Topics in Parallel and Distributed Computing: Introducing Algorithms, Programming, and Performance within Undergraduate Curricula[*][†][‡]

Chapter 6 – Scheduling for fault-tolerance: an introduction

Guillaume Aupy[1] and Yves Robert[2]

[1]Inria & Labri, Univ. of Bordeaux, France, guillaume.aupy@inria.fr
[2]ENS Lyon, France & Univ. Tennessee Knoxville, USA, yves.robert@ens-lyon.fr

[*]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms, Springer International Publishing, 2018, ISBN : 978-3-319-93108-1, Pages: 337.

[†]Free preprint version of this book: `https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cder_book_2`

[‡]Free preprint version of volume one: `https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book`

Abstract

Parallel execution time is expected to decrease as the number of processors increases. We show in this chapter that this is not as easy as it seems, even for perfectly parallel applications. In particular, processors are subject to faults. The more processors are available, the more likely faults will strike during execution. The main strategy to cope with faults in High Performance Computing is checkpointing. We introduce the reader to this approach, and explain how to determine the optimal checkpointing period through scheduling techniques. We also detail how to combine checkpointing with prediction and with replication.

Relevant core courses: Data Structures and Algorithms, Probabilities

Relevant PDC topics: Scalability in algorithms and architectures; Fault tolerance; Time

Context for use: Mid under-graduate curriculum. Having a minimal background in probabilities is better. The appendices are for students who are more advanced.

Learning outcomes: Comprehend that having access to more processors does not guarantee faster execution — introduce the notion of faults and easy algorithms to cope with faults

6.1 Introduction

In this chapter, we present scheduling algorithms to cope with faults on large-scale parallel platforms. We study *checkpointing* and show how to derive the optimal checkpointing period. Then we explain how to combine checkpointing with *fault prediction*, and discuss how the optimal period is modified when this combination is used. And finally we follow the very same approach for the combination of checkpointing with *replication*. But wait. First, we have to help Alice out: she is having trouble with her laptop while writing her thesis.

6.2 Checkpointing on a single processor

6.2.1 Alice needs help

The most natural fault-tolerance technique when considering a fault-prone environment is to save your work periodically. This is what we (should) do in every-day's life. Alice is doing a very long and fastidious work: she is writing her PhD thesis, using an unreliable resource, namely a four-year-old laptop. Because she is afraid of losing her precious work if the laptop crashes, she regularly saves her work on an external disk.

At first, because she knew that her laptop could not be trusted, Alice decided to save her work on the external disk every three hours. Writing her file to disk takes approximatively three minutes. On the mid-afternoon of day 3, Alice's laptop crashed, she had to reboot it, and as a consequence she lost the last hour and a half of her work! Indeed, the crash happened right ninety minutes after her last saving on the external disk; she could have lost much more if, say, the crash had happened only ten minutes before the next saving. Piqued, Alice decided that from now on, she would save her work on the external disk more frequently, every half hour of work instead of every three hours. But after three additional days of work without further problem, she compared what she did during days 1, 2 and 3, and during days 4, 5 and 6. She noticed that she did less work on days 4, 5 and 6 than on days 1, 2 and 3 (even though she lost ninety minutes of work on the third day). Alice is puzzled now: what is the best frequency to save her work?

The technique of saving intermediate work is called *checkpointing*. Because Alice works for a constant amount of time between two checkpoints, her technique is called *periodic checkpointing*. In the following, we explain why she did more work during the three first days, and how she could find the best period between each checkpoint.

6.2.2 Modeling the occurrence of faults

Computing environments, such as Alice's laptop, are prone to faults. The first question is to quantify the rate or frequency at which these faults strike. To that purpose, one uses probability distributions, and more specifically, Exponential probability distributions. The definition of $Exp(\lambda)$, the Exponential distribution law of parameter λ, goes as follows:

- The probability density function is $f(t) = \lambda e^{-\lambda t} dt$ for $t \geq 0$;
- The cumulative distribution function is $F(t) = 1 - e^{-\lambda t}$ for $t \geq 0$;
- The mean is $\mu = \frac{1}{\lambda}$.

Consider a process executing in a fault-prone environment. The time-steps at which fault strike are non-deterministic, meaning that they vary from one execution to another. To model this, we use IID (Independent and Identically Distributed) random variables $X_1, X_2, X_3, \ldots$. Here X_1 is the delay until the first fault, X_2 is the delay between the first and second fault, X_3 is the delay between the second and third fault, and so on. All these random variables obey the same probability distribution $Exp(\lambda)$. We write $X_i \sim Exp(\lambda)$ to express that X_i obeys an Exponential distribution $Exp(\lambda)$.

