On the Calculation of the Checkpoint Interval in Run-Time for Parallel Applications

Leonardo Fialho*, Dolores Rexachs and Emilio Luque
Department of Computer Architecture and Operating System, University Autonoma of Barcelona, Spain
leonardo.fialho@caos.uab.es, dolores.rexachs@uab.es, emilio.luque@uab.es

Abstract— *The growth in the number of components that compose parallel computers increases their fault frequency. Currently, in such systems faults are no longer a rare event but a common problem, thus some sort of fault tolerance should be provided. In general, fault tolerance protocols rely on checkpoints. A common question surrounding checkpointing is the definition of the checkpoint interval. Checkpoint interval models define variables which depends on application characteristics, e.g. the time need to take a checkpoint. The use of average values and/or statistical data to define these variables reduces the model's accuracy. In this paper we propose a methodology to define in run-time the variables value needed to calculate the checkpoint interval. While using uncoordinated checkpoint this interval can be defined individually for each process of the parallel application. The variables definition relies on the measuring of the time spent on fault tolerance tasks in run-time. Experimental evaluation shows that the use of our methodology reduces in more than 3% the overhead introduced by fault tolerance while tested applications are running in a faulty environment.*

Keywords: MPI; fault tolerance configuration; checkpoint interval; uncoordinated checkpoint.

1. Introduction

The growth in the number of components that compose parallel computers increases is notorious for increasing their fault frequency [1]. Currently, in such systems faults are no longer rare events but a common problem. Some systems such as the BlueGene/L the Mean Time Between Failures (MTBF) is counted in days. However, the commodity clusters exhibit a usual MTBF of tens of hours [2]. The natural answer for this problem is to provide some sort of fault tolerance for applications running on these systems. This permits applications to finish successfully despite faults.

To write applications with native support for faults seems to be a good option. There are many techniques that help developers to codify fault tolerant parallel applications [3]. Many of these techniques are suitable to be used in conjunction with MPI: a widely used message passing library for parallel programming. However, this approach requires the rewriting of legacy applications. Another solution is to provide fault tolerance at the communication library level and on the parallel environment. The combination of a resilient parallel environment and a fault recovery technique had been useful in MPI implementations like MPICH [4] and Open MPI [5][6].

To save the application state and to resume its execution in case of faults is commonly known as rollback-recovery. There are different rollback-recovery protocols that can be useful to assure application completion [7][8]. These protocols ultimately rely on checkpoints as the main state-saving technique or to save storage space while using combined with message logging *i.e.* reducing the space needed to store message log. The matter in question which surrounds checkpointing is the definition of the frequency in which checkpoints should be taken, better know as the checkpoint interval. If the checkpoint interval is smaller o bigger than the optimal the overhead added by the fault tolerance increases [9]. Because checkpointing is a widely used technique, there have been studies regarding the definition of its interval since the 70's [10] until today [11].

However, these studies are far from being the ultimate solution to the checkpoint interval. The major root cause of this resides in the definition of the variables value used by these models. The use of average values as input parameters for models reduces their accuracy. During the execution, some application characteristics may change over the time. Thus, models will experience a loss of accuracy because the checkpoint interval does not change to reflect such changes. Models variables depend on the application characteristics such as the memory footprint and the communication pattern, besides the system load such as the storage and the communication network.

In this paper we propose a methodology to define in run-time the checkpoint interval for parallel applications. The dynamic definition relies on the measuring of the time spent on fault tolerance tasks to obtain values for the checkpoint interval model variables. It turns the checkpoint interval model versatile enough to accommodate changes in the application characteristics throughout its execution.Experimental evaluation shows that the use of our methodology reduces in more than 3% the overhead introduced by fault tolerance while tested applications are running in a faulty environment.

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

*Contact author to whom correspondence should be addressed.

†This paper is addressed to the PDPTA conference.

The content of this paper is organised as follows. The related work is introduced in section 2. In section 3 a description of the model used and some improvements made on it is presented. Results are shown in section 4. Conclusions are stated in section 5 besides future work.

