Using ULFM to design resilient numerical schemes on HPC platforms

Master Degree Project

University of Perpignan Via Domitia

Faculty of Science and Engineering

High Performance Computing and Simulation

Author:

Said Amirouche

Supervisors:

Pr. Luc Giraud

Dr. Emmanuel Agullo

TABLE OF CONTENTS

Page

LIST OF FIGURES

Acknowledgements

I like to thank all the wonderful people whose support and encouragement made this project a possibility. I am highly indebted to my supervisors Luc Girgaud and Emmanuel Agullo for their guidance and constant supervision as well as for providing necessary informations regarding the project and also for their support in completing the project.

I would like to express my special gratitude to George Bosilca and Aurelien Bouteiller from the university of Tennesse for giving me such attention and time concerning the use of ULFM with linear solvers and Mawussi Zounon from the university of Manchester for his help in the numerical recovery strategies

I would like to express my gratitude towards my parents and my girlfriend and all my friends for their encouragement which help me in completion of this project.

My thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities.

CHAPTER 1

Introduction

As High-performance computing (HPC) systems grow computational power and scale, failure rates in these systems are expected to increase. However, although the Message-Passing Interface (MPI) is the prevalent programming model for large-scale HPC applications, it provides practically no support for fault tolerance, if a failure occurs, applications are either automatically aborted or can do little more than abort themselves, Several fault-tolerant models have been proposed for MPI, but none has been accepted in the MPI standard. Even under the most optimistic assumptions about the individual component's reliability, probabilistic amplification from using millions of nodes has a dramatic impact on the Mean Time Between Failure (MTBF) of the entire platform. The probability of a failure happening during the next hour on an Exascale platform is disturbingly close to 1; thereby many computing nodes will inevitably fail during the execution of an application.

Efforts toward fault tolerance in MPI have previously been attempted. Automatic fault tolerance is a compelling approach for users, as failures are completely masked and handled internally by the MPI library by aborting or ignoring them, which requires no new interface to MPI or application code changes. Unfortunately, many recent studies point out that automatic approaches, either based on checkpoints or replication, will exhibit poor efficiency on Exascale platforms.

Algorithm based fault tolerance (ABFT) is a class of approaches in which algorithms are adapted to encode extra data for fault tolerance at expected low cost. The basic idea consists in maintaining consistency between extra encoded data and application data. The extra encoded data can be exploited for fault detection and for loss data recovery. ABFT strategies may be excellent candidates to ensure the resilience of an application; however they induce extra costs for computing and storing the data encoding even when no fault occurs.

In this project, we present ULFM [13] which is a type of ABFT, that allows to detect the failure and to reconstruct the communicator in a way that the new respawned process takes the same rank as the dead one before, and we present numerical recovery strategies for linear system solvers who can be used once the communicator is restored as it was.To solve these systems of linear equations,which are used in many scientific and engineering applications, direct methods based on matrix decompositions, are commonly used because they are very robust. However, to solve large and sparse systems of linear equations, direct methods may require a prohibitive amount of computational resources (memory and CPU). To overcome the drawbacks of direct methods, iterative methods constitute an alternative widely used in many engineering applications. The basic idea is to approximate the solution of large sparse systems of linear equations, through successive iterations that require less storage and fewer floating-point operations. In addition to having attractive computational features for solving large sparse systems of linear equations, iterative methods are potentially more resilient. After a perturbation induced by a fault, the computed iterate can still serve as an initial guess as long as the key data that define the problem to solve, that are the matrix and the right-hand side, are not corrupted. the reminder of this project is organized as follows: the next chapter presents globally the techniques which allow fault tolerance in system level and user level, chapter 2 introduces ULFM, its mechanisms, functions and how it works and numerical recovery strategies used, chapter 3 an experimental example was analyzed,

CHAPTER 2

Background and related work

This chapter provides background informations about fault tolerance basics and recovery techniques

2.1 Overview of fault tolerance

The European Exascale Software Initiative (EESI) began their journey in the middle of 2010 with the hope of creating a common platform to tackle the issues that may arise in today's and upcoming HPC systems. This initiative seems as a driving force of participating in a competition among different nations for building the next generation supercomputers. For instance, in June 2011, the Japanese K computer achieved the number one placing on the TOP500 list of the world's fastest supercomputers, with a performance in excess of 10 petaflops. But today, china's Tianhe-2 replacing that positing, with a sustained performance of 33.86 petaflops, which is more that three times as powerful than the K computer. Since the rate of component failures of a system is roughly proportional to the number of cores of the system, an exascale system consisting of 1.000 times more cores than either the K computer or Tianhe-2 will certainly suffer more frequent component failures. The use of these parallel resources at large scale leads to a significant decrease of the mean time between faults (MTBF) of HPC systems.

The sources of such frequent failures include memory soft and hard errors, disks, file system or I/O node errors, network connection faults (fibers, connectors, etc.), operation system/runtime/library bugs, hardware errors (power supply, fans, water valves, water connectors, etc.), operators, user errors, and so on. An experiment was carried out in LANL HPC systems over a period of a few years to find out the root causes for system failures (both soft and hard). It is observed for these systems that hardware is the single largest source of faults with 64% of all failures assigned to this category. Software is the second largest contributor, with 18% of all failures, it is also important to consider that the number of failures obtaines from the undetermined cause is significant with 14%.

The set of possible solutions to deal with these failures is divided into three categories: the hardware approach, the system approach, the application approach.

The hardware approach will add additional hardware in an exascale system to deal with failures on the hardware level. Although this will require the least effort in porting current applications, it will incur additional power consumption in the system. Moreover, as the hardware in exascale system becomes more complex, the software will become more complex. In this scenario, new complexities arise in the system due to the introduction of additional hardware.

In the system approach, fault tolerance is achieved by applying both the hardware and system software in such a way that the application codes remain unchanged. Although it may be convincing that changing the system software is less costly than that of the hardware, this approach may add additional complexities in the system and, hence, may increase its energy consumption.

2.2 Failure Recovery Techniques

The classical failure recovery techniques are as follows.

Checkpoint/Restart

The classic system level/automatic fault tolerance technique is the Checkpoint/Restart [1] . It generally means the process of periodically storing the whole state of a computation in disk space such that its execution could be restarted from its recent saved state in the event of a failure. In other meaning once a failure happens the computation start working not from the initial state but from the saved state. This storing is operated in local disk storage or in remote disk storage or in both. In case of storing the state of large scale application, each of the tens of thousands of processes writes several gigabytes of data. This increases the overall checkpoint volume in the order of several tens of terabytes. Furthermore, an application will no longer progress if the MTBF is shorter than or equal to the application restart time. In such a scenario, without performing any actual computation, application starts next checkpoint just after restarting from the recent checkpoint

Diskless checkpointing

The diskless checkpointing approach [2] is proposed to reduce the overhead of the checkpoint/restart approach. It stores a reduced volume of checkpoint state data into compute node own memory without going to disk. It also needs some extra nodes to save a checksum of the memory checkpoint states so that it could be used to recover the memory checkpoints of the failed nodes. Although the performance of this technique is better than that of the disk-based Checkpoint/Restart method, there is a potentially significant memory I/O overhead to this method, especially in memory intensive applications. Moreoever, the number of additional nodes to store the checksum will grow in proportion to the number of nodes running the application.

Replication

the replication [3] technique is proposed to solve the problem of large time overheads of the Checkpoint/Restart technique and to exempt the requirements of storing checkpoints on memory of the diskless checkpointing approach. The idea of replication is that most applications leave some "wasted" spaces on the cluster machines on which they are executed. In order to efficiently use those spaces, multiple copies of the application are executed simultaneously. If there is any failure occurs one of the replicated processes taking the carge of the original version of the application and the computation can continue onwards. This technique is applicable for some types of machines, especially those where the system utilization is not greater than 50%.

