran MPI_Fint*". Após isso, utilizando a funcionalidade do MPI "*MPI_Comm_f2c*", o endereço *Fortran* é convertido em um endereço *MPI_Comm* compatível com o C++. O Código-fonte 3 mostra como essa conversão é realizada.

Código-fonte 3 – Conversão de endereços no DeLIApy

```
#DeLIApy.pyx
from mpi4py.libmpi cimport MPI_Comm, MPI_Comm_f2c,
   MPI_Fint
from libc.stdlib cimport malloc
cdef MPI_Comm* global_c_comm_ptr = NULL
cdef extern from "../../include/DeLIA.h":
    cdef void cpp_DeLIA_setMPIComm "DeLIA_setMPIComm"(
       MPI_Comm *comm) except +

def DeLIA_setMPIComm(py_comm):
    global global_c_comm_ptr
    if py_comm is None:
        raise ValueError("py_comm must be an mpi4py.MPI.
           Comm instance, not None")
    if not hasattr(py_comm, 'py2f'):
        raise TypeError("py_comm must be an mpi4py.MPI.Comm
            object")

    #python -> fortran -> c
    cdef MPI_Fint fortran_comm = py_comm.py2f()
    global_c_comm_ptr = <MPI_Comm*>malloc(sizeof(MPI_Comm))
    global_c_comm_ptr[0] = MPI_Comm_f2c(fortran_comm)

    cpp_DeLIA_setMPIComm(global_c_comm_ptr)
```

4.2 Teste em nuvem

Devido à limitação de recursos para esse trabalho, os testes na infraestrutura de nuvem da AWS ocorreram em instâncias EC2 sob demanda do tipo *t3.large*. Essas instâncias possuem 2 vCPUs e 8 GB.

4.2.1 Configuração da arquitetura em nuvem

O ambiente de nuvem foi desenvolvido com IaC por meio do *Terraform*. A inicialização e o gerenciamento remoto dos processos entre as instâncias EC2 são realizados por meio de *Secure Shell* (SSH), permitindo a execução distribuída de aplicações paralelas. Também, em *clusters* de memória distribuída que utilizam MPI, os *jobs* podem ser executados em diferentes nós por meio do SSH[8].

A configuração do SSH para essa conexão remota segura entre nós, por meio de autenticação baseada em chaves públicas e privadas, foi automatizada com o *Terraform*. Utilizar um único par de chaves SSH em todas as instâncias eliminou a necessidade de gerar e distribuir chaves individualmente, simplificando a comunicação e o gerenciamento do *cluster*. O *Terraform* disponibiliza o recurso *tls_private_key*, que permite a geração automatizada de pares de chaves criptográficas que podem ser distribuídos entre diferentes instâncias, conforme exemplificado no Código-fonte 4.

Código-fonte 4 – Uso do recurso *tls_private_key* do Terraform

```
1    resource "tls_private_key" "cluster_key" {
2      algorithm = "RSA"
3      rsa_bits  = 4096
4    }
```

A configuração das instâncias foi realizada por meio de um *shell script*. O *script* utilizado no *User Data* é definido previamente em um bloco *locals*. Essa abordagem permitiu encapsular toda a lógica de inicialização em uma única definição, que, posteriormente, é referenciada na criação da instância.

4.3 Teste em supercomputador

Para execuções de problemas complexos, com grandes malhas e alto volume de dados, relizam-se testes no supercomputador Santos Dumont[9]. Para o escalonamento e execução dos testes, utilizou-se o gerenciador de recursos Slurm, direcionando as cargas de trabalho para a partição "*sequana_cpu_dev*". Essa partição disponibiliza 4 nós computacionais para cada

[8] https://docs.open-mpi.org/en/v5.0.x/launching-apps/ssh.html
[9] https://sdumont.lncc.br/index.php

trabalho (*job*), com limite de 1 trabalho de até 20 minutos de execução por vez.

4.4 Validação das funcioalidades ULFM implementadas

Para testar as funcionalidades implementadas, foi desenvolvida uma aplicação *Python* com as funcionalidades de tolerância a falhas que utilizam o ULFM, que realiza uma operação coletiva de *Allreduce* e, durante a execução, simula-se uma falha em um dos nós para que a DeLIApy possa executar sua rotina de recuperação. A seguir, é detalhada a aplicação utilizada para a validação.

4.4.1 Aplicação *Python*

O Pseudocódigo 5 detalha a implementação desse algoritmo, nele, há a inicialização das variáveis e da DeLIApy (Linhas 1-7). As variáveis "*dado_local*", "*dado_global*"e "*vetor_de_tarefas*"representam, respectivamente, o dado local de cada processo, o dado global da aplicação e a lista de tarefas que cada processo está responsável. A Tabela 1 exemplifica a distribuição de 10 tarefas entre 4 processos, assim cada processo é responsável por um conjunto de tarefas.

Tabela 1 – Distribuição de tarefas entre processos

Processo	Tarefas
0	{0, 1}
1	{2, 3}
2	{4, 5}
3	{6, 7, 8, 9}

Fonte: Autoria própria, (2026)

Após isso, a operação *AllReduce* é executada dentro de um bloco *try-catch* que, em caso de falha, captura o código de erro para ser posteriormente enviado à biblioteca (Linhas 10–15). Se a biblioteca não detectar uma falha (por meio do código de retorno) na chamada de "*DeLIA_CheckProcess(rc)*", a iteração termina (Linhas 16–22). Caso contrário, o DeLIApy executa as funcionalidades de recuperação ULFM, redimensionando o comunicador e redistribuindo as tarefas do nó que falhou entre os processos ativos (Linhas 17–19).

Código-fonte 5 – Pseudocódigo do algoritmo de validação

```
inicializar MPI
obter rank, comm_sz e iterations
inicializar dado_local, dado_global e vetor_de_tarefas
DeLIA_SetTasksData(vetor_de_tarefas)
DeLIA_SetGlobalData(dado_global)
DeLIA_SetLocalData(dado_local)
DeLIA_Replication_Init()

PARA i DE 0 ATE iterations FACA:
    TENTAR:
        MPI_Allreduce(dado_local, dado_global, SOMA)
        rc := MPI.SUCCESS
    CAPTURAR erro:
        rc := codigo_do_erro
    fr := DeLIA_CheckProcess(rc)
    SE fr.hasFailure ENTAO:
        DeLIA_ReduceLostProcessData()
        atualiza comunicador
        MPI_Allreduce(dado_local, dado_global, SOMA)
        #usando novo comunicador
    FIM_SE
FIM_PARA

DeLIA_Replication_Finalize()
DeLIA_Finalize()
Finalizar MPI
```