2. Related Work

In the last years many fault tolerance MPI implementations has been designed. In general, those implementations rely on a rollback-recovery protocol. Such protocols are based on checkpointing, message logging, or both combined [7]. The era in which parallel applications are unable to finish due to faults in the parallel machine has gone.

However, due to the overhead introduced by fault tolerance tasks, especially by the recovery phase, researchers start to work on adaptive fault tolerance [12][13][14]. Adaptive fault tolerance requires information such as status and error reports about the machine in which applications are running. Indeed, a framework has been designed [15] to provide such information to fault tolerance libraries. This permits the creation of runtime strategies to dynamically reconfigure the parallel environment to avoid application being affected by faults [16].

Furthermore, there is a lack of studies regarding the configuration of the fault tolerant strategy according to specific applications characteristics. Working on this direction, Chen and Ren have published a study about the impact of the checkpoint interval on soft real-time applications [14]. Moreover, recently Jones et al. have published a work about the impact of a misconfigured checkpoint interval on the application efficient [9]. Despite of this, there are too few studies about the dynamic definition of the fault tolerance configuration according to specific application requirements.

3. The Methodology to Define the Checkpoint Interval in Run-Time

Our propose to define the checkpoint interval in run-time rests on two foundations: first on a checkpoint interval model and second on the measurement of the time needed to perform fault tolerant tasks. The second provides the values for the variables used by the first.

To help familiarise the reader with the checkpoint interval model used in this paper, the following list provides useful definitions:

- α as the mean time to interrupt (MTTI) for a given system, which is the inverse of the fault probability.
- σ as the checkpoint interval used to run the application.
- t_c as the time spent on a checkpoint operation including the storage time.
- t_l as the time needed to load a checkpoint from storage, not the rework time.
- Δ_{lp} as the time added to message delivery due to the logging procedure.
- Δ_{lr} as the time spent on processing the message log after a fault.
- ϕ as the factor which represents the inter-process dependency [17].

To define the value of these variables, we propose a monitoring mechanism of the fault tolerant tasks performed during application execution. The diagram shown in figure 1 depicts such a mechanism.

The time needed to perform a checkpoint operation (t_c) is measured by the timer depicted in events 1 and 2 of the diagram. The inter-process dependency factor is calculated by analysing sources and destinations of messages exchanged with other processes and is depicted in the diagram by event

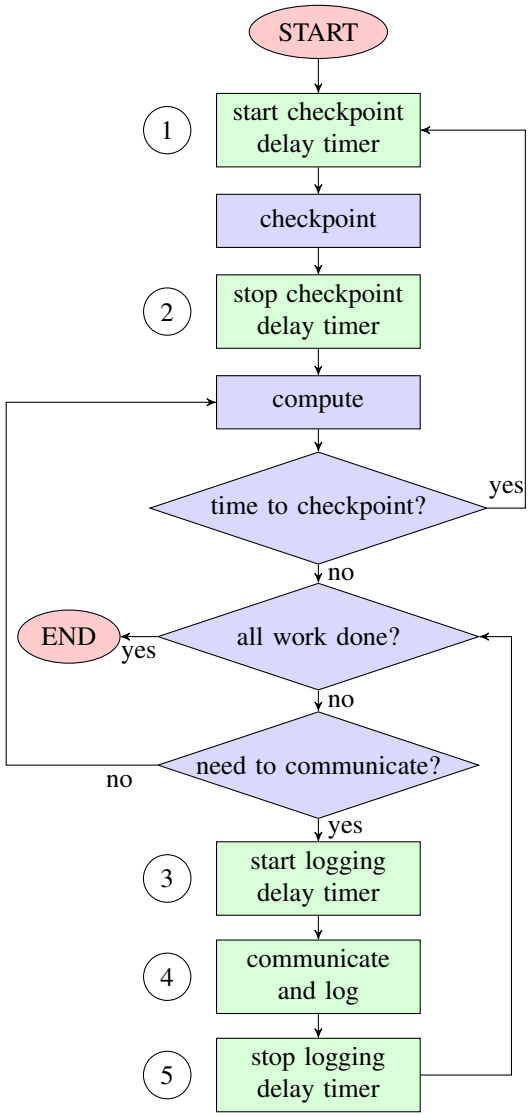


Fig. 1: Diagram of the methodology used to define model variables values in run-time.