Message logging

The message logging technique [4] is proposed to reduce the rollback overhead of the Checkpoint/Restart technique. It involves all processes to checkpoint their states without coordination and logging all communication operations in a stable media. Thus, in case of any failures, this log is analyzed to restart the execution of only the crashed processes, rather than every processes, from the recent local snapshot, and establishing the same communication with the help of the saved communication log. However, the overhead of this technique is proportional to the communication volume of the application. A significant amount of penalty is added by this technique for all messages transferred even if there is no fault throughout the whole computation.

Task pools with reassignment

The drawbacks of the replication technique can be solved by the task pools with reassignment technique [5] . Instead of running multiple copies of the application simultaneously, it splits the work into some discrete tasks by a manager at run-time. These tasks then can be executed by any worker node. In the event of a failure, there is a provision of reassigning the affected tasks to other nodes. It represents a very effective method for implementing fault tolerance and is already used in. However,

this method can be vulnerable to the failure of the manager node responsible for scheduling.

Proactive migration

The proactive migration technique [6] is proposed to solve the problem of recomputing the affected tasks from the beginning of the task pools with reassignment technique. In order to avoid the recomputation, it predicts the failures in nodes before they really happen and moves the running applications away from them before the fault occurs. Theoretically, this would allow applications to run on fault-prone systems without any modifications. But practically, the performance monitoring of the nodes must occur sufficiently quickly that the application can be migrated before the failure does occur. Otherwise, it will not be applicable. The success of the proactive fault tolerance solutions depends solely on the accurate prediction of the failures and the ranges of failures that it could cover. Prediction techniques used to achieve fault tolerance should incur less overhead, as well as, the work lost due to wrong prediction should be small.

Algorithm-Based or User Level Fault Tolerance

In the Algorithm-Based Fault Tolerance (ABFT) [7] technique, numerical algorithms are modified to include methods for detecting and correcting errors. An extension of this is to develop new algorithms that are naturally resilient to faults. The major advantage of dealing with faults at the algorithm level is that the time-to-solution is roughly unchanged in the presence of faults. There may be an impact in terms of some loss of accuracy, but in many cases these are an acceptable compromise in order to guarantee a solution within a given time window.

2.3 MPI level fault tolerance

There are several MPI level fault tolerance techniques available. There are as follows.

CoCheck

The CoCheck [8] MPI is a combination of the Checkpoint/Restart and migration techniques. It uses a single process checkpointer which plays an important role of migrating the processes by saving the in-flight messages in a safe buffer and clearing the channel. This is achieved by exchanging a special message between the processes to indicate the clearance of the channel. If a process receives the special message, it assumes that there is no in-flight message left in the channel. Otherwise, it stores the special message in a special buffer as this is the in-flight message. When finally a process collects either the special or in-flight messages from all the processes destined to this process, it assumes that there is no ongoing message left in the channel. Hence, processes can now safely migrate with the checkpointer.

Starfish

The Starfish [9] technique combines group communication technology and the Checkpoint/Restart technique. This group communication technology allows the application to run without any disruption in the event that some of the nodes fail. Failure recovery is achieved by recomputing the part of the application which are disrupted due to failures. Recomputation is done either from the beginning, or from the recent checkpointed state by the Checkpoint/Restart technique. This is usually performed on the extra nodes added to the application by the Starfish on-the-fly. This run-time node adding feature allows Starfish to migrate the application processes from one node to another node with the assistance of the Checkpoint/Restart technique. Moreover, Starfish has the capability of performing both the application independent and application dependent fault tolerance.

MPI FT

MPIFT [10] is a fault-tolerant version of MPI. It performs failure recovery by means of reassigning tasks to the replacements for the dead processes. A detection technique is used for the detection of process failures. A centralized monitoring process (called the Observer), responsible for notifying the failure event to the rest of the alive processes, performs the recovery action. There are two modes of the recovery action. The first one performs distributed buffering of message traffics on each process. When the Observer detects process failures, it performs recovery by resending buffered messages from all the processes to the replacement processes, those are originally destined for the dead ones. The second one is based on the idea of centrally storing every message traffics by the Observer, and resend these to the replacements for the dead processes.

One of the problems encountered with MPIFT for performing recovery action is the recovery of the dead communicator(s). This problem is solved by either preparing the spawning communicators in advance, covering every cases of the failure event by creating and using a communicator matrix, or spawning the replacement processes when the program starts executing. The disadvantages of this technique are that it pauses the synchronization due to the collective operations responsible for spawning the communicators, and the alive processes on the new communicator may have some pending messages destined for the old communicator which need to be synchronized

MPICH V

MPICHV [11] is a fault-tolerant version of MPI combining features from the uncoordinated Checkpoint/Restart and message logging techniques. With the MPICH-V run-time support, any application written in standard MPI could be made fault-tolerant. The key idea is that a Dispatcher coordinates the whole application execution by periodically collecting "alive" messages from all the nodes, and at the same time keeps records of all the communications in stable Channel Memories (CM). In addition to this, every node checkpoints their task images to a Checkpoint Server

(CS). If any "alive" message is not received for a certain period of time, the Dispatcher assumes that the particular node is dead, and restarts the execution from the point of failure with the support of the CS. By this time, if that faulty node rejoins the system, the duplicate instance removal is managed by the CM. Moreover, network connection management for the alive and dead nodes is achieved by the CM.

FT MPI

FT-MPI [12] offers a number of options for automatic process-level fault tolerance within the MPI library itself. This is achieved by simply calling a new communicator creation function, such as MPI_COMM_DUP or MPI_COMM_CREATE, in the application, with almost no impact on the user code.

FT-MPI provides the following three types of recovery modes to chose from by an application developer.

- The first recovery mode is SHRINK, which builds a new communicator excluding the failed processes and shrink the communicator size. Although the alive processes are ordered in the communicator, but for some applications where computation depends on the consistent values of the local ranks, this shrinkage could cause problems.

- The second recovery mode is BLANK. This is similar to SHRINK in the sense that all the failed processes are removed from the reconstructed communicators. However, without shrinking the communicator size, it replaces them with invalid process ranks. Although communication with the invalid ranks causes error, but those are left for future development to replace with new processes so that there is no disruption in inter-process communication.

- The third and most well-supported recovery mode is REBUILD. It automatically replaces failed processes by the newly created processes. The original communicator size and the process rank orders are left unchanged. With default communicator (MPI_COMM_WORLD), newly created processes are auto

matically restarted with the same command-line parameters as the original processes. However, for the other communicators they must be reconstructed manually.

Although FT-MPI had lots of functionalities to provide the fault tolerance support, it was never adopted into the MPI standard due to the lack of standardization, and its development was discontinued.

ULFM MPI

User Level Failure Mitigation (ULFM) [13] , MPI can be considered as an attempt of resolving the non-standardization issue of FT-MPI. The MPI Forums Fault Tolerance Working Group began the implementation of standard fault-tolerant MPI by introducing a new set of semantics on the top of the existing standard MPI library. Semantics of the draft standard include the detection and identification of process failures, propagating the failure information within the alive processes in the faulty communicator, and so on. Usually process failures are detected by checking the return code of the collective communication routines. With the run-through stabilization mode [Fault Tolerance Working Group] of ULFM MPI, surviving processes can continue their operations while others fail. The alive processes can form a fully operational new communicator without getting any disruption from the dead processes. It is also possible to create the replacement processes for the failed ones to recover the original communicator. Based on the requirements of the application, either the local or global recovery is also possible.

ABFT techniques could be easily implemented on top of ULFM MPI, which presents actually the project that i worked on during my internship

CHAPTER 3

Application Level Fault Recovery with ULFM MPI

3.1 Introduction

User Level Failure Mitigation (ULFM) is a draft standard for the fault-tolerant model of MPI [13]. Implementation of this standard is initiated by the MPI Forums Fault Tolerance Working Group by introducing a new set of tools into the existing MPI to create fault-tolerant applications and libraries. This draft standard allows application programmers to design their recovery methods and control them from the user level, rather than specifying an automatic form of fault tolerance managed by the OS or communication library.

