Parallel Computing 81 (2019) 58–67

Contents lists available at ScienceDirect

Parallel Computing

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

Resilient computational applications using Coarray Fortran

Alessandro Fanfarillo[a,*], Sudip Kumar Garain[b], Dinshaw Balsara[b], Daniel Nagle[a]

[a] National Center for Atmospheric Research, USA
[b] University of Notre Dame, USA

A R T I C L E I N F O

Article history:
Received 18 December 2017
Revised 26 October 2018
Accepted 6 December 2018
Available online 7 December 2018

Keywords:
Partitioned global address space
Coarray Fortran
Fault tolerance

A B S T R A C T

With the increase in the number of hardware components and layers of the software stack in High Performance Computing (HPC) there will likely be an increment in number of hardware and software failures, which will be user-visible. Even under the most optimistic assumptions about the individual components reliability, probabilistic amplification from using millions of nodes has a dramatic impact on the Mean Time Between Failure (MTBF) of the entire platform.

Although several techniques to address this problem have been developed, the support provided by the programming model, for the user to mitigate or work around this issue, is still insufficient. The Fortran 2018 standard defines *failed images*, a new feature that allows the programmer to detect and manage image failures in a parallel program.

In this paper we show how to use *failed images* and *teams*, another feature defined in the Fortran 2018 standard, to implement resilient computational applications.

1. Introduction

The presence of billions of hardware components and several levels of software stack will likely represent an increment in number of hardware and software failures. An exascale system, 1000 times faster than a petascale system, will fail, roughly (in the best case), 1000 times more frequently. Unfortunately, smaller transistors are much more error prone. Smaller circuits, carrying smaller charges, are much more subject to transient errors due to environment interferences. One of the major causes for transient errors is cosmic radiation: neutrons flux occasionally interacts with silicon creating a parasite cascade of charged particles. The increase in chip density (predicted by Moore's law) will probably be a limiting factor in processors design because of the cosmic-rays effect. Smaller transistors and wires will age more rapidly and more unevenly, so that permanent failures (physical hardware failures) will be more frequent. Although hardware vendors can face the increase of fault rates by using more powerful error detection and correction codes, researchers [1] estimate that the increase in error rate can be kept under control by using 20% more circuits and energy consumption.

One of the most used techniques for facing failures is checkpointing: saving all the work periodically on a persistent media in order to restart from a recent backup after an eventual fault. Even though this technique is quite effective, it requires a lot of I/O operations that usually require tens of minutes. At limit, the risk is that the time for checkpointing is close to the Mean Time Between Failures; in this case, a lot of time is spent for saving the work and only a small fraction is used for the actual computation. A valid alternative to pure checkpointing is to create fault tolerant applications. A fault

[*] Corresponding author.
E-mail addresses: elfanfa@ucar.edu, fanfarillo@ing.uniroma2.it (A. Fanfarillo), sgarain@nd.edu (S.K. Garain), dbalsara@nd.edu (D. Balsara), dnagle@ucar.edu (D. Nagle).

https://doi.org/10.1016/j.parco.2018.12.002
0167-8191/© 2018 Elsevier B.V. All rights reserved.

tolerant parallel application can handle the errors and execute some actions to terminate cleanly or follow some recovery procedures. The application has to be able to detect errors and access remote data to correct or compensate for the error and its effect. Obviously, such applications will pay this ability in terms of code complexity, energy consumption (mainly because of more frequent communication) and speed.

Although MPI remains the dominant communication layer in High Performance Computing, several alternative parallel systems are becoming increasingly present. A few examples of this new parallel programming systems are the ones belonging to the family of Partitioned Global Address Space (PGAS) Languages, such as Unified Parallel C (UPC) [2], Coarray Fortran (CAF) [3], Chapel [4] and X10 [6].

In this work, we show how the fault tolerant support provided by the *failed images* CAF feature of Fortran 2018 can be used effectively to make a Jacobi iteration algorithm resilient. In order to do so, we implemented the failed images feature in the GNU Fortran compiler and OpenCoarrays library [7], making this work a unique contribution.

2. Related work

Fault tolerance in transport layers is not a new concept; several efforts have been attempted in different directions.

In the PGAS family there have been several attempt to make a preexisting PGAS language fault tolerant. In [8] Chen et al. analyze the feasibility of providing fault tolerance for PGAS model showing X10-FT: a X10-based fault tolerant framework that leverages renowned techniques in distributed systems like distributed file systems and Paxos, as well as specific solutions based on the characteristics of the APGAS model to make checkpoints and consensus. In [9] Vishnu et al. present ARMCI-FT: a first step toward data-centric fault resilience. It implements a fault-resilient, one-sided communication run-time framework using Global Arrays and its communication system, ARMCI [10]. In [11], the authors present the Global Address Space Programming Interface (GASPI) and its fault tolerant support based on timeouts.