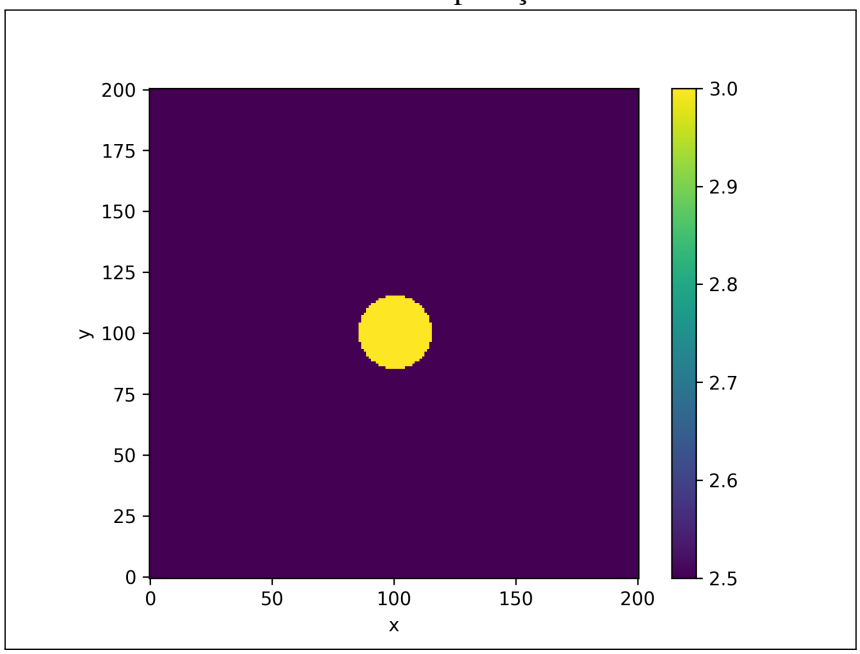
4.5 Integração da biblioteca na aplicação sísmica

Nessa etapa, foram desenvolvidas duas aplicações FWI, uma 2D e outra 3D, utilizando o Devito para a prototipação e a implementação, já que esse *framework* reduz a complexidade da implementação do imageamento sísmico e dos algoritmos de inversão.

A etapa de aquisição sísmica da aplicação 2D foi realizada utilizando um modelo de velocidade observado de *benchmark* 2D com velocidade de fundo de 2,5km/s, com um círculo ao centro com velocidade de 3km/s. A Figura 8 ilustra este modelo de velocidade. O modelo de velocidade inicial utilizado possui velocidade constante de 2,5km/s.

Na etapa de aquisição sísmica da aplicação 3D, utilizou-se um modelo de velocidade observado de *benchmark* 3D com velocidade de fundo constante de 2,5km/s, com uma esfera ao centro, com velocidade de 3km/s. A Figura 9 ilustra este modelo de velocidades. O modelo de

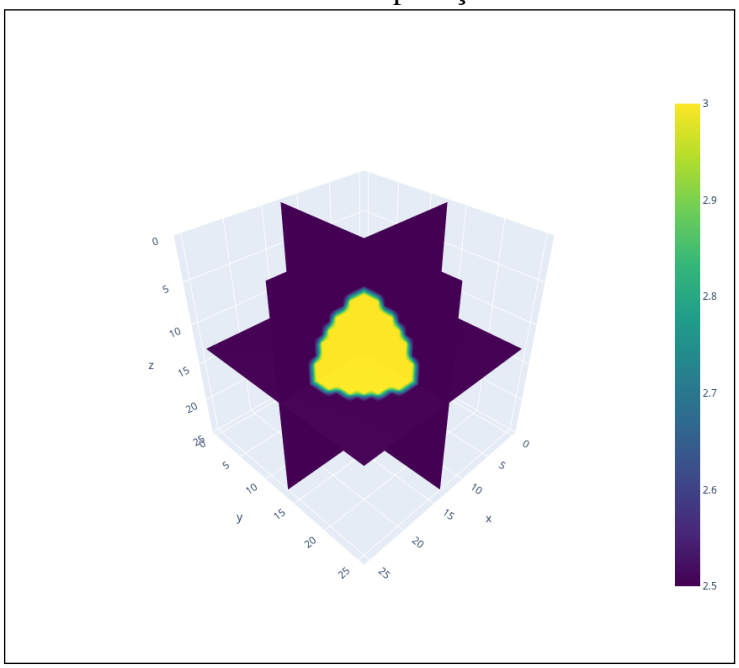
Figura 8 – Modelo de velocidade observado da aplicação 2D



Fonte: Autoria própria, (2026)

velocidade inicial utilizado possui velocidade constante de 2,5km/s.

Figura 9 – Modelo de velocidade observado da aplicação 3D



Fonte: Autoria própria, (2026)

Ambas as aplicações são paralelizadas, distribuindo os tiros sísmicos entre os nós computacionais. Consequentemente, rotinas de comunicação global também são utilizadas para a agregação do gradiente e do valor de desajuste (do inglês: *misfit*), que quantifica o erro entre os dados sísmicos modelados e os observados, guiando a minimização.

A tolerância a falhas foi integrada às aplicações por meio dos mecanismos de *checkpointing* e *rollback* globais e locais da DeLIApy. Como dado global, optou-se por salvar o modelo de velocidades resultante e o gradiente global em iteração na FWI. Assim, o estado macroscópico da otimização será preservado, permitindo que a inversão seja retomada a partir do último passo de atualização persistido em caso de falhas.

Os dados locais referem-se ao progresso individual de cada processo MPI durante o processamento do respetivo subconjunto de tiros. Para isso, foram escolhidos como dados locais o índice *i* do tiro sísmico corrente, o gradiente local acumulado até o tiro *i* e o valor da função objetivo local parcial. O armazenamento dessas variáveis evita o reprocessamento dispendioso das simulações *forward* e *adjoint* de tiros já concluídos por um nó, caso ocorra uma interrupção antes da sincronização global.

Para operacionalizar essa estratégia, foi desenvolvida uma função responsável pela inicialização da biblioteca e pela rotina de recuperação de falhas (*rollback*). O Pseudocódigo 6 mostra como essa função foi implementada. Nela, a DeLIApy é inicializada e as variáveis que correspondem ao estado da aplicação nos escopos local e global são registradas (Linhas 2-4). Depois, é feita a verificação de *rollback*, na qual a biblioteca verifica a persistência dos dados de execuções anteriores (Linha 6). Se houver, os dados globais são recuperados (Linhas 8–12); caso contrário, a aplicação inicia-se a partir do estado inicial (Linhas 13–14).

O Código-fonte 7 mostra o momento em que é feito o *checkpointing* global dos dados. Primeiro, é realizada a *k*-ésima iteração de FWI (Linha 6) e, após a agregação dos valores locais de cada processo MPI, o estado global da aplicação é salvo para garantir a recuperação segura em caso de falhas subsequentes (Linha 13).

O salvamento do estado local é detalhado no Pseudocódigo 8. Nessa etapa, após a inicialização das variáveis e dos dados residuais e sintéticos (Linhas 2-6), a aplicação utiliza a funcionalidade de recuperação de dados locais persistidos no disco (Linha 8). Posteriormente, é analisado se houve dados recuperados pela funcionalidade. Nessa etapa, é comparado se o valor de "*iteracao_local*"(dado configurado no escopo local durante a inicialização do DeLIApy) é zero (Linha 9). Se não for, significa que o DeLIApy recuperou os dados locais persistidos no disco e o valor de "*tiro_atual*", "*objetivo*"e "*gradiente_local*"são atualizados com esses dados (Linhas 10-12). Caso contrário, essas variáveis continuam com o valor que foram inicializadas. Após isso, são realizadas a modelagem direta e o cálculo do gradiente local do *i*-ésimo tiro, seguidas da atualização do valor objetivo parcial, do gradiente local acumulado e das variáveis

Código-fonte 6 – Pseudocódigo de inicialização do DeLIApy

```
1  def my_delia_init():
2      #Inicializar DeLIApy (arquivos de configuracao e
          parametros \acrshort{MPI})
3      #Registrar variaveis globais (modelo_atual,
          gradiente_global_atual)
4      #Registrar variaveis locais (gradiente_local_atual,
          iteracao_local, objetivo_local)
5
6      pode_recuperar := DeLIA_CanRecoverGlobalCheckpointing()
7
8      SE pode_recuperar ENTAO:
9          DeLIA_ReadGlobalData()
10         iteracao_inicial:=DeLIA_getCurrentGlobalIteration()
11         #Reconstruir modelo 3D a partir dos dados
              recuperados
12         #Reconstruir gradiente 3D a partir dos dados
              recuperados
13     SENAO:
14         DeLIA_SaveSettings()
15     FIM_SE
```

locais registradas da DeLIApy (Linhas 13–21). Por fim, realiza-se o registro do *checkpointing* local (Linha 23).

