# Defining the Checkpoint Interval for Uncoordinated Checkpointing Protocols

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**— *Parallel applications running on large computers suffer from the absence of a reliable environment. Fault tolerance proposals, in general, rely on rollback-recovery strategies supported by checkpoint and/or message logging. There are well-defined models that address the optimum checkpoint interval for coordinated checkpointing. Nevertheless, there is a lack of models concerning uncoordinated checkpointing combined with message logging. First we present a model designed for serial applications or coordinated checkpointing-based solutions. Our contribution is the extension of this model to a scenario based on uncoordinated checkpointing combined with message logging. We introduce two key points to minimise the fault tolerance overhead for parallel applications. The first is the use of a factor to represent the dependency relation between processes. The second is the use a specific checkpoint intervals for each process. Experiments show that our model performs as well as previous studies for serial applications or coordinated checkpointing. While running parallel applications using uncoordinated checkpointing combined with message logging, our checkpoint interval model effectively minimises the overhead introduced by the fault tolerance tasks. Moreover, the overhead prediction error is smaller than 5% for all applications tested.*

**Keywords:** MPI; fault tolerance; checkpoint interval; model; uncoordinated checkpoint.

## 1. Introduction

Fault tolerance has become an important issue for parallel applications in the last few years. The growth of the number of components, which form parallel machines, are the major root causes of the failures increasingly seen on these machines. In order to achieve the execution end, parallel applications should use some fault tolerance strategy. Strategies can be the use of redundant hardware or the incorporation of the redundancy by software. Actually, the second is cheaper than the first even though it represents an overhead on the application run time.

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.

Checkpointing is an established rollback-recovery technique used to achieve fault tolerance on applications. There are well-defined models [1], [2] to calculate the checkpoint interval to minimise the overhead introduced by fault tolerance and maximise the application efficiency, *i.e.*, "the ratio of time the job spends making forward progress compared to the entire wall-clock time" [3]. Nevertheless, these models have been designed based on serial applications. The use of models based on serial applications is acceptable for parallel applications when they are protected by coordinated checkpointing. However, for uncoordinated checkpointing-based strategies these models would not be useful.

Fault tolerance architectures designed for parallel machines such as MPICH-V [4] and RADIC [5] are based on uncoordinated checkpointing combined with message logging. Ergo, these architectures suffer from the lack of models to calculate the checkpoint interval as well as to predict the overhead introduced by the fault tolerance architecture on the application run time.

In this study we will propose a novel model to calculate a checkpoint interval to minimise the overhead introduced by fault tolerance architectures based on uncoordinated checkpointing combined with message logging in parallel applications. For the sake of deducing the parallel application fault tolerance model, a model based on serial applications will be introduced first. The model to calculate the checkpoint interval for parallel application incorporates the message logging influence on the overhead and a factor to measure the dependency relationship between processes. We present two key points to minimise the fault tolerance overhead in parallel applications. The first is the use of a factor to represent the dependency relationship between parallel application processes. The second is the use of different checkpoint intervals for each parallel application process based on its own characteristics.

Experiments show that our model performs as well as previous studies while running serial applications. For parallel applications the overhead prediction error is less than 5% while running with uncoordinated checkpointing combined with message logging.

The content of this paper is organised as follows. The related work is presented in section 2. Section 3 introduces the checkpoint interval models. The experimental evaluation comes in section 4. The conclusions are stated in section 5.

## 2. Related Work

The use of analytical models to define the optimum checkpoint interval for serial applications has been studied from the 70's until today. In 1974, Young [6] introduced the first order approximation, an analytical model to determine the optimum checkpoint interval. Using Young's model it is possible to calculate the optimum checkpoint interval once the user knows the time needed to perform a checkpointing operation and the system fault probability. In order to predict the overall running time while performing checkpoints, Young's model requires the original application run time as well as the time needed to recover a failed process. More recently, Lin *et al.* [7] and Gropp and Lusk [1] presented a model achieving the same result for the optimum checkpoint interval. Nevertheless, they use different approaches that lead to different overhead prediction equations.

Many years after Young, Daly presented a deeply analytical study [2] to determine the higher order estimation of the optimum checkpoint interval. Daly analyses different scenarios such as multiple failures between checkpoints, fractional rework (the amount of work completed after a checkpoint and prior to a failure), failures during restarts, *et cetera*.

