DeLIA: A Dependability Library for Iterative Applications applied to parallel geophysical problems

Carla Santana, Ramon C.F. Araújo, Idalmis Milian Sardina, Ítalo A.S. Assis, Tiago Barros, Calebe P Bianchini, Antonio D de S Oliveira, João M de Araújo, Hervé Chauris, Claude Tadonki, et al.

Computers & Geosciences 191 (2024) 105662

Contents lists available at ScienceDirect

Computers and Geosciences

journal homepage: www.elsevier.com/locate/cageo

Research paper

DeLIA: A Dependability Library for Iterative Applications applied to parallel geophysical problems

Carla Santana [a,*], Ramon C.F. Araújo [a], Idalmis Milian Sardina [a], Ítalo A.S. Assis [c], Tiago Barros [a], Calebe P. Bianchini [d], Antonio D. de S. Oliveira [a], João M. de Araújo [a], Hervé Chauris [b], Claude Tadonki [b], Samuel Xavier-de-Souza [a]

[a] Universidade Federal do Rio Grande do Norte, Brazil
[b] Mines Paris - PSL, France
[c] Universidade Federal Rural do Semi-Árido, Brazil
[d] Universidade Presbiteriana Mackenzie, Brazil

ARTICLE INFO

Keywords:
Fault tolerance
Fault detection
Checkpointing
Heartbeat monitoring
High-performance computing
Full-waveform inversion

ABSTRACT

Many geophysical imaging applications, such as full-waveform inversion, often rely on high-performance computing to meet their demanding computational requirements. The failure of a subset of computer nodes during the execution of such applications can have a significant impact, as it may take several days or even weeks to recover the lost computation. To mitigate the consequences of these failures, it is crucial to employ effective fault tolerance techniques that do not introduce substantial overhead or hinder code optimization efforts. This paper addresses the primary research challenge of developing fault tolerance techniques with minimal impact on execution and optimization. To achieve this, we propose DeLIA, a Dependability Library for Iterative Applications designed for parallel programs that require data synchronization among all processes to maintain a globally consistent state after each iteration. DeLIA efficiently performs checkpointing and rollback of both the application's global state and each process's local state. Furthermore, DeLIA incorporates interruption detection mechanisms. One of the key advantages of DeLIA is its flexibility, allowing users to configure various parameters such as checkpointing frequency, selection of data to be saved, and the specific fault tolerance techniques to be applied. To validate the effectiveness of DeLIA, we applied it to a 3D full-waveform inversion code and conducted experiments to measure its overhead under different configurations using two workload schedulers. We also analyzed its behavior in preemptive circumstances. Our experiments revealed a maximum overhead of 8.8%, and DeLIA demonstrated its capability to detect termination signals and save the state of nodes in preemptive scenarios. Overall, the results of our study demonstrate the suitability of DeLIA to provide fault tolerance for iterative parallel applications.

1. Introduction

High-performance computing (HPC) has revolutionized research involving complex calculations and large datasets, such as geophysical methods. Large-scale supercomputers are equipped with numerous components, making them capable of solving complex problems. However, the size and complexity of these systems also contribute to increased failures. Consequently, it is essential to employ techniques that reduce the effects of interruptions or failures in the system. One strategy is using a fault-tolerance solution that ensures the reliability of the overall system and the user applications (Rojas et al., 2021).

In supercomputers, the components are organized into nodes, each of which can have its own set of processing cores, memory hierarchies, and other resources. These nodes are distributed throughout the system, and communication among them often occurs through the use of the message-passing interface (MPI). When a node fails during the execution of an application, the error can propagate and lead to a global failure of the entire execution. To address this issue, programmers must implement fault tolerance (FT) mechanisms at the application level. This involves adopting application-specific strategies to minimize the adverse impact on the system's performance when faults occur (Weber, 2003). However, a significant challenge lies in developing FT techniques that do not introduce excessive overhead, as it is crucial to maintain the system's efficiency during normal operations. Overhead refers to the additional resource allocation (time, memory, and processing cycles) due to introducing some capability apart from the main application, which in this case is fault tolerance.

* Corresponding author.
E-mail address: carla.santana.058@ufrn.edu.br (C. Santana).

https://doi.org/10.1016/j.cageo.2024.105662
Received 22 November 2023; Received in revised form 30 March 2024; Accepted 20 June 2024
Available online 25 June 2024

Cloud computing has emerged as a popular option for utilizing HPC environments, providing an alternative to traditional on-premise clusters. Many research efforts in the HPC field are focused on assessing the cost–benefit analysis of migrating resource-intensive applications from on-premise environments to public cloud platforms (Netto et al., 2018). One cost-effective approach is using preemptive cloud instances, which are less expensive but can be reclaimed by the cloud provider at any time. Major cloud providers, such as Amazon EC2 Spot (AWS, 2023), Microsoft Azure Spot (Microsoft Azure, 2023), Google Cloud Preemptible (Google Cloud, 2023), and Oracle Preemptible Instances (Oracle Cloud, 2023), offer this type of instance, which utilizes spare computing resources. In such a scenario, employing FT methods to detect notifications before a cloud instance is reclaimed and save the application's state is essential. By employing FT techniques, researchers can ensure that meaningful progress is not lost when a preemptive instance is taken away. This would allow for the seamless resumption of computation from the last saved state, minimizing the effects of disruptions and maximizing the utilization of cloud resources.

Researchers have extensively explored the application of FT techniques in HPC applications, with checkpointing and rollback (CR) being the most commonly used methods. CR involves saving the state of an application at predetermined intervals. In the event of a failure, the application can resume execution from the last saved checkpoint. In distributed systems, researchers can save each process's state in local checkpoints or save the entire application state in global checkpoints (Kalaiselvi and Rajaraman, 2000).

Another critical area of investigation is failure detection, where heartbeat monitoring (HM) is commonly employed. This approach involves processes periodically sending heartbeat messages to a leader node. If the leader node does not receive the expected heartbeat messages from a particular node within a specific time interval, it is assumed that the node has failed (Chetan et al., 2005).

The implementation details of each FT technique can vary among researchers. For example, in their geophysical study, Gonçalves et al. (2011) applied interruption detection directly using MPI. On the other hand, Kayum et al. (2020) and Okita et al. (2022) implemented interruption detection in conjunction with a workload scheduler. A workload scheduler is a management software that allocates and distributes the application's work between different processes in a balanced manner to obtain better execution.

Typically, FT methods are designed to be integrated into specific problem domains or incorporated into workload schedulers. However, Bautista-Gomez et al. (2011) proposed an FT library that provides checkpointing and rollback functionality from the application level. Bland et al. (2013) applied interruption detection using MPI routines, and similarly, Laguna et al. (2016) presented a tool to give the MPI state checkpointing and rollback. These approaches offer more general-purpose solutions for implementing FT techniques.

Considering the specific requirements and constraints of various applications and the need for FT in the face of increasingly less reliable supercomputers or preemptive circumstances, providing a solution that minimizes the implementation effort is highly desirable. FT HPC techniques are usually attached to an application or workload scheduler, offering data recovery but not interruption detection or the opposite. To address this, we propose the dependability library for iterative applications (DeLIA), which provides functionality for saving the application state and detecting potential interruptions. DeLIA is designed explicitly for bulk synchronous programs (BSP), characterized by data synchronization among all processes, resulting in a globally consistent state after each iteration (Valiant, 1990). Many scientific methods that leverage HPC exhibit this behavior, including the geophysical method of full-waveform inversion (FWI), which we have chosen as a case study.

By utilizing DeLIA, BSP application developers can seamlessly integrate fault tolerance capabilities into their programs without spending significant time and effort on implementation. The library enables the saving of the application state. It provides interruption detection, ensuring that the execution can resume from the last saved state in the event of failures or interruptions. DeLIA offers a user-friendly and efficient solution for incorporating fault tolerance in BSP applications, providing researchers with a streamlined approach to safeguarding their work and mitigating the impact of interruptions in HPC environments. The application developers can adapt the parameters and features they will use.

The main features and contributions of this study to fault tolerance in HPC applications field are:

- High configurability: DeLIA offers extensive configuration options, allowing users to select which library functionalities they want to utilize in their applications, which critical data should be saved, and some important parameters such as the frequency to save.
- Adaptability to diverse BSP: DeLIA is designed to be adaptable to different BSPs with various workload schedulers, making it applicable to a wide range of scientific methods.
- Checkpoint and rollback functionality: DeLIA saves both the application's global state and the local state of individual processes, ensuring that the execution can be resumed from the last saved state.
- Interruption detection: DeLIA incorporates interruption detection mechanisms, including detecting termination signals sent by preemptive instances and heartbeat monitoring. These features help identify potential failures or interruptions in the execution of the application.
- Low communication overhead: In DeLIA, communication requirements are handled by a dedicated thread to minimize interference with the application's behavior. To reduce the possibility of conflict with the existing MPI communication in the application, we implemented communication for heartbeat monitoring using user datagram protocol (UDP) instead of MPI. This protocol ensures a lighter communication mechanism because it does not require a message-delivery guarantee.