For MPI, the most notable past effort is FT-MPI [12]. It aims at helping an application to express its failure recovery strategy by rebuilding internal MPI data structures (communicators, rank, etc.) and invoking user provided callbacks to restore a coherent application state when failures occur. Although this approach is very efficient to minimize the cost of failure recovery techniques, it still adds a significant level of complexity to the design and implementation of parallel applications. A more recent effort was the Run-Through Stabilization proposal [13]. This proposal introduced many new structs for MPI including the ability to “validate” communicators as a way of marking failure as recognized and allowing the application to continue using the communicator. Because of the implementation complexity imposed by resuming operations on failed communicators, this proposal was eventually unsuccessful in its introduction to the MPI Standard. Currently, the most promising solution seems to be the User-Level Failure Mitigation (ULFM) specification [14]. It features the basic interface and new semantics to enable applications and libraries to repair the state of MPI and tolerate failures. ULFM has been successfully used as fault-tolerant transport layer for frameworks like Fenix [16,18,19].

Lastly, it is worth to mention Fault Tolerant Messaging Interface (FMI) [17]: a fault-tolerant MPI-like framework based on in-memory checkpoint/restart.

Because our work is based on ULFM, we will provide more details about it in the following Section.

3. MPI User-level failure mitigation

The User-Level Failure Mitigation (ULFM [14]) proposal adds a small set of new routines to the MPI-3 standard allowing to detect failures, propagate the information about failures to other processes and recovery from a failure.

ULFM focuses on the so called *fail-stop failures*: unrecoverable failures that interrupt the execution of the application. These include all hardware faults, and some software ones for as long as they are escalated to the brutal disappearance of one of the participating processes. ULFM does not directly address the so called *silent errors*, where the application data is silently corrupted due to external causes such as bit flips. Detecting occurrences of silent errors requires a detection systems (usually such capability is provided by the hardware) external to MPI, which makes this class of errors hard to support in a generic or portable way. However, some of the ULFM constructs can be used to drive the recovery part, assuming that the silent errors are detected by some other means (i.e. interrupts generated by the OS.) More details about the fail-stop failures and silent errors can be found in [15].

Failure Notification. In order to report process crash failures and user-mandated interruptions, ULFM adds 3 additional error classes. These error classes are passed to the normal MPI error reporting system: MPI error handlers, or are returned to the application when the error handler is set to `MPI_ERRORS_RETURNS`.

An error handler is a callback function attached to the communicator that is automatically invoked before returning from an MPI function if an error occurs. The handler receives an error code as input (which can then be converted to its error class), giving the opportunity for the developer to react to the failure type. A predefined error handler permits bypassing the invocation of the error handler callback. In any case, the error code (or MPI_SUCCESS when no error is present) is returned from MPI function calls.

Failure detection (also called failure notification) in ULFM involves only processes participating in communications with dead processes. This minimal notification system allows the library/application to define the scope of a failure (deciding which ranks should be notified) based on the application needs. When a failure gets reported to an MPI process, two routines help identifying which processes have failed: `MPI_COMM_FAILURE_ACK` and `MPI_COMM_FAILURE_GET_ACKED`.

Both are local routines, the first one gives the users a way to acknowledge all locally notified failures on a specific communicator; the second one returns a group of failed processes that have been locally acknowledged as failed by `MPI_COMM_FAILURE_ACK`. These two functions can be combined to get the list of all failed MPI ranks.

Error Propagation. If the application requires that all the MPI processes belonging to a specific communicator should be notified about a failure, for example because the normal execution flow must be interrupted to regroup in a collective recovery procedure, ULFM provides a function called `MPI_COMM_REVOKE` that can be used for this purpose. `MPI_COMM_REVOKE` is locally non-collective but it has a collective effect: it revokes a communicator and interrupts all operations on the communicator, window (future or active, at all ranks) by raising `MPI_ERR_REVOKED`. Once a communicator or window has been revoked it cannot be used for normal communication anymore (it may be used only with select MPI recovery functions).

Error Recovery. If the developer wants the application to continue to run excluding the dead processes, an error recovery procedure must be executed. ULFM provides two functions for this purpose: `MPI_COMM_AGREE` and `MPI_COMM_SHRINK`. The first function performs a fault-tolerant consensus, or agreement, between all living MPI processes in the associated communicator and consistently return a value and an error code at all ranks. It is used to make sure that all living processes in a communicator have the same view of the application status. The second function creates a new communicator by excluding all known failed processes from the parent communicator. Once the new and smaller communicator has been created, the application can decide to keep going with a smaller number of ranks, or spawn new processes that will replace the dead ones, or use spare processes to substitute for the missing ranks.

It should be noted that not all recovery strategies require all of these features. For example, in an application that implements a Monte Carlo simulation, organized in a master-worker fashion, if a worker process dies only the master will be notified (assuming workers do not communicate in-between). Once the master has acknowledged the failure, the other processes do not need to be notified and the communicator does not need to be shrunk. By allowing this scenario, ULFM avoids mandating a recovery strategy, but only provides a minimalistic set of routines for implementing recovery strategies as simple or as complex as the application developers see fit.