4. The message logging overhead depends on the logging protocol used [7].

In this paper we will analyse the overhead introduced by a pessimistic receiver-based message logging. The time added to message delivery due to the logging procedure (Δ_{lp}) is measured by the timer depicted in events 3 and 4. When the message logging operation is performed on the receiver process, as shown in figure 2, in case of faults, only the faulty process is involved in recovery. Since messages do not need to be replayed, the time needed to process the message log (Δ_{lr}) tends to be unappreciable. Thus, the value of the Δ_{lr} variable can be considered as zero [19].

The time needed to load a checkpoint (t_l) cannot be measured using our methodology if no fault occurs. However, as a first approximation we consider this time equal to the time needed to perform a checkpoint. It does not reduces the checkpoint interval model accuracy because variables related to the recovery phase, with the exception of the rework time, tend to be inappreciable [10].

After all variables values needed by the checkpoint interval model had been already defined in run-time it is possible to use the following checkpoint interval model to calculate the checkpoint interval.

For a system with a known MTTI, the following equation[1] estimates the wall-clock time T_{est} required to run an application (which originally takes T_p time to conclude) in a faulty environment with fault tolerance:

$$T_{est} = T_p\left[1 + \frac{\phi\sigma^2 + \sigma(2\phi t_l + \phi t_c + 2\phi\Delta_{lr} - t_c + 2\Delta_{lp})}{\alpha(2\sigma + 2t_c)} + \frac{2t_c(\phi t_l + \phi\Delta_{lr} + \alpha - t_l - \Delta_{lr} + \Delta_{lp})}{\alpha(2\sigma + 2t_c)}\right] \quad (1)$$

[1]The definition of the checkpoint interval model used in this paper can be found in `http://caos.uab.es/~lfialho/ic/parallel_model.pdf`.

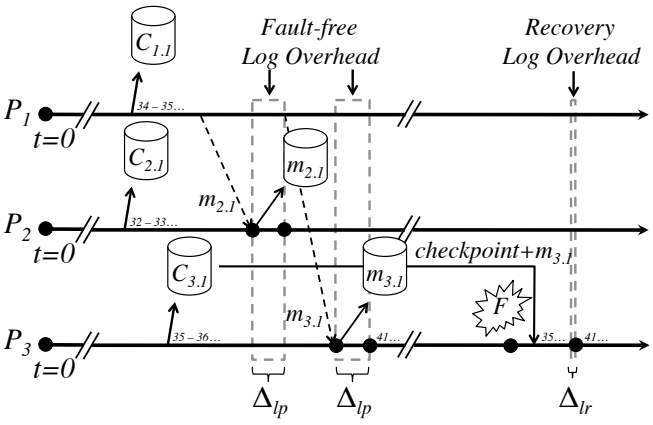


Fig. 2: Overhead introduced by a pessimistic receiver-based message logging protocol during a fault-free execution (Δ_{lp}) and during the recovery phase (Δ_{lr}).

In the equation above, the fault detection latency has been omitted. This variable can be safely omitted because it depends on the fault detection mechanism implemented by the fault tolerance architecture. If we suppose the use of a heartbeat/watchdog system, then the higher the heartbeat frequency, the smaller the detection time. Moreover, the heartbeat communication does not impose a considerable overhead on the system. In addition, there is always the possibility of using the application communication as another fault detection mechanism [18].

The checkpoint interval that minimises the fault tolerance overhead of the aforementioned model is:

$$\sigma = \frac{\sqrt{\phi t_c(t_c + 2\alpha - 2t_l - 2\Delta_{lr})}}{\phi} - t_c \quad (2)$$

and the inter-process dependency factor is defined by the following equation:

$$\phi_{global} = \frac{\sum_1^N P(n)}{N^2} \quad (3)$$

where $P(n)$ is the function which defines the number of processes that depend on the process n including itself. *N* is the total number of processes in the parallel application.