4.6 Análise de desempenho

Para avaliar o desempenho da biblioteca, analisou-se o custo adicional introduzido em termos de tempo de execução decorrente do uso das funcionalidades de salvamento de estado da biblioteca em comparação com a execução original da aplicação. A sobrecarga é obtida a partir da seguinte equação:

$$overhead = \frac{M_{cDeLIAP} - M_{sDeLIAP}}{M_{sDeLIAP}} \times 100$$

Onde $M_{cDeLIAP}$ representa a mediana dos tempos de execução da aplicação com a utilização da biblioteca, enquanto $M_{sDeLIAP}$ corresponde à mediana dos tempos de execução da aplicação sem a biblioteca. A mediana é utilizada como medida representativa do tempo de execução por ser menos sensível a valores extremos (*outliers*), que podem ocorrer em experimentos realizados em ambientes computacionais.

Código-fonte 7 – Adição de tolerância a falhas global no FWI

```
#inicializacao dos modelos e variaveis locais e globais
#distribuicao dos tiros

my_delia_init()
for k in range(start_it, fwi_iterations):
    phi_local, grad_local = fwi_gradient(model0.vp)
    # allreduce global via MPI do gradiente e da funcao
       objetivo
    # ...
    # Atualizacao do modelo
    # ...
    # Sincronizacao entre processos
    # ...
    DeLIApy.DeLIA_SaveGlobalData()
```

Além disso, é calculado o Desvio Padrão Relativo (RSD), utilizado para medir a variabilidade dos tempos de execução entre diferentes execuções de um mesmo experimento. O RSD é obtido pela seguinte equação:

$$RSD = \frac{SD}{AVG} \times 100$$

Onde SD corresponde ao desvio padrão dos tempos de execução e AVG representa a média desses tempos. O RSD fornece uma medida percentual da dispersão dos resultados em relação à média, permitindo avaliar o grau de estabilidade das execuções.

Foram utilizados dois cenários de teste:

1. No primeiro, essas métricas foram avaliadas com base na quantidade de tarefas por nó. Para isso, o tamanho do problema permaneceu fixo e o número de tiros variou.
2. No segundo, as métricas foram avaliadas com base no tamanho do problema. Nesse cenário, nas execuções 3D, as dimensões variaram, bem como o número de tiros e receptores proporcionalmente ao tamanho. Nas execuções 2D, as dimensões variaram, o número de tiros permaneceu fixo e o número de receptores variou de acordo com o tamanho do problema.

   Em ambos, as execuções ocorrerão sem a simulação de falhas.

Código-fonte 8 – Pseudocódigo de adição de tolerância a falhas local no FWI

```
def fwi_gradient(vp_in):
    zerar gradiente_local
    criar estruturas de receptores (residual e dados
       sinteticos)
    objetivo := 0
    tiro_atual := tiro_inicial
    iteracao_local := 0

    DeLIApy.DeLIA_ReadLocalData()
    SE (iteracao_local > 0) ENTAO:
        tiro_atual := tiro_inicial + iteracao_local
        objetivo := objetivo_salvo
        gradiente_local := gradiente_salvo
    FIM_SE
    PARA i DE tiro_atual ATE tiro_final FACA:
        Configurar geometria da fonte para o tiro 'i'
        Realizar simulacao forward
        Calcular o residual
        Acumular erro na funcao objetivo
        Calcular o gradiente_do_tiro

        Acumular gradiente_do_tiro no gradiente_local
        Atualizar iteracao_local e objetivo_local

        DeLIA_SaveLocalData()
    FIM_PARA
    Zerar variaveis locais de controle de estado (iteracao,
        objetivo)
    RETORNAR objetivo, gradiente_local
```

5 RESULTADOS

Este capítulo apresenta os resultados e as análises de desempenho obtidos a partir das execuções das aplicações desenvolvidas, com o objetivo primário de avaliar a viabilidade e o impacto da utilização da biblioteca DeLIApy. A avaliação é dividida em três etapas principais: Primeiramente, na Seção 5.1, analisamos o resultado da execução do teste de validação, a fim de avaliar o comportamento da biblioteca e a integridade dos resultados após a recuperação. Posteriormente, discutimos, na Seção 5.2, os resultados dos testes bidimensionais (FWI 2D) realizados em instâncias EC2 da AWS, analisando a sobrecarga e a estabilidade das execuções ao variar tanto o número de tiros quanto o tamanho do problema. Por fim, detalhamos, nas Seções 5.3 e 5.4, os resultados tridimensionais (FWI 3D), que, devido à sua maior exigência computacional e limitações de recursos nas instâncias em nuvem, demandaram a divisão dos experimentos entre o ambiente AWS e o supercomputador SDumont, respectivamente. Para todos os cenários, são relacionados os tempos de execução com e sem a integração da DeLIApy, utilizando o Desvio Padrão Relativo (RSD) como métrica de estabilidade, além da sobrecarga.

5.1 Teste de Validação

O teste de validação das funcionalidades implementadas aplica as funcionalidades de ULFM em uma operação de comunicação coletiva (*allReduce*). Para isso, a aplicação foi executada com o auxílio de um *shell script*, que automatiza as tarefas de lançamento dos processos e de envio do sinal de terminação *SIGKILL* a um processo.

O Exemplo 9 mostra uma execução do programa com 4 processos sem falhas, esse resultado servirá de base de comparação para o experimento com falhas. Nele é realizada a operação de *AllReduce* na variável "*gdata*".

Código-fonte 9 – Resultado de execução do teste de validação sem falhas

```
1 Sucess in all reduce gdata
2 DATA INT ALL REDUCE JUST 4:
3 _11110_22220_33330_44440_55550
4 _66660_77770_88880_99990_111100
```

O Exemplo 10 mostra o resultado de uma execução com falha do programa com 4

processos. Durante a execução, simulou-se a queda do processo 1 por meio do envio do sinal de terminação *SIGKILL* para o programa utilizar a tolerância a falhas da DeLIApy. Observa-se que a biblioteca, ao detectar a falha no processo 1, redimensiona corretamente o comunicador e atribui novos valores de *rank* aos processos ativos (Linhas 2-7). Ao comparar o resultado do *AllReduce* com e sem falhas, é possível concluir que a redistribuição das tarefas entre os *ranks* ocorreu com sucesso, gerando resultados iguais com 4 e 3 processos.