The remaining sections of this paper are organized as follows: Section 2 provides a detailed description of the proposed library, highlighting its main features. Section 3 focuses on the 3D FWI method, which serves as a case study for evaluating DeLIA. Section 4 presents the experimental results obtained from applying DeLIA in a 3D FWI scenario and their analysis. Section 5 discusses related works in the field of fault tolerance in HPC applications. Finally, Section 6 concludes the paper by summarizing the contributions and suggesting avenues for future research.

2. DeLIA

This study developed DeLIA using C++ for BSP applications structured according to Fig. 1. BSP problems are iterative, where processes perform tasks on their local data during each iteration. At the end of each iteration, all processes synchronize to update the application's global state.

The primary focus of DeLIA is to address fail-stop failures, which refer to interruptions in the execution of the application caused by hardware faults or specific software faults (Herault and Robert, 2015). These failures result in the abrupt termination of the application. Additionally, DeLIA is designed to handle preemptive scenarios, which are circumstances where the execution of the application is interrupted, even though it is not due to a failure.

By providing fault tolerance capabilities for both fail-stop failures and preemptive scenarios, DeLIA enhances the resilience of BSP applications. It allows for the detection and recovery from interruptions, ensuring the continuity of the application's execution and preserving the global or local state.

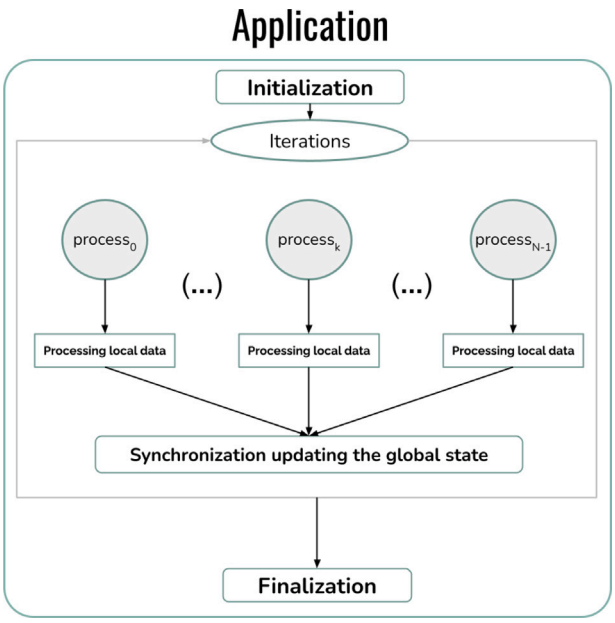

Fig. 1. Schematic of the behavior of the target application for using DeLIA.

DeLIA offers an application programming interface (API) that enables developers to incorporate FT features into their software. This API allows users to invoke DeLIA functions, abstract the library implementation, and define the main parameters of DeLIA in a JavaScript object notation (JSON) file. JSON was chosen for parameter input because it is user-friendly, lightweight, and supported by many libraries.

2.1. Interruption detection

Detecting a possible application failure is necessary for applying techniques to save and recover the data. In our library, interruption detection is performed via heartbeat monitoring (Chetan et al., 2005) and detection of termination signal.

2.1.1. Heartbeat messages

In the heartbeat monitoring system, the fault detector node receives periodic heartbeat messages from every other node. If it fails to receive these messages within a specific interval, it concludes that the node has failed. Avoiding overwhelming the network with excessive heartbeat messages when employing this method is crucial (Chetan et al., 2005).

DeLIA creates a thread in the nodes responsible for sending and receiving the messages. The leader node (also defined as a fault detector) receives the heartbeat messages from each node. These messages are sent in a defined interval (`SLEEP_THREAD_TIME`). If the leader node does not receive any heartbeat message from a given node within a maximum waiting time (`TIME_MAX_WAIT`), it assumes that this node failed (Fig. 2(b)). The leader then sends a trigger to the other nodes, which respond by saving their local data to disk (Fig. 2(c)). Unfortunately, we cannot detect this failure if the leader node fails.

The heartbeat monitoring communication was implemented using the UDP networking protocol in addition to MPI. This choice was made because the applications already utilize MPI, and to avoid any potential conflicts with the version used in the application, we preferred to implement our communication more generically. Furthermore, the UDP protocol incurs less overhead as it is a lighter form of communication that does not guarantee message delivery, unlike transmission control protocol (TCP), which ensures message delivery but has more overhead. Additionally, there is no issue with losing some messages during heartbeat monitoring.

The application developer specifies the `SLEEP_THREAD_TIME` and `TIME_MAX_WAIT` through the parameter file. `TIME_MAX_WAIT` should be more significant than `SLEEP_THREAD_TIME` to increase the chances of receiving one message. If this configuration is not respected, the nodes will spend more time (`SLEEP_THREAD_TIME`) to send the messages than the maximum tolerated by the leader (`TIME_MAX_WAIT`), so the leader will assume a failure occurred. In this situation, the nodes will receive a trigger and save their local data unnecessarily. As the data will be saved by a thread created for the HM, it is expected that the time spent saving this data, even if unnecessarily, does not considerably impact the application's performance.

It is important to emphasize that heartbeat monitoring in DeLIA is primarily intended to verify the operational status of the nodes rather than scheduling tasks. The responsibility of task scheduling lies with the application itself. This approach offers greater flexibility as the application can choose a workload scheduler that effectively achieves load balancing according to its specific requirements.

2.1.2. Termination signals

Supercomputers and cloud systems utilize termination signals to notify a job that it should terminate for some reason, such as system failures, exceeding the allocated execution time, or resource reclamation in preemptive environments. However, with the integration of fault-tolerance mechanisms, an application can safeguard critical data, thus preventing the loss of progress during execution. DeLIA can detect these termination signals and respond by triggering the nodes to save their local data. This ensures that even in the event of termination, the application's progress and data are preserved, allowing for seamless recovery and continued execution when possible. By implementing this capability, DeLIA enhances the dependability of applications in supercomputing and cloud environments, offering a robust solution for handling failures and interruptions effectively.

2.2. Checkpointing and rollback recovery

The main challenge in checkpointing is determining which data to save and how often (frequency) to save. Data size is critical because the overhead of saving and loading should be less than the recomputation time avoided. Frequency is another critical parameter, considering the higher the frequency, the higher the overhead (Kalaiselvi and Rajaraman, 2000).

Each application has its details, and because of that, the developer should know which critical data to save. DeLIA allows users to pass data directly to the library, which will save and read it. However, if the application is already implemented with object-oriented programming, the developer can implement DeLIA's interfaces related to the critical data.

DeLIA provides a solution for scenarios where an experiment is run with one configuration, producing a checkpoint, and subsequent execution is performed with a different configuration. The application cannot recover the last checkpoint because it was created using different settings. To address this issue, DeLIA allows the application to verify the settings of the last checkpoint before attempting to recover the data. The user can send the data corresponding to the settings to enable this functionality.

If, in execution, the application is interrupted and saves its global data, in the subsequent execution with the same settings, when the application calls the method `DeLIA_ReadGlobalData`, DeLIA will recover the global data. During synchronization, the application calls the function `DeLIA_SaveGlobalData`, which saves the global data in the disk. DeLIA can also write and read data from each process; in the case of the local data, each process can check if its local data can be recovered and read them. DeLIA writes the local data through the thread created to send the fault detection trigger.

