Outcomes of the Fault Tolerance Configuration

Leonardo Fialho[1] *, Angelo Duarte[2], Dolores Rexachs[1], and Emilio Luque[1]

[1] Computer Architecture and Operating Systems Department
University Autonoma of Barcelona. Bellaterra, Barcelona 08193, Spain
`{lfialho, guna}@caos.uab.es, {dolores.rexachs, emilio.luque}@uab.es`
[2] Departamento de Tecnologia
Universiade Estadual de Feira de Santana. Feira de Santana, Bahia, Brasil
`angeloduarte@ecomp.uefs.br`

Abstract. This paper presents the influence of the fault tolerance configuration on different applications using performance metrics. Two configuration parameters are analysed: the heartbeat/watchdog interval and the checkpoint interval. In addition, even message logging is mandatory, an analysis of its overhead on different applications is presented. The impact of message logging on applications has been analysed according to the nature of the communication primitives used on the application. This analysis shows why for different applications the message logging introduces different overhead.

Key words: RADIC, Fault Tolerance, HPC, Configuration.

1 Introduction

Parallel computers are growing in complexity and number of components which increases fault probability. In these machines, Message Passing Interface becomes a *de facto* library used to implement message passing parallel applications. RADIC [1] (*Redundant Array of Distributed Independent Fault Tolerance Controllers*) is a rollback/recovery based fault tolerance architecture which has been proposed to be integrated on message passing libraries. In order to achieve fault tolerance, actual implementations of the RADIC architecture performs message logging and uncoordinated checkpointing. In this work an implementation named RADIC/OMPI will be used as a testbed fault tolerant MPI library.

During the development of the RADIC architecture four main characteristics have been aimed: transparency, scalability, distributed operation and flexibility. This paper explores the flexibility characteristic focusing on the influence of configuration parameters on the observed overhead and degradation. What is the overhead introduced by the fault tolerance architecture? How to configure the fault tolerance parameters in order to achieve lowers execution time?

Due to its distributed characteristic, estimate the overhead introduced by fault tolerance operations is not trivial. While RADIC does not stops the entire

* This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974

application for checkpointing nor recovery and the logging procedure performance depends on the application's communication behaviour, determine the impact of these operations requires an extensive knowledge of the application performance over the target computer.

This paper presents the influence of the fault tolerance configuration on the application's performance. The content is organised as follows. Section 2 presents some related work. In section 3 the RADIC architecture is described, which includes it configurations parameters as well as the theoretical impact of them on applications. Section 4 concerns to experimental evaluation. Finally, conclusions are stated in section 5.

2 Related Work

Much effort has gone into studies about the impact of fault tolerance into applications and how to define fault tolerance parameters. Furthermore, the literature lacks an extensive analysis of fault tolerance tasks impact individually.

There is no consensus about the use of coordinated and uncoordinated checkpoints. While using uncoordinated checkpointing [2] presents an analysis of the impact of message logging on application's performance. While using coordinated checkpointing, proposals normally permits to adjust solely the checkpoint interval parameter. In this sense a model for predicting the optimum checkpoint interval is proposed on [3] and [4]. Therefore, for large clusters another approach should be used, as presented in [5] and [6] which analyse the use of non-blocking checkpointing.

As shown in [7], the performance of applications depends, in part, on the overhead introduced on communication. Although there are many studies about the impact of fault tolerance on applications, normally, these studies are limited to present the overhead introduced by an specific fault tolerance proposal. This papers aims to analyse the impact of each protection task performed by RADIC, which are common tasks performed by any fault tolerance architecture.

3 RADIC Architecture

RADIC architecture [1] must assures a secure recovery. Actual implementations rely on uncoordinated checkpoints combined with receiver-based pessimistic message logs. *Critical data* like checkpoints and message logs of one application process are stored on other node different from the one in which the application is running. This selection assures application completion if a minimum of three nodes is left operational after n non-simultaneous faults. Even more, simultaneous faults are supported if the faulty resource is not involved in recovery (*i.e.* a fault in a node which runs an application simultaneously with a fault in the node which stores its critical data is not supported).