Código-fonte 10 – Resultado da execução do teste de validação com falhas

```
1  FAILURE IN 1
2  0_PROCESS_1_FAILED
3  0_PROCESS_BEFORE_0_NOW
4  3_PROCESS_1_FAILED
5  3_PROCESS_BEFORE_2_NOW
6  2_PROCESS_1_FAILED
7  2_PROCESS_BEFORE_1_NOW
8
9  DATA INT ALL REDUCE JUST 3:
10 _11110_22220_33330_44440_55550
11 _66660_77770_88880_99990_111100
```

5.2 Execuções da FWI 2D em nuvem

Para todas as execuções FWI 2D foram utilizados quatro nós das instâncias *t3.large* com o MPI v5.0.8 configurado com suporte ULFM e 1 processo por nó. Para cada teste, foram realizadas 10 execuções sem ocorrência de falhas da aplicação com 1 iteração FWI. Os resultados das execuções variando o número de tiros e as execuções variando o tamanho do problema são apresentados nas Subseções a seguir.

5.2.1 Análise com base na quantidade de tarefas por nó

Nesse cenário, o tamanho do problema foi fixado em (201X201) e o número de tiros varia entre {32, 48, 64 e 80}. A Tabela 2 descreve os parâmetros base de simulação utilizados na aplicação, onde *nshots* é o número de tiros, *nreceivers* é o número de receptores, *nx* e *ny* representam o número de pontos da malha computacional nas direções X e Y, respectivamente. O espaçamento dessa malha é dado por *dx* e *dy* (em metros), com a origem do modelo definida

pelas coordenadas *ox* e *oy*. Por fim, a propagação da onda é modelada do tempo inicial *t*0 até o tempo total *tn* (milissegundos), utilizando uma fonte sísmica do tipo *Ricker* com frequência *f*0 (kHz).

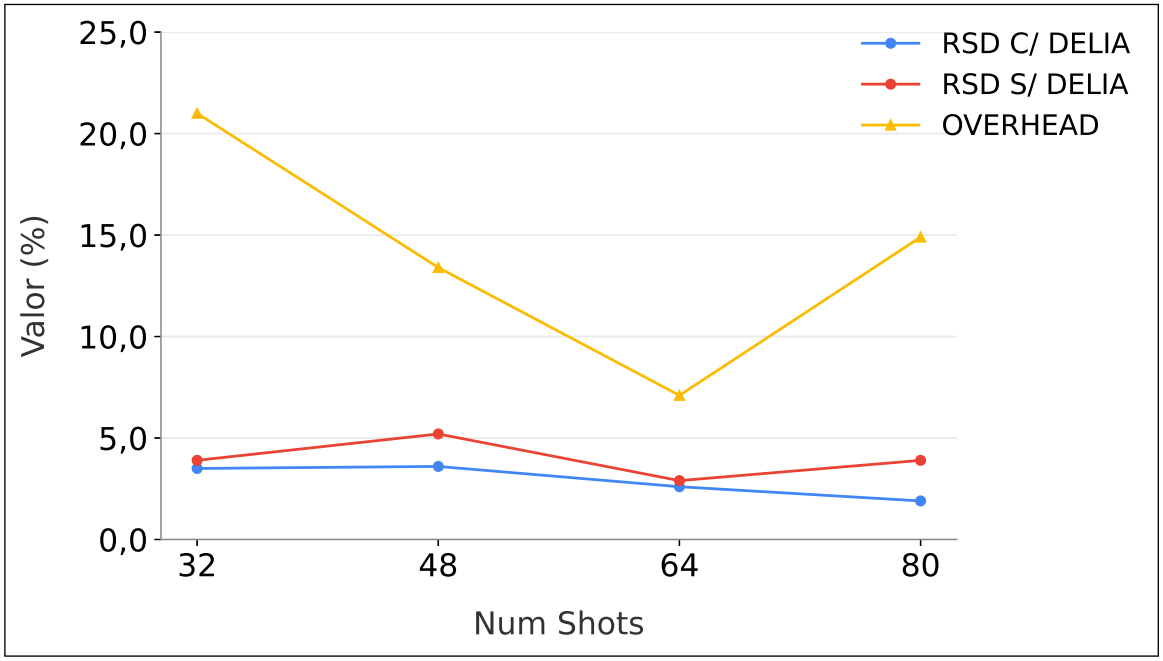
Tabela 2 – Parâmetros de simulação FWI 2D (Caso 1 em instâncias EC2)

Parâmetro	Valor
nshots	<VARIANDO>
nreceivers	201
(*nx*, *ny*)	(201, 201)
(*dx*, *dy*)	(10.0, 10.0)
(*ox*, *oy*)	(0.0, 0.0)
t0	0.0
tn	1000.0
f0	0.010

Fonte: Autoria própria, (2026)

Um resumo dos resultados pode ser observado na Figura 10.

Figura 10 – Sobrecarga e RSD com e sem DeLIApy da FWI 2D variando o número de tiros em instâncias EC2



Fonte: Autoria própria, (2026)

Como mostrado, a sobrecarga introduzida pela biblioteca diminui à medida que a quantidade de tiros aumenta. A análise do RSD revela que a variação nos tempos de execução com a inserção da biblioteca permanece na mesma ordem de grandeza da aplicação original (abaixo de 6%). Isso evidencia que a DeLIApy atua de maneira estável, não alterando o comportamento padrão da aplicação.

5.2.2 Análise com base no tamanho do problema

Nesse cenário, foi fixado o número de tiros em 48, enquanto o tamanho do problema variava em *nx* e *ny* nos valores {(201X201), (401X201), (401X401)}. A Tabela 3 descreve os parâmetros base de simulação utilizados na aplicação.

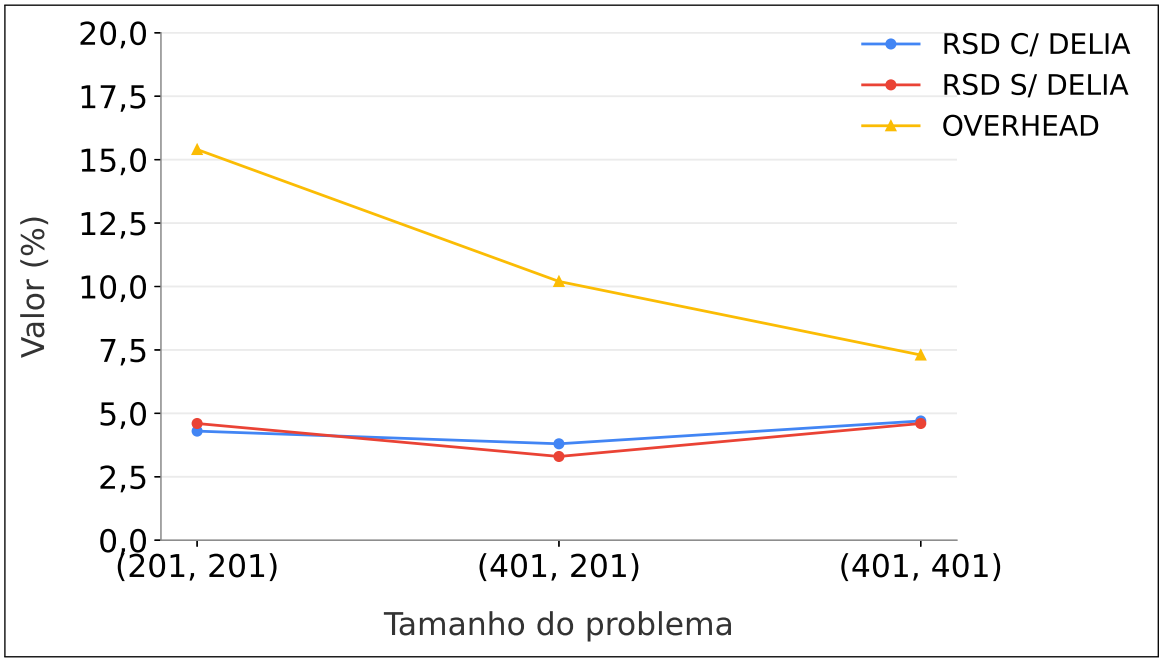
Tabela 3 – Parâmetros de simulação FWI 2D (Caso 2 em instâncias EC2)