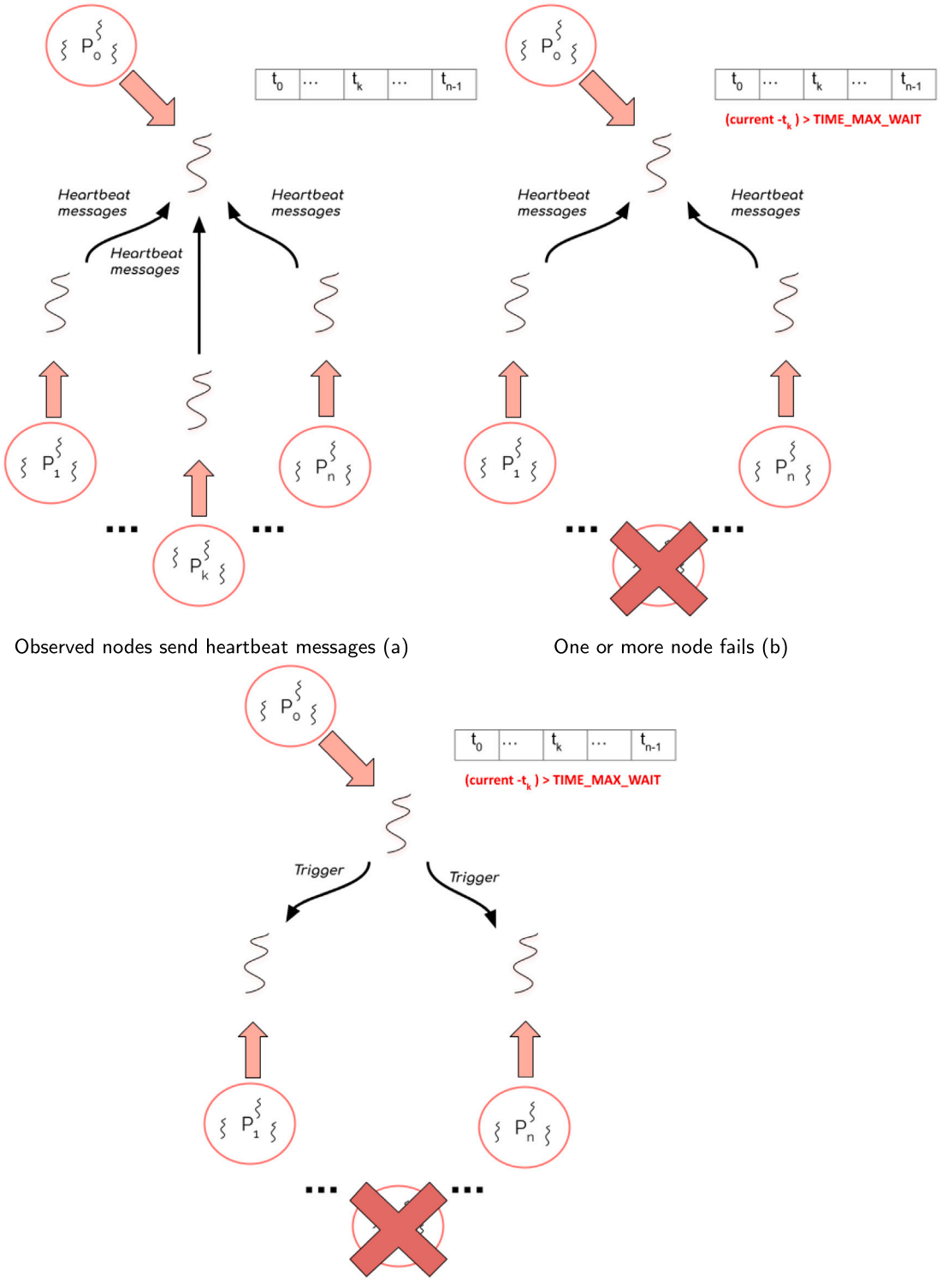

Fig. 2. Diagram of how heartbeat monitoring communication works.

3. Case study: 3D full waveform inversion

The case study to be analyzed is the integration of DeLIA with the FWI, a standard algorithm in geophysics processing for oil and gas exploration. This application is an ideal example of validating the library because it generates significant data, has built-in global data synchronization, and is used frequently in academia and the oil and gas industry (Virieux and Operto, 2009).

FWI can be implemented using shared memory and distributed memory. In this work, we parallelized the wave propagation with shared memory as Barros et al. (2018). For distributed memory, we use two different workload schedulers: decentralized static (DS) (Santana

et al., 2019) and the cyclic token-based work-stealing (CTWS) (Assis et al., 2019). DS scheduling distributes the task equally among the process, and CTWS can redistribute some tasks dynamically after a static distribution of tasks.

3.1. FWI definition

FWI is a numerical optimization technique to extract quantitative information from seismograms (Virieux and Operto, 2009). The main objective is to determine the velocity model vector $\mathbf{v}^*$ by solving the equation

$$\mathbf{v}^* = \arg\min_{\mathbf{v}} \|\mathcal{L}(\mathbf{v}) - \mathbf{d}\|_2^2, \tag{1}$$

where the vector $\mathbf{d}$ represents the seismic data obtained in the field (observed data), consisting of several seismograms recorded from different seismic shots, $\mathcal{L}(\mathbf{v})$ is the operator that represents the artificial modeling process of data using numerical methods to simulate the propagation of seismic waves. This process generates seismograms based on a predetermined velocity model ($\mathbf{v}$). (Virieux et al., 2009).

Eq. (1) minimizes the misfit f which is calculated as

$$f = \|\mathcal{L}(\mathbf{v}) - \mathbf{d}\|_2^2. \tag{2}$$

During the FWI algorithm, $\mathbf{v}$ is iteratively updated. The final velocity model should effectively represent the subsurface velocity. The updated of $\mathbf{v}$ applies the quasi-Newton optimization method, as follows:

$$\mathbf{v}_{j+1} = \mathbf{v}_j - \alpha_j \mathbf{H}_j^{-1} \mathbf{g}_j, \tag{3}$$

where the velocity model for the next iteration is updated using the gradient. Here, $j \geq 0$ represents the iteration number, $\mathbf{v}_j$ is the velocity model vector at the jth iteration, α_j is the step size in the direction of the gradient, $\mathbf{g}_j$ is the gradient at the jth iteration, and $\mathbf{H}_j^{-1}$ is an approximation of the inverse of the Hessian matrix.

Notably, in our implementation of FWI, the gradient $\mathbf{g}_j$ is computed using the adjoint state method (Plessix, 2006). The gradient is calculated after computing the forward propagation and the backpropagation. In this computation, it is possible to apply the optimal checkpointing proposed by Symes (2007), where part of the forward propagation wavefield is saved in the disk and reused in the backpropagation. We highlight that the optimal checkpointing proposed by Symes (2007) focuses on resolving the space problem in the main memory, and DeLIA's checkpointing is focused on fault tolerance.

To calculate $\mathbf{H}_j^{-1}$, we utilize the limited-memory bounded Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS-b) optimization method (Nocedal, 1980) with the library proposed by Zhu et al. (1997). The main parameters provided to L-BFGS-b include the function to minimize (f), the vector to be updated ($\mathbf{v}_j$), and the gradient ($\mathbf{g}_j$).

3.2. FWI algorithm

Next, we describe the algorithm of the FWI method (Algorithm 1) using the notation shown in Table 1. The algorithm begins by reading the observed seismic data of each shot and the initial model $\mathbf{v}_0$. We defined the list of shots for each node $\mathbf{T}_p$ as the list of tasks. Each of the P processes receives approximately the same number of shots. If FWI is employing CTWS, it can adjust the $\mathbf{T}_p$ dynamically because, first, CTWS allocates the tasks equally, and when a process stays idle, it will try to steal tasks from an overloaded process.

Every process calculates the misfit (f_j^p) and the gradient ($\mathbf{g}_j^p$) corresponding to their shots. We generate $\mathbf{g}_j$ and f_j at the end of the jth iteration, summing all f_j^p and $\mathbf{g}_j^p$. Next, we update the velocity model with L-BFGS-b using $\mathbf{g}_j$ and f_j. If the algorithm has converged, the current velocity model is the best fit; otherwise, we execute another iteration of FWI until the convergence is reached or up to maximum number of iterations.

Table 1
Notation for the FWI.

S	number of shots,
s	index for the shot number ($s = 0, \ldots, S-1$),
J	number of iterations,
j	index for the iteration number ($j = 0, \ldots, J-1$),
f_j	misfit from the iteration j,
f_j^s	partial misfit of shot s from the iteration j,
$\mathbf{v}_j$	velocity model from the iteration j,
$\mathbf{g}_j$	gradient from the iteration j,
$\mathbf{g}_j^s$	partial gradient of shot s from the iteration j,
P	number of processes,
p	process index, i.e. rank ($p = 0, \ldots, P-1$),
$\mathbf{T}_p$	list of tasks (shots) for the process p,
$\mathcal{L}(\mathbf{v})_j^s$	modeled data corresponding to the velocity model of the iteration j for the shot s,
$\mathbf{t}_p$	list of tasks already processed for the process p,
$\mathbf{g}_j^p$	partial gradient of process p from the iteration j,
f_j^p	partial misfit of process p from the iteration j,

Algorithm 1 Main steps of one process in FWI parallel

```
1: Read v_0 and d
2: for (J iterations) do
3:     for (T_p shots) do        ▷ If CTWS is applied T_p can be updated dynamically
4:         Generate L(v)_j^s
5:         Compute the misfit f_j^s
6:         Compute the gradient g_j^s
7:         f_j^p += f_j^s
8:         g_j^p += g_j^s
9:     end for
10:    Compute g_j and f_j
11:    Compute v_{j+1} from v_j using the L-BFGS-b library
12:    if v_{j+1} has converged then
13:        break
14:    end if
15: end for
```

3.3. Applying DeLIA to the FWI

As explained in Section 2, we need to identify the essential data in FWI, to the settings, global checkpointing, and local checkpointing. For the settings, we must save parameters such as the velocity model size, the number of iterations, and the number of shots that we can change between the runs. The setting data must be saved only in the initialization of the FWI.

Global checkpointing should save the critical data of an FWI iteration, which are essential to the next iteration. In the case of the FWI, the data are the total gradient ($\mathbf{g}_j$) and misfit (f_j); The updated velocity model in this iteration ($\mathbf{v}_{j+1}$); L-BFGS-b library parameters as iteration number, bounds and double precision.

The local checkpointing must save the data that diverges between the nodes, which is vital for its state. To the FWI, the local data are the partial gradient ($\mathbf{g}_j^s$), misfit (f_j^s), and the list of shots that are already processed ($\mathbf{t}_p$). We omit the wavefield in the local checkpointing process because data saving needs to be quick, and the wavefield contains significant data that would take longer to save. Additionally, we should add more information on the wavefield at each time step to ensure the return of the state during rollback, and it would require saving and reading a large amount of data, which would be time-consuming.