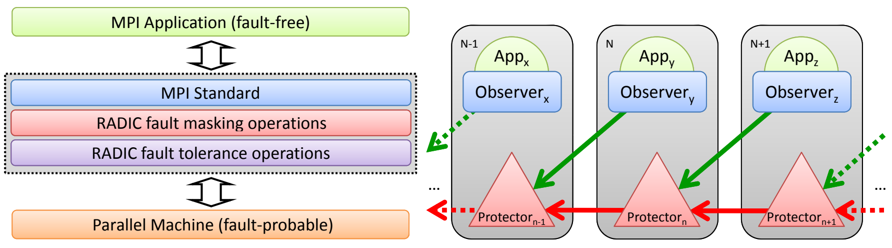
There are operations defined by rollback/recovery protocol as the pessimistic message logging. On the other hand, RADIC define configurations parameters which should be defined by the user before application launching. These parameters are the checkpoint interval, heartbeat/whatchdog interval and the use of spare nodes. Furthermore, the mapping of applications processes and RADIC fault tolerance entities impacts on the overall running time.

In short, RADIC defines two entities:

- **Observer:** this entity is responsible for monitoring the application's communications and masks possible errors generated by communication failures. Therefore, the observer performs message logs in a pessimistic way as well as periodically taking an application process checkpoint. Checkpoints and message logs are sent to protectors. There is an observer attached to each application process.
- **Protector:** according to RADIC specifications, each node runs only one protector, which can protect more than one application process. In order to protect the application's critical data, protectors store that on a non-volatile media. Periodically, protector sends a heartbeat message to other protector, which resets its watchdog timer while the reception of the message. This heartbeat/watchdog mechanism permits the fault detection. In case of failure, the protector recovers the failed application process with its attached observer.

These entities would not run on the same node because, in RADIC, fault tolerance is obtained by joining fault-probable resources. Thus, RADIC does not need any central or stable resource: nodes work together aiming for resilience. As shown in figure 1a, in order to be transparent for the application, observers manage MPI communication masking errors. In addition to masking errors, observers should request a recovery for the faulty application's protector before retrying communication.

Errors generated due MPI communication attempts are one of RADIC fault detection mechanisms. Moreover, protectors implement a heartbeat/watchdog mechanism between nodes which permits the configuration of the fault detection latency. Figure 1b depicts communication performed by RADIC in order to



(a) (b)

Fig. 1. a) RADIC layers which provide transparency for applications. b) Relationship between nodes running an application with RADIC fault tolerance architecture. Diagonal arrows represent critical data flow while horizontal ones represent heartbeats/watchdog communication.

achieve fault tolerance. Horizontal arrows represent heartbeats. Diagonal arrows represent critical data flow (*i.e.* message logging and checkpoint file transmission). Additionally, figure 1b shows that observers go attached to application processes while protectors are standalone processes.

The major advantage of RADIC is its intrinsic distributed characteristic. No collective operation is needed. Tasks performed by the fault tolerance mechanism involve solely two nodes and do not depend on the number of nodes neither any central element nor unique resource. In order to avoid performance degradation in the presence of faults, RADIC allows the use of spare nodes [8]. By default the architecture recovers the failed process on the same node where its protector runs. Summarising, the RADIC architecture is transparent, scalable, distributed and flexible.

According to flexibility, RADIC permits to modify its configurations in order to accomplish user's requirements. RADIC permits to the user to modify de checkpoint interval, observer/protector mapping, number of copies of each process, number and logical location of spare nodes, and the heartbeat/watchdog interval. On actual implementations the message logging is mandatory in order to achieve a consistent state for the application while using uncoordinated checkpointing.

The RADIC architecture has been tested using a prototype called RADICMPI [1] and nowadays there is a version implemented over Open MPI [9] called RADIC/OMPI. A short description of each configuration parameter is given as follow. Although the logging is mandatory in the rollback/recovery protocol used, the impact of this operation is described also.