Parâmetro	Valor
nshots	48
nreceivers	201
(*nx*, *ny*)	<VARIANDO>
(*dx*, *dy*)	(10.0, 10.0)
(*ox*, *oy*)	(0.0, 0.0)
t0	0.0
tn	1000.0
f0	0.010

Fonte: Autoria própria, (2026)

O resultado da execução está exposto na Figura 11.

Figura 11 – Sobrecarga e RSD com e sem DeLIApy da FWI 2D variando o tamanho do problema em instâncias EC2



Fonte: Autoria própria, (2026)

Como é possível observar, apesar da sobrecarga ser elevada, ela diminui à medida que aumentamos o tamanho do problema. Isso pode indicar que o DeLIApy talvez tenha um melhor desempenho à medida que aumentamos o tamanho do problema. A variação nos tempos de

execução com e sem DeLIApy permanece muito próxima, indicando que a adição da tolerância a falhas na aplicação não altera o comportamento padrão da aplicação à medida que o tamanho do problema aumenta.

5.3 Execuções da FWI 3D em nuvem

Apesar da sobrecarga elevada nas execuções da aplicação 2D na nuvem, é possível observar que ela apresenta uma redução quando o tamanho do problema aumenta. Para analisar esse comportamento, foram realizadas execuções da aplicação 3D na nuvem AWS. Para todas as execuções foram utilizados quatro nós das instâncias *t3.large* com o MPI v5.0.8 configurado com suporte ULFM e 1 processo por nó. Foram realizadas 10 execuções sem ocorrência de falhas da aplicação com 1 iteração FWI. Os resultados das execuções variando o número de tiros e as execuções variando o tamanho do problema são apresentados nas Subseções a seguir.

5.3.1 Análise com base na quantidade de tarefas por nó

Nesse cenário, o tamanho do problema foi fixado em (26X26X26) e o número de tiros varia entre {36, 49, 64 e 81}. A Tabela 4 descreve os parâmetros base de simulação utilizados na aplicação.

Tabela 4 – Parâmetros de simulação FWI 3D (Caso 1 em instâncias EC2)

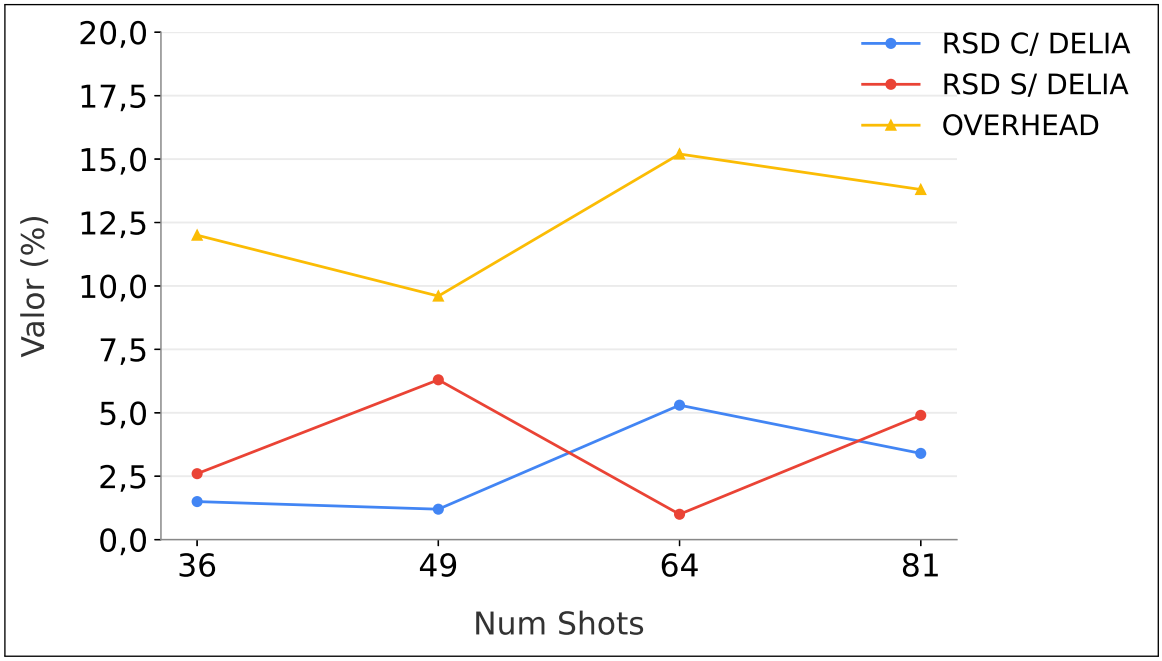
Parâmetro	Valor
nshots	<VARIANDO>
nreceivers	36
(*nx*, *ny*, *nz*)	(26, 26, 26)
(*dx*, *dy*, *dz*)	(10.0, 10.0, 10.0)
(*ox*, *oy*, *oz*)	(0.0, 0.0, 0.0)
t0	0.0
tn	1000.0
f0	0.010

Fonte: Autoria própria, (2026)

Os resultados das execuções estão expostos na Figura 12.

Com esses resultados, é possível observar que, para o tamanho de problema dado, a sobrecarga aumenta à medida que aumentamos o número de tiros. Esse aumento ocorre pelo motivo de o tamanho do problema ser pequeno, fazendo com que o custo de computação das operações numéricas não consiga superar o custo de orquestração da biblioteca. Isso reforça o argumento de que a biblioteca possui uma sobrecarga menor para tamanhos de problema

Figura 12 – Sobrecarga e RSD com e sem DeLIApy da FWI 3D variando o número de tiros em instâncias EC2



Fonte: Autoria própria, (2026)

maiores.

O RSD com e sem DeLIApy evidencia uma oscilação significativa ao longo do intervalo de valores observado. Por ocorrer tanto com quanto sem DeLIApy, isso pode indicar que essa oscilação é decorrente de instabilidade do ambiente de execução (como sobrecarga de rede).

5.3.2 Análise com base no tamanho do problema

Nesse cenário, o tamanho do problema variou em *nx*, *ny* e *nz* entre os valores: {(26X26X26), (51X26X26), (51X51X26) e (51X51X51)}. O tamanho de problema máximo para as execuções em nuvem foi (51X51X51) por limitação de recursos das instâncias para esse trabalho. Nesse conjunto de testes, o número de tiros e receptores variou proporcionalmente ao tamanho do problema, onde cada fonte possui um espaçamento de 120m entre elas e os receptores são colocados com uma distância de 50m entre eles, como mostra a Tabela 5.