When using DeLIA in an application, the first step is to initialize the library by calling the function `DeLIA_Init` (Line 2 of Algorithm 2). This function requires the parameter file of the library as input,

which contains information such as TIME_MAX_WAIT and the folder where the data should be saved. After the initialization of DeLIA, we should check if there are global data to be recovered by calling the function DeLIA_CanRecoverGlobalCheckpointing; if so, we read the last global data from the disk in the function DeLIA_ReadGlobalData and start the FWI from the last iteration; if not, we save the current settings in the disk using the function DeLIA_SaveSettings (Lines 3 to 7 of Algorithm 2). Then, we call the function to start the heartbeat monitoring DeLIA_HeartbeatMonitoring_Init (Line 8 of Algorithm 2), and if some node fails, the other nodes will receive a trigger. At the end of the FWI, we call DeLIA_HeartbeatMonitoring_Finalize to finish the threads created for the heartbeat monitoring in DeLIA (Line 27 of Algorithm 2).

Before computing the partial gradient and misfit, each process checks if it can read local data by calling the function DeLIA_CanRecoverLocalCheckpointing. If the local data is available, the process reads its last local data and starts processing from the next shot that has not been processed yet with the function DeLIA_ReadLocalData. On the other hand, if the local data is not available, the process will process all shots. In our experiments, DeLIA saves the local data when a process receives a trigger, ensuring that the progress is recorded and can be resumed. This behavior is captured in Lines 10 to 12 of Algorithm 2.

At the end of each FWI iteration, we utilize the function DeLIA_SaveGlobalData to save the global state of the iteration (Line 25 of Algorithm 2). These data are saved in accordance with the specified limits of iterations and time–frequency, as specified in the parameter file. This ensures that the global state is appropriately recorded and can be accessed as needed.

Algorithm 2 Main steps of one process in FWI parallel using DeLIA

```
1: Read v_0 and d
2: DeLIA_Init
3: if DeLIA_CanRecoverGlobalCheckpointing then
4:     DeLIA_ReadGlobalData                          ▷ Update J and v_0
5: else
6:     DeLIA_SaveSettings
7: end if
8: DeLIA_HeartbeatMonitoring_Init
9: for (J iterations) do
10:    if DeLIA_CanRecoverLocalCheckpointing then
11:        DeLIA_ReadLocalData                       ▷ Update t_p, T_p, g_j^p and f_j^p
12:    end if
13:    for (T_p shots) do
14:        Generate L(v)_j^s
15:        Compute the misfit f_j^s and the gradient g_j^s
16:        f_j^p += f_j^s
17:        g_j^p += g_j^s
18:        Put p on the t_p
19:    end for
20:    Obtain the total gradient and misfit
21:    Generate the v_{j+1} updates the v_j using the L-BFGS-b library
22:    if v_{j+1} has converged then
23:        break
24:    end if
25:    DeLIA_SaveGlobalData
26: end for
27: DeLIA_HeartbeatMonitoring_Finalize
28: DeLIA_Finalize
```

4. Experiments and results

We carried out validation and overhead analyses in experiments with the 3D FWI application integrated with DeLIA. Experiments were performed on the supercomputer located at the High-Performance Computing Center (NPAD) at the Federal University of Rio Grande do Norte, and also on the preemptive environment Amazon Cloud Spot. Each NPAD compute node has 128 GB RAM DDR4 2133.

In a supercomputing environment such as NPAD, the execution of the application can be interrupted earlier than expected when there is a failure in a node (or due to some hardware or software error specifically in that node and the application ends at the moment when the other nodes try to communicate with the fail node), connection error, or the execution exceeded the time limit that the supercomputer made available to the user. In the case of AWS Spot, if the execution takes too long, the probability of interruption increases due to the essence of preemptible instances. FWI typically runs a lot of data and uses a lot of resources, so there are possibilities for interruptions in both environments. By performing FWI in these environments using DeLIA, we can analyze the overhead generated by the library to check it does not hinder the HPC optimizations that have already been applied.

We investigate this overhead under two workload schedulers, DS and CTWS, and examine the application's behavior when used with the library in preemptive scenarios. In Section 4.1, we conduct experiments using data from an estimated velocity model obtained from the Gato do Mato (GdM) oil field in the Santos Basin, Brazil, where Shell Brazil operates (Lopez et al., 2020).

We are considering the overhead and the relative standard deviation (RSD) for the first analysis. The overhead is defined by:

$$overhead = \frac{M_{woDeLIA} - M_{wDeLIA}}{M_{wDeLIA}} \times 100, \tag{4}$$

where $M_{woDeLIA}$ and M_{wDeLIA} are the median runtime of a 3D FWI without and with DeLIA, respectively. RSD is defined by:

$$RSD = \frac{SD}{AVG} \times 100, \tag{5}$$

where AVG is the average time of the executions, and SD is the standard deviation. The RSD shows how dispersed the execution times are. If the overhead is less than the RSD of experiments of FWI without DeLIA, then we consider that the result is adequate for that experiment.

The same application was executed in preemptive circumstances as explained in Section 4.2. We executed in the situation of a typical supercomputer that will finish the job when the execution time exceeds the requested time and in a preemptive cloud environment. The goal of the experiments is to verify whether a program integrated with DeLIA behaves as expected in such a scenario.

For all experiments, the number of local data is the same as the nodes used in each investigation, and the number of measured seismograms (observed data) is the same as the shots. For the experiments in the NPAD, DeLIA saved the global and local data in the global file system, which all nodes have access to.

4.1. Analysis of overhead

We executed the FWI application with GdM to test the DeLIA with a bigger problem. We use the segment of the velocity model shown in Fig. 3 to generate the observed data for FWI (more details of FWI configuration are presented in Annex B). For this experiment, the data sizes were: global data 806.4 MB, each local data 28.8 MB, each observed data 49.9 MB, and velocity model 28.8 MB. For each iteration, DeLIA saves the global data (806.4 MB), and for each trigger in each alive node, DeLIA saves the local data (28.8 MB). We executed six iterations of FWI. The library configuration was saving the global data in each iteration, using the interruption detection with TIME_MAX_WAIT = 60 and SLEEP_THREAD_TIME = 1, and saving the local data after a trigger.

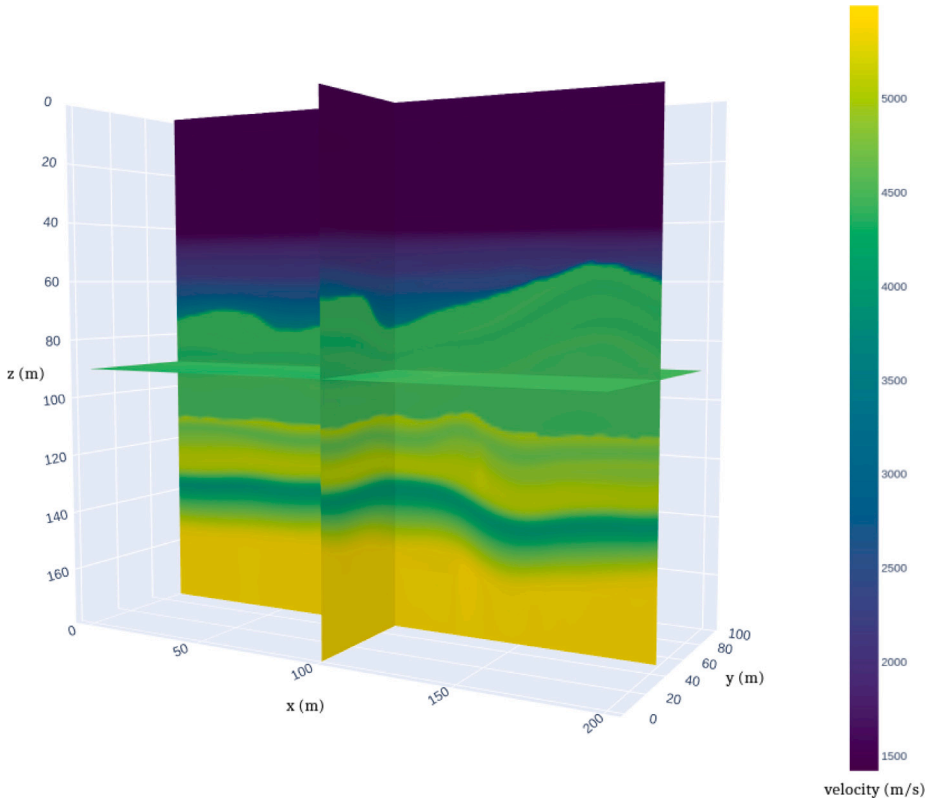

Fig. 3. Velocity model of Gato do Mato oil & gas field.

Table 2
Analysis of FWI using DS.

	4 nodes 8 shots	8 nodes 8 shots	8 nodes 32 shots
DeLIA overhead	2.54%	8.8%	3.84%
RSD without DeLIA	6.88%	3.37%	3.78%
RSD with DeLIA	17.40%	7.20%	5.14%

Table 3
Analysis of FWI with GdM using CTWS using 32 shots.