4. Introduction to Coarray Fortran

Coarray Fortran (also known as CAF) is a syntactic extension of Fortran 95/2003 which was proposed in the late 1990s by Robert Numrich and John Reid [3] and is now part of the Fortran 2008 standard (ISO/IEC 1539-1:2010) [20]. The main goal of coarrays is to allow Fortran users to create parallel programs without the burden of explicitly invoking communication functions or directives such as with MPI and OpenMP.

Coarrays are based on the Partitioned Global Address Space (PGAS) parallel programming model, which attempts to combine the SPMD approach used in the distributed memory systems with the semantic of the shared memory systems. In the PGAS model every process has its own memory address space but it can share a portion of its memory to other processes. The coarray syntax adds a few specific keywords and leaves the Fortran user free to use the regular array syntax within a parallel programs.

A program that uses coarrays is treated as if it were replicated at the start of execution; each replication is called an *image*. Images execute asynchronously and explicit synchronization statements are used to maintain program correctness; a typical synchronization statement is `sync all`, acting as a barrier for all images. Each image has an integer image index varying between one and the number of images (inclusive); the run time environment provides the `this_image()` and `num_images()` functions to identify the executing image and the total number of them.

Variables can be declared as *coarrays*: they can be scalars or arrays, static or dynamic, and of intrinsic or derived type. All images can reference coarray variables located on other images, thereby providing data communications; the Fortran standard further provides other facilities such as locks, critical sections and atomic intrinsics.

4.1. Coarray features in Fortran 2018

The coarray definition included in the Fortran 2008 standard defines a simple syntax for accessing data on remote images, synchronization statements and collective allocation and deallocation of memory on all images. Although these features allow one to write a totally functional coarray program, they do not allow to express more complex and useful mechanisms for synchronization, images organization and failure management.

Technical Specification 18508 [21] proposes the following extensions to the coarray facilities defined in Fortran 2008:

- teams;
- failed images;
- events;
- new atomic and collective procedures.

Teams allows one to execute more effectively and independently parts of a larger problem by grouping the images into non-overlapping teams. A class of problems that can benefit of such feature is multiphysics codes (e.g., climate models).

Table 1
Program behavior in case of failures based on `stat=` attribute presence.

	`stat=` present	`stat=` not present
Image Ctrl Stmt	continue	terminate with error
CAF Operation	continue	continue

Failed images provides a mechanism to identify what images have failed during the execution of a program. This obviously affects the resilience of programs running on large systems. Because this feature provides only the capability to detect which images have failed, the programmer is still in charge to implement a recovery policy.

Events provide a convenient mechanism for ordering execution segments on different images without requiring that those images arrive at synchronization point before any is allowed to proceed. This feature implements a fine grain synchronization mechanism based on a limited implementation of the well known semaphore primitives.

Fortran 2008 does not provide intrinsic procedures for commonly used collective operations and provides only minimal support for atomic memory operations. Such procedures can be highly optimized for the target computational system, providing significantly improved program performance. A typical example of collective operation introduced by TS-18508 is *co_broadcast*. This intrinsic allows one to broadcast data from a source image to a group of images as one single command. In Fortran 2008, the only way to implement this operation is to run a do-loop on the source image and perform a “put” operation on each target image, one at a time. TS-18508 enriches the available set of atomic intrinsics (e.g., new *atomic_fetch_and_op* intrinsics).

All the features defined in TS-18508 are going to be part of the Fortran 2018 standard.

4.1.1. Failed images description

The *failed images* feature provides a way to detect if one or more images have failed during the execution by adding a new image status, three intrinsic functions, one statement and few changes to the usual coarray syntax. To fully understand these changes, it is important to keep in mind that *failed images* aims to make a coarray program aware of the fact that failures may happen. This means that a coarray operation, like a transfer or a barrier, may not complete correctly and the user should have a way to check it. The most natural and already standardized way to check whether a coarray operation has completed correctly (e.g. it is involved with a failed image) is via a `stat=` attribute, as for IO operation and `allocate` statement.

Every coarray operation relies on an optional status variable, passed as argument, that gets defined according to the outcome of the operation. The programmer is supposed to check the value of the status variable after every coarray operation; in case a failure has been detected, the programmer can use the new intrinsic functions to detect which images have failed and adjust the execution flow accordingly.

The `stat=` attribute is optional for both image control statements (e.g. `sync all`) and coarray operations (e.g. “puts” and “gets”). Such attribute is necessary to check the correct behavior of the routine. If no failures or errors occur the `stat=` attribute, if present, is set to zero. When `stat=` attribute is omitted and there is a failure, calls to image control statements will cause the entire program to terminate with an error condition. On the other hand, calls to coarray operations like “puts” and “gets” will not cause a crash, as summarized in Table 1. However, coarray operations to and from a failed image, with or without the stat= attribute, cannot complete successfully; in such a case, the Fortran standard requires the “puts” to not have effect and the “gets” to return undefined data.