ULFM works on the run-through stabilization mode where surviving processes can continue their operations while others fail. Any MPI operation that involves a failed process will raise an MPI exception rather than waiting for an indefinite period of time to succeed the operation. If a point-to-point communication operation is unsuccessful due to a process failure, surviving process reports the failure of the partner process. With collective communication where some of the participating processes fail, some processes perform successful operations while others report process failures, which leaves the state as non-uniform. In such a scenario, the knowledge about failures is explicitly propagated and prohibit any further communication on the given communicator by setting the communicator in the revoked state.

3.2 The main ULFM Functions

MPI_COMM_FAILURE_ACK & MPI_COMM_FAILURE_GET_ACKED [14] : These two calls allow the application to determine which processes within a communicator have failed. The acknowledgement function serves to mark a point in time which will be used as a reference. The function to get the acknowledged failures refers back to this reference point and returns the group of processes which were locally known to have failed. After acknowledging failures, the application can resume MPI_ANY_SOURCE point-to-point operations between non-failed processes, but operations involving failed processes (such as collective operations) will likely continue to raise errors.

MPI_COMM_REVOKE [14] : Because failure detection is not global to the communicator, some processes may raise an error for an operation, while others do not. This inconsistency in error reporting may result in some processes continuing their normal, failure-free execution path, while others have diverged to the recovery execution path. As an example, if a process, unaware of the failure, posts a reception from another process that has switched to the recovery path, the matching send will never be posted. Yet no failed process participates in the operation and it should not raise an error.

The revoke operation provides a mechanism for the application to resolve such situations before entering the recovery path. A revoked communicator becomes improper for further communication, and all future or pending communications on this communicator will be interrupted and completed with the new error code MPI_ERR_REVOKED.

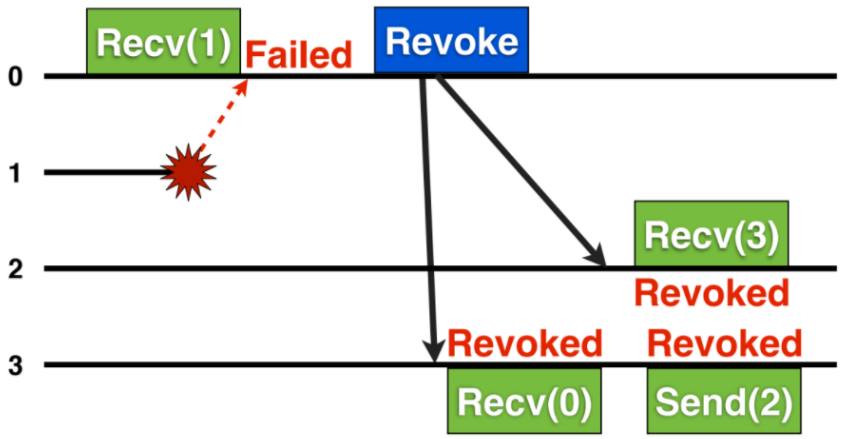


Fig. 3.1. The concept of the Revoke function

MPI_COMM_SHRINK [14] : The shrink operation allows the application to create a new communicator by eliminating all failed processes from a revoked communicator. The operation is collective and performs a consensus algorithm to ensure that all participating processes complete the operation with equivalent groups in the new communicator. This function cannot return an error due to process failure. Instead, such errors are absorbed as part of the consensus algorithms and will be excluded from the resulting communicator.

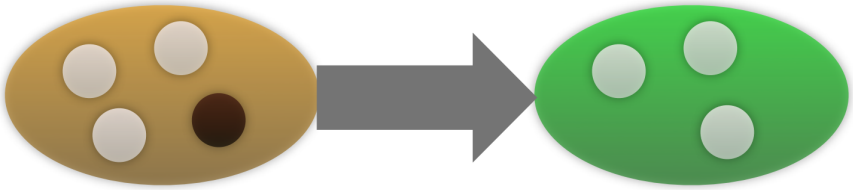


Fig. 3.2. The concept of the Shrink function

MPI_COMM_AGREE [14] : This operation provides an agreement algorithm which can be used to determine a consistent state between processes when such strong consistency is necessary. The function is collective and forms an agreement over a

boolean value, even when failures have happened or the communicator has been revoked. The agreement can be used to resolve a number of consistency issues after a failure, such as uniform completion of an algorithmic phase or collective operation, or as a key building block for strongly consistent failure handling approaches (such as transactions).

3.3 Fault Detection and identification

It is considered as the most important step for the implementation of resilience on any application by detecting if any component failure occurs, and list them if they occur.

For that, there are two ways to do it, the first by taking advantage of the MPI error handler, which is set by default to MPI_ERRORS_ARE_FATAL, the communicator's error handler must be changed to MPI_ERRORS_RETURN and then some necessary modifications must be done in the source code. And the second way is by creating and attaching a customized error handler to each Communicator,the customized error handler includes ULFM functions like the MPIX_Comm_failure_ack() and MPIX_Comm_failure_get_acked() functions to acknowledge and list the local failures.

An application can either check the return code of the collective functions like MPI_Barrier() or MPI_Allreduce() or check the return code of particular communications between processes. With failure it returns code other than MPI_SUCCESS, and propagates the knowledge about the failure with MPI_Comm_agree, and in the same time it revokes the communicator with MPI_Comm_revoke so that no communication can occur in the communicator until it is fixed. Then it shrinks the communicator containing only the surviving processes by the MPI_Comm_shrink() function call. Then an old and new groups are created from the broken and shrunk communicators, respectively, by the MPI_Comm_group() function call. Finally, these two groups are compared by the MPI_Group_compare() and MPI_Group_difference() function calls to

create a group containing only the lost processes, and then this group is translated to create a globally consistent list of failed processes by the MPI_Group_translate_ranks() function call.

3.4 Fault Recovery

The second step of fault-tolerant implementation after detecting and identifying component failures is to reconstruct the faulty communicator. There are two ways of accomplishing this. One is to reconstruct the faulty communicator by preserving the original communicator size and rank distribution by creating and restoring the replacement processes for the dead ones, and the other one is with sacrificing the original communicator size by excluding the dead processes.

3.4.1 Shrinking & Spawning based communicator reconstruction

the reconstruction of the faulty communicator is done firstly by knowing the failed processes ranks using the features of ULFM functions MPIX_Comm_failure_ack() and MPIX_Comm_failure_get_acked(), after revoking the communicator with MPIX_Comm_revoke(), the survival processes are put in a temporary communicator by shrinking the old one using MPIX_Comm_shrink(), then create a new process by spawning the replacement MPI process with the MPI_Comm_spawn() function call. With process failures, spawning is done for the processes which experience process failures. The spawned process which is the failed one is considered as Child process and the survival processes from the old communicator as Parents. each own has his own intercommunicators through which they can communicate between themselves, in order to put them together in the same one, a merging step between their intercommunicators is necessary with the MPI_Intercomm_merge(). After the merge, the rank of the Child process should be the same as it was before in the old communicator (in the initialization of the execution), to do so an ordering of ranks by the

MPI_Comm_split() is done using features of the functions MPIX_Comm_failure_ack() and MPIX_Comm_failure_get_acked(),.

The Figure 3.3 shows a demonstration of the communicator reconstruction's method.

