RADIC scalability analysis: Functional Model

Sandra A. Mendez[1], C. Marcelo Pérez Ibarra[1], Dolores I. Rexachs del Rosario[2], Leonardo Fialho[2], Emilio Luque Fadón[2], Nilda M. Pérez Otero[1], Cecilia M. Lasserre[1], Héctor P. Liberatori[1]

[1]Universidad Nacional de Jujuy, Facultad de Ingeniería. Gorriti 237. San Salvador de Jujuy. Argentina
[2] University Autonoma of Barcelona. Bellaterra, Barcelona 08193, España.
{fdolores.rexachs, emilio.luqueg}@uab.es, lfialho@caos.uab.es,
{smendez, cmperezi, nilperez, classerre,
hliberatori}@fi.unju.edu.ar

Abstract. In parallel systems, a number of measures of performance are not accurate or representative of their functioning. These measures allow to quantify the benefit of parallelism. Very often, programs are designed and tested for smaller problems on fewer processing elements. However, the real problems these programs are intended to solve are much larger, and the machines contain a great number of processing elements. Hence, it is necessary to create a model that allows to extrapolate the application execution over a few processing elements to larger machine configurations. These measures are more complex if we consider faults. When we take measures we must understand the interaction among system architecture, application architecture and fault tolerant system. In this paper we present a model which analyzes the combination parallel computer, parallel application and RADIC fault tolerance architecture.

Keywords: Scalability, Performability, High Performance Computing, Cluster, Overhead, RADIC.

1 Introduction

The demand for computational power has been leading the improvement of the High Performance Computing (HPC) area, generally represented by the use of distributed systems like clusters of computers running parallel applications. In this area, fault tolerance plays an important role in order to provide high availability isolating the application from the effects of the node faults.

Performance and availability form an undissociable binomial for some kind of applications. Therefore, the fault tolerant solutions must take into consideration these two constraints when it has been designed.

RADIC is a fault tolerant architecture for message passing systems. This architecture has the following features: transparency, decentralization, flexibility and scalability [1] and [2].

There are several measures of performance. Perhaps the simplest of these is the wall-clock time taken to solve a given problem on a given parallel platform. However, as we shall see, a single figure of merit of this nature cannot be extrapolated to other problem instances or larger machine configurations. Other intuitive measures quantify the benefit of parallelism, i.e., how much faster the parallel program runs with respect to the serial program. However, this characterization suffers from other drawbacks, in addition to those mentioned above [3]. For instance, what is the impact of using a poorer serial algorithm that is more amenable to parallel processing? What is the impact of using fault tolerance? What is the impact of increasing the problem size or the number of processing elements?. For these reasons, more complex measures for extrapolating performance to larger machine configurations or problems were often necessary.

Very often, programs are designed and tested for smaller problems on fewer processing elements. However, the real problems these programs are intended to solve are much larger, and the machines contain a great number of processing elements. Whereas code development is simplified by using scaled-down versions of the machine and the problem, their performance and correctness (of programs) is much more difficult to establish based on scaled-down systems. In this paper, we define a functional model that allows to increase problem size and the computational system capacity to analyze if RADIC can adapt to these new systems and keeps on with the performances.

The rest of this paper is organized as follows. In Section 2 we describe some metrics of parallel systems. In section 3, we briefly describe RADIC. In Section 4 we describe some metrics of fault tolerance parallel system and our functional model of RADIC. Finally, section 5 presents the experiments and conclusions about the scalability of RADIC.

2 Metrics of Parallel Systems

A sequential algorithm is usually evaluated in terms of its execution time, expressed as a function of the size of its input. The execution time of a parallel algorithm depends not only on input size but also on the number of processing elements used, and their relative computation and interprocess communication speeds. Hence, a parallel algorithm cannot be evaluated in isolation from a parallel architecture without some loss in accuracy. A parallel system is the combination of an algorithm and the parallel architecture on which it is implemented. In addition, a parallel system of high availability is the combination of the parallel application, parallel computer and fault tolerance.

It is important to study the performance of parallel programs with a view to determining the best algorithm, evaluating hardware platforms, and examining the benefits from parallelism. Several metrics have been used based on the desired outcome of performance analysis.