Despite Gelenbe *et al.* [8] and Ken and Mark's [9], whom were analysing models for distributed computing, there is a lack of knowledge surrounding the optimum checkpoint interval for parallel applications. Currently, many studies about fault tolerance for parallel applications are limited to presenting the approach used to achieve protection. Most of these studies discuss differences between coordinated and uncoordinated checkpointing, logging strategies, implementation details, architecture design, *et cetera*. In these studies in general, authors have omitted the method used to calculate the checkpoint interval used in experiments. A rare exception is Bouteiller *et al.* [10] who describe the model employed to calculate the checkpoint interval used to run experiments to depict the impact of fault frequency on the application run time. Nevertheless, this model is not useful to define the checkpoint interval to minimise the fault tolerance overhead.

Models designed to be used on serial applications can also be used on parallel applications under certain circumstances (*e.g.* if the application is protected by coordinated checkpointing). However, after a fault the system restores the last checkpoint and work done after the checkpoint and before the fault is lost.

To understand the impact of faults on parallel applications let us suppose a parallel application running with a rollback-recovery fault tolerance assisted by coordinated checkpointing. Checkpointing and recovery are collective operations that involve all processes in the parallel application. Thus, all processes are checkpointed at the same time and after a fault all processes should roll back and resume their operations from the last checkpoint. In the case of coordinated checkpointing models designed for serial applications can also be used.

However, parallel applications running with a fault tolerance system which implements an uncoordinated checkpointing protocol cannot use existing models to calculate the optimum checkpoint interval. In this scenario, after the fault occurrence only the faulty process needs to be rolled back. In this case just the work done by the faulty process is lost. Other processes continue to compute.

As far as we know, there is no model to calculate the checkpoint interval to minimise the overhead introduced by fault tolerance in this case. Also, there is no model to predict the overhead introduced by fault tolerance on parallel applications running with uncoordinated checkpointing.

## 3. Developing the Checkpoint Interval Models

In order to better understand the deduction of the checkpoint interval models let us consider figure 1 and the following naming system:

- $t_c$ as the time spent on a checkpoint operation including the storage time. In other words, it is the application interruption time necessary to take a checkpoint.
- $t_d$ as the time needed to detect a fault, also known as fault detection latency.
- $t_l$ as the time needed to load a checkpoint from storage.
- $t_r$ as the amount of time needed to recover a failed process and achieve the computation point just before fault. It is the reworking of the previous lost computation, also known as fractional rework.
- $Q$ as the quantity of checkpoints that should be performed between two faults.
- $T_c$ as the total protection time represented by the sum of all $t_c$ between two faults. This value can be obtained multiplying $t_c$ by $Q$. If there is some overhead introduced by the logging procedure this time need to be take in account.
- $T_r$ as the total recovery time per fault represented by the sum of $t_d$, $t_l$, and $t_r$.
- $\alpha$ as the mean time to interrupt (MTTI) for a given system, which is the inverse of the fault probability.
- $\sigma$ as the checkpoint interval used to run the application. It can also be considered as the useful time for the application to compute.
- $\Delta_{lp}$ as the time added to message delivery due to the logging procedure, if it exists.
- $\Delta_{lr}$ as the time spent no processing the message log after a fault. The majority of this is the replaying time, if it exists.

As shown in figure 1, the recovery task occurs at the beginning of the period between faults $F_x$ and $F_y$. Recovery takes $T_r$ time (segment $\overline{BE}$) to conclude and after this

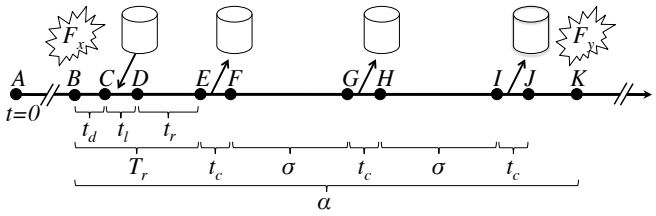



Fig. 1: Between faults $F_x$ and $F_y$ there is a recovery time $T_r$ and 3 checkpoints $t_c$ among computational periods $\sigma$.