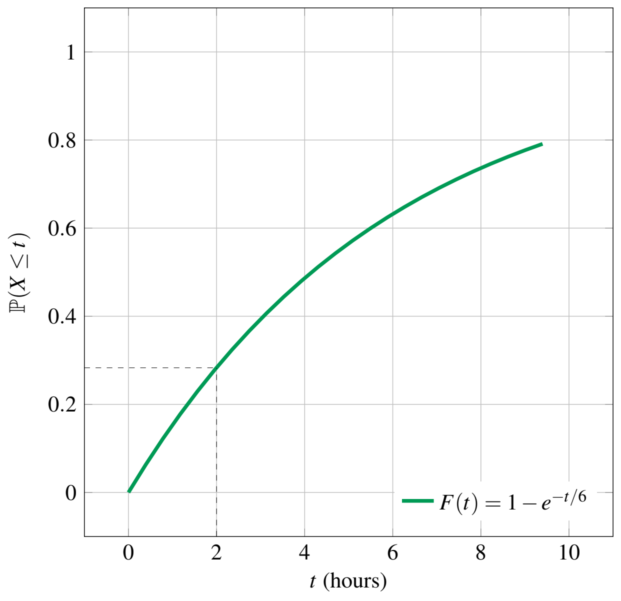


Figure 6.1: Assuming $\lambda = 1/6$ (counting time in hours), the probability that a failure will strike within two hours is $F(2) = \mathbb{P}(X < 2) = 1 - e^{-2/6} \approx 0.283$.

Each random variable X_i has the same cumulative distribution function $F(t) = 1 - e^{-\lambda t}$: by definition, $F(t)$ gives the probability of the event $X_i < t$. In other words, $F(t) = \mathbb{P}(X_i < t)$ is the probability of having the next fault strike after a delay not larger than t. See Figure 6.1 for the cumulative distribution function of $Exp(\frac{1}{6\times 3{,}600})$. For simplicity, time is counted in hours in the figure, so that $\lambda = \frac{1}{6}$ and $\mu = 6$: in average, a fault will strike every 6 hours. Reading the plot, we have $F(2) \approx 0.283$, which means that there is a 28% chance of having the next fault strike within 2 hours.

We already observed that each random variable X_i has the same mean $\mathbb{E}(X_i) = \mu$. In average, a fault will strike every μ seconds. This is why μ is called the MTBF of the process, where MTBF stands for *Mean Time Between Faults*. The MTBF is a key parameter to Alice's problem. One can show (see Appendix 3 for a proof) that the expected number of faults $N_{\text{faults}}(T)$ that will strike during T seconds is such that

$$\lim_{T\to\infty} \frac{N_{\text{faults}}(T)}{T} = \frac{1}{\mu} \tag{6.1}$$

Why are Exponential distribution laws so important? This is because of their *memoryless* property, which writes: if $X \sim Exp(\lambda)$, then $\mathbb{P}(X \geq t + s \,|\, X \geq s) = \mathbb{P}(X \geq t)$ for all $t, s \geq 0$. This equation means that at any instant, the delay until the next fault does not depend upon the time that has elapsed since the last fault. The memoryless property is equivalent to saying that the fault rate is constant. The fault rate at time t, $\textsc{rate}(t)$, is defined as the (instantaneous) rate of fault for the survivors to time t, during the next instant of time:

$$\textsc{rate}(t) = \lim_{\Delta\to 0} \frac{F(t+\Delta) - F(t)}{\Delta} \times \frac{1}{1-F(t)} = \frac{f(t)}{1-F(t)} = \lambda = \frac{1}{\mu}$$

The fault rate is sometimes called a *conditional* fault rate since the denominator $1 - F(t)$ is the probability that no fault has occurred until time t, hence converts the expression into a conditional rate, given survival past time t.

We have discussed Exponential laws above, but other probability laws could be used. For instance, it may not be realistic to assume that the fault rate is constant: indeed, computers, like washing machines, suffer from a phenomenon called *infant mortality*: the probability of fault is higher in the first weeks than later on. In other words, the fault rate is not constant but instead decreasing with time. Well, this is true up to a certain point, where another phenomenon called *ageing* takes over: your computer, like your car, becomes more and more subject to faults after a certain amount of time: then the fault rate increases! However, after a few weeks of service and before ageing, there are a few years during which it is a good approximation to consider that the fault rate is constant, and therefore to use an Exponential law $Exp(\lambda)$ to model the occurrence of faults. The key parameter is the MTBF $\mu = \frac{1}{\lambda}$.

6.2.3 Problem statement

We start by stating the problem formally. Let $\textsc{Time}_{\text{base}}$ be the base time of the work that needs to be done, without any overhead (neither checkpoints nor faults). Assume that Alice's computer is subject to faults with a mean time between faults (MTBF) equal to μ.