	4 nodes	8 nodes
DeLIA overhead	2.27%	2.68%
RSD without DeLIA	1.93%	8.22%
RSD with DeLIA	4.51%	7.69%

4.1.1. Using DS

We first ran the FWI with DS using eight sources, six iterations, and each experiment ten times. To compare the overhead of the FWI with DeLIA according to the number of nodes, we ran the experiments with four and eight nodes. The execution with DeLIA using four nodes shows good behavior, as shown in Fig. 4(a); the overhead was 2.54% less than the RSD without DeLIA (6.88%), but the RSD with DeLIA was more significant (17.40%), which shows that probably in the set of experiments, HM detects many fake failures (Table 2).

In the experiment with eight nodes and eight shots, we do not change the DeLIA and FWI configuration while employing further resources. We have only one shot per node, so the local data feature is underused; consequently, we have an overhead (8.8%) bigger than RSD without DeLIA (3.37%). In the execution, a significant number of tasks per node is adequate to save local data effectively. So we increased the number of shots to 32; in this scenario, the overhead of FWI with DS was 3.84%. The results are shown in Fig. 4 and Table 2.

4.1.2. Using CTWS

We executed the FWI with CTWS with six iterations and each experiment ten times, with four and eight nodes. For this set of experiments, we use 32 sources for more tasks per process because CTWS is a dynamic scheduler that performs better in these scenarios. The DeLIA main difference between a static and dynamic scheduler is that in the last one, each process should read the list of tasks already processed by the other in the recovery time.

The execution with DeLIA using four nodes shows a tolerable behavior, as shown in Fig. 5(a); the overhead was 2.27% a slightly more than RSD without DeLIA (1.93%), but the RSD with DeLIA was more significant (4.51%) because of the fake failures detection (Table 3).

The execution using eight nodes was suitable; the overhead was 2.68%, the RSD without DeLIA 8.2%, and the RSD with DeLIA 7.69%. As a dynamic scheduler, the execution times of the experiments can differ from each other due to the changes in execution time, such as stealing a task; because of that, the RSD without DeLIA was more significant.

4.2. Behavior in preemptible conditions

Fault tolerance techniques can also be applied for preemptive conditions without fault. To maintain the execution progress if the execution is interrupted preemptively, fault tolerance actions must be performed at the application level. To allow the use of DeLIA for such situations, we analyzed the behavior of 3D FWI with this library in preemptive circumstances; we used the supercomputer NPAD and Amazon Elastic Compute Cloud (Amazon EC2) Spot for this set of experiments.

4.2.1. Maximum resource usage time in a supercomputer environment

In this set of experiments, we utilized the 3D FWI application with DeLIA, using GdM data consisting of 174 shots and four nodes. We employed the CTWS workload scheduler in NPAD, setting a time limit

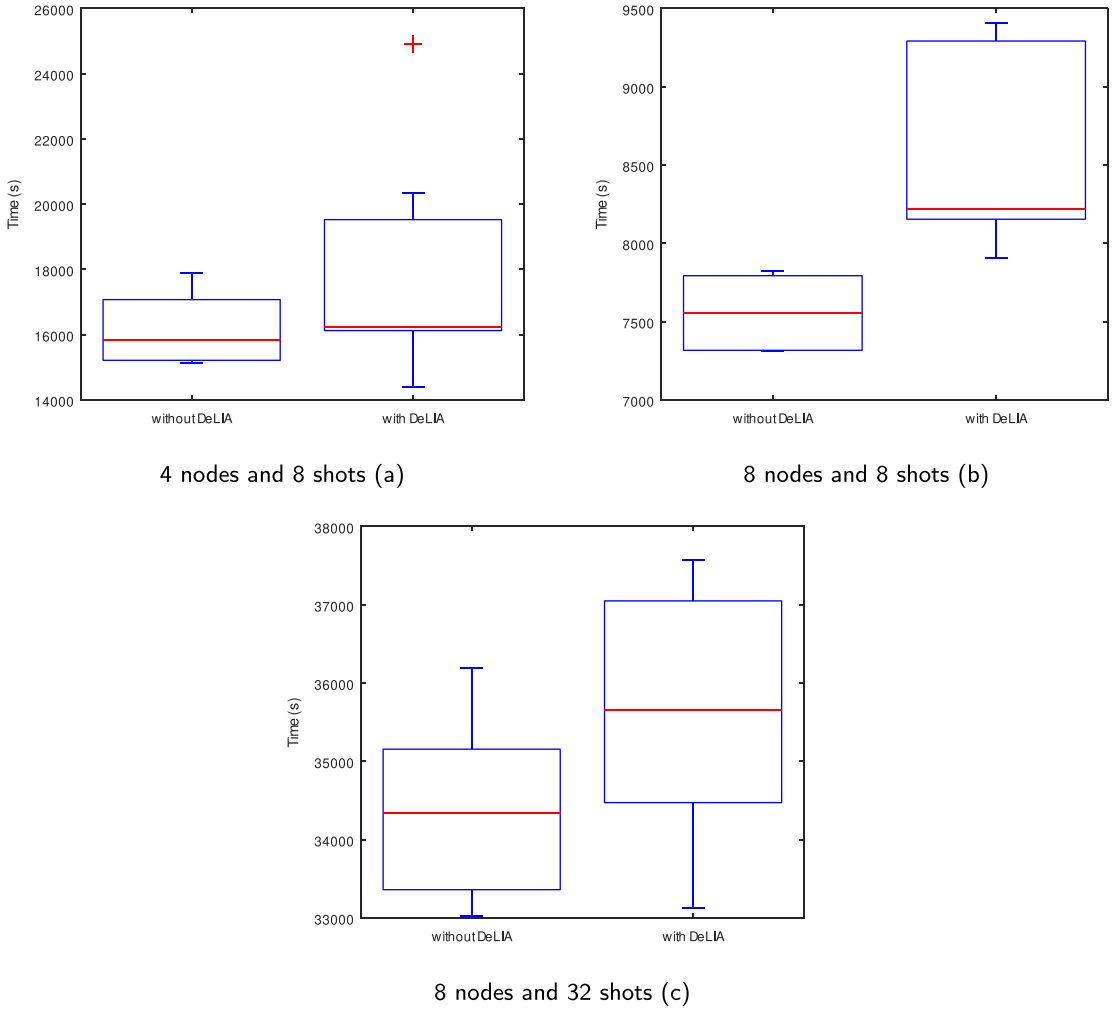

Fig. 4. Overhead analysis of FWI with and without DeLIA using DS.

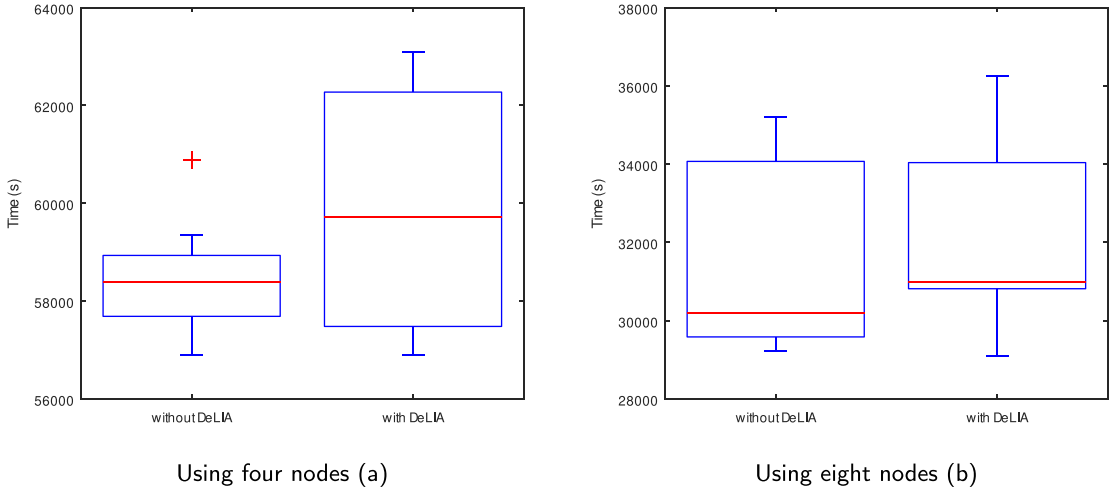

Fig. 5. Overhead analysis of FWI with and without DeLIA using CTWS.

of one day and configuring it to send a termination signal to the application two minutes before the end of the time limit.

The first execution involved running one FWI iteration. We then executed the application again, starting from the middle of the second iteration by reading the last application state. This second execution continued until iteration 9, spanning an additional three days.

Using DeLIA, we successfully carried out up to 9 iterations at different execution moments. The progress made during the first execution, which lasted one day, was seamlessly carried forward into the second execution, spanning three days, without any loss of progress.