When the `stat=` attribute is present, the status variable will be set to one of three possible values: (1) zero in case the execution is successful; (2) `STAT_STOPPED_IMAGE` if the operation is involved with an image that has initiated normal termination (called *stopped image*); (3) `STAT_FAILED_IMAGE` if the operation is involved with a failed image. The exact value of `STAT_STOPPED_IMAGE` and `STAT_FAILED_IMAGE` are constant and processor-dependent; they are defined in the `ISO_FORTRAN_ENV` module.

Once a failure has been detected using the mechanism described so far, the most convenient way to get the list of failed images is through the `failed_images` intrinsic function. This function simply returns an array of integers representing the images that have failed from the beginning of the execution. In order to check for possible stopped images, the `stopped_images` function is also provided; similar to `failed_images`, it returns the list of stopped images since the beginning of the execution.

It is also possible to check the status of a group of images using the `images_status(images)` function. This function allows one to check the status of a group of images, defined by the image index specified by the argument passed to the function. Because the function is defined as “elemental” by the Fortran standard, if an array of integers is passed as argument, the same function is applied to each element of the array.

5. Resilience in CAF applications

In scientific computing, the most used approach for parallel programming is usually based on the data-parallel model. This strategy consists in dividing the data to be analyzed (used for computation) among the processes and performing communication when needed in order to get the final result. This approach is also called the “Owner-Computes rule”: each process performs all the computation involving the data it owns (same code executed on each process, applied on different data). After this computation phase, each process waits in a barrier for everyone to complete in order to exchange the results. People refer to this pattern as “bulk-synchronous” because it recalls the computation/communication phases of the Bulk Synchronous Parallel (BSP) model [22–24]. The data-parallel model and the bulk-synchronous execution style provide very high performance when all the compute units are homogeneous (in terms of speed) and the data partitioning is well balanced. In fact, having all the compute units running at the same speed, on the same amount of data, leads to perfect parallelization and thus, high performance. If some processes fail to complete their portion of work in due time, a large amount of time may be wasted during the wait at the barrier.

Load balancing is usually performed in a static way at the beginning of the execution. No matter what strategy is adopted for saving and restoring data, if the dead processes are not replaced, the information that they used to own must be partitioned among the active processes. In this scenario, if the load cannot be re-partitioned in an equal way, the unbalance created will impact dramatically the overall performance.

Detecting failures is thus of little help without the possibility of replacing the dead processes with new ones. Even though fault tolerance is critical for exascale computing, what is really needed is designing *resilient* applications. Snir et al. [1] define resilience in exascale computing as *the collection of techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults*. “Correct,” “timely,” and “efficient” are context-dependent. In some contexts “correct” may mean “bit reproducible”; in another context, it could mean “within a rounding error”. “Timely” and “efficient” are relative rather than absolute (as in before the hurricane arrives and within our power budget).

The general-purpose technique usually adopted in HPC relies on checkpointing and rollback recovery: when a failure is detected, the fault-tolerant protocol uses past checkpoints to recover the application in a consistent state. There are two main approaches for checkpointing: coordinated and uncoordinated checkpointing. In the coordinated checkpointing, all the images must ensure that their checkpoints are consistent. When a failure strikes, all the application stops and the execution gets restarted from the last checkpoint. In the uncoordinated checkpointing, each image checkpoints its own data independently. When a failure strikes, a replacing image recovers the data previously saved by the dead image and restart the execution. Uncoordinated checkpointing does not ensure global consistency, other images usually need to roll back to their own checkpoints.

Because the time for checkpointing on disk is the real bottleneck, in-memory checkpointing protocols try to mitigate this phenomenon using the main memory of other images as reliable storage. PGAS languages (and thus Coarray Fortran) are well suitable for this sort of protocols. In fact, every image can expose remote accessible memory usable for backing up the data belonging to another image as in a shared-memory environment.

5.1. Failed images implementation in Opencoarrays

The definition of the *failed images* feature and the definition of the ULFM proposal show lots of similarities in the way failures are detected, notified and managed. Furthermore, the ULFM proposal seems to be the preferred way to introduce fault tolerance support in the standard MPI-4. For this reasons we decided to use ULFM to implement *failed images* in the MPI-based version of OpenCoarrays.

The first step we took to use ULFM in OpenCoarrays was to associate a MPI Error Handler routine to be invoked every time a failure gets detected. This operations consists in two MPI invocations: one to create the Error handler using the function `MPI_Comm_create_errhandler` and one to associate the error handler to a specific communicator: `MPI_Comm_set_errhandler`.

The routine associated with the error handler is the core of the *failed images* implementation in OpenCoarrays. As we mention in Section 3, when a failure occurs the first thing to do is to invoke the routines `MPI_Comm_failure_ack` and `MPI_Comm_failure_get_acked` in order to identify which processes have failed. This part represents the “Failure notification” support provided by ULFM. OpenCoarrays keeps a list of failed images that gets returned when the intrinsic function *failed images* is invoked, as described in Section 4.1.1. To do so, OpenCoarrays has to create to MPI groups, one for the failed images and one for the surviving images, and translate the ranks of failed images into their original ranks in the global communicator. This last operation is performed by calling the MPI routine `MPI_Group_translate_ranks`.