The time to take a checkpoint is C seconds ($C = 180$ in the example). We say that the period is T seconds when a checkpoint is done each time Alice has completed $T - C$ seconds of work. When a fault occurs, the time between the last checkpoint and the fault is lost. After the fault, there is a *downtime* of seconds to account for the temporary unavailability (for example Alice's laptop is restarted, or the mouse is changed, or she now needs to use her brother Bob's laptop). Finally, in order to be able to resume the work, the content of the last checkpoint needs to be *recovered* which takes a time of R seconds (the external disk is connected and the checkpoint file is read). The sum of the time lost after the fault, of the downtime and of the recovery time is denoted T_{lost}. All these notations are depicted in Figure 6.2.

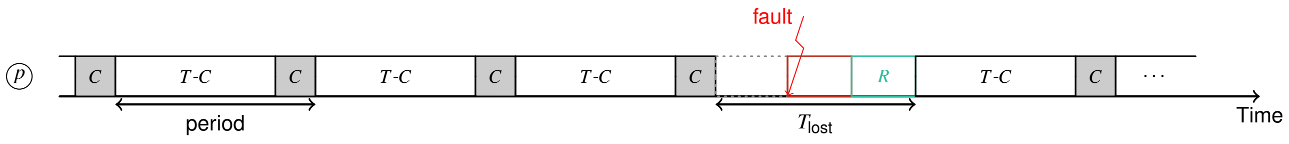


Figure 6.2: An execution.

6.2.4 Example

The difficulty of the problem is to trade-off between the time spent checkpointing, and the time lost in case of a fault. Consider an application such that $\textsc{Time}_{\text{base}} = 30$ minutes, and assume a checkpoint time of $C = 3$ minutes, a downtime of $= 1$ minute and a recovery time of $R = 3$ minutes.

We consider the following combinations:

Strategies

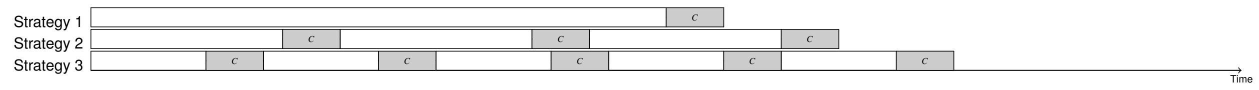
1. Only one checkpoint at the end of the execution;
2. Three checkpoints during the execution, after every 10 minutes of work;
3. Five checkpoints during the execution, after every 6 minutes of work.

Scenarios

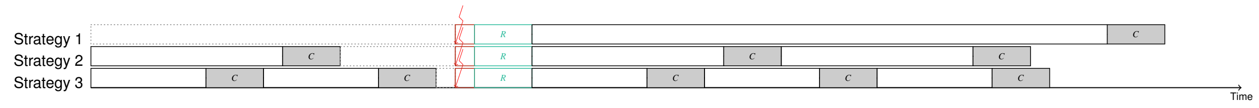
(a) A large time between faults (in this example, no fault during the execution);

(b) A medium time between faults (only one fault at the 19th minute during the first hour);

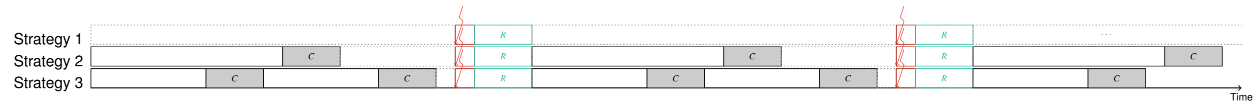
(c) A small time between faults (one fault at the 19th, 42nd, 62nd minutes).



(a) Large MTBF: there are no or very few faults. Checkpointing is too expensive. The first strategy wins.



(b) Medium MTBF: there are more faults. It is good to checkpoint, but not too frequently, because of the corresponding overhead. The second strategy wins.



(c) Small MTBF: there are many faults. The cost of the checkpoints is paid off because the time lost due to faults is dramatically reduced. The third strategy wins.

Figure 6.3: The three strategies obtain different results depending upon the MTBF.

In Figure 6.3, we picture the execution of the application for the three different strategies, under the three different scenarios. This example shows that the lower the time between faults, the higher the frequency of checkpoints should be. However, the checkpointing strategy with the smallest period is not always the best one: sometimes, there are not enough faults to pay off the overhead of frequent checkpoints.