comes one or more computational segments followed by checkpoints. Each checkpoint requires $t_c$ time (segments $\overline{EF}$, $\overline{GH}$, and $\overline{IJ}$) to be taken. Checkpoints are separated by an application computational period represented by $\sigma$ (segments $\overline{FG}$ and $\overline{HI}$). Ergo, $\sigma$ is the interval between checkpoints, *ipso facto* the checkpoint interval. Segment $\overline{JK}$ will be lost due to fault $F_y$. This work will be redone after the next recovery phase (not depicted in figure 1). As aforementioned, $T_c$ is the sum of all $t_c$, it represents the sum of segments $\overline{EF}$, $\overline{GH}$, and $\overline{IJ}$ in figure 1. Time spent on protection and recovery tasks is not useful application time, thus these tasks are considered to be overhead. This assumption leads us to the following equation:

$$Overhead = T_r + T_c \tag{1}$$

## 3.1 The Model for Serial Applications

As mentioned previously, $T_r$ is the sum of the fault detection latency ($t_d$), checkpoint loading from storage ($t_l$) and the fractional rework ($t_r$). It can also be seen in figure 1. As proved by Daly [2], it is accepted to assume interrupts occur halfway through the checkpoint interval. Thus, we consider the fractional rework ($t_r$) as half of the checkpoint interval ($\sigma$), which leads us to the following equation:

$$T_r = t_d + t_l + \sigma/2 \tag{2}$$

The same demonstration can be used in relation to the fault detection latency. However, there are many fault detection mechanisms that can be used. Because it is a user-defined variable we will leave this variable untouched.

To calculate $T_c$ the number of checkpoints ($Q$) should be defined. $Q$ represents the number of segments composed of checkpoint ($t_c$) and compute time ($\sigma$) that fits the period between faults ($\alpha$), excluding the recovery time. It is reflected in the equation below:

$$T_c = Q * t_c \tag{3}$$

$$Q = (\alpha - T_r)/(\sigma + t_c) \tag{4}$$

Using equations 1, 2, 3, and 4 and applying some algebraic operations the following overhead equation is obtained:

$$Overhead = (\sigma^2 + 2(\sigma t_d + \sigma t_l + \alpha t_c))/(2(\sigma + t_c)) \tag{5}$$

We can find the value of $\sigma_{opt}$ that minimises the fault tolerance overhead by deriving the overhead equation 5 with respect to $\sigma$ and setting the result to zero [1]. Considering the positive solution, this operation brings us to the following optimum checkpoint interval:

$$\sigma_{opt} = \sqrt{t_c^2 - 2t_c t_d - 2t_c t_l + 2\alpha t_c} - t_c \tag{6}$$

## 3.2 The Model for Parallel Applications

The message logging disturbance depends on the logging protocol used [11]. For the development of our model a pessimistic receiver-based message logging has been used. To better understand the deduction of the new model let us consider figure 2. This figure depicts the disturbances added by message logging operations and how failures impact on different application processes.

In general, receiver based logging doubles the time needed for message delivery because the data storage cannot be overlapped with message delivery as shown in figure 2. Modifying equations 2 and 3 to reflect the message logging overhead, leads us to the following equations for process *n*:

$$T_{r_n} = t_d + t_l + \sigma/2 + \Delta_{lr} \tag{7}$$

$$T_{c_n} = (Q * t_c) + \Delta_{lp} \tag{8}$$

In case of faults, only the faulty process needs to rollback and during its recovering phase there is no interaction with non-faulty processes [11]. However, once a process fails other processes that depend on the first could continue waiting for data from the first before continuing their execution. It means that processes have an intrinsic inter-dependent relationship. In this paper this relationship will be named *inter-processes dependency factor*. This factor affects the $T_r$ lowering its weight on the overhead equation. Rewriting the

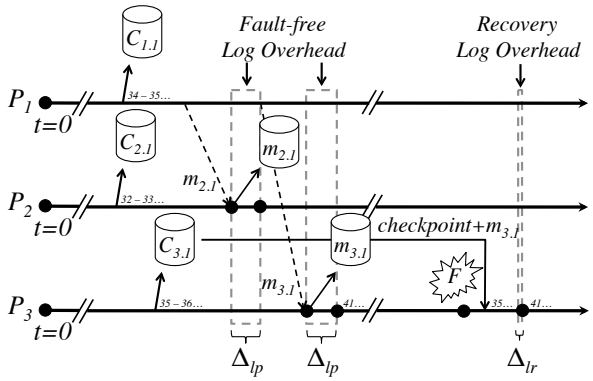



Fig. 2: Disturbances introduced by the receiver-based message logging protocol for protection $\Delta_{lp}$ and recovery $\Delta_{lr}$.

