HADAB: enabling fault tolerance in parallel applications running in distributed environments

Vania Boccia[1], Luisa Carracciuolo[2], Giuliano Laccetti[1], Marco Lapegna[1], and Valeria Mele[1]

[1] Dept. of Applied Mathematics, University of Naples Federico II,
Naples. 80126, Complesso Universitario Monte S. Angelo, Via Cintia, Italy
{vania.boccia,giuliano.laccetti,marco.lapegna,valeria.mele}@unina.it
[2] Italian National Research Council, Italy
{luisa.carracciuolo}@cnr.it

Abstract. *The development of scientific software, reliable and efficient, in distributed computing environments, requires the identification and the analysis of issues related to the design and the deployment of algorithms for high-performance computing architectures and their integration in distributed contexts. In these environments, indeed, resources efficiency and availability can change unexpectedly because of overloading or failure i.e. of both computing nodes and interconnection network. The scenario described above, requires the design of mechanisms enabling the software to "survive" to such unexpected events by ensuring, at the same time, an effective use of the computing resources. Although many researchers are working on these problems for years, fault tolerance, for some classes of applications is an open matter still today. Here we focus on the design and the deployment of a checkpointing/migration system to enable fault tolerance in parallel applications running in distributed environments. In particular we describe details about HADAB, a new hybrid checkpointing strategy, and its deployment in a meaningful case study: the PETSc Conjugate Gradient algortithm implementation. The related testing phase has been performed on the University of Naples distributed infrastructure (S.Co.P.E. infrastructure).*

Keywords: Fault tolerance, checkpointing, PETSc library, HPC and distributed environments

1 Introduction

In recent decades the focus of the scientific community moved from the traditional parallel computing systems to high performance computing systems for distributed environments, generally consisting of a set of HPC resources (clusters) geographically scattered. They provide increasing computing power and are characterized by a great resources availability (typical of distributed systems) and a high local efficiency (typical of traditional parallel systems). These systems may be used to solve the so-called "challenge problems". However, employing a distributed infrastructure, where HPC resources are geographically scattered

and are not dedicated to a specific application, is not priceless. Indeed such kind of environments are characterized by high dynamicity in resources load and by a high failure rate, thus applications fault tolerance and efficiency are key issues [13].

For many years by now researchers are working to identify standard methods to solve the problem of fault tolerance and efficiency of software designed for distributed environments. However, today this is still an open issue.

In this paper we focus on the design and deployment of a checkpointing/ migration system, in order to enable fault tolerance in parallel applications running in distributed environments.

In Sec. 2 is presented a short overview on fault tolerance and checkpointing mechanisms. In Sec. 3 is presented our checkpointing strategy, HADAB (**H**ybrid, **A**daptive, **D**istributed and **A**lgorithm-**B**ased), underlining criteria to enhance strategy robustness and to narrow the overhead. In Sec. 4 the HADAB deployment in a case study (the parallel version of PETSc library Conjugate Gradient) is shown. In Sec. 5, are described some tests and results, and finally, in Sec. 6, are presented some conclusions and a preview on future works.

2 Fault Tolerance and Checkpointing: State of the Art

Fault tolerance is the ability of a system to react to unexpected events such as sudden overload, temporary or persistent failure of resources [12]. An application is called fault tolerant if it is able to complete its execution in spite of the occurrence of faults. In "complex" computing systems, application survival to failures during execution, depends on the proper behavior of all software layers that the application uses and on the integrity and coherence of the execution environment. There are applications that, because of special properties of the algorithms on which they are based, are "naturally fault tolerant" (i.e. "super scalable" applications)[5, 6]. For all the others, it is necessary to provide "mechanisms" [9] to detect and report the presence of faults (detect/notify), making it possible "to take a snapshot" of the current execution state (checkpointing) and allow the application to resume its execution from the point where the fault occurred (rolling back/migration).