6.2.5 Solution

Let $\textsc{Time}_{\text{final}}(T)$ be the expectation of the total execution time of an application of size $\textsc{Time}_{\text{base}}$ with a checkpointing period of size T. The optimization problem is to find the period T minimizing

$\text{TIME}_{\text{final}}(T)$. However, for the sake of convenience, we rather aim at minimizing

$$\text{WASTE}(T) = \frac{\text{TIME}_{\text{final}}(T) - \text{TIME}_{\text{base}}}{\text{TIME}_{\text{final}}(T)}.$$

This objective is called the *waste* as it corresponds to the fraction of the execution time that does not contribute to the progress of the application (the time *wasted*). Of course minimizing the waste WASTE is equivalent to minimizing the total time $\text{TIME}_{\text{final}}$, because we have

$$(1 - \text{WASTE}(T))\ \text{TIME}_{\text{final}}(T) = \text{TIME}_{\text{base}},$$

but using the waste is more convenient. The waste varies between 0 and 1. When the waste is close to 0, it means that $\text{TIME}_{\text{final}}(T)$ is very close to $\text{TIME}_{\text{base}}$ (which is good), whereas, if the waste is close to 1, it means that $\text{TIME}_{\text{final}}(T)$ is very large compared to $\text{TIME}_{\text{base}}$ (which is bad).

First source of waste. Consider a *fault-free* execution of the application with periodic checkpointing. By definition, during each period of length T we take a checkpoint, which lasts for C time units, and only $T - C$ units of work are executed. Let TIME_{FF} be the execution time of the application in this setting. The fault-free execution time TIME_{FF} is equal to the time needed to execute the whole application, $\text{TIME}_{\text{base}}$, plus the time taken by the checkpoints:

$$\text{TIME}_{\text{FF}} = \text{TIME}_{\text{base}} + N_{\text{ckpt}}C,$$

where N_{ckpt} is the number of checkpoints taken. Additionally, we have

$$N_{\text{ckpt}} = \left\lceil \frac{\text{TIME}_{\text{base}}}{T - C} \right\rceil \approx \frac{\text{TIME}_{\text{base}}}{T - C}.$$

To discard the ceiling function, we assume that the execution time $\text{TIME}_{\text{base}}$ is large with respect to the period or, equivalently, that there are many periods during the execution. Plugging back the (approximated) value $N_{\text{ckpt}} = \frac{\text{TIME}_{\text{base}}}{T-C}$, we derive that

$$\text{TIME}_{\text{FF}} = \frac{T}{T - C}\text{TIME}_{\text{base}}. \tag{6.2}$$

Similarly to the WASTE objective, the waste due to checkpointing in a fault-free execution, WASTE_{FF}, is defined as the fraction of the fault-free execution time that does not contribute to the progress of the application:

$$\text{WASTE}_{\text{FF}} = \frac{\text{TIME}_{\text{FF}} - \text{TIME}_{\text{base}}}{\text{TIME}_{\text{FF}}} \Leftrightarrow \big(1 - \text{WASTE}_{\text{FF}}\big)\text{TIME}_{\text{FF}} = \text{TIME}_{\text{base}}. \tag{6.3}$$

Combining Equations (6.2) and (6.3), we get:

$$\text{WASTE}_{\text{FF}} = \frac{C}{T}. \tag{6.4}$$

This result is quite intuitive: every T seconds, we waste C for checkpointing. This calls for a very large period in a fault-free execution (even an infinite period, meaning no checkpoint at all). However, a large period also implies that a large amount of work is lost whenever a fault strikes, as we discuss now.

Second source of waste. Consider the entire execution (with faults) of the application. Let $\textsc{Time}_{\text{final}}$ denote the expected execution time of the application in the presence of faults. This execution time can be divided into two parts: (i) the execution of chunks of work of size $T-C$ followed by their checkpoint; and (ii) the time lost due to the faults. This decomposition is illustrated in Figure 6.4. The first part of the execution time is equal to $\textsc{Time}_{\text{FF}}$. Let N_{faults} be the number of faults occurring during the execution, and let T_{lost} be the average time lost per fault. Then,

$$\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}} + N_{\text{faults}} T_{\text{lost}}.$$

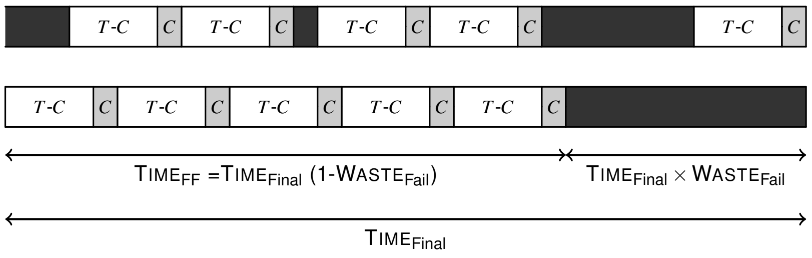


Figure 6.4: An execution (top), and its re-ordering (bottom), to illustrate both sources of waste. Blackened intervals correspond to time lost due to faults: downtimes, recoveries, and re-execution of work that has been lost.