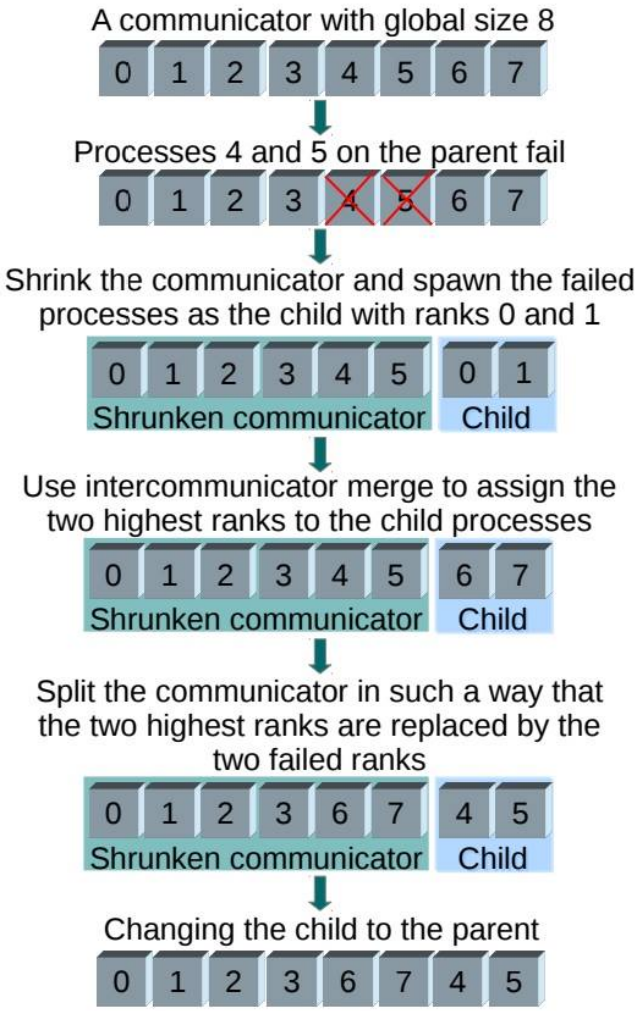


Fig. 3.3. The reconstruction of a faulty communicatorś steps

3.4.2 Shrinking based communicator reconstruction

Fault tolerance in this type of recovery is achieved without creating another processes, after the detection of a failure of a process, the failed communicator is shrunk,

it contains only the survival processes to continue the computation, after the shrink phase, the data structure is updated for the survival processes.

For example, suppose that an application runs with 16 processes, and the process 6 and 13 fail, after the detection of the failure and entering the maintaining phase, the communicator is shrunk excluding the processes 6 and 13, and update the data structure for the remaining 14 processes.

3.5 Lost data recovery

The final step for the implementation of the fault tolerance after the construction of the communicator is to recover the lost data of the failed processes. For that there are many recovery methods which have been mentioned in the previous chapter, but for the type of work that we worked on (the linear solvers with iterative methods) we restrict ourselves to some of the strategies that behave and give good results after the recovery. On parallel distributed platforms, the crash of a node is usually easy to detect. we consider the solution of Sparse linear system

$$Ax = b,$$

where A is a nonsingular matrix, the righthand side b and the solution x are vectors; they represent all the data of the problem.

We assume that the lost data is categorized into three categories : *Computational environment*, *Static data* and *Dynamic data*. The first one is the general data needed for the computation, including the source code of the program, MPI environment variables). Static data are the data initialized in the beginning of the execution, and it remains unchanged during the computation. It represents the input data of the problem, which represents the matrix A and the vector b. Dynamic data is the data generated during the computation, they represent the results of computations, in our case it represents the vector x. When a failure happens, all the data in its memory are lost, by that it means that its Static data and Dynamic data are lost.

In the rest of this section, we present the strategies used in this work, ones that are used in global context and applicated in any context or MPI Application type, and a specific strategy for the linear solvers and iterative methods.

3.5.1 Reset

This method represents the simplest reaction to a failure, it consists in re-initializing the data for the failed processes so that they can restart the work from the beginning. It restores just the Static data of the failed processes at their initial values.

The Reset method is widely used in parallel programming, but the efficiency of its results depends on the type and the context of the MPI application, for the iterative methods its results has a big impact in the speed of the convergence, it might assure the convergence if not too many failures occur.

3.5.2 Checkpoint/Restart

Its one of the classic system level fault tolerance technique, basically it stores the state of the state of data in iteration k in a disk space or a node for the disckless checkpointing, and loads it only if a failure occurs, in general, it stores the dynamic data which represent the vector x the solution of the linear system. Its default is that it takes extra sources (disk space) for the storage, and if the MTBF (mean time between failures) is shorter than or equal to the application restart time, the convergence will not be affected, without performing any changes for that, the application starts the next checkpoint just after the restarting from the previous checkpoint

3.5.3 Linear interpolation

The Linear Interpolation (LI) strategy [15], its principe consist at interpolating the lost data by using the data generated from the survival processes. Let $x^{(k)}$ be the approximative results of the linear system at the iteration k when a fault occurs,

the entries of $x^{(k)}$ are known for all the survival processes except the one that has dead. The LI strategy computes a new approximate solution by solving a local linear system associated with the failed one. If the node p fails, $x^{(LI)}$ is computed via

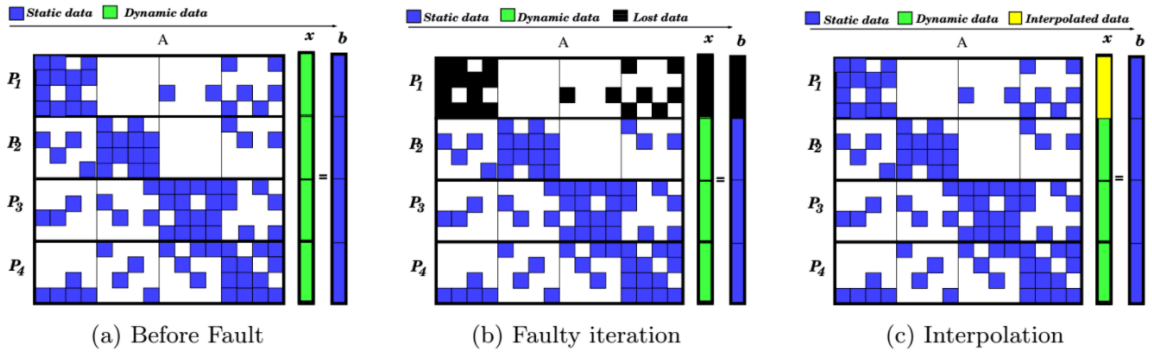
$$\begin{cases} x_{I_q}^{(LI)} = x_{I_q}^{(k)} & \text{for } q \neq p, \\ x_{I_p}^{(LI)} = A_{I_p,I_p}^{-1}(b_{I_p} - \sum_{q \neq p} A_{I_p}, I_q x_{I_q}^{(k)} \end{cases}$$

The motivation for this interpolation is that, at convergence, it regenerates the conservative solution ($x^{(LI)} = x$) as long as A_{I_p,I_p} is non singular. Furthermore it is shown that such an interpolation exhibits a property in term of A-norm of the error for symmetric positive definite (SPD) matrices as expressed in the proposition below

Proposition 1. *Let A be SPD, let $k+1$ be the iteration during which node p crashes, the regenerated entries $x_{I_p}^{(LI)}$ defined by the previous equation are always uniquely defined. Furthermore, let $e^{(k)} = x - x^{(k)}$ denote the forward error associated with the iterate before the fault occurs, and $e^{(LI)} = x - x^{(LI)}$ be the forward error associated with the recovered iterate computed using the LI strategy, we have :*

$$||e^{(LI)}||_A \leq ||e^{(k)}||_A \tag{3.1}$$

In the general case whether for SPD or non-SPD matrix, it can be noticed that the LI strategy is only defined if the diagonal block A_{I_P,I_P} has full rank. The figure 3.4 demonstrates how LI strategy works. Before the fault occurs (a) all the processes compute using the Static data each one its own Dynamic data, once a fault occurs the Dynamic and the Static data of the failed process are lost (b), after the phase of reconstruction of communicator and reset the static data of the new spawned process, its dynamic data are interpolated using the dynamic data of the other processes and its own static data.