Tabela 5 – Valores de *num_shots* e *num_receivers* por dimensão nas execuções em instâncias EC2

Dimensões	*num_shots*	*num_receivers*
26×26×26	9	36
51×26×26	15	66
51×51×26	25	121
51×51×51	25	121

Fonte: Autoria própria, (2026)

A Tabela 6 descreve os parâmetros base de simulação utilizados na aplicação.

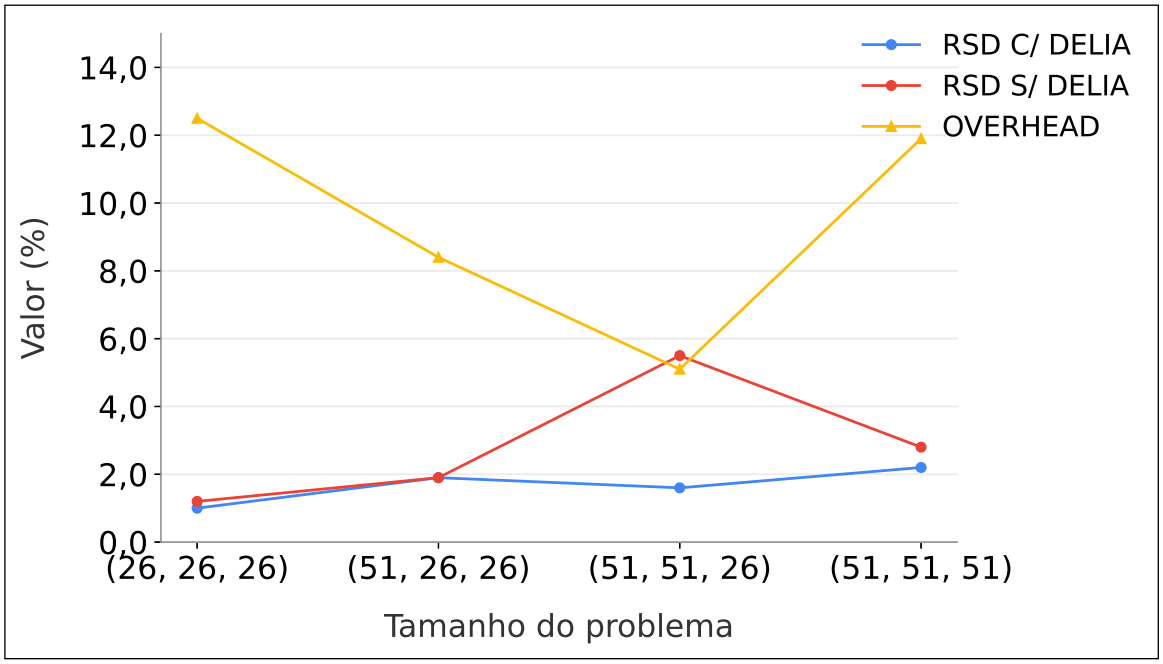
Tabela 6 – Parâmetros de simulação FWI 3D (Caso 2 em instâncias EC2)

Parâmetro	Valor
nshots	<VARIANDO>
nreceivers	<VARIANDO>
(*nx*, *ny*, *nz*)	<VARIANDO>
(*dx*, *dy*, *dz*)	(10.0, 10.0, 10.0)
(*ox*, *oy*, *oz*)	(0.0, 0.0, 0.0)
t0	0.0
tn	1000.0
f0	0.010

Fonte: Autoria própria, (2026)

Os resultados são mostrados na Figura 13.

Figura 13 – Sobrecarga e RSD com e sem DeLIApy da FWI 3D variando o tamanho do problema em instâncias EC2



Fonte: Autoria própria, (2026)

Nesses resultados, é possível observar que o RSD com e sem DeLIApy permanece muito próximo, indicando que, a biblioteca não introduz instabilidade na execução da FWI. A sobrecarga diminui à medida que o tamanho do problema aumenta, com uma redução de 12,5% para 5,1% no intervalo de (26X26X26) a (51X51X26).

5.4 Execuções da FWI 3D em um supercomputador

As execuções da aplicação 3D na nuvem mostram que o comportamento da sobrecarga se mantém semelhante à aplicação 2D: à medida que aumentamos o tamanho do problema, a sobrecarga introduzida pela biblioteca diminui. Para analisar se esse comportamento se mantém em tamanhos de problema ainda maiores, foram realizados testes no supercomputador Santos Dumont, que possibilita o uso de malhas computacionais maiores que as instâncias disponibilizadas para esse trabalho. Para os testes no supercomputador, foram utilizados 4 nós exclusivos da fila "*sequana_cpu_dev*", nos quais foi executado 1 processo por nó. O Código-fonte 11 detalha a configuração de ambiente escolhida.

Código-fonte 11 – Configuração de ambiente SLURM para testes

```
#Numero de Nos
#SBATCH --nodes=4
#Numero de tarefas por No
#SBATCH --ntasks-per-node=1
#Numero total de tarefas MPI
#SBATCH --ntasks=4
#Numero de threads
#SBATCH --cpus-per-task=48
#Fila (partition) a ser utilizada
#SBATCH -p sequana_cpu_dev
#Nome job
#SBATCH -J exec
#Utilizacao exclusiva dos nos durante a execucao do job
#SBATCH --exclusive
```

Foram realizadas 10 execuções sem ocorrência de falhas da aplicação com 1 iteração FWI. Os resultados são discutidos nas Subseções a seguir.

5.4.0.1 Análise com base na quantidade de tarefas por nó

Nesse cenário, o tamanho do problema foi fixado em ($101X101X101$), a maior dimensão da malha que foi possível executar sem atingir o tempo limite da fila. O número de tiros varia entre {49, 64, 81 e 100}.

A Tabela 7 descreve os parâmetros base de simulação utilizados na aplicação durante as execuções.

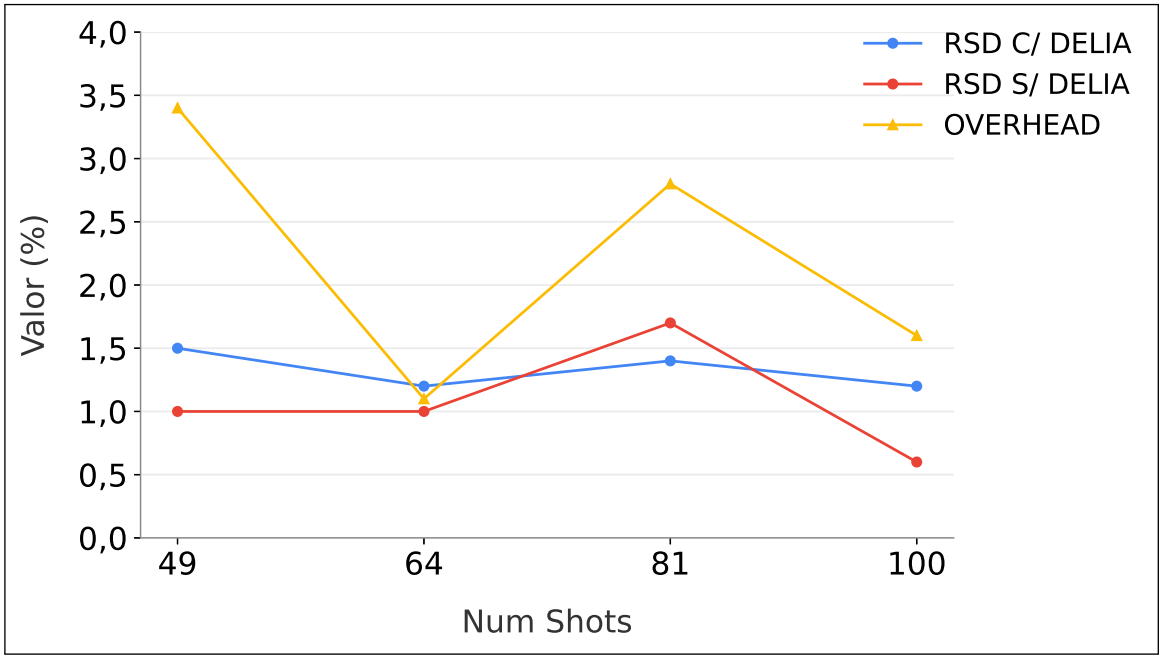
Tabela 7 – Parâmetros de simulação FWI 3D (Caso 1 em supercomputador)