At this point, the original communicator is no longer valid and the library needs to create a new communicator composed only be the surviving images. This step represents the “error recovery” support provided by ULFM. In order to create a new communicator, the `MPI_Comm_shrink` routine provided by ULFM has to be invoked. This routine creates a healthy communicator where all the ranks are in numbered sequence from 0 to the number of surviving images minus one. To reestablish the original ranks, the `MPI_Comm_split` routine has to be called. Finally, in order to get a consensus among all the surviving images about the correct creation of the new communicator, the routine `MPI_Comm_Agree` provided by ULFM has to be invoked.

Once a healthy communicator has been correctly created, OpenCoarrays has to make sure that no failed image was holding a lock or entered a critical section at the time of the failure (locks and critical sections belong to the standard Fortran 2008 and they have not been described or used in this paper).

```
real, allocatable :: critical_variable(:,:)
real, allocatable :: backup_critical_variable(:,:)


!in-memory checkpoint after iteration
backup_critical_variable(:,:)[buddy] = critical_variable(:,:)
```

Listing 1. CAF-based in-memory checkpoint.

5.2. Failed images and teams: Tandem for full resilience

The Fortran 2008 and Fortran 2018 coarray specifications do not allow to spawn new images at run-time. The most effective way to provide resilience in CAF applications is to use a small group of *spare* images that will replace the dead ones at run-time. This recovery strategy requires the use of both *Failed Images* and *Teams* features. *Failed Images* is needed to tolerate the failures and detect which images have failed. *Teams* allows to create two non-overlapping groups of images: active and spares.

In case of failure, the rank associated with the dead image must be assigned to a spare image and thus both teams, active and spare, must be re-formed.

6. Resiliency for Jacobi iteration code

In order to demonstrate the effectiveness of the strategy described in Section 5.2, we decided to use a Jacobi iteration code derived from the more complex computational fluid dynamics application described in [25]. The Jacobi code reproduces the "halo exchange" communication pattern used for solving partial differential equations, where the computational domain is divided into patches and the representation of each patch is held on a single process.

Currently, the only compiler supporting *failed images* is GFortran 7.1+ with OpenCoarrays 1.9+ as run-time library. A partial support for *teams* is currently available but it has not been included in the official release yet.

Creating two teams of images, running and spare, and performing a replacement in case of failure requires a fully functional implementation of *failed images* and *teams*. In order to overcome this limitation we assumed that the last *s* images are part of the spare images team. The number of spare images *s* can be defined either at compile time, as a fixed number, or at run-time as a percentage of the total number of images composing the application (e.g. 2% of the images are spare). For the spare images, all the code segments involving computation have been skipped with a combination of `if` and `goto` statements. The only commands executed by both teams are synchronization statements (e.g. `sync all`).

6.1. Recovery strategy and in-memory checkpointing

In a Jacobi iteration code, every image involved in the computation has a portion of the solution matrix. A communication phase is thus required during each iteration step in order to exchange the halo regions.

In order to make the Jacobi iteration code resilient, a recovery strategy is needed. For each iteration, the partial solution of the iterative method should be saved by each image on a persistent media. As mentioned in Section 5, in-memory checkpointing can be a viable solution when the cost of checkpointing becomes the bottleneck. Because coarray Fortran adopts a one-sided communication paradigm, the in-memory checkpoint on buddy memory can be easily implemented with a coarray "put" operation as shown in Listing 1.

The implementation of the in-memory checkpoint needs to be implemented by the programmer.

In our case, every image picks a "buddy" image that stores the critical data. The selection of the "buddy" image is based on an image index offset. For example, for an offset of two, image one will store its critical data on image three, image two will store the data on image four and so on. The critical data is the partial solution of the iterative method, which is represented by a 2-D Fortran array. The detection and notification of the failure(s) is performed only during the `sync all` statements at the end of the compute phase or at the end of the iteration step, as shown in Listing 2.

In case of failure(s), if the number of spare images is greater or equal than the number of failed images in that particular iteration step, the team of the computing images will be reformed by assigning the images index of the failed images to the spare images selected for the replacement, as shown in Listing 3. The replaced images will then get the critical data with a coarray get operation from the "buddy" images. After this recovery step, the computation can restart from the point where it was interrupted.

Although simple and effective, this recovery strategy may fail for several reasons that we will cover in Section 6.2.