There is an issue with the whole model presented related to the definition of the inter-process dependency factor. Equation 3 reflects this inter-process dependency factor for the entire application. This factor should be redefined to represent the dependency of an individual process in relation to other processes. Below we present our propose to the redefinition of this factor:

$$\phi = \frac{P(n)}{N} \quad (4)$$

and finally, the value for this factor is defined in run-time based on the application monitoring. As in previous equation, $P(n)$ is the function which defines the number of processes that have sent to or received messages from process n, including itself. *N* is the total number of processes in the parallel application.

Our methodology is based on measurements taken during the most recently checkpoint cycle. When the application changes its behaviour, *i.e.* the communication pattern, or its memory footprint, after one checkpoint cycle the checkpoint interval will already be adapted to the new application characteristics. Moreover, except during the start-up and finalisation it is expected that applications do not change their behaviour or memory footprint too frequently in comparison to the checkpoint interval [20].

4. Experimental Evaluation

To evaluate our proposal, three experiments were designed. In the first experiment we depict the adaptation of the checkpoint interval to changes in the process size and we

also demonstrate the influence of the communication pattern on the inter-process dependency factor (ϕ). The second experiment depicts the reduction in the overhead introduced by fault tolerance while using our methodology to define the checkpoint interval in run-time. Finally, the third experiment shows the accuracy of the inter-process dependency factor defined in run-time.

Experiments run in an 8-node cluster. Each node was equipped with two Dual-Core Intel Xeon processors running at 2.66GHz, 12 GBytes of main memory, and a 160 GBytes SATA disk for local storage. Nodes were interconnected via two Gigabit Ethernet interfaces. One of these networks was used for storage while the other was used for process communication. RADIC/OMPI [6] was used as a fault tolerant MPI library.

To create a fault scenario processes are killed during the application execution. The fault moment is defined by the MTTI (α) and its distribution along the MTTI is defined using the MT19937 PRNG algorithm [22]. The processes to be killed is selected using the same algorithm. After the fault has been injected the node is available to be reused by a recovered process.

4.1 Checkpoint Interval Adaptation

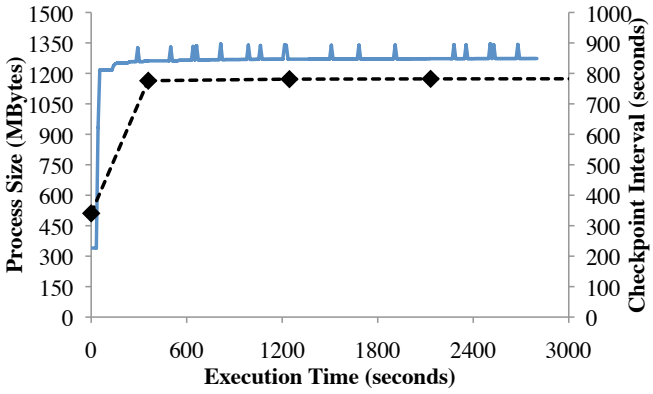
Model variables such as t_c and t_l depends on the amount of memory used by application processes. And processes on the same application may present different memory footprints. This occurs because processes compute different data or processes play different roles in the parallel application.

To depict the adaptation of the checkpoint interval to the process memory footprint we have used the NAMD molecular dynamics application [21]. NAMD is implemented over a Master/Worker paradigm where workers also communicate between themselves; the master process requires more memory in comparison to the workers. The experiment has been executed using with a fault frequency (α) of 3600 seconds and the heartbeat frequency (t_d) was set to 1 second. Values for t_c, t_l, Δ_{lp}, and Δ_{lr} were measured during the execution. For this application we have manually calculated the inter-process dependency factor (ϕ) and its value is 1.

Dashed lines in figures 3 depict the checkpoint interval used throughout the application execution. Figure 3(a) refers to the master process, while figure 3(b) refers to a worker process. Figure 3 depicts only one worker processes, however others present a similar behaviour.