In average, during a time $\textsc{Time}_{\text{final}}$, $N_{\text{faults}} = \frac{\textsc{Time}_{\text{final}}}{\mu}$ faults happen (recall Equation (6.1)). We need to estimate T_{lost} (see Figure 6.2). A natural estimation for the moment when the fault strikes in the period is $\frac{T}{2}$. Intuitively, faults strike anywhere in the period, hence in average they strike in the middle of the period. Daly [6] give the proof of this result for Exponential distribution laws. We conclude that $T_{\text{lost}} = \frac{T}{2} + D + R$, because after each fault there is a downtime and a recovery. This leads to:

$$\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}} + \frac{\textsc{Time}_{\text{final}}}{\mu}\left(+R+\frac{T}{2}\right).$$

Let $\textsc{Waste}_{\text{fault}}$ be the fraction of the total execution time that is lost because of faults:

$$\textsc{Waste}_{\text{fault}} = \frac{\textsc{Time}_{\text{final}} - \textsc{Time}_{\text{FF}}}{\textsc{Time}_{\text{final}}} \Leftrightarrow (1-\textsc{Waste}_{\text{fault}})\,\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}}$$

We derive:

$$\textsc{Waste}_{\text{fault}} = \frac{1}{\mu}\left(+R+\frac{T}{2}\right). \tag{6.5}$$

Equations (6.4) and (6.5) show that each source of waste calls for a different period: a large period for $\textsc{Waste}_{\text{FF}}$, as already discussed, but a small period for $\textsc{Waste}_{\text{fault}}$, to decrease the amount of work to re-execute after each fault. Clearly, a trade-off is to be found. Here is how. By definition we have

$$\begin{aligned}\textsc{Waste} &= 1 - \frac{\textsc{Time}_{\text{base}}}{\textsc{Time}_{\text{final}}} \\ &= 1 - \frac{\textsc{Time}_{\text{base}}}{\textsc{Time}_{\text{FF}}} \frac{\textsc{Time}_{\text{FF}}}{\textsc{Time}_{\text{final}}} \\ &= 1 - (1 - \textsc{Waste}_{\text{FF}})(1 - \textsc{Waste}_{\text{fault}}).\end{aligned}$$

Altogether, we derive the final result:

$$\begin{aligned}\textsc{Waste} &= \textsc{Waste}_{\text{FF}} + \textsc{Waste}_{\text{fault}} - \textsc{Waste}_{\text{FF}}\textsc{Waste}_{\text{fault}} && (6.6)\\ &= \frac{C}{T} + \left(1 - \frac{C}{T}\right)\frac{1}{\mu}\left(+R + \frac{T}{2}\right). && (6.7)\end{aligned}$$

In Figure 6.5, we plot $\textsc{Waste}$ as a function of the period T for a set of parameters.

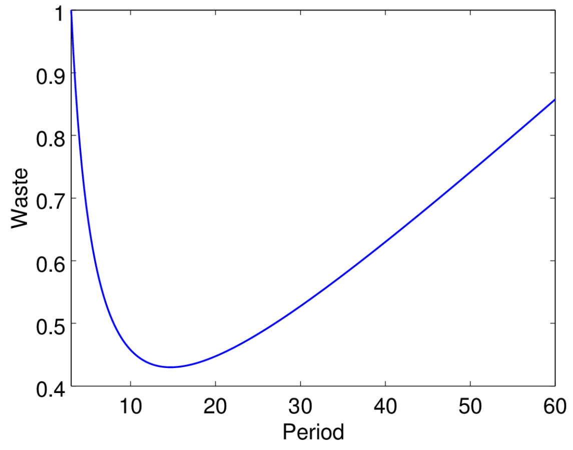


Figure 6.5: Waste as a function of the period T, for $C = 3, = 1, R = 3$ and $\mu = 40$. $T_{\text{FO}} \approx 14.7$. Shorter periods increase $\textsc{Waste}_{\text{FF}}$ too much. Longer periods increase $\textsc{Waste}_{\text{fault}}$ too much. T_{FO} achieves the best trade-off between both sources of waste.

We obtain $\textsc{Waste} = \frac{u}{T} + v + wT$, where $u = C\left(1 - \frac{+R}{\mu}\right)$, $v = \frac{+R - C/2}{\mu}$, and $w = \frac{1}{2\mu}$. It is easy to see that $\textsc{Waste}$ is minimized for $T = \sqrt{\frac{u}{w}}$. The First-Order (FO) formula for the optimal period is thus:

$$T_{\text{FO}} = \sqrt{2(\mu - (+R))C}. \tag{6.8}$$

and the optimal waste is $\textsc{Waste}_{\text{FO}} = 2\sqrt{uw} + v$, therefore

$$\textsc{Waste}_{\text{FO}} = \sqrt{\frac{2C}{\mu}(1 - \frac{+R}{\mu})} + \frac{+R - C/2}{\mu}. \tag{6.9}$$

Finally, we show in Appendix 1 why the computation above is a first order approximation.