overhead equation 1 to consider this assertion, the following equation is produced:

$$Overhead = \phi(T_r) + T_c \quad (9)$$

where $\phi$ represents the inter-process dependency factor. Regarding this factor, small values represent less dependency between processes while the higher value (1) means that when one process fails all other processes continue waiting for it. Replacing equations 7 and 8 in equation 9 and after applying some algebraic operations, the following equation is obtained:

$$Overhead = [\phi\sigma^2 + \sigma(2\phi t_d + 2\phi t_l + \phi t_c + 2\phi\Delta_{lr} - t_c + 2\Delta_{lp}) + 2t_c(\phi t_d + \phi t_l + \phi\Delta_{lr} + \alpha - t_d - t_l - \Delta_{lr} + \Delta_{lp})] /(2\sigma + 2t_c) \quad (10)$$

We can find the value of $\sigma_{opt}$ that minimises the fault tolerance overhead for parallel applications by deriving equation 10 with respect to $\sigma$, and setting the result to zero[1]. Considering the positive solution, this operation brings us to the following optimum checkpoint interval:

$$\sigma_{opt} = \sqrt{\phi t_c(t_c + 2\alpha - 2t_d - 2t_l - 2\Delta_{lr})}/\phi - t_c \quad (11)$$

### 3.3 The Inter-Process Dependency Factor

All processes in the parallel application can fail. However, each process failure may impact other processes in a distinct way. This means that if process $n$ fails, one or more processes can hang waiting for the recovery of $n$ to be completed (*e.g.* considering a master/worker application, if the master process fails, all workers may wait for the recovery of the master). The following equation shows how to define the dependency factor using such analysis:

$$\phi = \frac{\sum_1^N P(n)}{N^2} \quad (12)$$

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

Based on the example above, let us assume an application running with 8 processes and written under a master/worker paradigm in which workers do not communicate among themselves and supposing a function $P(n)$ which considers the existence of communication as the only dependency between processes. If the master process fails all workers should wait for its recovery, then $P(master)$ is 8. If any worker fails just the master may wait for it, then $P(worker)$ is 2 for all 7 workers. In compliance with this assumption the dependency factor for this application is 0.34375.

Besides message logging modelling, the introduction of the inter-process dependency factor is a key difference between the previous models and ours. The introduction of this factor is crucial to the accuracy of the predicted checkpoint interval that minimises the fault tolerance overhead in parallel applications.

## 4. Experimental Evaluation

Hereunder, a comparison between models will be presented. The comparison was made using simulation and running real applications. The fault distribution is defined by the MTTI, and faults are displaced in time with a 100% of deviation calculated using the MT19937 PRNG algorithm [12]. Moreover, in this section we evaluate the checkpoint interval model for uncoordinated checkpointing.

To run experiments a 32 node cluster has been used. Each node is 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 are interconnected via two Gigabit Ethernet interfaces. RADIC/OMPI [13] has been used as a fault tolerant MPI library.

To inject faults a program has been designed. This program runs on a machine external to the cluster. According to the fault distribution, the program connects to the target node and kills the application process. The target machine is selected in a round-robin fashion. The killed process is recovered from the last checkpoint by the daemon used to launch the application process.

### 4.1 Models for Serial Applications

To compare models we have used a simple matrix multiplication algorithm. This experiment tries to verify the accuracy of the calculated checkpoint interval to minimise the overhead introduced by fault tolerance. For that, the matrix multiplication has been executed with different checkpoint intervals. The application overhead is compared with the predicted overhead of the models. The fault detection latency is virtually zero and the MTTI ($\alpha$) has been defined to be 100 seconds. Each experiment has been executed at least 16 times and values are the average of all data that fall in a 95% confidence interval.