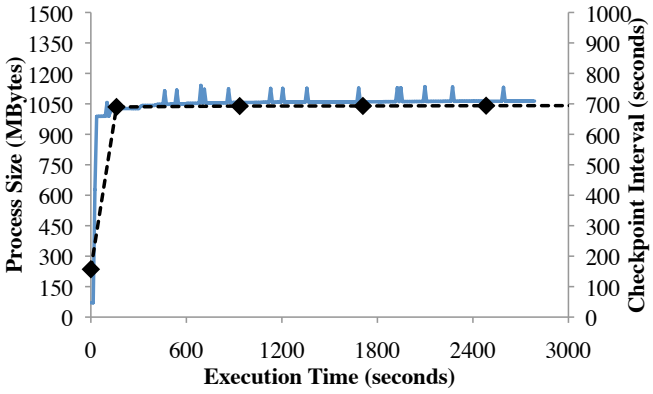
As the figures depict, processes use a small amount of memory in the startup phase. As a consequence of this the model calculates a short checkpoint interval initially. However, after the startup phase the application increases its memory footprint. After the second checkpoint the checkpoint interval changes to reflect the changes on the process memory footprint. Tables in figure 3(a) and 3(b) summarise the checkpoint instances and sizes for the master and a worker process, respectively.

To depict the adaptation of the checkpoint to the inter-process dependency factor we have used a dynamic matrix multiplication application built under a Master/Worker paradigm where workers only communicate with the master



Execution Instant	Process Size	Checkpoint Interval
0.31 seconds	339.96 MB	340.56 seconds
358.45 seconds	1261.75 MB	775.93 seconds
1245.35 seconds	1270.05 MB	779.04 seconds
2132.25 seconds	1272.37 MB	779.41 seconds

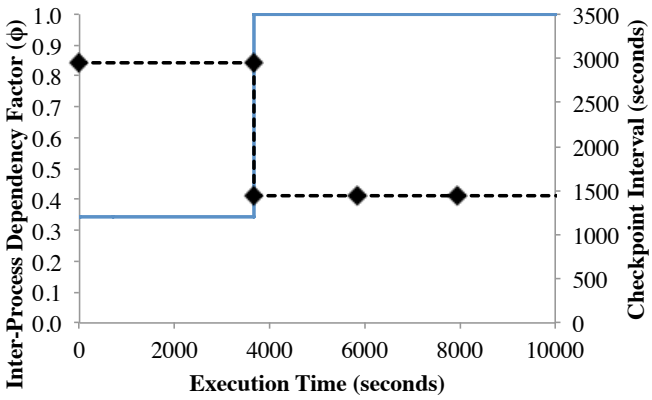
(a)



Execution Instant	Process Size	Checkpoint Interval
0.19 seconds	70.57 MB	157.35 seconds
160.95 seconds	1026.79 MB	689.77 seconds
934.97 seconds	1056.99 MB	693.29 seconds
1708.81 seconds	1060.23 MB	693.87 seconds

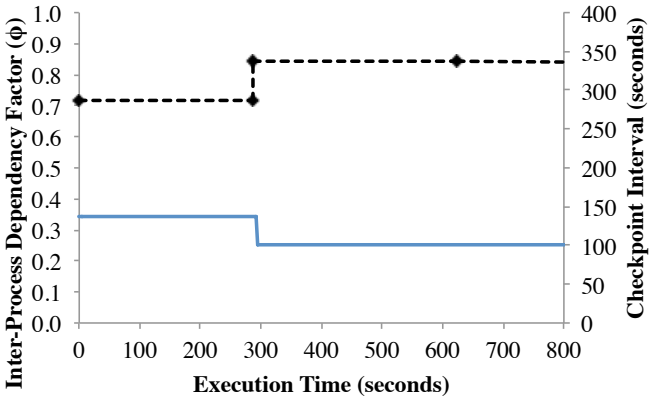
(b)

Fig. 3: The continuous line shows the memory footprint of the NAMD (a) master and (b) worker processes running the "stmv" workload; values are shown on the left axes. The dashed line represents the checkpoint interval used; values are shown on the right axes. The rhombus points depict checkpoint instances. Tables depict first four values of the checkpoint size and the calculated next checkpoint interval for each type of process according to the execution instant.