In 1974, Young [18] obtained a different formula, namely $T_{\mathrm{FO}} = \sqrt{2\mu C} + C$. Thirty years later, Daly [6] refined Young's formula and obtained $T_{\mathrm{FO}} = \sqrt{2(\mu + R)C} + C$. Equation (6.8) is yet another variant of the formula, which we have obtained through the computation of the waste. There is no mystery, though. None of the three formulas is correct! They represent different first-order approximations, which collapse into the beautiful formula $T_{\mathrm{FO}} = \sqrt{2\mu C}$ when μ is large in front of the resilience parameters , C and R. This latter condition is the key to the accuracy of the approximation (see Appendix 1). Let us formulate our result as a theorem:

The optimal checkpointing period is $T_{\mathrm{FO}} = \sqrt{2\mu C} + o(\sqrt{\mu})$ and the corresponding waste is $\textsc{Waste}_{\mathrm{FO}} = \sqrt{\frac{2C}{\mu}} + o(\sqrt{\frac{1}{\mu}})$.

Theorem 6.2.5 has a wide range of applications. We discuss three of them in the following sections.

6.3 Checkpointing on a parallel platform

In this section, we deal with the problem of checkpointing a parallel application. We show how to reduce the optimization problem with p processors to the previous problem with only one processor. Most high performance applications are *tightly-coupled* applications, where each processor is frequently sending messages to, and receiving messages from the other processors. This implies that the execution can progress only when all processors are up and running. This also implies that when a fault strikes one processor, the whole application must be restarted from the last checkpoint. Indeed, even though the other processors are still alive, they will very soon need some information from the faulty processor. But to catch up, the faulty processor must re-execute the work that it has lost, during which it had received messages from the other processors. But these messages are no longer available. This is why all processors have to recover from the last checkpoint and re-execute the work in parallel.

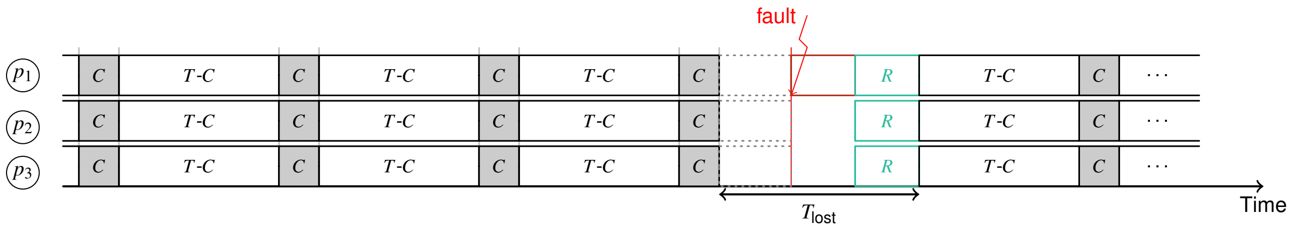
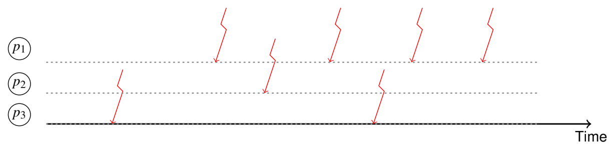
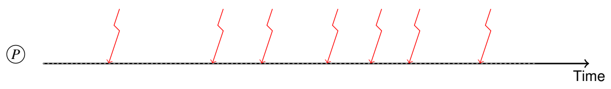


Figure 6.6: Behavior for a tightly coupled application.

Let us recap. Each time a fault strikes somewhere on the platform, the application stops, all processors perform a downtime and a recovery, and they re-execute the work during a time T_{lost}. This sounds familiar. We can see the whole platform as a single *super-processor*, very powerful (its speed is p times that of individual processors) but also very prone to faults: all the faults strike this poor super-processor! See Figure 6.7 for an illustration.

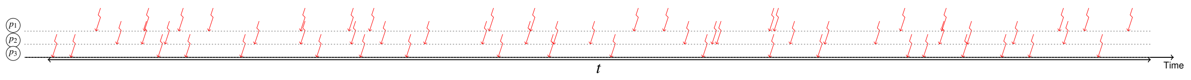


(a) Three faulty processors...

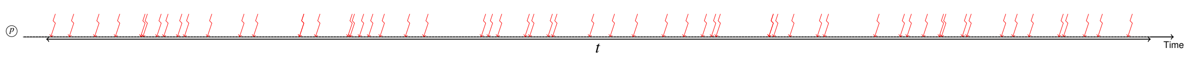


(b) ...make up for an equivalent even more faulty processor!