Figure 3 depicts a comparison between overhead prediction of the models and a real execution. All models have calculated an optimum checkpoint interval between 9.75 and 10.3 seconds. All models have presented an overhead relative error smaller than 3%. Close to the optimum checkpoint interval the relative error is smaller than 2% for all models.

To evaluate the influence of the fault frequency a discrete event simulator has been used. Table 1 shows variables used to simulate a 500 days application as well as the values of the optimum checkpoint interval calculated by models for each scenario. The detection latency has been set to zero.

Figure 4(a) compares the simulated run time with the four models using a 24 hour MTTI. The results of the models are very close to the simulation. With regard to the predicted overhead, our model presents a relative error of 0.64% on values close to the calculated checkpoint interval (114.89

Table 1: Variables used to simulate the influence of MTTI on model accuracy and the optimum checkpoint interval.

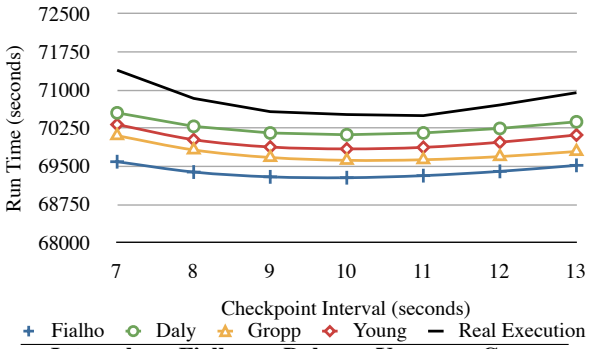
| **MTTI** | $t_c$ | $t_l$ | **Fialho** | **Daly** | **Gropp** | **Young** |
|---|---|---|---|---|---|---|
| *hours* | *minutes* | *minutes* | *minutes* | *minutes* | *minutes* | *minutes* |
| 24 | 5 | 5 | 114.89 | 115.00 | 120.00 | 120.00 |
| 6 | 5 | 5 | 54.79 | 55.00 | 60.00 | 60.00 |

minutes). Daly's model presented an error smaller than 0.2% for all checkpoint intervals simulated.

Figure 4(b) demonstrates that Young's model cannot predict the overhead with less than 15% error when the MTTI is smaller than 6 hours and the time needed to take or load a checkpoint is 5 minutes. However, its calculated checkpoint interval is still close to the optimum. The smaller simulated run time is for a 55 minute checkpoint interval. At this point our model, Gropp's, and Daly's present a relative error of 2.54%, 1.23%, and 0.67%, respectively.

## 4.2 Analysing the Results of the Checkpoint Interval Model for Parallel Applications

To evaluate the results of the checkpoint interval model for parallel application two sets of experiments have been designed: *1)* one analyses the effectiveness of the inter-process dependency factor, and *2)* verifies the correctness of the message logging modelling. To show that the using




| **Interval** | **Fialho** | **Daly** | **Young** | **Gropp** |
|---|---|---|---|---|
| 7 | 2.6% | 1.2% | 1.5% | 1.8% |
| 8 | 2.1% | 0.8% | 1.2% | 1.5% |
| 9 | 1.8% | 0.6% | 1.0% | 1.3% |
| 10 | 1.8% | 0.6% | 1.0% | 1.3% |
| 11 | 1.7% | 0.5% | 0.9% | 1.2% |
| 12 | 1.9% | 0.7% | 1.0% | 1.5% |
| 13 | 2.1% | 0.8% | 1.2% | 1.7% |
| **Calculated** | 9.752s | 9.766s | 10.300s | 10.300s |

Fig. 3: Comparison of real execution and overhead prediction of the models for $\alpha = 100$, $t_c = 0.530$, $t_l = 0.505$, $t_d = 0$, values in average. Application runs in 62,830 seconds without fault tolerance and in absence of faults. The table shows the relative error of predicted overhead of the models for each checkpoint interval used. The last line presents the optimum checkpoint interval estimation of the models.

of current models is inappropriate in this scenario, the values achieved with other models will be included on the following experiments[1].

To run experiments presented in this section the MTTI ($\alpha$) has been set to 100 seconds. The RADIC/OMPI library has been configured to send a heartbeat every 1 second. Thus, the fault detection latency ($t_d$) is 0.5 seconds. As this library performs receiver-base message logging during the recovery phase messages are already available in the log. Thus, the time needed to process the message log ($\Delta_{lr}$) tends to be unappreciable because there is no message replaying [14].