Parâmetro	Valor
nshots	<VARIANDO>
nreceivers	441
(*nx*, *ny*, *nz*)	(101, 101, 101)
(*dx*, *dy*, *dz*)	(10.0, 10.0, 10.0)
(*ox*, *oy*, *oz*)	(0.0, 0.0, 0.0)
t0	0.0
tn	1000.0
f0	0.010

Fonte: Autoria própria, (2026)

Os resultados são mostrados na Figura 14.

Figura 14 – Sobrecarga e RSD com e sem DeLIApy da FWI 3D variando o número de tiros em supercomputador



Fonte: Autoria própria, (2026)

Com esse resultado, analisamos que a sobrecarga introduzida pela DeLIApy é baixa, com pico em 3,4%. Isso indica que o custo computacional da tolerância a falhas se dispersa com a computação dos tiros sísmicos. O RSD indica que a biblioteca não causa interferência no comportamento das execuções da aplicação.

5.4.1 Análise com base no tamanho do problema

Nesse cenário, o tamanho do problema variou em *nx*, *ny* e *nz* entre os valores: {(76*X*51*X*51), (76*X*76*X*51), (76*X*76*X*76), (101*X*76*X*76), (101*X*101*X*76) e (101*X*101*X*101)}. Nesse conjunto

de testes, o número de tiros e receptores varia proporcionalmente ao tamanho do problema. Cada fonte possui um espaçamento de 120m entre elas e os receptores são colocados com uma distância de 50m entre eles, como mostra a Tabela 8.

Tabela 8 – Valores de *num_shots* e *num_receivers* por dimensão nas execuções em supercomputador

Dimensões	*num_shots*	*num_receivers*
76×51×51	35	176
76×76×51	49	256
76×76×76	63	256
101×76×76	63	336
101×101×76	81	441
101×101×101	81	441

Fonte: Autoria própria, (2026)

A Tabela 9 descreve os parâmetros base de simulação utilizados na aplicação.

Tabela 9 – Parâmetros de simulação FWI 3D (Caso 2 em supecomputador)

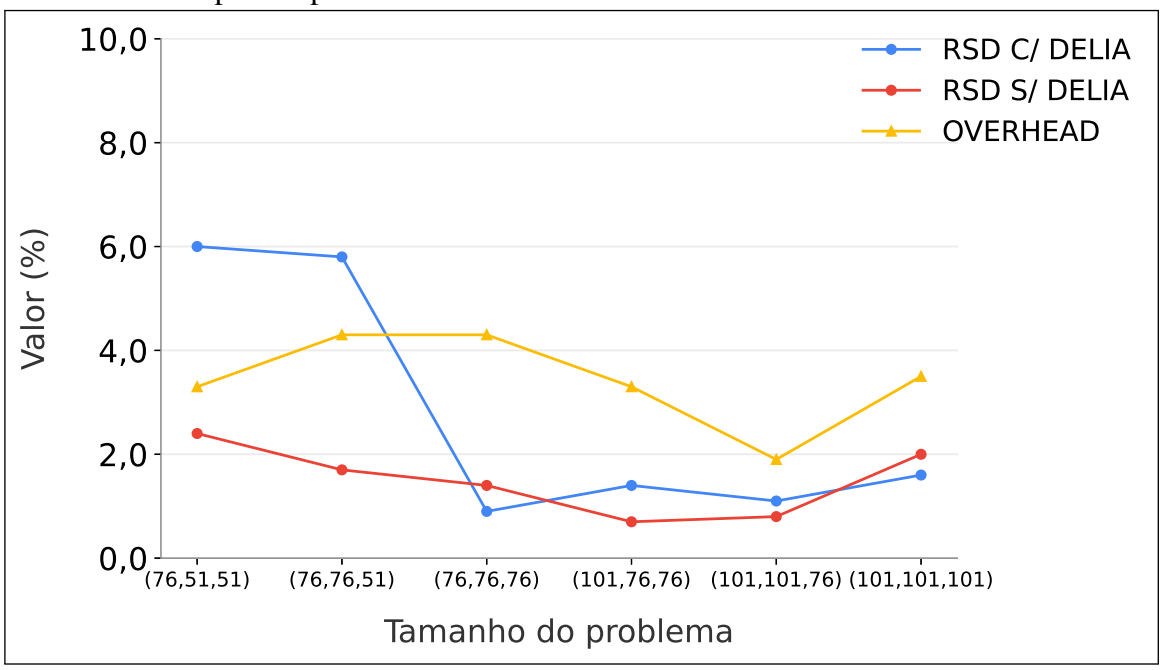
Parâmetro	Valor
nshots	<VARIANDO>
nreceivers	<VARIANDO>
(*nx*, *ny*, *nz*)	<VARIANDO>
(*dx*, *dy*, *dz*)	(10.0, 10.0, 10.0)
(*ox*, *oy*, *oz*)	(0.0, 0.0, 0.0)
t0	0.0
tn	1000.0
f0	0.010

Fonte: Autoria própria, (2026)

Os resultados são mostrados na Figura 15

Esses resultados fortalecem o argumento de que a sobrecarga da biblioteca diminui à medida que o tamanho do problema cresce, já que no menor cenário (76X51X51) o custo computacional da biblioteca é de apenas 3,3%. Além disso, os valores de RSD com e sem DeLIApy reforçam o argumento de que a biblioteca não causa instabilidade ou variações imprevisíveis na execução da aplicação.

Figura 15 – Sobrecarga e RSD com e sem DeLIApy da FWI 3D variando o tamanho do problema em supercomputador



Fonte: Autoria própria, (2026)

6 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo expõe as considerações finais do trabalho realizado, abordando a viabilidade da biblioteca em aplicações de memória distribuída com a linguagem Python, além de apresentar, na Seção 6.1, sugestões para pesquisas futuras.

O trabalho apresentou o desenvolvimento da biblioteca de tolerância a falhas DeLIApy. A biblioteca é destinada a aplicações BSP de memória distribuída desenvolvidas em *Python*. Foi realizada a análise do desempenho dessa biblioteca em uma aplicação sísmica de inversão de forma de onda completa (FWI), desenvolvida utilizando o *framework* Devito.

Os resultados dos testes de validação mostram que as funcionalidades implementadas realizam corretamente o redimensionamento do comunicador e agregação dos dados, gerando, assim, resultados válidos. Dessa forma, a DeLIApy consegue melhorar a resiliência de aplicações de memória distribuída.