Detect/notify mechanisms are generally in charge to the runtime environment, while checkpointing/migration mechanisms are in charge to the application (or to its runtime environment) and consist of:

- procedures to store data (checkpointing) that, in case of fault, enable the process restart from the point of execution where the fault occurred (checkpoint)
- procedures to resume the execution (rolling back), from where it left off because of the fault, on alternative resources, by recovering and using checkpointing data.

In literature there are different approaches that can be followed to realize checkpointing mechanisms: algorithm-based vs. transparent, disk-based vs. diskless and an exhaustive description of these can be found in [3, 7, 8, 14, 15].

The approaches used to implement such mechanisms are numerous and each one has advantages and disadvantages in relation to checkpointing technique efficiency and robustness [3]. Sometimes, it is not sufficient to use a single strategy to realize a robust checkpointing strategy, but new strategies can arise from the combination of multiple methodologies (strategies for hybrid checkpointing [11]). In general, strategies combination can be used to increase the robustness with respect to the single methodologies, but it is necessary to limit checkpointing overhead.

3 HADAB: the Hybrid, Adaptive, Distributed, Algorithm-Based Checkpointing

The strategy described in this work is:

- hybrid, because combines two strategies: a variant of diskless parity-based [10] and coordinated [11] checkpointing;
- adaptive, because different checkpointing techniques are performed each with different frequency, with the aim to reduce the total overhead;
- distributed, because checkpointing data are periodically saved on a remote storage resource;
- algorithm-based because, although hard to implement, this approach is still the safest method to select and reduce the checkpointing data amount.

We focus on parallel applications based on Message Passing paradigm, in particular based on MPI standard. Currently FT-MPI [2] is the only existing message passing library, implementing MPI standard, that providing the software tools to identify and manage faults, makes applications able to use a diskless approach. Indeed, FT-MPI allows to re-spawn failed processes redefining the MPI context. Unfortunately, its development has been stopped in 2003 and so, on some new architectures, the library seems to be not stable.

Other implementations of MPI standard, as Open MPI [4], promise to introduce the important features of FT-MPI, but developers are waiting for the MPI3 standard to implement these in the production release.

In absence of the software tools needful to use a diskless approach, we chose a disk-based approach and a stop/restart method to implement our checkpointing strategy. In the rest of this section we describe how we built the HADAB strategy.

First, we considered a disk-based variant of the parity-based checkpoint, where the checkpointing phase can be divided into two sub-phases:

1. each application processor [10] saves its checkpointing data locally and sends them to the checkpoint processor
2. the checkpoint processor [10] calculates the bitwise-XOR of the received data and stores it on its local storage device.

In a similar way, the rolling back phase can be divided into two sub-phases:

1. survived processors recover their data from local disks

2. the processor, that took the place of the failed processor, reconstructs its portion of data, by both local data coming from other processors and encrypted data sent by the checkpoint processor.

The strategy described above is already quite robust because it ensures the checkpointing integrity (the last successfully saved checkpointing data are removed only after that new coherent checkpointing data are saved) and allows the application to survive to the fault of one processor at a time (application or checkpoint processor).

Moreover, the use of encryption offers advantages in terms of efficiency, because the amount of data that checkpoint processor has to store is drastically reduced (for a problem of dimension N, using p processors, checkpoint processor data dimension is equal to N/p). So it is clear that this strategy has a lower I/O overhead than a non-coding one, but it can tolerate only one fault at a time.

Thus, to improve checkpointing robustness, we decided to add also "few" phases of coordinated checkpointing, realizing an hybrid strategy.

In general checkpointing frequency cannot be high (to avoid the increasing of the total checkpointing overhead) and, anyway, the time for data saving should not be relevant in relation to the total execution time. So we developed our hybrid strategy using an automatic choice of the checkpointing rate that depends on the extimated execution time.