(a) Before Fault (b) Faulty iteration (c) Interpolation

Fig. 3.4. The Linear interpolation method

CHAPTER 4

Experimental Example

4.1 Introduction

In this section, we introduce the use of ULFM for the design of a resilient linear system solver, specially for the iterative methods (Jacobi, Gauss Seidel, Krylov, etc.). The solution of the linear systems represents a perfect context for fault tolerence using ULFM. because of its type of communications, which are built on Point-To-Point communication. like MPI_Send and MPI_Recv to assure the communication between different processes, and the collective ones too like MPI_Reduce, MPI_Barrier, MPI_Allreduce. The detection and identification of a failure can be done in those two types of communication depending on the strategy of reconstruction of the communicator (Shrink & Spawn based reconstruction, or Shrink based reconstruction).

Each iterative method that runs in parallel mode, the first step is to distribute the data to all the processes, then these processes communicate with each other for data, informations , etc., once they finish communicating, they compute each one their own work in their local memory, and the last step is to reduce the results whether to one process or to all processes. The scenario of failures can appear any where in one of these steps.

To study and design a resilient scheme for any type of linear solvers, we choose the solution of the *Laplace Equation* [16] in 2D with finite differences using jacobi iterative method as an application to set fault tolerance using ULFM on it.

Laplace Equation

The Laplace equation [16] is a second-order partial differential equation which is writen as :

$$\Delta^2 \varphi = 0 \tag{4.1}$$

for our experimentation, we solve the discrete Laplace equation in two dimensions, discreted with finite differences, the amount of work resumes in one single computation

```
while (not converged) {
  for (i,j)
    xnew[i][j] = (x[i+1][j] + x[i-1][j] + x[i][j+1] + x[i][j-1])/4;
  for (i,j)
    x[i][j] = xnew[i][j];   }
```

For simplicity, consider a $m \times m$ mesh on n processes :

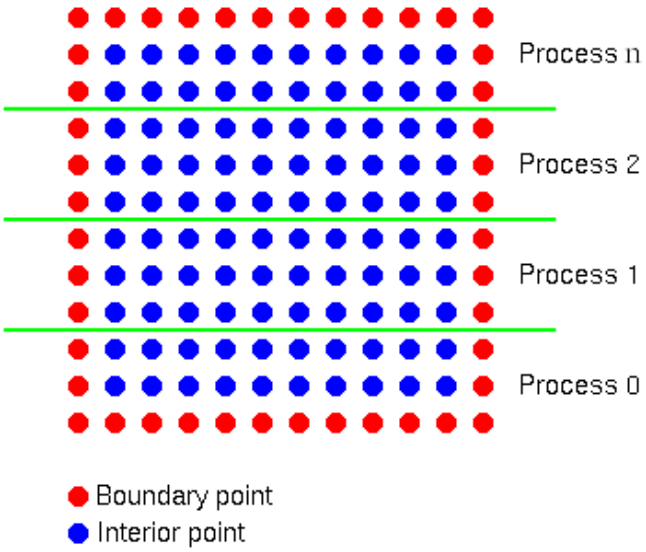


Fig. 4.1. A 12 x 12 mesh on 4 processor

4.2 Building Blocks

In general, making a MPI application resilient consists in updating the structure of the source code of the problem, by adding the feature ULFM functions and making the necessary changes in the source code. For the linear solvers with iterative methods, the resilient code has basically the same structure. We can see it as Building Blocks for the final resilient MPI version of the problem.

To construct a resilient MPI application, we need a set of blocks, each one will contribute for the composition of the final resilient MPI application it can be seen as the composition of 4 blocks, one that contains the main goal of the application, the other is seen as a reference for the use of ULFM to build a resilient application, and the two left blocks are designed to recover static and dynamic data.

4.2.1 MPI numerical algorithms

It represents the main problem to solve, the simple working MPI source code without fault tolerance, it can be any kind of MPI application. For our case it represents the solution of The *Laplace equation with difference finite in 2D*

4.2.2 MPIX reference rebuild steps

There are many ways to use ULFM, reconstructing the communicator just by shrinking the dead processes, identifing the error, revoking the communicator and continue working, but for our case, we need these main steps

Shrink Create a new communicator that contains only the survival processes.

Get knowledge Know the processes that failed and their informations (i.e. rank) which will be used for respawning just after and for rank re-assignment in the spawning phase.

Respawn Create processes that will replace the dead ones after the fault occurs, the creation is based on the assignment on the same rank to keep up the semantics of the problem correct.

Reconstruct Merge the new safe communicator with the respawned processes in a way that each one takes its previous rank as in the old communicator.

4.2.3 Recovery schemes for static data

Re-initialize the static data which represent in our example the matrix A and the vector b just by initializing the data given at the beginning of the solution.

4.2.4 Numerical recovery schemes for dynamic data

There are several ways to recover the dynamic data which represents the data that we obtain after computing, We used for our example *Reset, Checkpoint/restart and Linear interpolation* that we introduced previously in this chapter.

4.3 Generate the faults

To see the effictiveness of ULFM, and observe its behavior with faults, we need to generate some faults, the faults in the real context can originate from many sources: memory soft and hard errors, disks,file system or I/O node errors, network connection faults (fibers,connectors, etc.), operation system/runtime/library bugs, hardware errors (power supply, fans, water valves, water connectors, etc.), because we cannot generate such errors by ourselves, the only errors that can be done are whether killing the processes from the exterior with the command *Kill ¡PID¿* from the terminal, or from the interior of the program by simulating a suicide scenario with the function *exit()*.

4.3.1 The expected scenarios

Generating the faults from the exterior represents a scenario more realistic to observe the fault tolerance of the solver then the simulation of suicide, because the simulation of suicide is not random and so we cannot be sure that it works for all cases. In the example of linear solvers with iterative methods there is four main parts in the code where a fault can occur

- **Before and during the communication phase**. In the beginning of a computation, the processes are initialized then they communicate between themselves to exchange data or to synchronize, an error can occur during the phase of sending and receiving data between processes.

- **During the local computation**. Once the communications are finished, each process compute the data on its own locally.
- **During the reducing of the results**. After obtaining the local results from each neighbor process, the final step is to reduce the results to compute the stopping criterion
- **During the reconstruction of the communicator**. We suppose that an error occured in the previous iteration, so the program enter in a maintaining phase to reconstruct the communicator and respawn a new processes if needed, another error can occur during this maintaining phase, which requires re-entering to that phase again to fix the second error just before entering to Numerical data recovery.

4.4 Failure detection and identification in iterative methods

As mentioned in the previous chapter, in order to determine and detect a failure, there are two type of detection, a detection between two processes in communication, and detection in the collective functions.

4.4.1 Point to Point communication

In order to compute, each process needs some data from its neighbor processes which represents the adjacent values for each one, we can call them the ghost-points. A communication of the type SendRecv is needed for this type of work, one way to detect errors is to check the return value of the Send or the receive, if the return value is MPI_ERR_PROC_FAILED, the process that detects the error send the failure notification to the other processes with Revoke, the other processes will receive value of MPI_ERR_REVOKED, so they stop computing and enter to the maintaining phase to fix the problem. as shown in the figure 4.2.