Figure 6.7: Platform model: the super-processor replaces $p = 3$ processors.



(a) If three processors have around 20 faults during a time t ($\mu_{\text{ind}} = \frac{t}{20}$)...



(b) ...during the same time, the equivalent processor has around 60 faults ($\mu = \frac{t}{60}$)

Figure 6.8: Intuition of the proof of Proposition 6.3.

We can apply Theorem 6.2.5 to the super-processor and determine the optimal checkpointing period as $T_{\text{FO}} = \sqrt{2\mu C} + o(\sqrt{\mu})$, where μ now is the MTBF of the super-processor. How can we compute this MTBF? Have a look at Figure 6.8. We see that the super-processor is hit by faults p times more frequently than the individual processors. We should then conclude that its MTBF is p times smaller than that of each processor. We state this result formally:

Consider a platform with p identical processors, each with MTBF μ_{ind}. Let μ be the MTBF of the platform. Then

$$\mu = \frac{\mu_{\text{ind}}}{p} \tag{6.10}$$

If the inter-arrival times of the faults on each individual processor are IID random variables (recall that IID means Independent and Identically Distributed) with probability distribution $Exp(\lambda)$ (where $\lambda = \frac{1}{\mu_{\text{ind}}}$), then the inter-arrival times of the faults on the super-processor are IID random variables with probability distribution $Exp(p\lambda)$, which will prove the result.

The arrival time of the first fault on the super-processor is a random variable $Y_1 \sim Exp(\lambda)$. This is because Y_1 is the minimum of $X_1^{(1)}, X_1^{(2)} \dots, X_1^{(p)}$, where $X_1^{(i)}$ is the arrival time of the first fault on processor P_i. But $X_1^{(i)} \sim Exp(\lambda)$ for all i, and the minimum of p random variables following an Exponential distribution $Exp(\lambda_i)$ is a random variable following an Exponential distribution $Exp(\sum_{i=1}^{p} \lambda_i)$ (see the textbook by Ross [16, p. 288]).

The memoryless property of Exponential distributions is the key to the result for the delay between the first and second fault on the super-processor. Knowing that first fault occurred on processor P_1 at time t, what is the (conditional) probability distribution of a random variable for the occurrence of the first fault on processor P_2? This probability distribution is conditioned on the information that P_2 has been alive for t seconds. The memoryless property states that the probability distribution of the arrival time of the first fault on P_2 is not changed at all by when given this information! It is still an Exponential distribution $Exp(\lambda)$. Of course this holds true not only for P_2, but for each processor. And we can use the same minimum trick as for the first fault. Finally, the reasoning is the same for the third fault, and so on.

This concludes the proof. We refer the reader to Appendix 3 for another proof, where we also prove Equation (6.1).

Proposition 6.3 shows that scale is the enemy of fault-tolerance. If we double up the number of components in the platform, we divide the MTBF by 2, and the minimum waste automatically increases by a factor $\sqrt{2} \approx 1.4$ (see Equation (6.9)). And this assumes that the checkpoint time C remains constant. With twice as many processors, there is twice more data to write onto stable storage, hence the aggregated I/O bandwidth of the platform must be doubled to match this latter requirement.

6.4 Fault prediction

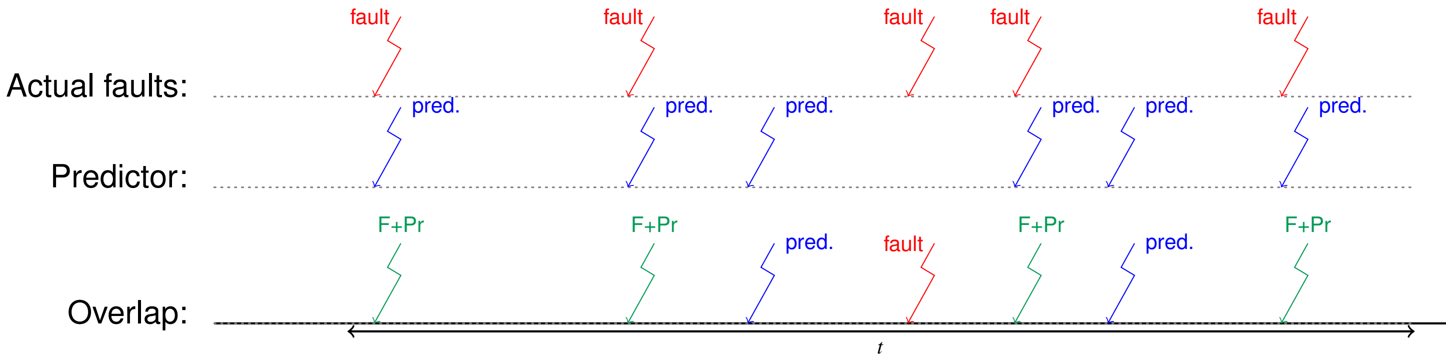
A possible way to cope with the numerous faults and their impact on the execution time is to try and predict them. In this section we do not explain how this is done, although Gainaru et al. [10], Yu et al. [19] and Zheng et al. [21] provide more details for the interested reader.