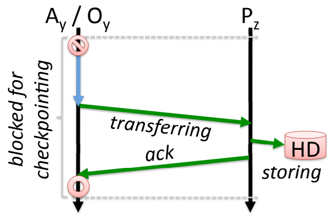
3.1 Checkpoint Interval

The checkpoint interval parameter defines the frequency in which checkpoints are performed. During a checkpointing operation, the process is unavailable for computation and communication. Pending messages do not progress and new messages are not allowed. The time needed to make a checkpoint varies according to the process size and transferring/storing performance.

Figure 2 depicts tasks performed during checkpointing. At first, the observer (O_y) closes all communications channels used by the application (A_y) in order to save its state (vertical blue arrow) to a checkpoint file. Thus, the observer starts the transferring of the checkpoint file to its protector and after conclusion reopens all application's communication channels.

During start up, a fixed value is defined by the user on configuration files. This value is used by RADIC along all the process execution. As discussed in [4], the optimum checkpoint interval (t_0) should consider the fault probability (α), and the time needed for checkpointing (k_0), as depicted on equation 1.

There is no certainty about the fault frequency due to unpredictable characteristic of the event. Thus, the optimum checkpoint interval, which is the result of an analytical equation should be used considering an error margin. The error depends on the accuracy of measures used to obtain the optimum checkpoint interval as well as the expected fault probability.



$$t_0 = \sqrt{\frac{2k_0}{\alpha}} \qquad (1)$$

Fig. 2. Tasks performed during checkpointing.

Equation 1 minimises the execution time reducing the number of checkpoints. Furthermore, it considers that the fault probability is a well-know and fixed value, but normally it is not true. On the other hand, to reduce the number of checkpoints increases the recovery time. In addition, equation 1 has been designed for coordinated checkpointing. There is no guarantee that in a uncoordinated checkpointing environment this equation can be used.

3.2 Observer/Protector Mapping

The observer/protector mapping refers to the assignation in which node critical data of each application should be stored. The default configuration in RADIC plays a simple algorithm which defines the next logical node as a protector node, while current node stores data from previous logical node. The last and first node are considered neighbours nodes, which assures the treatment for all of them.

The mapping can be changed to fit a logical task mapping or the network topology, for example. The impact of the RADIC original mapping algorithm over an application is unpredictable without knowing the performance of the application over the target parallel computer. To achieve a better performance the application's communication pattern, the network topology and node's available resources and load should be considered.

3.3 Heartbeat/Watchdog Interval

The heartbeat/watchdog interval controls the maximum fault detection latency. Actually, exchange small messages between a protectors pair does not introduces a perceptively overhead. Thus, too small heartbeat/watchdog interval could flood the network. Thus, a possible outcome of choosing a too small heartbeat/watchdog interval is the increase of MPI message latency.

Heartbeat/watchdog interval and checkpoint interval represent the major influence on the recovery time. Furthermore, due to the logging operation there are application's specific characteristics with impact on the recovery time.

3.4 Message Logging

Different than other overhead sources introduced by the fault tolerance architecture, the message logging overhead depends, majority, on the application's

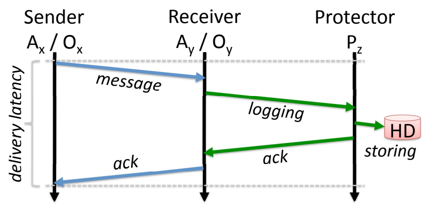


Fig. 3. Tasks performed during message logging.

communication frequency. Figure 3 depicts tasks performed during message logging. For each message received, the observer replies this message to its protector which stores that locally. Once logging if performed for each message received, its impact is depends on the application's performance over the target machine. The number of logging operations performed depends on the application.

Message logging permits the rollback of an individual process of the entire parallel application. In addition, the recovery time can be reduced drastically because during recovering the communication latency tends to zero.

4 Experimental Evaluation

Different applications has been used to depict better specific configuration parameters. Experiments run on a 32 node Linux cluster equipped with two Dual-Core Intel Xeon processors running at 2.66GHz. Each node has 12 GBytes of main memory and a 160 GByte SATA disk. Nodes are interconnected via two Gigabit Ethernet interfaces.