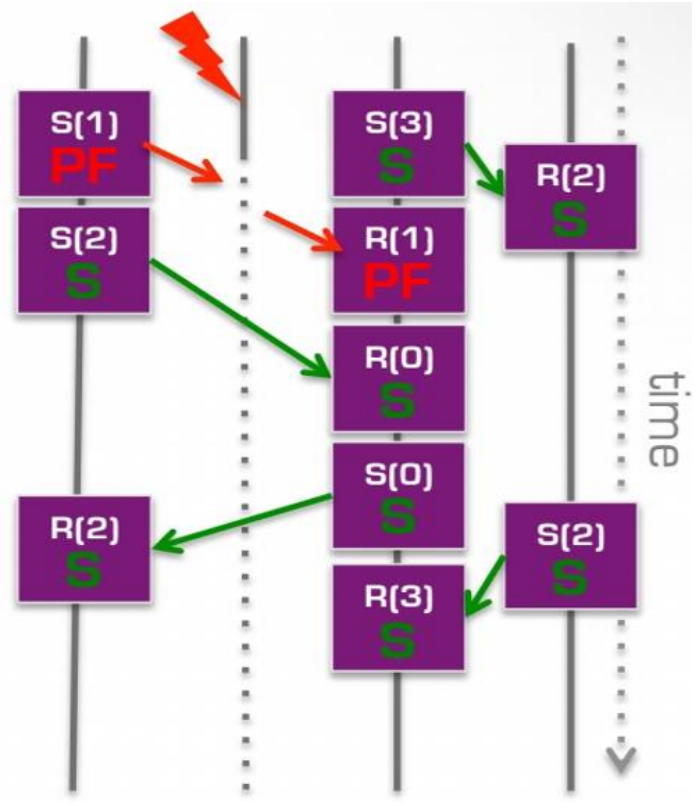


Fig. 4.2. Failure detection in Point to Point communication

4.4.2 Collective communication

Once the ghost-points are exchanged between processes, each one computes the new iterate locally, if an error occurs at this level, its identification is impossible, because each one works without enter in contact with the others, but it can be detected when the results are reduced throw collective functions MPI_Reduce and MPI_Allreduce, as the Point to Point detection method, we check the return value of one of these functions : MPI_reduce, MPI_Barrier, MPI_Allreduce, if it is not MPI_SUCCESS the processes with be informed that the communicator is revoked and it will enter in maintaining phase

4.5 Reconstruction of the communicator

In the beginning we initialize data structure over n processes, and if p processes fail, it would be impossible to continue working without them unless we change completely the data structure and distribute it on the $n - p$ remaining processes. Thats why on our communicator reconstruction, we choosed Shrink & Spawn based recovery, that allows us to make sure that the communicator will be restored as it was before with n processes without changing the data structure and without changing the way the processes communicate with each other. The Shrink & Spawn based recovery was explained in the previous chapter.

4.6 Numerical recovery methods

In this type of MPI Applications, the choice of the Numerical recovery method is some thing very important, because each method has its advantage and has its defaults. As explained in the previous chapter after the reconstruction of the communicator, the spawned process needs to have the same data that the failed one had before the fault occurs. For this example of Jacobi iteration we choose to apply the *Reset*, *Checkpoint/Restart*, and *Linearinterpolation*, for the *Reset* it's done by re-initializing the *Static data* of the failed processes so that they can restart the work from the beginning, The strategy doesn't cost so much, it's easy to be apply but the efficiency of its results has a big impact in the accuracy of the convergence, it assures the convergence. The *Checkpoint/restart* on its turn it's done by saving the state of the whole computation on each iteration and rollback to that iteration's computed data if there is a failure, it assures an efficiency of convergence but it's a costy strategy due to the stores in each iteration even if there is no failure. and for the *Linear Interpolation* (LI) we interpolate the data using the computed data of the survival processes, and since our example is a mesh of $n \times n$ not of the form $Ax = b$ so the interpolation is a local computation of the initialized *Staticdata* of the new process with the use of the ghostpoints of the computed x of neighbors. This strategy assures

the efficiency with a bit loss of accuracy in term of number of iterations needed to convergence to the solution.

4.7 Coordinated and Uncoordinated iterations

In the numerical recovery strategies used the processes are coordinated, means that all the processes must achieve the iteration k before passing to the iteration $k+1$, in other term the processes run each iteration simultaneously. while for the uncoordinated version, the processes can be at the iteration k while others in iteration $k+1$ or $k-1$, the uncoordination between the processes can impact on the convergence in term of number of iteration needed for that. the uncoordinated version can seen as a version of Jacobi iteration but with possible delay, known as chaotic relaxatin.

4.8 Experimental platform

To put into practice these theoretical notions, we wanted to use PlaFRIM which is the INRIA cluster to see a real footage of killing processes from the exterior and see the behave of the application with it (failure detection, communicator re-construction), Unfortenetely i had some technical issues to set ULFM on PlaFRIM, so instead of using it, i used a Docker package ULFM image, and since it's Jacobi iteration example, the convergence is fast to set a lot of processes for it and instead of that i put 8 processes.

4.8.1 Docker

Docker is a tool designed to make easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized

settings that machine might have that could differ from the machine used for writing and testing the code.

In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application. And importantly, Docker is open source. This means that anyone can contribute to Docker and extend it to meet their own needs if they need additional features that aren't available out of the box.

4.8.2 Docker's users

Docker [17] is a tool that is designed to benefit both developers and system administrators, making it a part of many DevOps (developers + operations) toolchains. For developers, it means that they can focus on writing code without worrying about the system that it will ultimately be running on. It also allows them to get a head start by using one of thousands of programs already designed to run in a Docker container as a part of their application. For operations staff, Docker gives flexibility and potentially reduces the number of systems needed because of its small footprint and lower overhead.

4.9 Experimental setup

As mentioned for this experiment, we used the Docker package ULFM image that was applied and integrated for the solution of the Laplace equation with the Jacobi iterative method. we used eight processes for the experiment, but before running and seeing the efficiency of ULFM, the source code must be adapted for fault tolerance and for that, there is some steps that were taken.

- **Change the error handler to MPI_ERROR_RETURN**, the error handler is set by default to MPI_ABORT, which will stop working and abort the computation once there is an error, MPI_ERROR_RETURN allows us to check the return value of MPI functions such as MPI_Send and MPI_Recv which will be used after to detect and identify the failure if it occurs. One of the modifications is to make the program recognize MPI_ERR_PROC_FAILED and MPI_ERR_REVOKED.

- **Errors injection** In order to see ULFM working, we have to simulate some errors, these errors can be created in the interior of the program by simulating a process suicide or by a real kill from the exterior using the command KILL ¡PID¿. For the sake of reliability of tests the error apparency must be controlled and for that the injection of error is the simulation of a failure by killing processes in different iterations.

- **Check communications** the most important step for fault tolerance is to be able to detect and identify an error or failure when it occurs, to do so there is two ways, one is to check the Point to Point communications in other meaning to check the Sendrecv communications between processes, or to check the collective functions such as MPI_Barrier, MPI_Reduce, MPI_Allreduce. If there is an error it would be detected directly in one of these two communication types.

To see the impact of data loss and recovery just after, we did four scenarios of suicide, in each one we kill four processes in different interval of iterations, to see the efficiency of the recovery strategies in different situations (in the beginning of the computation, in the middle and in the end). We display the convergence history as a function of iterations to see the features of the numerical recovery strategies, and we display it too as a function of time to see the cost of the ULFM functions and the numerical recovery strategies in term of complexity and the cost, The tests were done by single fault at the time or multiple faults in the same time. We refer to Checkpoint/restart as ER, the linear interpolation as LI, and as a reference to

compare the results, a version of the program without fault is set and refered to as NF (No faulty execution). Because all the strategies were coordinated we did uncoordinated one too and denote it as Relaxed. the acronyms used to denote the names of different curves in the next graphs are shown in this table.

Acronym	**Definition**	**Fault tolerance support**
Reset	Replace lost data by its initial value	Multiple fault
ER	Checkpoint / Restart	Multiple fault
LI	Linear Interpolation	Single fault
Relaxed	Uncoordinated version of ER	Multiple fault
NF	No faulty execution	-
NF - no Agreement	No faulty execution without ULFM functions	-

Table 4.1
Definition of the acronyms used in the captions of forthcoming plots

4.10 Experimental Results

In Figure 4.3, we can see the results for the run with eight processes with four scenarios where the number of faults is identical and they occur at the same iteration for all the runs. It can be observed that Reset strategy had difficulty to converge, we can see picks that return roughly to the initial scaled residuel before starting again to converge. In the other hand, the Checkpoint/Restart and the Linear interpolation strategy ensure convergence with very similar behavior as the Non-fault curve. Furthermore, the convergence behavior of these recovery strategies does not seem to be affected by the loss data.