The advantage introduced by the hybrid strategy is that, if p is the number of processors, the application can tolerate up to $p-1$ simultaneous faults, except the fault of the checkpoint processor. So, paying a not so relevant price in terms of total overhead, we gain in terms of checkpointing strategy robustness.

In case of checkpoint processor unavailability, the hybrid strategy is not able to recover the application, because all checkpointing data are lost. Hence the idea to build a distributed version of the hybrid strategy (HADAB) that periodically saves, in an asynchronous way, checkpointing data on an "external" storage resource.

HADAB is able to guarantee up to p faults at a time:

- up to $p-1$ faults there always is a local copy of checkpointing data available (from coordinated or parity-based strategies) to recover from the fault;
- if all processes fail, a remote saved copy of all checkpointing data is available for application resumption: an external stop/restart system migrates application execution on a new set of computational resources, using the remote copy of all checkpointing data (Fig. 1).

The next section describes the work done to deploy the HADAB checkpointing strategy in a case study.

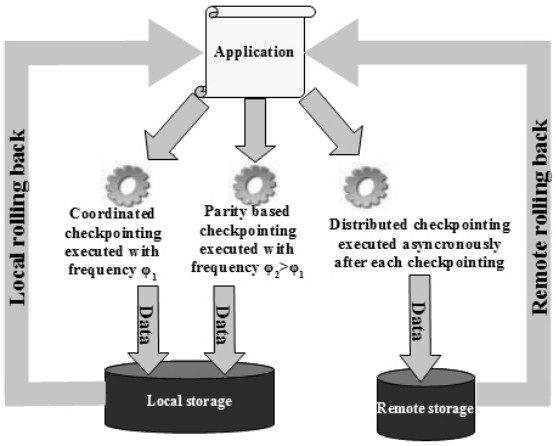

Fig. 1. Migration system schema.

4 HADAB Deployment on the PETSc Conjugate Gradient (CG)

We deployed HADAB into the parallel version of Conjugate Gradient (CG) algorithm implemented in PETSc library [1] with the objective to realize a fault tolerant version of such a procedure (CGFT).

In order to develop a fault tolerant version of CG algorithm (Fig. 2), we followed an algorithm-based approach [3]:

- first we identified the data needed for the checkpointing in the CG algorithm: four vectors and four scalars
- then we added to the PETSc CG routine, the code needed to implement checkpointing phases (see lines 19-32 in Fig. 2) and rolling back phases (see lines 5-14 in Fig. 2) of the HADAB strategy.

Application starts with checkpointing frequency chosen by the user (i.e., parity-based checkpointing is performed at each iteration while the coordinated one is performed every k iterations); during execution the `PetscCheckFreq` routine modifies the checkpointing frequency on the bases of both:

- the average of previous iterations execution time and
- the real time spent to save data (for each checkpointing type).

During the recovery phase, the `CheckCheckpoint` routine selects the most "convenient"[1] checkpointing type to be used in application resumption: if it is parity-based, the `PetscRollbackCodif` routine is called, otherwise the `PetscRollbackCoord` routine is executed.

[1] Metric for convenience is the total cost of the recovery phase that is related to both the data "freshness" and to the overhead in data reading.

Fault tolerant version of Conjugate Gradient with hybrid adaptive distributed checkpointing: code fragment.

```
1  PetscErrorCode KSPSolve_CGFT (KSP ksp)
2  PetscFunctionBegin;
3  /* Initialization phase */
4  ...
5  IF (restart)
6          rt =  CheckCheckpoint(...);
7          IF ( rt == 1 )
8                  ierr =  PetscRollbackCoord(...);
9          ELSE IF ( rt == 0 )
10                 ierr =  PetscRollbackCodif(...);
11         ELSE
12                 printf("It is not possible to recover locally from the fault");
13         ENDIF
14 ENDIF
15 REPEAT
16         ...
17         /* repeat-until loop of the CG algorithm */
18         ...
19         IF (chkenable)
20                 /* ck_coord is the iteration number when
21                    coordinated checkpointing is performed */
22                 /* ck_codif is the iteration number when
23                    coded checkpointing is performed */
24                 IF (i % ck_coord == 0 )
25                         ierr =  PetscCheckpointingCoord(...);
26                         ierr =  PetscStartCopyThreads(...);
27                 ELSE /* case  i % ck_codif == 0  */
28                         ierr =  PetscCheckpointingCodif(...);
29                         ierr =  PetscStartCollectThreads(...);
30                 ENDIF
31                 ierr =  PetscCheckFreq(...);
32         ENDIF
33 UNTIL (i < max_it && r > tol)
34 /* finalization phase */
35 PetscFunctionReturn(0);
```