Experimental evaluation concerns two configuration parameters and an analysis of the message logging overhead source: *a)* the checkpoint interval on BT and LU class D applications from the NAS benchmarks, *b)* heartbeat/watchdog interval while running a matrix multiplication program, and *c)* the impact of message logging over blocking and non-blocking communication used by SP and FT class A applications from the NAS benchmarks respectively. During these experiments the use of spare nodes has been considered.

4.1 Checkpoint Interval

In order to analyse the influence of the checkpoint interval on applications, two class D applications from the NAS benchmarks has been used: BT and LU. These applications has been select due to their execution time. BT class D runs in 1774.8 seconds while LU class D runs in 1237.8 seconds. These measures refer to applications running without fault tolerance.

To create a faulty scenario a simple task has been introduced to randomly kills one of the running processes of the entire application every 600 seconds. An instrumentation has been introduced on the MPI library in order to obtain

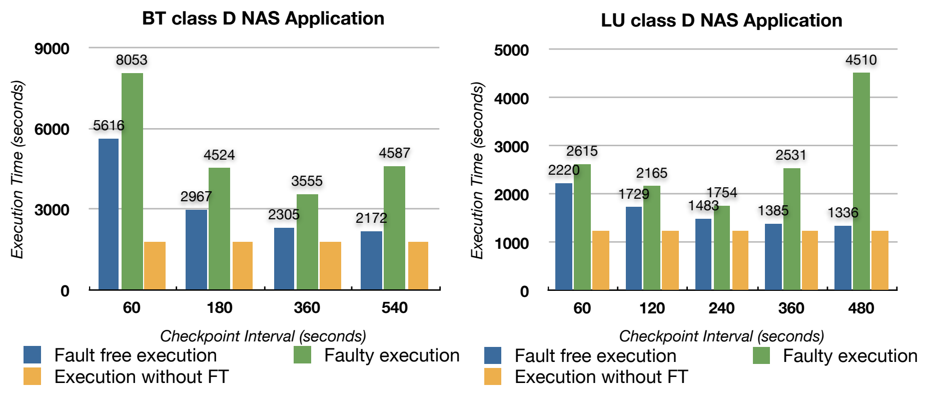
Table 1. Values used to calculate the optimum checkpoint interval and run BT and LU class D applications.

	BT/D	LU/D
Fault probability	every 600 seconds	
Process size in memory	1.4 GB	722 MB
Time needed to checkpoint	132.47	49.09
Optimum interval	**398.74**	**242.72**
Heartbeat/watchdog interval	1 second	
Nodes used	25	32

the time needed for checkpointing BT and LU class D applications. BT class D performs a checkpoint in 132.47 seconds while LU class D needs 49.09 seconds. The difference refers to the total amount of memory used by applications. Table 1 presents other fault tolerance parameters used on these experiments.

Applying these values to equation 1 the optimum checkpoint interval obtained for BT and LU class D applications were 398.7 and 242.7 seconds respectively. To depict the influence of checkpoint interval on these applications a set of experiments were run using parameters shown in table 1 and with different checkpoint intervals. For BT applications 60, 180, 360 and 540 seconds has been used, and for LU application 60, 120, 240, 360 and 480 seconds. For better comparison three scenarios will be presented: fault free, faulty, and for the sake of comparison, without fault tolerance.

Figure 4a shows the execution of the BT class D application. As shown, too small checkpoint interval increases the execution time because the application stops many times to perform a checkpoint. On the other hand, the recovery time is shorter. The opposite behaviour occurs when a bigger interval is used.



(a) (b)

Fig. 4. BT and LU class D application running with different checkpoint intervals. Faults occurs every 600 seconds. Values are expressed in seconds.

As depicts figure 4b the same behaviour seen in BT could be observed in LU class D application. Too small checkpoint interval reduces the recovery time and increases the execution time because it produces more checkpoints and with a larger execution time more faults occurs. On the other hand, too big checkpoint interval reduces the execution time increasing the time spent in recovering.