Execution Instant	Process ϕ	Checkpoint Interval
0.84 seconds	0.34375	2,949.62 seconds
3,658.74 seconds	1.00000	1,436.16 seconds
5,823.01 seconds	1.00000	1,433.98 seconds
7,928.74 seconds	1.00000	1,439.21 seconds
10,054.94 seconds	1.00000	1,433.73 seconds

(a)



Execution Instant	Process ϕ	Checkpoint Interval
0.31 seconds	0.34375	286.52 seconds
286.84 seconds	0.25000	336.68 seconds
623.52 seconds	0.25000	334.99 seconds
960.20 seconds	0.25000	337.09 seconds
1,296.88 seconds	0.25000	336.96 seconds

(b)

Fig. 4: The continuous line shows the value of ϕ for the matrix multiplication (a) master and (b) process; values are shown on the left axes. The dashed line represents the checkpoint interval used; values are shown on the right axes. The rhombus points depict checkpoint instances. Tables depict first five values of ϕ and the calculated next checkpoint interval for each type of process of the matrix multiplication execution according to the execution instance.

process. This parallel application was executed using 8 nodes.

Considering the equation 3, the initial value for the ϕ variable is 0.34375. This value represents a global view of the relationship established between all processes on this parallel application. The fault frequency (α) has been defined as 3600 seconds and the heartbeat frequency (t_d) has been set to 1 second. Values for t_c, t_l, Δ_{lp}, Δ_{lr}, and ϕ are measured during the execution. Equation 4 has been used to define in run-time the value of ϕ for each process.

Continuous lines in figure 4 depict the calculated values for the ϕ in run-time for the master 4(a) and for a worker 4(b) process, respectively. In addition, the dashed line on these figures depict the values of the checkpoint interval during the application execution as well as the checkpoints instances.

As shown in figure 4(a), the initial value of 0.34375 was redefined to 1. This occurs because between the first and the second checkpoint the master process communicated with all 7 workers. As a consequence of this increase in the value of ϕ, the model has changed the checkpoint interval. Similarly, in figure 4(b) the decrease in the value of ϕ increases the time between checkpoints for a worker process.

Figure 4 depicts only one worker process, however, other worker processes present similar behaviour. The huge difference between the checkpoint interval calculated for the master and for the worker process is caused by the difference in the memory footprint of these processes.

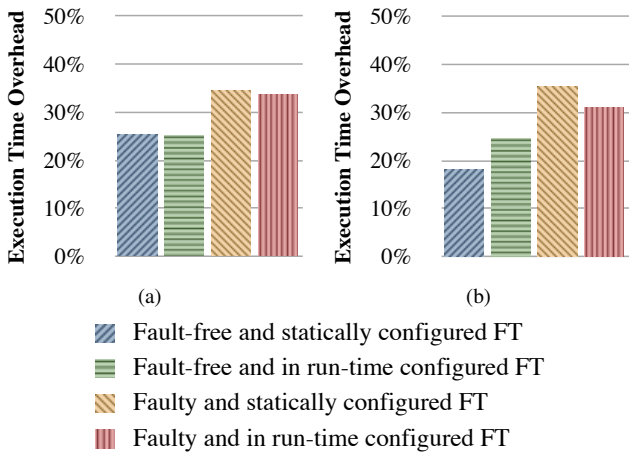
4.2 Reduction in the Overhead

The next experiments depict the performance gain in using our methodology to define the checkpoint interval in run-time. This experiment compare the performance of our proposal with a static configuration in a faulty and fault-free scenario. The comparison was made using the aforementioned NAMD and dynamic matrix multiplication applications. In this experiment only one fault was injected in each execution. The moment of the fault differs from one execution to other. The fault is distributed along the application execution according to the MT19937 PRNG algorithm. Each experiment has been executed at least in 16 times and values are the average of all data that fall in a 95% confidence interval.

As shown in figure 5(a) the use of the fault tolerance provided by the RADIC/OMPI library introduces an overhead of about 25% in a fault-free execution and about 34% in a faulty scenario.