4.2.2. Preemptive cloud instances (AWS EC2 spot)

The program was executed from the beginning in one instance (8 vCPUs, 32 GB RAM) of Amazon EC2 Spot, a preemptive computing solution offered by AWS. After about 7 h, iteration 1 of FWI was completed, and the global checkpoint was saved. Some hours later, during iteration 2, due to the supply and demand dynamics at the time, AWS decided to reclaim the resources and issued a two-minute interruption notice. DeLIA detected the signal associated with the interruption notice and saved the partial progress of iteration 2 (local checkpoints) correctly.

Later, another EC2 Spot instance was started, and a 3D FWI simulation was executed with the same settings. DeLIA detected the checkpoints from the previous run and loaded their state of execution. Iteration 2 of FWI was completed in 4.5 h, much less than the regular 7 h per iteration, because of the partial progress loaded from checkpoints. Then, the following iterations were processed one after the other until the program was completed, with no interruptions from AWS.

The application behaved as expected during a preemptive interruption, as it detected the interruption notice, saved a consistent state of the application, and later resumed execution from the saved checkpoints. Therefore, the DeLIA's termination signal detection, checkpointing (local and global), and rollback recovery capabilities have been validated in preemptive environments.

5. Related works

Researchers have investigated geophysics problems using high-performance computing techniques with many computations, but few have employed fault tolerance techniques for such problems. Gonçalves et al. (2011) embedded checkpointing and rollback recovery for reverse time migration (RTM) using heartbeat monitoring with MPI for fault detection. Fault-tolerance strategies with checkpointing are common and add an overhead that may be acceptable depending on the application. However, the communication of the heartbeat monitoring using MPI absorbs additional overhead of the MPI library and may not be suitable if the application is an independent project that already uses MPI because conflicts may occur. In our work, we choose to implement heartbeat monitoring communication without MPI and to promote reusability, we have built DeLIA as a library that can be easily employed in other application codes.

Some authors proposed fault tolerance methods in the application with a workload scheduler. Kayum et al. (2020), for example, applied the method leader and workers, where the leader node sends the tasks to the worker nodes, and if a worker fails, the leader can discard this node. The authors concentrate on developing the FT methods for network communication, which uses a TCP socket instead of MPI because MPI demands all nodes to operate correctly to complete a collective communication and to finalize. Borin et al. (2015) proposed the scalable partially idempotent tasks system (SPITS), a model that develops a workload scheduler with FT. SPITS architecture includes job manager (JM) and worker. The JM is responsible for sending the tasks to the workers; the workers execute them. The JM sends a task and adds it to a list of work in progress (WIP). After a while, if the task is consolidated, it is removed from the WIP list; otherwise, the worker probably failed, and JM sends it again to another worker. Okita et al. (2018) applied SPITS in the Non-hyperbolic Common Reflection Surface method, and Okita et al. (2022) applied to RTM using SPITS and checkpointing; both works have investigations in preemptive cloud scenarios. Combining FT with workload scheduling has been promising but requires the use of a specific scheduler. On the other hand, our approach can be used with other workload schedulers.

Bautista-Gomez et al. (2011) proposed the checkpoint library called fault tolerance interface (FTI) implemented in C/MPI and Python. This library spawns one extra MPI process per node to employ the FTI features and to guarantee that the library does not cause any damage to the application communication; on the other hand, it causes the overhead of creating a new process. The authors have developed FTI with a checkpoint using the Reed–Solomon method to encode the data and used the Mw9.0 Tohoku Japan earthquake simulation as a case study. Parasyris et al. (2020) extended the FTI to support checkpoints with multiple nodes and multiple graphics processing units (GPUs), reducing the overhead of the checkpoint and using a method that detects and stores only the data that has changed. Our work does not rely on an extra MPI process and does not use any MPI functionally, and it does so to avoid additional overhead and conflicts with other MPI features used in the application. DeLIA also adds fault detection methods creating a dedicated thread for this.

Applications for distributed-memory HPC architectures often rely on the MPI to handle node communication. However, the MPI standard lacks a comprehensive fault tolerance strategy, as noted by Bouteiller and Bosilca (2022). Several projects have emerged to address this issue by introducing fault tolerance techniques to MPI, such as MPI-FT (Louca et al., 2000) and FT-MPI (Fagg and Dongarra, 2000). One notable project in this domain is user-level failure mitigation (ULFM), which extends the MPI standard with fault-tolerance constructs, as discussed by Bland et al. (2013). ULFM provides a set of essential interfaces that enable users to adapt their recovery methods to suit the characteristics of their applications. This flexibility has led researchers and production teams to propose various recovery strategies at the application level. ULFM defines MPI routines and definitions that enable MPI applications to communicate despite failures. It is important to note that ULFM is not designed to be a specific recovery approach itself; instead, it supports the development of fault tolerance packages by offering the minimal interface required to repair communication channels.

Recent significant advancement in the ULFM project has been presented by Bouteiller and Bosilca (2022). This work focuses on fault reports and the concurrent execution of recovery activities. Another work that builds upon ULFM is introduced by Munhoz et al. (2022), which proposes an in-memory rollback restart technique utilizing ULFM. Furthermore, Godón (2022) investigate fault-tolerant strategies applied to ant colony optimization using MPI with ULFM. The application must be configured with a compatible MPI version that supports ULFM to use it. ULFM enables the continued operation of MPI programs even after a node failure by providing information about the error that occurred in a node. However, it does not save the application's state after a failure, as in our proposal.

Sultana et al. (2019) proposed MPI Stages to save the state of the MPI in BSPs. MPI Stages is an extension of the project proposed by Laguna et al. (2016) called Reinit, which arises to fill the gap left by UFML about the checkpoint of the MPI state. MPI Stages divide the application into two functions, `main_loop` (which executes all code) and `main` (which initializes MPI and calls `main_loop`). When the application fails, it is not necessary to re-queue the job or restart the application from the beginning; the MPI Stages permits restart without initializing the MPI and continue from `main_loop`. It is crucial noting that MPI Stages saves and recovers the state of the MPI communication, but the user should guarantee the application data checkpoint. DeLIA, on the other hand, saves the application's state instead of MPI's.

In summary, this work stands out from others by introducing a fault tolerance library with several key features. Firstly, it provides a user-friendly API allowing flexible integration into existing applications.

Table 4
Literature review on tools of FT and geophysical problems applying FT.

	Flexible FT tool	Continue after node failure	CR application state	Interruption detection	Preemptive cloud environment	Applied to geophysical problems
Gonçalves et al. (2011)			x	HM with MPI		x
Kayum et al. (2020)		x		through workload scheduler		x
Okita et al. (2018)		x		through workload scheduler	x	x
Okita et al. (2022)		x	x	through workload scheduler	x	x
Bautista-Gomez et al. (2011)	x		x			
Bland et al. (2013)	x	x		MPI routines		
Laguna et al. (2016)	x	x	only MPI State	MPI routines		
Our proposal	x		x	HM with a dedicated thread using UDP	x	x

Secondly, it incorporates checkpointing and rollback mechanisms, enabling the preservation and restoration of the application's state in the event of failure. Additionally, the library includes interruption detection capabilities through heartbeat monitoring and the detection of termination signals. Furthermore, communication in DeLIA is achieved without relying on the MPI to avoid conflict with the MPI already applied in the application. This design choice allows for efficient communication between processes. Lastly, DeLIA is compatible with various workload schedulers, making it versatile and compatible with different HPC environments.

Table 4 presents an overview of the works related to tools of FT and geophysical problems applying FT techniques cited above. We consider flexible FT tools the works developed uncoupled to an application or workload scheduler.

6. Conclusion

We present DeLIA, a fault-tolerance library written in C++ for bulk synchronous programs (BSP). DeLIA supports checkpointing the application's global state and each process's state, heartbeat monitoring, and termination signal detection. DeLIA provides interface classes to handle the application's state or settings and an application programming interface (API) to simplify its use.

We validated DeLIA's features in the context of the geophysical problem of 3D full waveform inversion (FWI), a widely used application in the scientific community that requires high-performance computing resources. To ensure the library's effectiveness, we performed experiments using data from an estimated velocity model acquired from the Gato do Mato oil field in the Santos Basin, Brazil (Lopez et al., 2020) to analyze the overhead in a more realistic scenario.

Since the FWI application is employed in industrial and academic domains with various schedulers, we tested DeLIA with two different schedulers to ensure its adaptability in different settings. Through these experiments, we observed that the number of tasks per process directly impacts DeLIA's overhead. Specifically, when saving local data, many tasks are required to utilize this feature effectively.

We performed tests in the supercomputing NPAD and the preemptive instance of AWS Cloud. DeLIA showed a light overhead of 8.8% in the worst experiment and presented itself as suitable for employment in preemptive circumstances. We currently have some limitations on continuing the execution even if any node has been interrupted. We intend to perform this in prospective work. The library is available on https://lappsufrn.gitlab.io/delia/ with its documentation.

Code availability section

Dependability Library for Iterative Applications (DeLIA)

Email contact: carla.santana.058@ufrn.edu.br

Requirements to apply: A C++11-compatible compiler is required. The library has been tested to compile with CMake version 3.5 or more recent.