### 4.2.1 Inter-Process Dependency Factor Effectiveness

To assure that the inter-process dependency factor is the only variable which changes between executions a synthetic application has been designed. The message logging interference and the time needed to take and load a checkpoint are quite similar for the same number of process *per* node.

[1]Other models had been designed to be used with serial applications. To compare these models with ours may be unfair while running applications protected by uncoordinated checkpoint combined with message logging. However, it is important to show de benefits of using models specifically designed for uncoordinated checkpointing.

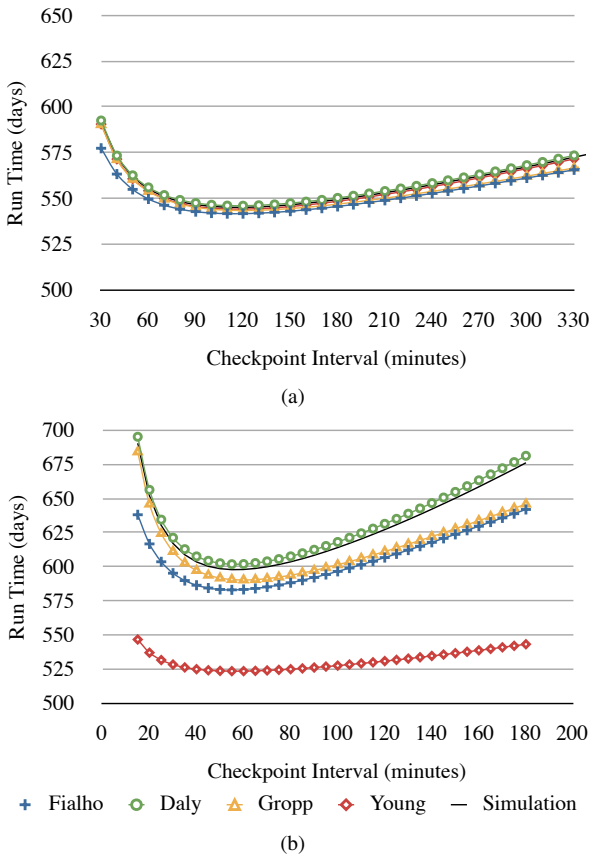



Fig. 4: Comparison of simulation results for values depicted in table 1 using a 24 hours (a) and 6 hours (b) MTTI.

Table 2: Relevant characteristics of the synthetic application used to verify the effectiveness of the inter-process dependency factor. For each model, the first column shows the optimum checkpoint interval calculated and the second column shows the predicted overhead error. $\alpha = 100$ and $t_d = 0.5$. Values are expressed in seconds.

| # of | | | | | | Fialho | | Daly | | Gropp | | Young | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Processes | $\phi$ | $t_c$ | $t_l$ | $\Delta_{lp}$ | $\Delta_{lr}$ | $\sigma$ | Error | $\sigma$ | Error | $\sigma$ | Error | $\sigma$ | Error |
| 4 | 1.0000 | 1.630 | 1.643 | 2.771 | 0.000 | 16.30 | 3.67% | 16.42 | 2.8% | 18.05 | 4.7% | 18.05 | 3.4% |
| 9 | 0.5556 | 1.622 | 1.596 | 2.763 | 0.000 | 22.39 | 3.15% | 16.39 | 1.3% | 18.01 | 0.6% | 18.01 | 0.0% |
| 16 | 0.3125 | 1.691 | 1.610 | 2.765 | 0.000 | 31.00 | 2.21% | 16.70 | 5.3% | 18.39 | 3.3% | 18.39 | 3.3% |
| 25 | 0.2000 | 1.650 | 1.634 | 2.779 | 0.000 | 38.70 | 2.11% | 16.52 | 7.0% | 18.17 | 5.1% | 18.17 | 4.8% |
| 16 (4×4) | 0.3125 | 4.954 | 5.131 | 4.188 | 0.000 | 50.46 | 3.08% | 26.52 | 12.1% | 31.48 | 7.0% | 31.48 | 6.4% |
| 36 (4×9) | 0.1389 | 5.032 | 5.199 | 4.160 | 0.000 | 78.73 | 2.65% | 26.69 | 17.4% | 31.72 | 12.2% | 31.72 | 10.4% |
| 64 (4×16) | 0.0781 | 4.981 | 5.287 | 4.399 | 0.000 | 106.06 | 1.88% | 26.58 | 20.4% | 31.56 | 15.3% | 31.56 | 12.7% |
| 100 (4×25) | 0.0500 | 5.284 | 5.330 | 4.328 | 0.000 | 137.76 | 1.63% | 27.22 | 22.9% | 32.51 | 17.6% | 32.51 | 14.6% |