Fig. 2. PETSc CG fault tolerant version.

During the checkpointing phase `PetscCheckpointingCodif` routine (for parity-based checkpointing) and `PetscCheckpointingCoord` routine (for coordinated checkpointing) are called respectively with a frequency equal to `ck_codif` and `ck_coord` (φ_2 and φ_1 respectively, see Fig. 1).

Finally, `PetscStartCopyThreads` and `PetscStartCollectThreads` routines perform the asynchronous distributed checkpointing data saving on external storage resources. Distributed checkpointing phase does not add any overhead bacause of the use of threads.

When ever the local rolling back phase is impossible (see line 14 in fig. 2), application stops and the migration system migrates the execution on a new set of computational resources. The remote rolling back phase, included in the migration task, introduces an overhead that may significantly change on the basis of several parameters depending on distributed environment characteristics.

In the next section we report some tests performed at the University of Naples Federico II on the HPC computational resources available in the S.Co.P.E. GRID infrastructure. Tests results provided a first validation of HADAB checkpointing strategy and some useful information about migration system overheads.

5 Tests and Remarks

The first tests have been carried out with the aim to verify the behavior of both the checkpointing strategies: coordinated and parity-based. Tests are related to the solution, by means of CGFT, of a linear system where sparse matrix has $N = 3.9 * 10^{17}$ non-zero elements[2]. The linear system is solved after 6892 iterations.

Indeed, depending on when the fault occurs, the overhead introduced by the checkpointing/rolling back phases becomes more or less relevant in comparison to the total execution time of the application.

Table 1. Execution times in seconds: to execute one CG iteration (T_{iter}), to save checkpoinitng data with parity-based strategy ($T_{checkCodif}$) and to save checkpoinitng data with coordinated strategy ($T_{checkCoord}$). p is the number of processors. Checkpointing data are written on a shared area based on Lustre File System. Data dimension is $2.5 * 10^9$. $T_{checkCoord}$ value is indipendent from p.

p	T_{iter}	$T_{checkCodif}$	$T_{checkCoord}$
8	8.76	362.68	417
12	5.65	206.81	417
16	5.46	196.19	417
20	4.67	192.85	417
24	4.16	189.72	417
28	2.31	188.12	417

Table 1 is useful to understand the optimal value for checkpointing frequencies. Focusing on the test performed with 16 processors, the total execution time for the application, in absence of faults, is about 10 hours and 45 minutes. The `PetscCheckFreq` routine chooses to execute a coordinated checkpointing every 196 iterations and a parity-based one every 14 iterations.

In the following tables, when HADAB is enabled, we consider checkpointing frequencies defined on the bases of results reported in table 1.

Looking at the tables 2, 3 and 4 it is possible to evaluate the benefits, if any, due to the use of HADAB checkpointing in the following scenarios:

- Case 1: failure free execution (see table 2)
- Case 2: a single fault during execution (see tables 3 and 4)

[2] Checkpointing data are $M = 2.5 * 10^9$ and their amount is of about $18GB$

Table 2. Application execution with HADAB checkpointing enabled: T_{comp} is the time related to the computational phase, T_{check} is the time due to HADAB checkpointing, $T_{tot} = T_{comp} + T_{check}$ and $Overhead_{check} = T_{check}/T_{tot}$ is the overhead introduced by HADAB in a failure free execution. All times values are expressed in seconds.