These metrics could be applied to determinate the performability (performance and availability) of a parallel system modifying both problem size and number of processing elements.

2.1 Execution Time

The serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The parallel runtime is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution. The serial runtime is denoted by T_S and the parallel runtime by T_P.

2.2 Total Parallel Overhead

The overheads incurred by a parallel program are encapsulated into a single expression referred to as the overhead function. We define overhead function or total overhead of a parallel system as the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol T_o.

The total time spent in solving a problem summed over all processing elements is pT_P. T_S units of this time are spent performing useful work, and the remainder is overhead. Therefore, the overhead function (T_o) is given by

$$T_o = pT_p - T_S \tag{1}$$

2.3 Speedup

When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with p identical processing elements. We denote speedup by the symbol S.

$$S = \frac{T_S}{T_P} \tag{2}$$

2.4 Efficiency

Efficiency is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to p and efficiency is equal to one. In practice, speedup is less than p and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol E. Mathematically, it is given by

$$E = \frac{S}{p} \Longrightarrow E = \frac{T_S}{pT_p} \tag{3}$$

2.5 Problem Size

The size or the magnitude of the problem is the number of basic computation steps in the best sequential algorithm to solve the problem on a single processing element. Because it is defined in terms of sequential time complexity, the problem size is a function of the size of the input. The symbol we use to denote problem size is *W*.

The variation both *W* and number of processing elements (*p*) produces changes in the compute-communication relationship.

2.6 Average Computational Capacity (*ACC*)

The *ACC*, defined by Koren and Krishna [4], is a function of how good the parallel application uses the available processors in a system. Therefore, the ACC depends on the probability $P_i(t)$ that exactly *i* processors, from the *N* total processor of the parallel computer, are operational at time *t*, and of the function *C(i)* that defines how good the application adapts to the *i* available processors, as described in equation 3.

$$ACC = \sum_{i=1}^{N} C(i) \cdot P_i(t) \quad (4)$$

2.7 Scalability of a Parallel Systems

We know that the total overhead function T_o is a function of both problem size and the number of processing elements (equation 1). In many cases, T_o grows sublinearly with respect to T_s. In such cases, we can see that efficiency increases if the problem size is increased keeping the number of processing elements constant. For such algorithms, it should be possible to keep the efficiency fixed by increasing both the size of the problem and the number of processing elements simultaneously. This ability to maintain efficiency at a fixed value by simultaneously increasing the number of processing elements and the size of the problem is exhibited by many parallel systems.

We call such systems scalable parallel systems. In other words, the scalability of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processing elements. It reflects a parallel system's ability to utilize increasing processing resources effectively [5] and [6].

Performance analysis in fault tolerant parallel systems is as important as in parallel systems. In a particular way we'll analyze the performance of RADIC fault tolerance architecture. Hence, in the next section we explain RADIC architecture, and then we start studying the metrics of parallel fault tolerant systems with RADIC in order to evaluate their impact in the performance.

3 RADIC

RADIC (Redundant Array of Distributed Independent fault tolerance Controllers) was presented in [1] and [2] as a flexible, decentralized, transparent and scalable architecture that provides fault tolerance to message passing based parallel systems

using rollback-recovery techniques. RADIC acts as a layer that isolates an application from the possible nodes failures. RADIC increases system availability. RADIC, in its basic protection level, does not demand any passive resources to perform its activities. Thus, after a failure the controller recovers the faulty process in some existent node of the cluster. Such level increases the availability of the system, however it may degradate the overall system performance after the recovery.

RADIC [7], can incorporate a flexible dynamic redundancy feature, allowing to mitigate or to avoid some recovery side-effects. This functionality allows restoring a changed process per node distribution (from now, also called system configuration) and it can avoid the configuration changes (while there are spare nodes the computational capacity doesn't change). RADIC allows dynamically inserting new spare nodes during the application execution in order to replace the requested ones (hot swap). The use of spare nodes avoids performance degradation, except during the short period of the recovery process, which leads to an increment of system performability.

Figures 1 (a, b) depict the basic elements that RADIC architecture uses for fault tolerance.