The fault tolerant scheme proposed by Fenix [5,16,18,19] relies on in-memory checkpoint and it is very similar to what we propose in this paper. Anyways, since *failed images* only provides the tools to detect failures, the fault tolerant implementation of the Jacobi algorithm we present in this work is just one of many other possible implementations.

```
 !end of computation
 SYNC ALL(stat=stat_sync)
 if(stat_sync /= 0) then
   CALL RECOVERY ( ilevel)
!   We reset so that the living images know that there is no failure.
    stat_sync = 0
 endif
```

Listing 2. Failure Detection.

```
integer, allocatable :: ifailarr(:)

ifailarr = FAILED_IMAGES ()
nfailed = size(ifailarr)

if(nfailed > nspares) ERROR STOP

DO i = 1, size ( ifailarr)
      ifailed_pe = ifailarr ( i)
      spare = NUM_IMAGES() - nfailed
      SYNC ALL
      CALL INIT_NEW_SPARE ( ilevel , ifailed_pe , spare)
      CALL REPLACEMENT ( ifailed_pe , spare)
      nfailed = nfailed + 1
      nspares = nspares - 1
END DO
```

Listing 3. Recovery Subroutine.

6.2. Resiliency trade-off

In presence of failures, a fully resilient parallel application is very hard to implement and sometimes counterproductive. In the recovery strategy described in Section 6.1, if a failure strikes during the replacement of the failed images with the spares, the application will reach an inconsistent state that will prevent a successful recovery. Furthermore, if an image and its “buddy” fail at the same time, the critical data will be unrecoverable.

The trade-off between the level of resilience and code complexity is influenced by several factors such as: the nature of the application, the reliability of the machine, the number of processes, the compute node architecture and many others.

A good rule-of-thumb is to “protect” from failures only the portion(s) of the code that takes most of the running time, because more exposed to failures. In our case, we decided to protect from failures only the compute part of the code that takes most of the running time. In case of failures happening outside the “protected” portion of the code, the application terminates with an error message.

Two important parameters to choose properly are the number of spare images to reserve and the image index offset used for the “buddy” image selection. The number of spare images should be influenced by the reliability of the system and by the number of processes q that are allocated on a single node. In fact, failures are more likely to happen at node level instead of process level, thus it will be very likely to loose at least q images at time. The image index offset is almost completely influenced by the quantity q as well. In fact, it is important to store the critical data on a different compute node in order to reduce the probability of loosing an image and its “buddy” at the same time.

The optimal selection of these parameters is beyond the scope of this paper, but good insights can be found in [15].

Lastly, since we implemented an in-memory checkpoint mechanism, memory overhead is another critical factor to consider. In fact, in our implementation, we decided to backup the entire portion of the matrix computed by a single image in the buddy memory (2X memory overhead).

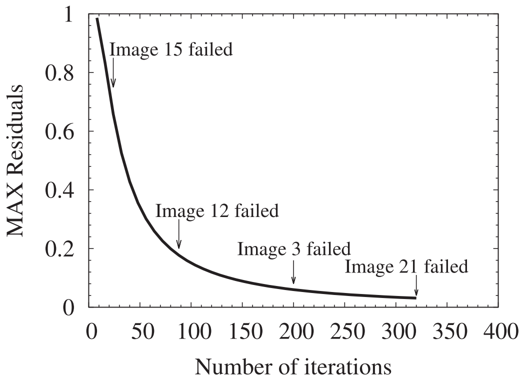

Fig. 1. Max residual.

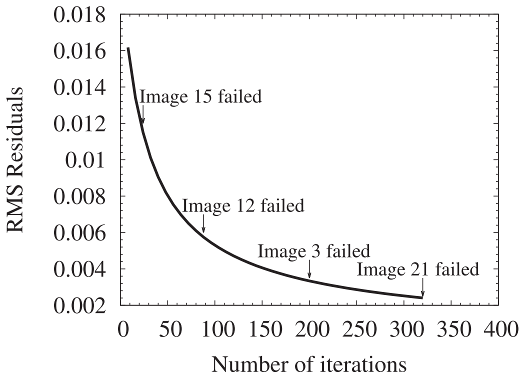

Fig. 2. Max root mean square residuals.

7. Results

The resilient Jacobi iteration code we implemented has been run on two different machines: (1) Mach: a Linux cluster equipped with Intel Xeon E5-4640 v2 processors running at 2.40GHz, each node composed by 4 processors, each of which equipped with 10 cores (2 threads per core), and (2) Cheyenne: NCAR's supercomputer. Each node is equipped with 2 Intel Xeon E5-2697 V4 processors running at 2.3GHz, each of which equipped with 18 physical cores, for a total of 36 physical cores per node.

On Mach, we decided to run the application using 32 images, 4 of which reserved as spare images. On Cheyenne, we decided to use 36 images total, using 4 images as spare.

In Figs. 1 and 2 we show the results obtained on Mach. Despite several failures happening over time, the application does not get affected and the numerical results in each iteration are correct. In particular, Fig. 1 shows how the maximum of the residuals between iterations decays despite the failures.