The synthetic application has been programmed using the SPMD paradigm. A computing and a communication phase compose each process. The computing phase is represented by a 2000×2000 matrix multiplication and during the communication phase processes communicate to the right and lower neighbours. These phases are repeated until a defined amount of work has been done. The computing and communication load are the same for all executions. Thus, the interference caused by the message logging is the same in all experiments regardless of the number of processes used. However, the value of the inter-process dependency factor changes accordingly to the number of processes.

Besides other variables, table 2 depicts the value of $\phi$ for all executions. Values of message logging operation ($\Delta_{lp}$ and $\Delta_{lr}$), checkpoint taking ($t_c$), and checkpoint loading ($t_l$) are averages of all measurements done during application execution. The value of the checkpoint interval has been previously calculated based on the applications characteristics and is used to configure the RADIC/OMPI library.

As shown in figure 5, as the number of processes increases (or the value of $\phi$ decreases) so does the accuracy of our model. On the execution with 4 processes the value of $\phi$ is 1. In this case all models perform similarly. However, as the number of processes increases other models depicts a loss of accuracy. Analysing figure 5(b) it is easy to conclude that previous models cannot be used with parallel applications protected by uncoordinated checkpoints combined with message logging. Especially with a high number of processes.

### 4.2.2 Correctness of the Message Logging Modelling

These experiments use the LU application from the NAS Parallel Benchmarks [15] running with 8 processes, one *per* node. LU has been executed using class B and C. Table 3 depicts relevant characteristics of LU. $\Delta$ values reflect the average measurements done during application execution. The number of iterations of LU class B and C has been modified to 300,000 and 37,500 respectively. The LU application presents a $\phi$ value equal to 0.5625 for 8 processes.

As shown in figure 6 our model performs better than any other. Because other models do not consider the message logging time they present an overhead prediction relative error greater than 20% for LU class B. It means that these models are not useful to predict the overhead for parallel applications using uncoordinated checkpointing combined with pessimist receiver-based message logging. Nevertheless, our model presents a modest overhead prediction error for both class B and C of the NAS LU. Notice that from figure

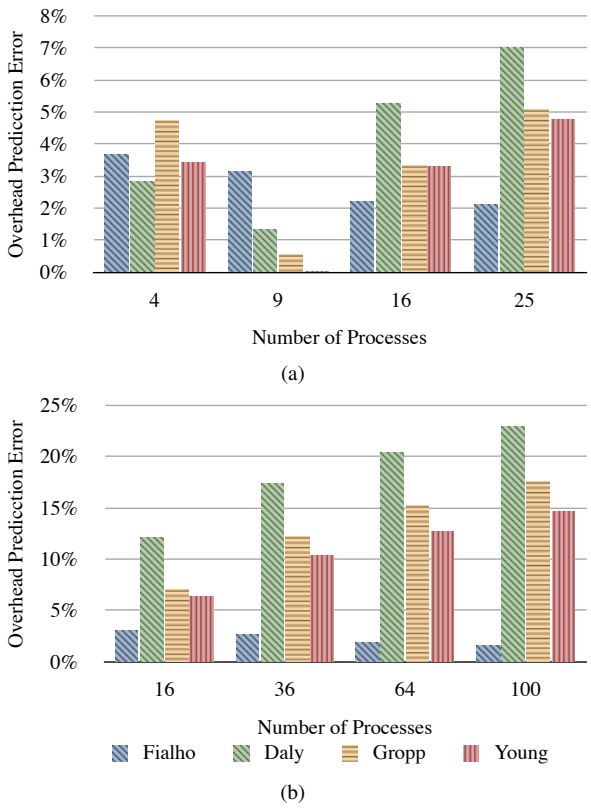



Fig. 5: Overhead prediction error for a synthetic application running with (a) 4, 9, 16, and 25 processes, 1 *per* node, and (b) 16, 36, 64, and 100 processes, 4 *per* node. Values of variables are depicted in table 2, $\alpha = 100$, and $t_d = 0.5$.