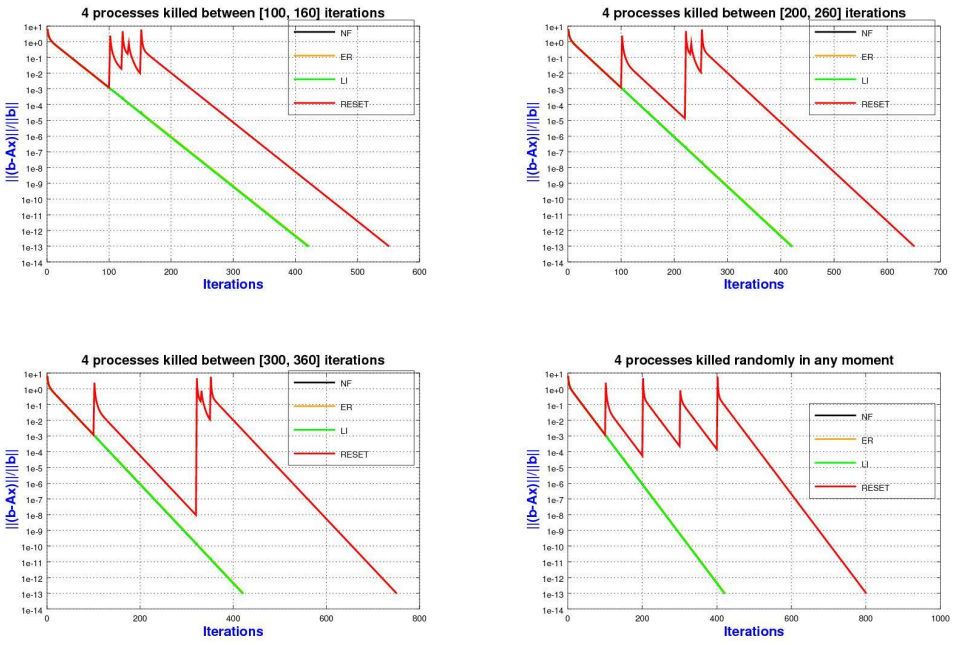


Fig. 4.3. Numerical recovery strategies behavior in term of iterations

In Figure 4.4, we get interested in the amount of time that it needs to compute and convergence, for that we run the same scenarios but instead of observing the convergence in term of number of iterations, we observe it in term of needed time

to converge, As shown, the checkpoint/restart strategy (ER) takes the most of time to converge due to the "store on disk" operations in each iteration and the rollback operations in case of failure, The linear interpolation (LI) takes less time but we can observe plateaus in the convergence when a fault occurs, the process once it is spawned, he takes some times to interpolate its data linearly. while the reset strategy takes time each time it tries to converge until it converge when there is no failure.

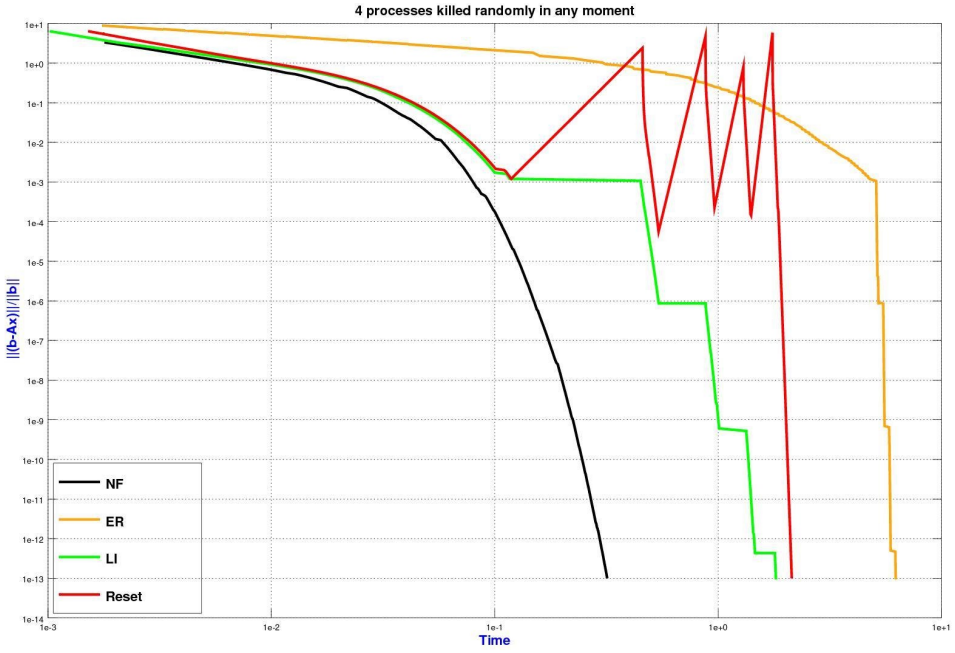


Fig. 4.4. Numerical recovery strategies behavior in term computed time

The costs of MPI_Comm_agree

MPI_Comm_agree represents one of the features of ULFM, it is a method to read a consensus about a given value, this means that it performs a fault tolerant agreement algorithm over a boolean flag, The agreement can be conceptualized as a failure-resilient reduction on a boolean value. The algorithm first performs a reduction of the input values to an elected coordinator in the communicator. The coordinator then makes a decision on the output value and broadcasts that value back to all of

the still alive processes in the communicator.

For these reasons the use of MPI_Comm_agree is very costy and it adds an important overhead to the computation time. As shown in Figure 4.5

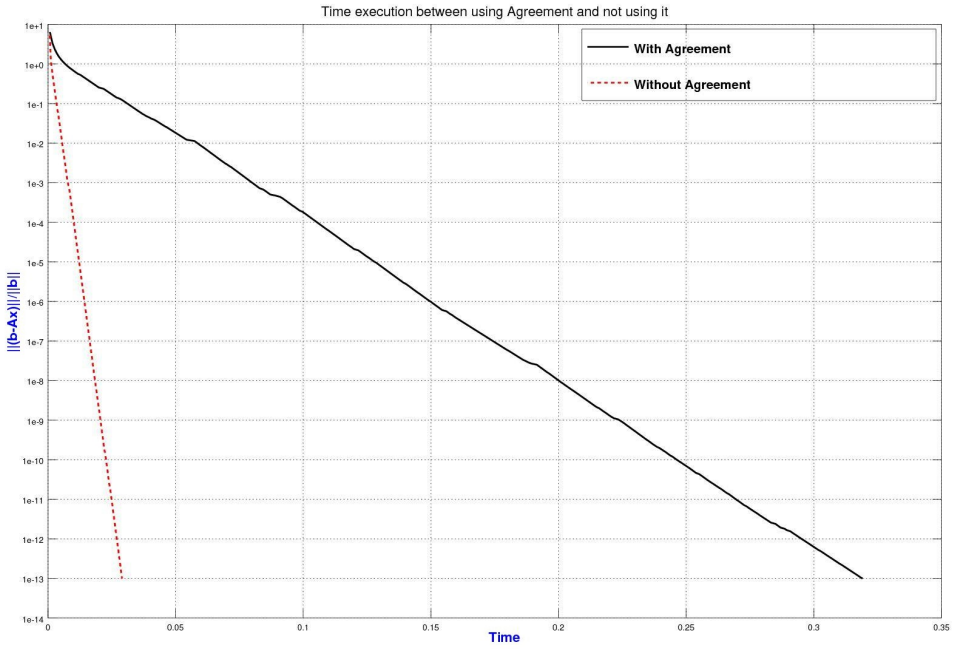


Fig. 4.5. The costs of Agreement on a Numerical recovery strategy

In Figure 4.6, we gathered all the recovery strategies used during this project with and without using the Agreement, and we let just one verification in the collective function MPI_Allreduce for the residual, and since the agreement allows to synchronize and coordinate between the processes in each iteration, so taking them off is the best scenario for the relaxed version (uncoordinated version of ER) where some processes are in the iteration k while others are either in iteration $k+1$ or $k-1$.