Program language: C++

Program size: 4.3 MB

DeLIA source codes are available for download at the link: https://gitlab.com/lappsufrn/delia-submission

DeLIA documentation is available at the link: https://lappsufrn.gitlab.io/delia/

FWI source codes are available for download at the link: https://gitlab.com/lappsufrn/seismic/ufrn-fwi/mamute

CRediT authorship contribution statement

Carla Santana: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Ramon C.F. Araújo:** Investigation, Validation, Writing – original draft, Writing – review & editing. **Idalmis Milian Sardina:** Conceptualization, Formal analysis, Methodology, Writing – review & editing. **Ítalo A.S. Assis:** Data curation, Software, Validation, Writing – review & editing. **Tiago Barros:** Data curation, Software, Validation, Writing – review & editing. **Calebe P. Bianchini:** Software, Validation. **Antonio D. de S. Oliveira:** Data curation, Software. **João M. de Araújo:** Data curation, Funding acquisition, Project administration, Software. **Hervé Chauris:** Supervision, Writing – review & editing. **Claude Tadonki:** Supervision, Writing – review & editing. **Samuel Xavier-de-Souza:** Conceptualization, Funding acquisition, Investigation, Methodology, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgments

The authors would like to acknowledge the support received from Shell Brazil through the projects "Novos Métodos de Exploração Sísmica por Inversão Completa das Formas de Onda" and "Novas metodologias computacionalmente escaláveis para sísmica 4D orientado ao alvo em reservatórios do pré-sal" at the Universidade Federal do Rio Grande do Norte (UFRN), as well as the support from the Brazilian Oil and Gas regulatory agency ANP. They are grateful for the assistance provided by the "Centre de Géosciences" of Mines Paris - PSL. Furthermore, the authors express their appreciation to the High-Performance Computing Center at UFRN (NPAD), and the "Centre de Recherche en Informatique" at Mines Paris - PSL for providing support and computing resources.

Appendix A. DeLIA quick start

The purpose of the appendices is to give an idea of how to apply DeLIA in an application, but for more details, go to our documentation https://lappsufrn.gitlab.io/delia/ .

The project example we are using has two loops: the external loop represents the iterations of the global data, and the internal loop represents the local data processing. The inner loop calculates the variable `local_data`, and the external loop reduces these variables in the `global_data`.

Listing 1: main file without DeLIA

```
int main(int argc, char **argv) {
   int data_size, iterations, ind=0;
   int rank, comm_sz;
   double *local_data, *global_data;
   Settings *st;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   // Class which represents the application
       configuration
   st = new Settings("config_file.json");
   // Size of the data
   data_size = st->getDataSize();
   // Iterations number
   iterations = st->getIterations();
   // Data that will be calculated in each
       process
   local_data  = new double[data_size];
   // Data that will be result of the
       reduction of the local data
   global_data = new double[data_size];
   if (local_data == NULL || global_data ==
       NULL) {
      fprintf(stderr, "Can't allocate vectors
          \n");
      exit(-1);
   }
   for (ind; ind < iterations; ind++) {
      for (int i = 0; i < data_size; i++) {
         // Calculation of local data
         local_data[i] = rank*i*ind;
      }
      // Reduction of the data
      MPI_Allreduce(local_data, global_data,
          data_size, MPI_DOUBLE,
                        MPI_SUM, MPI_COMM_WORLD)
                            ;
      if (rank ==0 )
         printf("Iteration %d ends\n", ind);
   }
   MPI_Finalize();
   delete (local_data);
   delete (global_data);
}
```

A.1. DeLIA initialization

The application has a file representing the settings information (`config_file.json`) in this example. To initialize DeLIA, we pass the id of the process and the number of processes, a JSON file with DeLIA parameters (`delia_param.json`), and the application configuration file. We are using checkpointing and interrupt detection. Then, we pass the local and global data to DeLIA.

Listing 2: delia_param.json

```
{
    "FT_FOLDER" : "./data",
    "CHECKPOINTING_GLOBAL_ITERATION": 1,
    "TRIGGER_SIGNAL" : true,
    "TRIGGER_HEARTBEAT_MONITORING": {
        "TIME_MAX_WAIT" : 20,
        "SLEEP_THREAD_TIME" : 1
    }
}
```

In our DeLIA configuration example, we saved the data in the folder `./data`, to save the global data in each iteration (`CHECKPOINTING_GLOBAL_ITERATION : 1`), activated the trigger by termination signal and heartbeat monitoring. The heartbeat monitoring defined node failed if it did not send a message in 20 s, and every 2 s, the observed node sends a message to the node leader.

Listing 3: DeLIA initialization in the code

```
// Initialization passing the rank of the
    process, the number of processes
// the configuration file from DeLIA
// the configuration file from application
DeLIA_Init(rank, comm_sz, "delia_param.json",
    "config_file.json");
if (!DeLIA_CanWork()) {
   std::cerr << "DeLIA can not work" << std::
       endl;
   exit;
};

// Passing the global and local data to DeLIA
DeLIA_SetGlobalData(global_data, data_size,
    DOUBLE_CODE);
DeLIA_SetLocalData(local_data, data_size,
    DOUBLE_CODE);
```

A.2. Check if you can recover data

After DeLIA's initialization, we can check if there is data to be recovered. We check if it is possible to recover the global data by calling the function `DeLIA_CanRecoverGlobalCheckpointing`, which will check if there are data and the data corresponds to the current application configuration.

We should guarantee the processes synchronization, and we can read the global data or write the current settings.

Listing 4: Recover data

```
// Check if there are data to be read with
    the current settings
bool canRecover =
    DeLIA_CanRecoverGlobalCheckpointing();
// Barrier necessary because if there aren't
    global data
// to be recover all process should check in
    the same moment
MPI_Barrier(MPI_COMM_WORLD);
if (canRecover)
   DeLIA_ReadGlobalData();
else
   DeLIA_SaveSettings();
ind = DeLIA_getCurrentGlobalIteration();
```

A.3. Interruption detection

To apply the heartbeat monitoring, we should initialize this feature before the main data processing (in this example, before the external loop) and finalize it after the data processing finishes (in this example, when the outer loop ends).

Listing 5: Heartbeat monitoring

```
DeLIA_HeartbeatMonitoring_Init();
for (ind; ind < iterations; ind++) {
   // ...
}
DeLIA_HeartbeatMonitoring_Finalize();
```

To initialize the trigger by signal, you only need to define it in the configuration file and this feature will be initialized with DeLIA. Heartbeat monitoring is only initialized with the function `DeLIA_HeartbeatMonitoring_Init()` because it is a more expensive feature, and depending on the circumstances, it may not need to be active at the same time as DeLIA.

A.4. Checkpointing

To save the global data, the function `DeLIA_SaveGlobalData` should be called when the global data represents the application's state. In this example, it is after the `MPI_Allreduce`. The checkpointing frequency is according to the `delia_param.json` parameters. In this parameters file, there is nothing about local checkpointing; in this case, the local checkpointing is saved just when the node receives a trigger.

Listing 6: Save global data

```
for (ind; ind < iterations; ind++) {

   for (int i = 0; i < data_size; i++) {
      // Calculation of local data
      local_data[i] = rank*i*ind;
   }
   // Reduction of the data
   MPI_Allreduce(local_data, global_data,
       data_size, MPI_DOUBLE,
                  MPI_SUM, MPI_COMM_WORLD);
   if (rank ==0 )
      printf("Iteration %d ends\n", find);

   // DeLIA will save the global data
       according the frequency already
   // informed in the delia_params.json
   DeLIA_SaveGlobalData();
}
```

A.5. DeLIA finalize

You should call the function `DeLIA_Finalize` to finish DeLIA.

Listing 7: Finalize functions

```
DeLIA_HeartbeatMonitoring_Finalize();
DeLIA_Finalize();
MPI_Finalize();
delete (local_data);
delete (global_data);
```

A.6. Application with DeLIA

Example complete with DeLIA.