As shown in table in figure 5 there is a modest reduction in the overhead while the checkpoint interval is calculated in run-time. This occurs because there is no significative change in the NAMD processes characteristics, except for the memory footprint in the start-up phase. However, as shown in figure 5(b) the matrix multiplication application presents different results.

In the fault-free environment the execution using statically configured checkpoint interval presents a smaller overhead than the execution running with in run-time configuration.



App	Statically Configured		Configured in Run-Time	
	Fault-free	Faulty	Fault-free	Faulty
NAMD	25.4%	25.1%	34.5%	33.6%
MM	18.1%	24.6%	35.4%	31.1%

Fig. 5: Comparison of the (a) NAMD and (b) a matrix multiplication execution time using different fault tolerance configuration strategies on different environments.

This is because the initial global value of ϕ increases the checkpoint interval for the master process. This reduces the number of checkpoints performed. In this situation, the overhead introduced by a fault increases. This can be verified when we compare the total wall time clock in a faulty environment for the configurations made statically and in run-time.

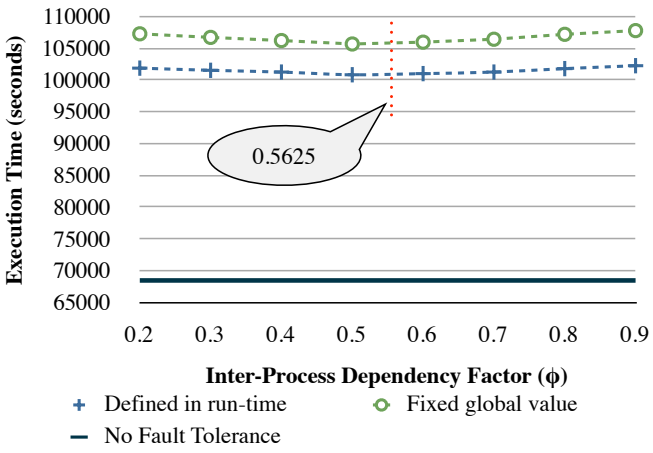
4.3 The Accuracy of the Inter-Process Dependency Factor Defined in Run-Time

To verify the accuracy of our model we executed the NAS [23] LU class B with 8 processes modified to iterate 300,000 times. This modified version of the LU has been executed using different global values for the ϕ, from 0.9 to 0.2. Table 1 shows the correct value of this factor individually and globally for this application.

Analysing the curve in figure 6 and the data present in the table it is possible to guess that the optimum value for the ϕ for this execution should be a value between 0.45 and 0.60. Despite of the small difference between the global and the individualised values for the ϕ, to use of a precise value for this factor reduces in more than 3% the overhead introduced

Table 1: Values for the inter-process dependency factor for the entire LU application and for each process individually.

Process Rank *(Global Value)*	$P(n)$	ϕ
Running with 8 processes		**0.56250**
0, 3, 4, 7	4	0.50000
1, 2, 5, 6	5	0.62500



ϕ	Overhead Using Fixed Global Value	Overhead Using a Defined in Run-Time Value	Difference
0.2	36.2%	32.8%	3.38%
0.3	35.8%	32.6%	3.27%
0.4	35.5%	32.4%	3.16%
0.5	35.2%	32.1%	3.11%
0.6	35.4%	32.2%	3.17%
0.7	35.7%	32.4%	3.28%
0.8	36.1%	32.7%	3.39%
0.9	36.5%	33.0%	3.43%

Fig. 6: Comparison between static and in run-time configured values of the inter-process dependency factor.

by the fault tolerance tasks for this application.

5. Conclusions

Checkpoint interval models used to rely on input variables based on average values. This reasoning is valid for applications running for a long time on systems that present a high fault frequency. However, this is not the common environment faced by parallel application users. This paper has presented a methodology to dynamically define the input variables used by models based on measurements performed during the application execution.

We propose the monitoring of processes that compose the parallel application to achieve the values for the variables used by checkpoint interval model. We monitor the time needed to perform fault tolerant tasks as well as the number of peers each process communicates with.

This instrumentation allow the definition of the checkpoint interval in run-time with a high degree of precision, process by process. The use of this methodology reduces in about 3% the overhead introduced in the execution time for applications running in faulty environments.