Table 3: Characteristics of the NAS LU class B and C. For each model, the first column shows the optimum checkpoint interval and the second the predicted overhead error. $\alpha = 100$, $t_d = 0.5$, $\phi = 0.5625$. Values are expressed in seconds.

| LU | | | | | Fialho | | Daly | | Gropp | | Young | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Class | $t_c$ | $t_l$ | $\Delta_{lp}$ | $\Delta_{lr}$ | $\sigma$ | Error | $\sigma$ | Error | $\sigma$ | Error | $\sigma$ | Error |
| B | 0.605 | 0.559 | 38.257 | 0.005 | 10.353 | 3.0% | 10.395 | 25.1% | 11.000 | 25.9% | 11.000 | 22.0% |
| C | 2.057 | 2.102 | 13.961 | 0.007 | 18.065 | 0.5% | 18.065 | 5.6% | 20.283 | 8.0% | 20.283 | 5.8% |

6(a) to figure 6(b) other models presented a decrease in the overhead prediction error while the opposite occurs with our model. This occurs because the ratio between compute and communication changes reducing the interference of message logging.

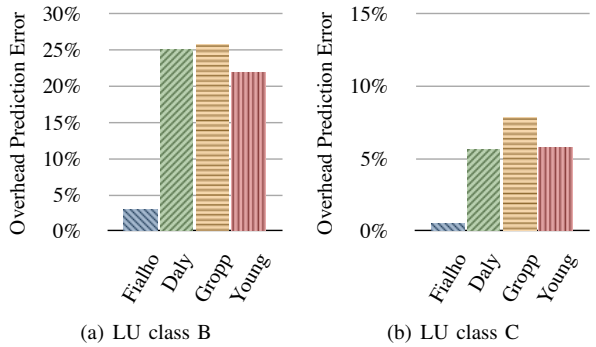
# 5. Conclusions

This paper has presented a novel model to calculate the checkpoint interval to minimise the overhead introduced by fault tolerance on parallel applications.

We have shown that our serial model presents a difference of less than 1.2% from other models on average and the execution time prediction error is smaller than 3% in comparison with a real application execution. With regard to our parallel model, it presents an overhead prediction error smaller than 5% for the applications tested. Furthermore, we have demonstrated that our models perform better when the number of processes increases and there is less dependency between processes.

## 5.1 Model Utilisation and Future Work

To reduce the number of variables in the checkpoint interval model that minimises the fault tolerance overhead, an analysis of variable sensitivity can be conducted. However, we consider our model short enough to be incorporated into any fault tolerant MPI library. This permits the dynamic definition of the checkpoint interval based on measurements of the time needed to perform fault tolerance procedures. Using this approach, users can achieve better results because the checkpoint interval value reflects the application characteristics at a given moment.




(a) LU class B (b) LU class C

Fig. 6: Model overhead prediction relative error for LU class B and C. Values of variables are depicted in table 3.

Another important issue is to prove that the parallel model is suitable for libraries that implement uncoordinated checkpointing combined with sender-based message logging.

# References


[1] 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.

[2] 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.

[3] 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.

[4] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 1—18, 2002.

[5] A. Duarte, D. Rexachs, and E. Luque, "Increasing the cluster availability using RADIC," *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 2006.

[6] J. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530—531, 1974.

[7] Y. Lin, B. Preiss, W. M. Loucks, and E. D. Lazawska, "Selecting the Checkpoint Interval in Time Warp Simulation," *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 3—10, 1993.

[8] E. Gelenbe, D. Finkel, and S. Tripathi, "Availability of a distributed computer system with failures," *Acta Informatica*, vol. 23, no. 6, pp. 643—655, 1986.

[9] K. Wong and M. Franklin, "Distributed Computing Systems and Checkpointing," *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pp. 224—233, 1993.

[10] A. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello, "Coordinated checkpoint versus message log for fault tolerant MPI," *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, pp. 242—250, 2003.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.