As shown we can see the extra time to compute between each recovery strategy with and without agreement, the agreement ensure the propagation of failure noti-

fication and identification but in the same time it introduces a large extra costs in term of computing time.

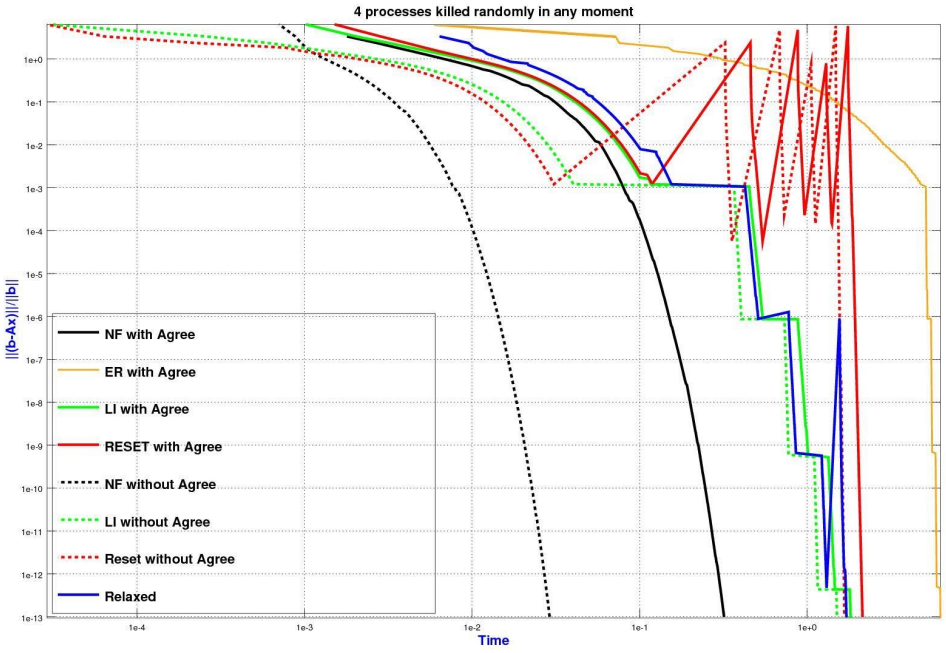


Fig. 4.6. Comparison of all the Numerical recovery strategies with and without Agreement

Finally, it can be observed as general remark that the Linear interpolation gives good performance-time ratio for this example as a Numerical recovery strategy.

CHAPTER 5

Conclusion

Considering the power consumption, clock frequencies, node size, performance, and other system requirements, frequent hardware failures seem to be inevitable in upcoming exascale systems. There are many possible recovery strategies in the context of the MPI application, for linear system solvers with iterative methods, The checkpoint/restart and the Linear interpolation were applied efficiently as numerical recovery strategy better then the reset one which ensure the convergence but in a large delay.

In this project, we have presented one of the techniques used for fault tolerance which is ULFM that proved to be applied to recover HPC applications from processes failures, with the Building block system, we designed and implemented toolkit that can be directly integrated into any linear solver with any iterative method. The use of the Agreement function is costy if it is used to verify each Point to point communication, and the better use of it, is to check only in the collective function in each iteration.

We simulated the failure of processes from the inside of the application, and we did the same thing by killing them from the exterior with $KILL < PID >$ instructions, and it happened what we were expecting, detection of failure , reconstruction of the communicator, and at leat the data recovery.

We have also presented an overview of some numerical recovery strategies that can be applied in any linear system using any iterative method, each strategy has its advantages but its defaults too. The linear interpolation had shown to offer the best Performance/time ratio. we tested the effectiveness of our resilience algorithms by simulating the suicide of processes in a parallel distributed system memory environment from the interior of the program and killing the processes from the exterior which represents a perfect footage of the efficacy

CHAPTER 6

Prospects

The main prospect is to use the same communicator recovery technique on a Krylov linear solver [15], it can be easly implemented due to the similarities in the structure of the codes. and the type of solution which are bouth iterative methods and a block system for a resilient version of the linear solver can be applied the same as the one studied in this project

The challenge to come after is to optimize the use of functions that costs like MPI_Comm_agree, and find the optimal amount of its use, in a simple example based on results reduction, an optimal use is to check the return value of the MPI reduction function in each iteration.

The numerical recovery strategies has proven that they give good performance in term of speed of convergence, especialy the Linear interpolation, with scenarios where we kill one process at time or two in the same time, we'd like to see the behave and stability of these numerical recovery strategies when we push it to the ultim conditions by killing several processes or even the majority of them and see its limits.

LIST OF REFERENCES

LIST OF REFERENCES

[1] I. Hursey, J.; Squyres and Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for open mpi," *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, pp. 2–8, 2007.

[2] K. Plank, J. S.; Li and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distribed Systems*, pp. 9–18, 1998.

[3] G. Ferreira, K.; Stearley and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems.," *Proceedings of the 2011 International Conference for High Performance Computing, Networking*, p. 9, 2011.

[4] P. K. G. Bouteiller, A.; Lemarinier and F. Capello, "Coordinated checkpoint versus message log for fault tolerant mpi.," *Proceedings of the IEEE International Conference on Cluster Computing (IEEE Cluster 2003)*, pp. 242–250, 2003.

[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, pp. 107–119, 2003.

[6] C. L. Chakravorty, S.; Mendes and L. Kal, "Proactive fault tolerance in mpi applications via task migration.," *Proceedings of the 13th International Conference on High Performance Computing*, p. 10, 2006.

[7] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, pp. 193–203, 1984.

[8] G. Stellner, "Cocheck: Checkpointing and process migration for mpi," *Proceedings of the 10th International Parallel Processing Symposium*, pp. 526–530, 1996.

[9] A. M. Agbaria and R. Friedman, "Starfish: fault-tolerant dynamic mpi programs on clusters of workstations.," *Proceedings of the 8th International Symposium on High Performance Distributed Computing*, pp. 167–176, 1999.

[10] N. L. A. Louca, S.; Neophytou and P. Evripidou, "Mpi-ft: Portable fault tolerance scheme for mpi.," *Parallel Processing Letters*, pp. 371–382, 2000.

[11] Bouteiller, A. Cappello, F. Djilali, S Bosilca, G.; "Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes," *Proceedings of the ACM/IEEE 2002 Conference of Supercomputing*, pp. 1–18, 2002.

[12] A. Fagg, G. E.; Bukovsky and Dongarra, "Harness and fault tolerant mpi," *Parallel Computing*, pp. 1479–1495, 2001.

[13] W. Bland, "User level failure mitigation in mpi," *Proceedings of the 2012 Parallel Processing Workshops (Euro-Par 2012)*, pp. 499–504, 201.

[14] G. B. Aurelien Bouteiller, Thomas Herault; Joshua Hursey, "An evaluation of user-level failure mitigation support in mpi," *The International Journal of High Performance Computing Applications*, vol. 13, pp. 2–6, 2016.

[15] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, Mawussi Zounon "Numerical recovery strategies for parallel resilient krylov linear solvers," *Numerical Linear Algebra with Applications*, vol. 22, pp. 3–8, 2016.

[16] "Laplace equation in two dimensions with finite differences." http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/jacobi/C/main.html. Accessed: 2010-09-30.

[17] "What is docker?." https://opensource.com/resources/what-docker. Accessed: 2017-06-21.