In Fig. 3 we show the impact of failures on the total execution time on Cheyenne. The execution time gets impacted by two main factors: (1) time needed to replace the failed image with the spare and get the data from "buddy" image; (2) performance degradation of the MPI implementation after failure. As shown in Fig. 3, the moment when a failure happens during the execution makes a big difference; when 4 failures strike during the early iterations ("early" line), the performance is worst. Because the time needed to replace the image and recovery is roughly constant, we conclude that there is a performance degradation of the MPI implementation when a failures happen. Such a degradation does not seem to get worse when multiple failures occur. In Fig. 3 we also report the performance of the non fault-tolerant version of the Jacobi code ("NO-FT" horizontal line). As expected, the non fault tolerant version is much faster than the fault tolerant version, even when no failures occur. This is mainly due to the overhead imposed by the data transfer needed to save the critical data in every iteration in the fault tolerant version.

Fig. 2 shows how the maximum of the Root Mean Square on the residuals decays as in a fault-free scenario.

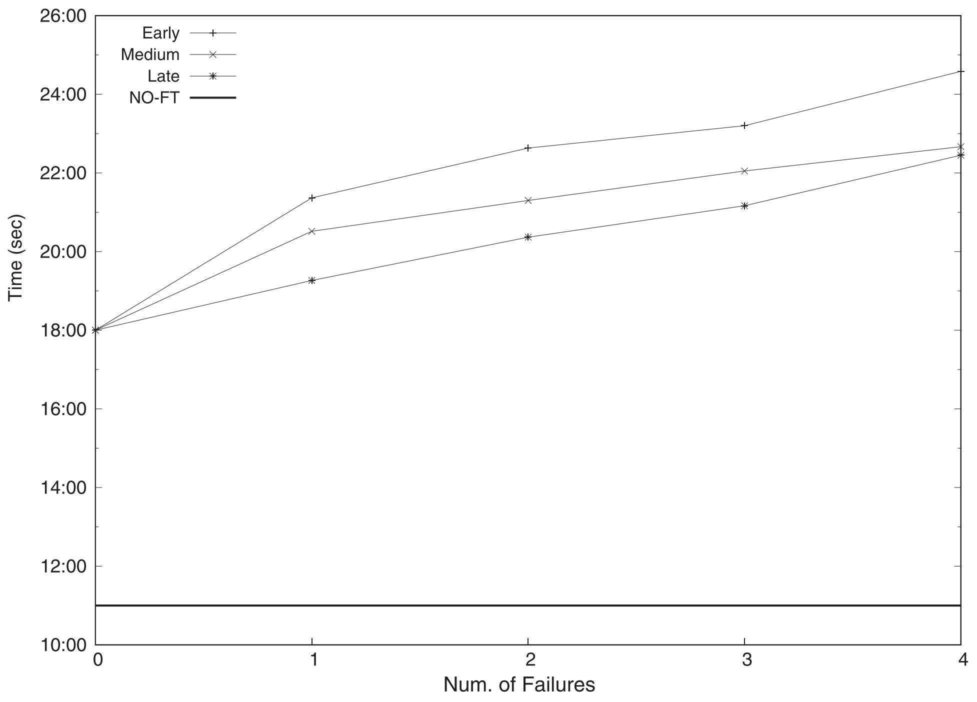

Fig. 3. Performance degradation after failures.

```
real, allocatable :: critical_variable(:,:)

call protect_variable(critical_variable, &
                      auto_select_buddy = .false.,&
                      buddy = buddy_image)
```

Listing 4. Library routine connecting variable and backup with explicit buddy selection.

8. Conclusions, future work and open questions

In this paper we presented a resilient Jacobi iteration code based on the *failed images* feature defined in the Fortran 2018 standard. At the time of this writing, GNU Fortran 7.1+ is the only Fortran compiler supporting such a feature using OpenCoarrays 1.9.1+ as run-time library. We proposed a general recovery strategy based on the presence of “spare images”, used as a replacement for failed images during the execution.

We also implemented a simple in-memory checkpointing mechanism based on CAF that has been shown effective in terms of performance and simplicity. In a future work we would like to explore the opportunity of combining the Burst Buffers technology (non-volatile RAM) with the in-memory checkpointing strategy used in this work.

Writing efficient and resilient parallel applications implies answering many different questions about the nature of the application, the system architecture and the desired level of resilience. The *failed images* feature defined in the Fortran 2018 standard, aims to provide a set of tools general enough to provide support to detect failures in virtually any possible application. Programmers may rather prefer some sort of “black box” tool, able to make the application fault tolerant just by specifying number of spare images and portion of the code to cover. Such an advanced tool is definitively attractive but quite difficult to implement in a general form. We wish to see in the future more of these advanced tools built on top of the *failed images* and *teams* features presented in this paper in the form of libraries for domain specific applications.

The definition of an Application Programming Interface able to provide high-level fault tolerance concepts to the users is beyond the scope of this paper, anyways Listing 4 provides a sketch of a routine used to declare a variable as critical and subject to replication on a specific buddy image. This routine replaces completely the code shown in Listing 1. This simple `protect_variable` routine alone relieves the user from manually implementing the backup strategy and buddy image selection, thus providing a lower programming overhead in the implementation of fault tolerant applications.