Além disso, a avaliação de desempenho mostra, através do RSD, que a biblioteca não causa instabilidade ou variações imprevisíveis na execução das aplicações. Adicionalmente, a análise da sobrecarga introduzida nas aplicações FWI desenvolvidas em *Python* revela que a DeLIApy apresenta um bom desempenho à medida que o tamanho do problema cresce. Na AWS, a sobrecarga é alta devido ao tamanho das malhas computacionais ser pequeno, fazendo com que o custo dos cálculos numéricos não consiga superar o custo de orquestração da biblioteca, porém nota-se que a sobrecarga diminui à medida que a dimensão da malha aumenta. Isso ocorre porque, para os tamanhos de problema utilizados, à medida que o volume de dados sísmicos e a malha computacional aumentam, o custo das operações numéricas domina o tempo total de execução. Consequentemente, o custo das rotinas de tolerância a falhas torna-se percentualmente baixo, diluindo o impacto da biblioteca no desempenho global.

6.1 Trabalhos Futuros

Para trabalhos futuros, pode ser analisada a sobrecarga adicional do custo de recuperação da biblioteca após um cenário de falha utilizando os mecanismos de reinicialização a partir do estado salvo (*rollback*). Além disso, a biblioteca pode ser analisada em aplicações FWI com tamanho de problemas maiores, com mais quantidade de nós computacionais e em ambientes preemptivos de nuvem.

Também, pode-se avaliar o desempenho de outros métodos de tolerância a falha disponi-

bilizados pela DeLIApy, como as rotinas ULFM.

A DeLIApy pode também disponibilizar tolerância a falhas em outras aplicações paralelas. Um exemplo são as aplicações de treinamento de modelos de Inteligência Artificial (IA). Essas aplicações demandam alto poder computacional e podem ser paralelizadas para distribuir a carga de trabalho entre diversos nós computacionais. Essa distribuição aumenta a probabilidade de falhas isoladas de hardware ou rede interromperem todo o processo. Dessa forma, técnicas de tolerância a falhas fazem-se estritamente necessárias para garantir a resiliência da aplicação e evitar a perda de progresso durante o treinamento.

O desenvolvimento da versão Julia das interfaces de portabilidade também pode ser realizado, uma vez que possui a mesma limitação que o *wrapper Python* possuía em sua versão anterior: as funcionalidades que utilizam o ULFM não estão presentes nessa versão.

REFERÊNCIAS

ASSIS, Í. A. S. de. **Um algoritmo paralelo eficiente de migração reversa no tempo (rtm) 3d com granularidade fina**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 01 2015. Disponível em: https://repositorio.ufrn.br/jspui/handle/123456789/19828.

CHARD, K. *et al.* Extended abstract: Productive parallel programming with parsl. **Ada Lett.**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 2, p. 73–75, abr. 2021. ISSN 1094-3641. Disponível em: https://doi.org/10.1145/3463478.3463486.

CHETAN, S.; RANGANATHAN, A.; CAMPBELL, R. Towards fault tolerant pervasive computing. **IEEE Technology and Society Magazine**, v. 24, n. 1, p. 38 – 44, 2005. Disponível em: https://www.scopus.com/inward/record.uri?eid=2-s2.0-17444391683&doi=10.1109%2fMTAS.2005.1407746&partnerID=40&md5=a47e7ca11593280a2ef2a9d13cba037e.

DALCIN, L. *et al.* Cython: The Best of Both Worlds . **Computing in Science & Engineering**, IEEE Computer Society, v. 13, n. 02, p. 31–39, mar. 2011. ISSN 1558-366X. Disponível em: https://doi.ieeecomputersociety.org/10.1109/MCSE.2010.118.

ELECTRICAL, I. of; ENGINEERS, E. Ieee standard for information technology–portable operating system interface (posix(tm)) base specifications, issue 7. **IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)**, p. 1–3951, 2018.

IRIGOYEN, M. *et al.* Deliap e deliaj: Interfaces de biblioteca de dependabilidade em pad para python e julia. In: **Anais da VI Escola Regional de Alto Desempenho da Região Nordeste**. Porto Alegre, RS, Brasil: SBC, 2025. p. 1–4. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/erad-ne/article/view/36388.

KALAISELVI, S.; RAJARAMAN, V. A survey of checkpointing algorithms for parallel and distributed computers. **Sadhana**, v. 25, n. 5, p. 489–510, 10 2000.

LAGUNA, I. *et al.* Evaluating and extending user-level fault tolerance in mpi applications. **The International Journal of High Performance Computing Applications**, v. 30, n. 3, p. 305–319, 2016. Disponível em: https://doi.org/10.1177/1094342015623623.

LOUBOUTIN, M. *et al.* Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. **Geoscientific Model Development**, v. 12, n. 3, p. 1165–1187, 2019. Disponível em: https://www.geosci-model-dev.net/12/1165/2019/.

MESSAGE Passing Interface Forum. **MPI: a message-passing interface standard version 3.1**. [S.l.], 2015. Disponível em: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

PACHECO, P.; MALENSEK, M. **An Introduction to Parallel Programming**. 2. ed. [S.l.]: Katey Birtcher, 2022. ISBN 978-0-12-804605-0.

RAUBER, T.; RUNGER, G. **Parallel Programming For Multicore and Cluster Systems**. 2. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 2010. ISBN 978-3-642-04817-3.

ROJAS, E. *et al.* Understanding failures through the lifetime of a top-level supercomputer. **Journal of Parallel and Distributed Computing**, v. 154, p. 27–41, 2021. ISSN 0743-7315. Disponível em: https://www.sciencedirect.com/science/article/pii/S0743731521000782.

SANTANA, C. *et al.* Delia: A dependability library for iterative applications applied to parallel geophysical problems. Elsevier, v. 191, p. 105662, 2024.

SANTANA, C. d. S. **A configurable dependability library for high-performance computing iterative applications with interruption detection, data preservation and failover capabilities**. 90 p. Tese (Tese (Doutorado em Engenharia Elétrica e de Computação)) — Universidade Federal do Rio Grande do Norte, Natal, 2024. Orientador: Dr. Samuel Xavier de Souza.

SANTOS, A. W. G. dos. **Inversão de forma de Onda Aplicada á Análise de Velocidades Sísmicas Utilizando uma Abordagem Multiescala.** Dissertação (Mestrado) — Universidade Federal da Bahia, 11 2013. Disponível em: https://oasisbr.ibict.br/vufind/Record/BRCRIS_cbf89a2cb50b6e17df0d3c7f12e2db1e.

STEEN, M. van; TANENBAUM, A. S. **Distributed Systems**. 4. ed. [S.l.]: Marteen van Steen, 2023. v. 2. ISBN 978-90-815406-4-3.

TARANTOLA, A. Inversion of seismic reflection data in the acoustic approximation. **Geophysics**, v. 49, n. 8, p. 1259–1266, 08 1984. ISSN 0016-8033. Disponível em: https://doi.org/10.1190/1.1441754.

TERESHCHENKO, D.; GANKEVICH, I. Distributed fault-tolerant computing with sbn-python on a real company case. **9th International Conference "Distributed Computing and Grid Technologies in Science and Education"**, 2021.

VALIANT, L. G. A bridging model for parallel computation. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 8, p. 103–111, ago. 1990. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/79173.79181.