Figures 4a and 4b depicts that the smaller execution time for these applications running on a faulty environment occurs when the checkpoint interval used is closer to the optimum checkpoint interval. Thus, these experiments show that the optimum checkpoint interval calculated using equation 1 can be applied on fault tolerance architectures based on uncoordinated checkpointing.

4.2 Heartbeat/Watchdog Interval

To analyse the influence of the heartbeat/watchdog interval a matrix multiplication application has been used. The selected matrix multiplication algorithm solely communicate during the start up and finalisation phases avoiding fault detection due to communication tries. Thus, in this case, the fault detection depends solely on the heartbeat/watchdog mechanism.

Figure 5 presents the execution time of the selected application multiplying two 10.000 X 10.000 matrices running on 16 nodes. Different heartbeat/watchdog intervals has been chosen: from 1 second to 90 seconds. Checkpoints are made every 600 seconds, and just after the second checkpoint a fault has been inserted in one of the 16 nodes.

As shown is figure 5 a shorter heartbeat/watchdog interval reduces the fault detection latency. Additionally, there is no perceivable overhead caused by the heartbeat/watchdog communication. It occurs due to the lack of communication existent on the computation phase of this application.

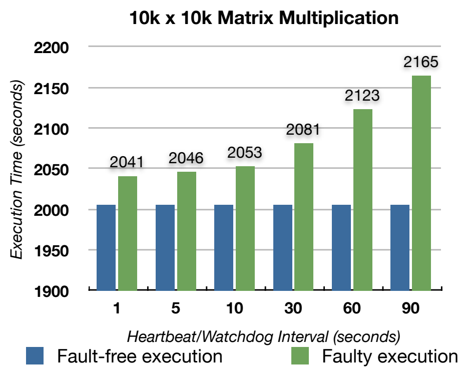


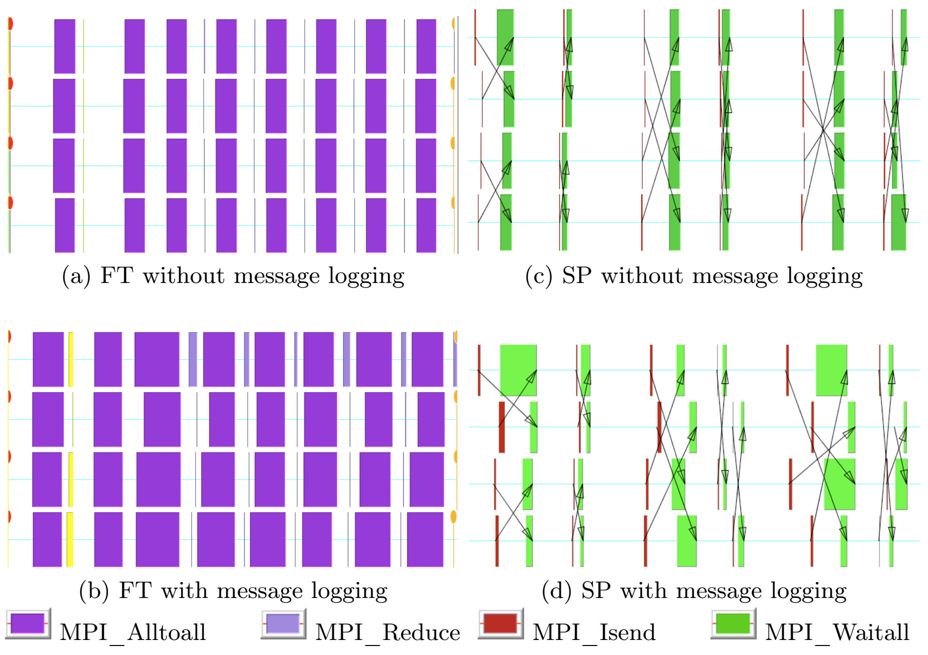
Fig. 5. 10k X 10k matrix multiplication execution using different heartbeat/watchdog intervals. Checkpoints are made every 600 seconds and a fault has been inserted just after the second checkpoint. Values are expressed in seconds.