N	T_{comp}	T_{check}	T_{tot}	$Overhead_{check}$ (%)
$3.9 * 10^{17}$	37630	18666	56296	33.1%

Table 3. Application execution with HADAB checkpointing disabled in an execution with one fault occurring at It_{fault}. $T_{it-lost}$ is the time spent to re-execute It_{fault} iterations where It_{fault} is: 1000, 2000, 3000, 4000, 5000, 6000. $T_{tot}^{NoC} = T_{comp} + T_{it-lost}$. All times values are expressed in seconds.

It_{fault}	$T_{it-lost}$	T_{tot}^{NoC}
1000	5460	37630+5460 = 43090
2000	10920	37630+10920 = 48550
3000	16380	37630+16380 = 54010
4000	21840	37630+21840 = 59470
5000	27300	37630+27300 = 64930
6000	32760	37630+32760 = 70390

Table 4. Application execution with HADAB checkpointing enabled: $T_{it-lost}$ is the time spent to execute again only the It_{lost} iterations from the last saved checkpointing to It_{fault}. In last column we report overhead, $Overhead_{chkp}$, introduced by HADAB where $Overhead_{chkp} = (T_{tot}^{C} - T_{tot}^{NoC})/T_{tot}^{NoC}$. All times values are expressed in seconds.

It_{fault}	It_{lost}	$T_{it-lost}$	T_{tot}^{C}	$Overhead_{chkp}$
1000	6	32.76	56296+32.76 = 56328.76	31%
2000	12	65.52	56296+65.52 = 56361.52	16%
3000	4	21.84	56296+21.84 = 56317.84	0%
4000	10	54.60	56296+54.60 = 56350.60	-1%
5000	2	10.92	56296+10.92 = 56306.92	-13%
6000	8	43.68	56296+43.68 = 56339.68	-20%

From table 2 we can observe that HADAB adds about the 33% of overhead on the total execution time in absence of faults.

However, if we consider execution with faults, the use of HADAB checkpointing becomes ever more affordable when the iteration number, where the fault occurs, increases (see table 4). Indeed, in the last three rows of the table 4, the $Overhead_{chkp}$ is negative, because the T_{tot}^{NoC} is greater than T_{tot}^{C}. Thus HADAB use is even profitable for the application: i.e. if the fault occurs at iteration 6000, we gain about the 20% on the time T_{tot}^{NoC}.

Looking at all the tables above it is possible to evaluate the benefits due to the use of HADAB checkpointing also in an execution where more than a fault occurs. Indeed, in case where a fault occurs twice, i.e. one at iteration 2000 and the other at iteration 5000, the application recovers twice from the fault,

re-executing only 14 iterations. In this case, thanks to HADAB checkpointing we gain about 26% on the time T_{tot}^{NoC} [3].

Finally table 5 reports the case when all processors p fail at the same time. Here, a migration phase is needed. HADAB checkpointing saved asyncronously data on external storage resource during application execution. Thus when p fault occurr at the same time, migration system selects a new cluster for execution, moves checkpointing data from external storage to the new cluster and restarts application execution from the point when it left out.

Table 5. Application execution with HADAB checkpointing enabled and all processors fail at the same time: $T_{Data-trans}$ is the time to move checkpointing data from storage external resource to the new execution cluster (remote rolling back phase).

N	$Check_{Data-dim}$ (GB)	$T_{Data-trans}$ (secs.)
$3.9 * 10^{17}$	18	134

$T_{Data-trans}$ are related only to the rolling back phase [4], but we have to consider, as overhead, also times due i.e. to new computational resource recruitment and to the queue time on the scheduling system before application restarts. These times are not fixed but depend on the distributed infrastructure characteristics.