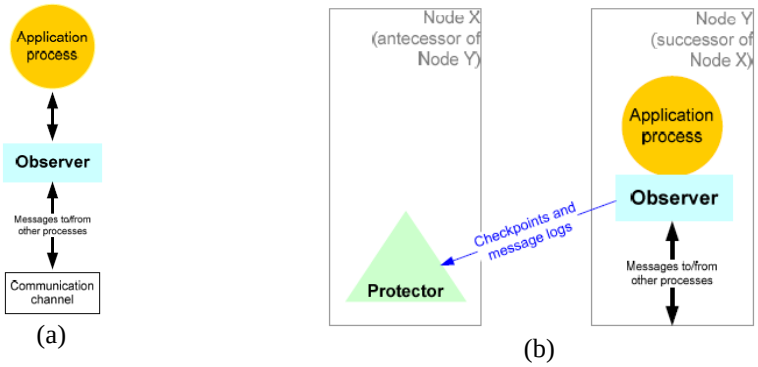

Fig. 1. (a) Application process associated observer. (b) RADIC's protector and observer processes interaction.

Figure 2 shows how in normal operation protectors monitor computer's nodes and observers take checkpoints and message logs of the distributed application processes (2a). Together, protectors and observers function as a distributed controller for fault tolerance. When protectors and observers detect a failure (2b), both work to reestablish the consistent state of the distributed parallel application and to reestablish the structure of the RADIC controller (2 c and d)

Considering that RADIC adds overhead, we should now ask: Which is the impact of using RADIC? How many overhead is introduced by RADIC? Which is the impact of increasing the problem size or number of processing elements of an application that is running over RADIC cluster?

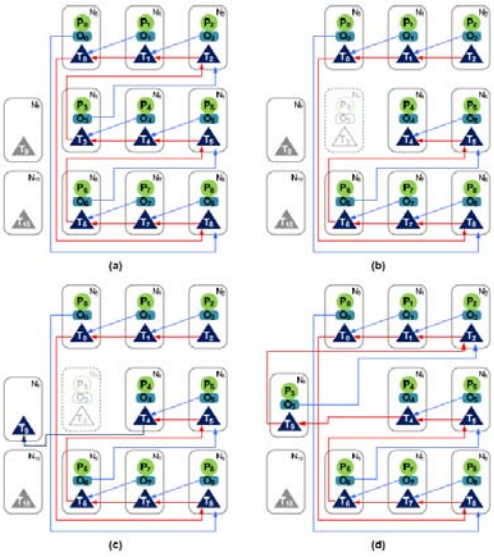

Fig. 2. Phases in the fault recuperation process of node 3, using 2 spare nodes.

4 Scalability Metrics of Fault Tolerant Parallel Systems

In order to carry out this study we need to redefine some terms. That is because many alternatives exist to implement fault tolerance: information redundancy, time redundancy, software redundancy and hardware redundancy. Each strategy has different impact on cost and performance.

4.1 Execution Time with Fault Tolerance

Execution time (T_{ETF}) of an application running in a parallel computer which includes fault tolerance is the time between the application launching and the finalization of the application by the last processing element. This time includes the overhead due to fault tolerance (T_{TFO}) plus the parallel systems overhead itself (processing elements communication and coordination).

$$T_{ETF} = T_P + T_{TFO}$$

4.2 Availability *A(t)*

The following function defines the availability of a parallel system with fault tolerance:

$$Av = \frac{T_{p_{\text{without faults}}}}{T_{p_{\text{with faults}}}} \quad (5) \quad \text{where} \quad T_{p_{with\,faults}} = T_{p_{without\,faults}} + T_{p_{faulttreatment}} \quad (6)$$

4.3 Total Parallel Overhead with Fault Tolerance

Parallel systems overhead (T_{PO}) becomes more complex when fault tolerance is added.

$$T_{OT} = T_{PO} + T_{TFO}$$

RADIC uses different kinds of redundancies.

- Time redundancy (distributed application takes their checkpoints in an uncoordinated manner and receiver-based pessimistic messages log)
- Dynamic redundancy (spare nodes): the spare is chosen when an active node faults.