Acknowledgments

We gratefully acknowledge the support received from George Bosilca and Aurelien Bouteiller (University of Tennessee, Knoxville).

References

[1] M. Snir, R.W. Wisniewski, J.A. Abraham, S.V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A.A. Chien, P. Coteus, N.A. Debardeleben, P.C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, E.V. Hensbergen, Addressing failures in exascale computing, Int. J. High Perform. Comput. Appl. 28 (2) (2014) 129–173, doi:10.1177/1094342014522573.
[2] U. Consortium, UPC Language Specifications, v1.2, In: Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.
[3] R. Numrich, J. Reid, Co-array fortran for parallel programming, SIGPLAN Fortran Forum 17 (2) (1998) 1–31.
[4] B. Chamberlain, D. Callahan, H. Zima, Parallel programmability and the chapel language, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 291–312, doi:10.1177/1094342007078442.
[5] M. Gammel, D. Katz, H. Kolla, J. Chen, S. Klasky, M. Parashar, Exploring automatic, online failure recovery for scientific applications at extreme scales, in: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014.
[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, SIGPLAN Not. 40 (10) (2005) 519–538, doi:10.1145/1103845.1094852.
[7] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, D. Rouson, OpenCoarrays: Open-source transport layers supporting Coarray Fortran compilers, in: Proc. PGAS '14, ACM, 2014.
[8] C. Xie, Z. Hao, H. Chen, X10-ft: transparent fault tolerance for apgas language and runtime, in: Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '13, ACM, New York, NY, USA, 2013, pp. 11–20, doi:10.1145/2442992.2442994.
[9] A. Vishnu, H.V. Dam, W.D. Jong, P. Balaji, S. Song, Fault-tolerant communication runtime support for data-centric programming models, in: 2010 International Conference on High Performance Computing, 2010, pp. 1–9, doi:10.1109/HIPC.2010.5713195.
[10] J. Nieplocha, B. Carpenter, Armci: a portable remote memory copy library for distributed array libraries and compiler run-time systems, in: Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 533–546.
[11] C. Simmendinger, M. Rahn, D. Gruenewald, The gaspi api: a failure tolerant pgas api for asynchronous dataflow on heterogeneous architectures, in: Sustained Simulation Performance 2014, Springer, 2015, pp. 17–32.
[12] G.E. Fagg, J. Dongarra, Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world, in: Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,, Springer-Verlag, London, UK,UK, 2000, pp. 346–353. http://dl.acm.org/citation.cfm?id=648137.746632.
[13] J. Hursey, R.L. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, D.G. Solt, Run-through stabilization: an mpi proposal for process fault tolerance, in: Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface, EuroMPI'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 329–332. http://dl.acm.org/citation.cfm?id=2042476.2042516.
[14] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, J. Dongarra, An Evaluation of User-level Failure Mitigation Support in mpi, Springer, Vienna, Austria, 2012.
[15] T. Hérault, Y. Robert, Fault-Tolerance Techniques for High-Performance Computing, Springer, 2015, doi:10.1007/978-3-319-20943-2. https://hal.inria.fr/hal-01200479.
[16] M. Gamell, D.S. Katz, H. Kolla, J. Chen, S. Klasky, M. Parashar, Exploring automatic, online failure recovery for scientific applications at extreme scales, in: SC '14, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 895–906, doi:10.1109/SC.2014.78.
[17] K. Sato, A. Moody, K. Mohror, N. Maruyama, S. Matsuoka, FMI: Fault tolerant messaging interface for fast and transparent recovery, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2014, doi:10.1109/IPDPS.2014.126.
[18] M. Gamble, R.V.D. Wijngaart, K. Teranishi, M. Parashar, Specification of fenix mpi fault tolerance library version 1.0.1. doi:10.2172/1330192.
[19] M. Gamell, D.S. Katz, K. Teranishi, M.A. Heroux, R.F.V.d. Wijngaart, T.G. Mattson, M. Parashar, Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques, 2016 45th International Conference on Parallel Processing Workshops (ICPPW), 2016. 346–355. doi:10.1109/ICPPW.2016.56.
[20] R.W. Numrich, J. Reid, Co-arrays in the next Fortran standard, SIGPLAN Fortran Forum 24 (2) (2005) 4–17.
[21] ISO/IEC/JTC1/SC22/WG5, TS 18508 additional parallel features in Fortran, 2015. Aug.
[22] J.M.D. Hill, B. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, R.H. Bisseling, Bsplib: The BSP programming library, 1998.
[23] D.B. Skillicorn, J.M.D. Hill, W.F. Mccoll, Questions and answers about BSP, 1996.
[24] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111, doi:10.1145/79173.79181.
[25] S. Garain, D.S. Balsara, J. Reid, Comparing Coarray Fortran (caf) with mpi for several structured mesh pde applications, J. Comput. Phys. 297 (Supplement C) (2015) 237–253, doi:10.1016/j.jcp.2015.05.020. http://www.sciencedirect.com/science/article/pii/S002199911500354X.