5.1 Future Work

The overhead added to the application execution by the monitoring mechanism tends to be unappreciable. However,

it is necessary to quantify this overhead.

The use of uncoordinated checkpointing is the only solution that allows the use of different checkpoint intervals for each application process. However, the use of uncoordinated checkpointing assisted by message logging may not be the solution that presents the lowest overhead. There is the need to analyse if a sender-based message logging or a coordinated checkpointing solution present better results.

References

[1] F. Cappello, "Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities," *International Journal of High Performance Computing Applications*, pp. 212—226, 2009. [Online]. Available: http://dx.doi.org/10.1177/1094342009106189

[2] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the Message Logging Model for High Performance," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196—2211, 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.1589

[3] W. Gropp and E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363—372, 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004046045

[4] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging," *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, p. 25, 2003. [Online]. Available: http://dx.doi.org/10.1109/SC.2003.10027

[5] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*, 2007. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2007.370605

[6] L. Fialho, G. Santos, A. Duarte, D. Rexachs, and E. Luque, "Challenges and Issues of the Integration of RADIC into Open MPI," *Proceedings of the 16th European PVM/MPI Users' Group Meeting*, pp. 73—83, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1612227

[7] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375—408, 2002. [Online]. Available: http://dx.doi.org/10.1145/568522.568525

[8] S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," *Sadhana*, vol. 25, no. 5, pp. 489—510, 2000. [Online]. Available: http://dx.doi.org/10.1007/BF02703630

[9] W. Jones, J. Daly, and N. DeBardeleben, "Impact of Sub-optimal Checkpoint Intervals on Application Efficiency in Computational Clusters," *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 276—279, 2010. [Online]. Available: http://dx.doi.org/10.1145/1851476.1851509

[10] J. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530—531, 1974. [Online]. Available: http://dx.doi.org/10.1145/361147.361115

[11] J. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303—312, 2006. [Online]. Available: http://dx.doi.org/10.1016/j.future.2004.11.016

[12] Y. Li and Z. Lan, "Exploit Failure Prediction for Adaptive Fault-Tolerance in Cluster Computing," *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, pp. 531—538, 2006. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2006.45

[13] Z. Lan, Y. Li, Z. Zheng, and P. Gujrati, "Enhancing Application Robustness through Adaptive Fault Tolerance," *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, 2008. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2008.4536383

[14] N. Chen and S. Ren, "Adaptive Optimal Checkpoint Interval and Its Impact on System's Overall Quality in Soft Real-time Applications," *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 1015—1020, 2009. [Online]. Available: http://dx.doi.org/10.1145/1529282.1529506

[15] R. Gupta, P. Beckman, B. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems," *Proceedings of the 2009 International Conference on Parallel Processing*, pp. 237—245, 2009. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2009.20

[16] Y. Li, Z. Lan, P. Gujrati, and X. Sun, "Fault-Aware Runtime Strategies for High Performance Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 4, pp. 460—473, 2009. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2008.128

[17] L. Fialho, D. Rexachs, and E. Luque, "What Is Missing in Current Checkpoint Interval Models?" **To appear in** *the Proceedings of the 31th International Conference on Distributed Computing Systems*, 2011.

[18] A. Duarte, D. Rexachs, and E. Luque, "Increasing the cluster availability using RADIC," *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 2006. [Online]. Available: http://dx.doi.org/10.1109/CLUSTR.2006.311872

[19] S. Rao, L. Alvisi, and H. Vin, "The Cost of Recovery in Message Logging Protocols," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 160—173, 2000. [Online]. Available: http://dx.doi.org/10.1109/69.842260

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 45—57, 2002. [Online]. Available: http://dx.doi.org/10.1145/605397.605403

[21] J. C. Philips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable Molecular Dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781—1802, 2005. [Online]. Available: http://dx.doi.org/10.1002/jcc.20289

[22] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3—30, 1998. [Online]. Available: http://dx.doi.org/10.1145/272991.272995

[23] W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow, "New Implementations and Results for the NAS Parallel Benchmarks 2," *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.3199