These redundancies are implemented using the following strategies:

- Distributed controller: The fully decentralized feature means that in any moment of the fault-tolerance process, including recovery, the solution does not need any central elements. Besides, RADIC necessarily needs to keep decentralize all information about the spares. All nodes should work independently, exchanging information as needed just with a few neighbor nodes.
- Distributed stable storage: The checkpoints and log messages are saved decentralized in the neighbor nodes. Decentralization reduces the storage overhead caused by central storage and distributes the redundant data saving activity among the nodes
- Reduce synchronization overhead: The RADIC controller uses checkpoint/restart and receiver-based pessimistic log rollback-recovery protocol to handle the faults in order to satisfy the scalability requirement. This protocol allows a recovery mechanism that does not demand synchronization between the in-recovering process and the processes not affected by the fault.
- Reduce load imbalance (flexible dynamic redundancy): RADIC makes possible the implementation of several strategies to face the load balance problem after process recovery.

Thus, the overhead function for a fault tolerant parallel system (T_O) is the parallel systems overhead (T_{OP}) plus the overhead due to the fault tolerant mechanism (T_{OFT}) (equation 5).

$$T_O = T_{OP} + T_{OFT} \quad (5)$$

From that equation we already knew T_{OP}, so we must study T_{OFT}. In the case we implement fault tolerance with RADIC, that overhead depends on the four classic phases that RADIC implements: Protection, Detection/Diagnosis, Recuperation and Reconfiguration.

- Protection: in this phase Checkpoints (there is one for each process P and time increases due to P, state size and bandwidth because the checkpoint must be sent to another node) and pessimist messages log (it depends on the application behavior: more or less messages, strong or weak coupled) affects overhead.
- Detection/Diagnosis: the HB/WD operation impacts the overhead. The number of nodes N has influence but it is a totally distributed operation in detection. We can consider that it only depends on the relationship with 1 neighbor (2 nodes

interaction), in diagnosis with 2 neighbors (3 nodes interaction), and it doesn't increase with N.

- Recuperation: it is a local process in which one node interacts with another.
- Reconfiguration: it is a completely distributed operation, the system is reconfigured meanwhile a fault is discovered by other node controllers.

Due to all that has been mentioned above, we need to define a functional model in order to evaluate the performability of RADIC architecture considering the different elements and tasks that are involved in the fault tolerant mechanism.

4.4 Functional Model

T_{OFT} is produced by RADIC components which implement fault tolerance.
If we consider RADIC as a parallel application which runs beside the application and interacts with it, we'll be able to analyze the operations performed, the required time and the resulting overhead. Those parameters help us to define a functional model to validate the performance of a parallel application running in a parallel computer with RADIC fault tolerant architecture. Table 1 depicts the functional model of RADIC taking into account the operations that impact on the overhead considering the operations that influence on the overhead caused by fault tolerance.

Table 1. Functional Model of RADIC.

Application running without faults	Application running with faults
• Detection: Watchdog and heartbeat (interval and bandwidth, application type) • Protection: Checkpoint frequency and size (problem size, bandwidth, creation and sending time, affected process while the checkpoint is done) • Protection: Message log (application type, compute or communication addressed, bandwidth, message size)	• Recovery: Spare node search, spare node connection to the protection chain, WD/HB mechanisms re-establishment and failed process recovery. Sending the checkpoint to the spare for its recovery and "forced" checkpoint (made by the observer after the re-execution in order to have protection again) • Recovery: If there aren't spare nodes, the failed process is recovered in a node that already belongs to the cluster. Moreover, we must consider the degradation caused by the loss of computational power. Reconfiguration: Update of Radictable and Sparetable. Each observer must update its Radictable when it finds out that a process was changed to another node after a fault.

In a first approach this model leads us to make the following reflections:

1. RADIC tasks (protection, detection, recovering and reconfiguration) are defined in function of some parameters defined by the user (checkpoint interval, heartbeat interval) and application features (state size in memory, communications)
2. Besides, we must also consider the relation between computing and communication. For example, when there is not a log, communication and

computing are overlapped. Two things may happen if a log is added: a) the log remains overlapped or b) the log overflows computing time.
3. Checkpoints always cause an overhead which can have more o less impact according to the behavior more or less synchronized of the processes.
4. In asynchronic communications, the overhead that RADIC introduces depends on the application computing/communication ratio. In synchronic communications computing and communication don't overlap. In this case communication time is at least (in a regular case) the original communication times two. The same happens with broadcast communication.