4.3 Message Logging

To depict the influence of the message logging on applications two benchmarks from the NAS has been selected: FT and SP. FT implements collective operations like `MPI_Alltoall` and `MPI_Reduce` while SP implements asynchronous communication using `MPI_Isend` and `MPI_Waitall` primitives.

As result of the additional communication introduced by the logging procedure, the total execution time increases. The execution time of FT is 7.758 seconds without logging and 14.529 seconds with logging. This behaviour is observed mainly while application uses blocking or collective communication, which occurs with FT kernel. Message logging has introduced an overhead of 87.27% on FT class A.

Figure 6c depicts an slice of the communication trace of SP without message logging and figure 6d presents the same slice while using logging. The slice without logging refers to 0.098 seconds while the slice with logging refers to 0.120 seconds of the application execution. In this case the message logging has introduced an overhead of 55.68% on SP class A. As figure depicts, the waiting time while using message logging just some times is bigger than while logging is not in use. This unpredictable behaviour of the waiting time depends on the individual process computation and on the network load during communication.



(a) FT without message logging (c) SP without message logging

(b) FT with message logging (d) SP with message logging

Fig. 6. Communication trace of FT and SP class A application. (a) and (b) without message logging. (c) and (d) with message logging.

These experiments has demonstrated that the overhead introduced by the message logging operation depends on the application's communication pattern. Applications which uses collective or blocking communication primitives are more sensitive to delays introduced by the logging.Depending on the application's communication/computarion ratio the message logging can be completely overlapped with computation.

5 Conclusions

This paper has presented the impact of the configuration of the fault tolerance on different applications. For the heartbeat/watchdog interval an matrix multiplication application has been used, while to present the influence of checkpoint interval and message logging applications from the NAS benchmarks been used.

The experimental evaluation permits to conclude that the fault detection latency depends on the heartbeat/watchdog interval and small values are better than large ones for some application while to others there is no influence because the message exchange frequency is smaller than the heartbeat/checkpoint interval. About the checkpoint interval, experiments has demonstrated that the same equations used to uncoordinated checkpointing can be applied to a coordinated checkpointing scenario. Additionally, the impact of message logging on applications which uses collective and blocking communication primitives is higher than on applications which uses non-blocking ones.

References

1. Duarte, A.: RADIC: a powerful fault-tolerant architecture. recolecta.net (Jan 2007)
2. Bouteiller, A., Collin, B., Herault, T., Lemarinier, P.: Impact of event logger on causal message logging protocols for fault tolerant MPI. 19th IEEE International Parallel and Distributed Processing Symposium (Jan 2005) 97—97
3. Daly, J.: A model for predicting the optimum checkpoint interval for restart dumps. Lecture Notes in Computer Science (Jan 2003) 724
4. Gropp, W., Lusk, E.: Fault tolerance in message passing interface programs. International Journal of High Performance Computing Applications (Jan 2004) 363—372
5. Oliner, A., Sahoo, R., Moreira, J., Gupta, M.: Performance implications of periodic checkpointing on large-scale cluster systems. 19th IEEE International Parallel and Distributed Processing Symposium. (Jan 2005) 299—306
6. Coti, C., Herault, T., Lemarinier, P., Pilard, L.: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. Proceedings of the ACM/IEEE 2006 Conference on Supercomputing. (Jan 2006) 18—18
7. Doerfler, D., Brightwell, R.: Measuring MPI send and receive overhead and application availability in high performance network interfaces. Lecture Notes in Computer Science (Jan 2006) 331—338
8. Santos, G.: RADIC II: a fault tolerant architecture with flexible dynamic redundancy. recercat.cat (Jan 2007)
9. Fialho, L., Santos, G., Duarte, A., Rexachs, D., Luque, E.: Challenges and Issues of the Integration of RADIC into Open MPI. Proceedings of The European PVM/MPI Users' Group Conference (May 2009) 73—83