6 Conclusion and Future Works

Checkpointing mechanisms deployment in scientific libraries, as PETSc, always is a "good investment". Indeed, if a library is fault tolerant so are all applications that use it. The work here reported, gave us the opportunity to evaluate the benefit in using hybrid strategies for the implementation of checkpointing mechanisms even when disk-based approaches are used.

The implemented system is robust and efficient enaugh; further improvements in efficiency can arise from the use of diskless strategies, currently not feasible, while more improvements in robustness can arise i.e. from the use of virtual resources, fault tolerant networks and fault tolerant message passing libraries.

The remarks made in Sec. 3 and 5 about the overhead introduced by the HADAB checkpointing, are related to an application that, on a big amount of data, performs a "small" amount of computations.

Thus the utility of the checkpointing mechanisms is much more evident in other contexts as i.e.:

- applications handling the same amount of data, but using algorithms with more complexity than that here considered;

[3] $T_{tot}^{C} = T_{tot} + T_{it-lost}(2000) + T_{it-lost}(5000)$, by using data in tables 2 and 4; $T_{tot}^{NoC} = T_{tot} + T_{it-lost}(2000) + T_{it-lost}(5000)$, by using data in tables 2 and 3.

[4] Data have been moved among two clusters of the S.Co.P.E. distributed infrastructure, by using grid protocols.

– computer centers where it is permitted the use of computing resources for a time not adequate to terminate the application execution.

References

1. Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G, Curfman McInnes, L., Smith, B.F., Zhang, H.: PETSc Users Manual. ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.
2. Chen, Z., Fagg, G.E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., Dongarra, J.: Building Fault Survivable MPI Programs with FT MPI Using Diskless Checkpointing. In In Proceedings for ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 213223, 2005.
3. Dongarra, J., Bosilca, B., Delmas, R., Langou, J.: Algorithmic Based Fault Tolerance Applied to High Performance Computing. Journal of Parallel and Distributed Computing, Volume 69, pp 410-416, 2009.
4. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H, Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. Proceedings of 11th European PVM/MPI Users' Group Meeting, 2004.
5. Geist, A., Engelmann, C.: Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors. 2002.
6. Geist, A., Engelmann, C.: Super-Scalable Algorithms for Computing on 100,000 Processors. In Proceedings of ICCS, pages 313-321. Springer, 2005.
7. Hung, E., Student, M.P.: Fault Tolerance and Checkpointing Schemes for Clusters of Workstations. 2008.
8. Kofahi, N.A., Al-Bokhitan, S., Journal, A.A.: On Disk-based and Diskless Checkpointing for Parallel and Distributed Systems: An Empirical Analysis. Information Technology Journal, 4:367376, 2005.
9. Lee, K., Sha, L.: Process resurrection: A fast recovery mechanism for real-time embedded systems. Real-Time and Embedded Technology and Applications Symposium, IEEE, 0:292301, 2005.
10. Plank, J.S., Li, K., Puening, M.A.: Diskless Checkpointing. Technical Report CS-97-380, University of Tennessee, December, 1997.
11. Silva, L.M., Silva, G.J.: The Performance of Coordinated and Independent Checkpointing. Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, pages 280-284, Washington, DC, USA, 1999. IEEE Computer Society.
12. Simon, H.D., Heroux, M.A., Raghavan, P.: Faul Tolerance in Large Scale Scientific Computing, Chapter 11, pages 203-220. Siam Press, 2006.
13. Song, H., Leangsuksun, C., Nassar, R.: Availability Modeling and Analysis on High Performance Cluster Computing Systems. First International Conference on Availability, Reliability and Security, 0:305313, 2006.
14. Vadhiyar, S.S., Dongarra, J.: SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. In In: Parallel Processing Letters. Volume, pages 291312, 2002.
15. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In Parallel and Distributed Processing Symposium, 2007.