5 Scalability measurement, an evaluation using RADIC

Taking into account the functional model, performance metrics of distributed parallel system and scalability concept, we designed an experiment over a RADIC cluster and a parallel system (with similar features) without fault tolerance. This experiment consists of obtaining runtimes of matrices multiplication programs. The used matrices had 5000, 7500 and 10000 elements. The programs were running over a parallel system with 4, 8 and 16 nodes, and over a RADIC cluster with 1 spare node. The measures were taken without faults and introducing a fault at 25% runtime.

The application consists of the product of a Hilbert Matrix programming as a master/worker with dynamic load balance.

The experiments were conducted on a multicore cluster DELL with 4 nodes, each node has 2 Quadcore Intel Xeon E5430 of 2.66 Ghz processors, and 6 MB of cache L2 shared by each two core and RAM memory of 12 GB by blade.
Finally the 3 sizes experiments were performed in workers with 4, 8 and 16 nodes taking into account 4 scenarios: a. without RADIC (without faults), b. with RADIC (with faults), c. with RADIC, without spare node and 1 fault (checkpoint interval: 60 seconds), d. with RADIC, without spare nodes and 1 fault (checkpoint interval: 60 seconds).

Table 2 depicts the obtained results in seconds.

Table 2. Experiment results.

Scenarios	Matrices of 5000			Matrices of 7500			Matrices of 10000		
	4w	8w	16w	4w	8w	16w	4w	8w	16w
a	454,6	232,6	124,3	1410,3	711,8	367,5	3582,5	1808,5	950,3
b	472,2	241,3	134,8	1454,7	737,5	406,1	3702,9	1870,9	1001,2
c	584,9	272,6	148,2	1817,9	831,5	436,8	4624,6	2105,3	1069,8
d	479,8	248,1	142,0	1465,5	744,9	413,7	3710,9	1877,2	1009,9

During experimentation:

1. The master was always isolated in a node and workers shared 3 nodes. All workers have the same distance to the master and they also share the same network. There is no communication between the workers and the load remains balanced (overlapped communications, even with messages log).

2. The four executions of each experiment don´t present a significant variance. Thus, we can assume that with more executions the numbers will remain the same.

This first analysis of the obtained results shows that time arises a lot when the application runs over 16 workers, whatever the size of the application is. Perhaps that was caused because the load of the application wasn't balanced in the workers. Because of that we decided to begin with the design of a new experimentation, based on the proposed functional model. This time we'll try to isolate time variations done by the application and of those produced by RADIC.

6 References

1. Duarte, A., Rexachs, D., and E. Luque.: Increasing the cluster availability using RADIC. Cluster Computing, 2006 IEEE International Conference on, pages 1–8. (2006)
2. Duarte, A., Rexachs, D., and E. Luque.: Functional Tests of the RADIC Fault Tolerance Architecture. Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on, pages 278–287. (2007)
3. Grama, A., Gupta, A., Karypis, G., and V. Kumar.: Introduction to Parallel Computing, Second Edition. Addison Wesley. January 16. ISBN 0-201-64865-2. (2003)
4. Koren , I., and M. Krishna.: Fault-Tolerant Systems. Morgan Kaufmann Publishers is an imprint of Elsevier. San Francisco. United States. (2007)
5. Grama, A., Gupta, A., and V. Kumar.: Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Parallel & Distributed Technology. Vol. 1, No 3, pp. 12-21. (1993)
6. Sun, X. and D. T. Rover.: Scalability of parallel algorithm-machine combinations. IEEE Transactions on Parallel and Distributed Systems. Vol. 5, no. 6, pp. 599-613. (1994)
7. Santos, G., Duarte, A., Rexachs, D., and E. Luque.: Providing Nonstop Service for Message-Passing Based Parallel Applications with RADIC. In: Proceedings of the 14th international Euro-Par conference on Parallel Processing. Springer-Verlag, pp.58-67. (2008)