Listing 8: main file with DeLIA

```
int main(int argc, char **argv) {

   int data_size, iterations, ind=0;
   int rank, comm_sz;
   double *local_data, *global_data;
   Settings *st;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   // Class which represents the application
       configuration
   st = new Settings("config_file.json");
   // Size of the data
   data_size = st->getDataSize();
   // Iterations number
   iterations = st->getIterations();

   // Data that will be calculated in each
       process
   local_data  = new double[data_size];
   // Data that will be result of the
       reduction of the local data
   global_data = new double[data_size];

   if (local_data == NULL || global_data ==
       NULL) {
      fprintf(stderr, "Can't allocate vectors
          \n");
      exit(-1);
   }

   //initialization passing the rank of the
       process, the number of processes
   // the configuration file from DeLIA
   // the configuration file from application
   DeLIA_Init(rank, comm_sz, "delia_param.
       json", "config_file.json");
   if (!DeLIA_CanWork()) {
      std::cerr << "DeLIA can not work" <<
          std::endl;
      exit;
   };

   // Passing the global and local data to
       DeLIA
   DeLIA_SetGlobalData(global_data, data_size
       , DOUBLE_CODE);
   DeLIA_SetLocalData(local_data, data_size,
       DOUBLE_CODE);
```

```
    // Check if there are data to be read with
        the current settings
    bool canRecover =
        DeLIA_CanRecoverGlobalCheckpointing();

    // Barrier necessary because if there aren
        't global data
    // to be recover all process should check
        in the same moment
    MPI_Barrier(MPI_COMM_WORLD);
    if (canRecover)
       DeLIA_ReadGlobalData();
    else
       DeLIA_SaveSettings();
    ind = DeLIA_getCurrentGlobalIteration();

    //initialization of the Hearbeat
        monitoring
    DeLIA_HeartbeatMonitoring_Init();

    for (ind; ind < iterations; ind++) {

       for (int i = 0; i < data_size; i++) {
          // Calculation of local data
          local_data[i] = rank*i*ind;
       }
       // Reduction of the data
       MPI_Allreduce(local_data, global_data,
           data_size, MPI_DOUBLE,
                      MPI_SUM, MPI_COMM_WORLD)
                          ;
       if (rank ==0 )
          printf("Iteration %d ends\n", ind);

       // DeLIA will save the global data
           according the frequency already
       // informed in the delia_params.json
       DeLIA_SaveGlobalData();
    }

    DeLIA_HeartbeatMonitoring_Finalize();
    DeLIA_Finalize();
    MPI_Finalize();
    delete (local_data);
    delete (global_data);
}
```

Appendix B. FWI parameters

We apply DeLIA in the FWI available in https://gitlab.com/lappsufrn/seismic/ufrn-fwi/mamute; for this work, we used the configuration below:

- number of points in the velocity model we have in the dimension x: 200
- number of points in the velocity model we have in the dimension y: 100
- number of points in the velocity model we have in the dimension z: 180
- distance from one point to another in the x, y and z dimension in meters: 50
- number of points of the border we are considering: 50
- number of timesteps: 2400
- time sampling in seconds: 0.003343213190252791
- peak frequency in hertz: 2.0
- number of receivers: 2600

References

Assis, Í.A., Oliveira, A.D., Barros, T., Sardina, I.M., Bianchini, C.P., De-Souza, S.X., 2019. Distributed-memory load balancing with cyclic token-based work-stealing applied to reverse time migration. IEEE Access 7, 128419–128430.

AWS, 2023. Amazon EC2 spot instances. URL https://aws.amazon.com/ec2/spot/?nc1=h_ls.

Barros, T., Fernandes, J.B., de Assis, I.A.S., de Souza, S.X., 2018. Auto-tuning of 3D acoustic wave propagation in shared memory environments. In: First EAGE Workshop on High Performance Computing for Upstream in Latin America. EarthDoc, Santander, pp. 1–5. http://dx.doi.org/10.3997/2214-4609.201803072, URL http://www.earthdoc.org/publication/publicationdetails/?publication=94579.

Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S., 2011. FTI: High performance fault tolerance interface for hybrid systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–32.

Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J., 2013. Post-failure recovery of MPI communication capability: Design and rationale. Int. J. High Perform. Comput. Appl. 27 (3), 244–254.

Borin, E., Rodrigues, I.L., Novo, A.T., Sacramento, J.D., Breternitz, M., Tygel, M., 2015. Efficient and fault tolerant computation of partially idempotent tasks. In: 14th International Congress of the Brazilian Geophysical Society & EXPOGEF. Rio de Janeiro, Brazil, 3-6 August 2015, Brazilian Geophysical Society, pp. 367–372.

Bouteiller, A., Bosilca, G., 2022. Implicit Actions and Non-blocking Failure Recovery with MPI. arXiv preprint arXiv:2212.08755.

Chetan, S., Ranganathan, A., Campbell, R., 2005. Towards fault tolerance pervasive computing. IEEE Technol. Soc. Mag. 24 (1), 38–44.

Fagg, G.E., Dongarra, J.J., 2000. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. Springer, pp. 346–353.

Godón, M.B., 2022. Estratexias de tolerancia a fallos nun algoritmo paralelo de colonia de formigas. (Bachelor's thesis). Universidade da Coruña.

Gonçalves, A., Bersot, M., Bulcao, A., Boeres, C., Drummond, L., Rebello, V., 2011. Fault tolerance in an industrial seismic processing application for multicore clusters. In: European MPI Users' Group Meeting. Springer, pp. 264–271.

Google Cloud, 2023. Preemptible VM instances. URL https://cloud.google.com/compute/docs/instances/preemptible.

Herault, T., Robert, Y., 2015. Fault-Tolerance Techniques for High-Performance Computing. Springer.

Kalaiselvi, S., Rajaraman, V., 2000. A survey of checkpointing algorithms for parallel and distributed computers. Sadhana 25 (5), 489–510.

Kayum, S.N., Alsalim, H., Tonellot, T.-L., Momin, A., 2020. A fault tolerant implementation for a massively parallel seismic framework. In: 2020 IEEE High Performance Extreme Computing Conference. HPEC, IEEE, pp. 1–8.

Laguna, I., Richards, D.F., Gamblin, T., Schulz, M., de Supinski, B.R., Mohror, K., Pritchard, H., 2016. Evaluating and extending user-level fault tolerance in MPI applications. Int. J. High Perform. Comput. Appl. 30 (3), 305–319.

Lopez, J., Neto, F., Cabrera, M., Cooke, S., Grandi, S., Roehl, D., 2020. Refraction seismic for pre-salt reservoir characterization and monitoring. In: SEG Technical Program Expanded Abstracts 2020. pp. 2365–2369. http://dx.doi.org/10.1190/segam2020-3426667.1.

Louca, S., Neophytou, N., Lachanas, A., Evripidou, P., 2000. MPI-FT: Portable fault tolerance scheme for MPI. Parallel Process. Lett. 10 (04), 371–382.

Microsoft Azure, 2023. Azure spot virtual machines pricing. URL https://azure.microsoft.com/en-us/pricing/spot-advisor/.

Munhoz, V., Castro, M., Mendizabal, O., 2022. Strategies for Fault-Tolerant Tightly-Coupled HPC Workloads Running on Low-Budget Spot Cloud Infrastructures. In: 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing. SBAC-PAD, IEEE, pp. 263–272.

Netto, M.A., Calheiros, R.N., Rodrigues, E.R., Cunha, R.L., Buyya, R., 2018. HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges. ACM Comput. Surv. 51 (1), 1–29.

Nocedal, J., 1980. Updating quasi-Newton matrices with limited storage. Math. Comput. 35 (151), 773–782.

Okita, N.T., Camargo, A.W., Ribeiro, J., Coimbra, T.A., Benedicto, C., Faccipieri, J.H., 2022. High-performance computing strategies for seismic-imaging software on the cluster and cloud-computing environments. Geophys. Prospect. 70 (1), 57–78.

Okita, N., Coimbra, T., Rodamilans, C., Tygel, M., Borin, E., 2018. Using spits to optimize the cost of high-performance geophysics processing on the cloud. In: First EAGE Workshop on High Performance Computing for Upstream in Latin America, vol. 2018, (no. 1), EAGE Publications BV, pp. 1–5.

Oracle Cloud, 2023. Preemptible instances. URL https://docs.oracle.com/en-us/iaas/Content/Compute/Concepts/preemptible.htm.

Parasyris, K., Keller, K., Bautista-Gomez, L., Unsal, O., 2020. Checkpoint restart support for heterogeneous HPC applications. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing. CCGRID, IEEE, pp. 242–251.

Plessix, R.-E., 2006. A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. Geophys. J. Int. 167 (2), 495–503.

Rojas, E., Meneses, E., Jones, T., Maxwell, D., 2021. Understanding failures through the lifetime of a top-level supercomputer. J. Parallel Distrib. Comput. 154, 27–41.

Santana, C., Barros, T., Milian, I., Bianchini, C., Xavier De Souza, S., 2019. Workload scheduling comparison in a full waveform inversion distributed memory implementation. In: 16th International Congress of the Brazilian Geophysical Society. Rio de Janeiro, Brazil, pp. 1–5.

Sultana, N., Rüfenacht, M., Skjellum, A., Laguna, I., Mohror, K., 2019. Failure recovery for bulk synchronous applications with MPI stages. Parallel Comput. 84, 1–14.

Symes, W.W., 2007. Reverse time migration with optimal checkpointing. Geophysics 72 (5), SM213–SM221.

Valiant, L.G., 1990. A bridging model for parallel computation. Commun. ACM 33 (8), 103–111.

Virieux, J., Operto, S., 2009. An overview of full-waveform inversion in exploration geophysics. Geophysics 74 (6), WCC1–WCC26.

Virieux, J., Operto, S., Ben-Hadj-Ali, H., Brossier, R., Etienne, V., Sourbier, F., Giraud, L., Haidar, A., 2009. Seismic wave modeling for seismic imaging. Leading Edge 28 (5), 538–544.

Weber, T.S., 2003. Tolerância a falhas: Conceitos e exemplos. Apostila do Programa de Pós-Graduação–Instituto de Informática-UFRGS. Porto Alegre 24.

Zhu, C., Byrd, R.H., Lu, P., Nocedal, J., 1997. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. ACM Trans. Math. Softw. 23 (4), 550–560.