A *fault predictor* (or simply a predictor) is a mechanism that warns the user about upcoming faults on the platform. More specifically, a predictor is characterized by two key parameters, its recall r, which is the fraction of faults that are indeed predicted, and its precision pr, which is the fraction of predictions that are correct (i.e., correspond to actual faults). In this section, we discuss how to combine checkpointing and prediction to decrease the platform waste.

We start with a few definitions. Let μ_{Pr} be the mean time between predicted events (both true positive and false positive), and μ_{NPr} the mean time between unpredicted faults (false negative). The relations between μ_{Pr}, μ_{NPr}, μ, r and pr are as follows:

- Rate of unpredicted faults: $\frac{1}{\mu_{\text{NPr}}} = \frac{1-r}{\mu}$, since $1-r$ is the fraction of faults that are unpredicted;
- Rate of predicted faults: $\frac{r}{\mu} = \frac{pr}{\mu_{\text{Pr}}}$, since r is the fraction of faults that are predicted, and pr is the fraction of fault predictions that are correct.

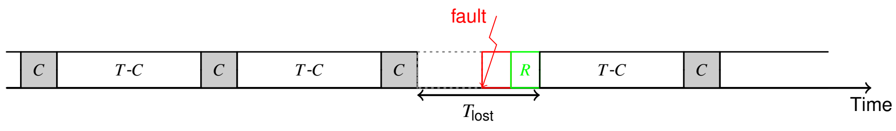
To illustrate all these definitions, consider the time interval below and the different events occurring:



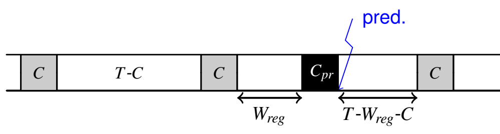
During this time interval of length t, the predictor predicts six faults, and there were five actual faults. One fault was not predicted. This gives approximatively: $\mu = \frac{t}{5}$, $\mu_{\text{Pr}} = \frac{t}{6}$, and $\mu_{\text{NPr}} = t$. For this predictor, the recall is $r = \frac{4}{5}$ (green "F+Pr" arrows over red "fault" arrows), and its precision is $pr = \frac{4}{6}$ (green "F+Pr" arrows over blue "pred." arrows).

Now, given a fault predictor of parameters pr and r, can we improve the waste? More specifically, how to modify the periodic checkpointing algorithm to get better results? In order to answer this question, we introduce *proactive checkpointing*: when there is a prediction, we assume that the prediction is given early enough so that we have time for a checkpoint of size C_{pr} (which can be different from C). We consider the following simple algorithm:

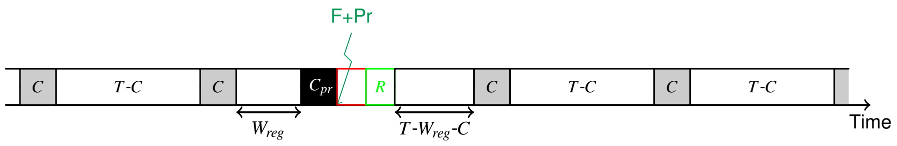
- While no fault prediction is available, checkpoints are taken periodically with period T;
- When a fault is predicted, we take a proactive checkpoint (of length C_{pr}) as late as possible, so that it completes right at the time when the fault is predicted to strike. After this checkpoint, we complete the execution of the period (see Figures 6.9b and 6.9c);



(a) Unpredicted fault



(b) Prediction taken into acco



(c) Prediction taken into account - with actual fault

Figure 6.9: Actions taken for the different event types.

We compute the expected waste as before. We reproduce Equation (6.6) below:

$$\text{WASTE} = \text{WASTE}_{\text{FF}} + \text{WASTE}_{\text{fault}} - \text{WASTE}_{\text{FF}}\text{WASTE}_{\text{fault}} \tag{6.11}$$

While the value of $\textsc{Waste}_{\text{FF}}$ is unchanged ($\textsc{Waste}_{\text{FF}} = \frac{C}{T}$), the value of $\textsc{Waste}_{\text{fault}}$ is modified because of predictions. As illustrated in Figure 6.9, there are different scenarios that contribute to $\textsc{Waste}_{\text{fault}}$. We classify them as follows:

(1) Unpredicted faults: This overhead occurs each time an unpredicted fault strikes, that is, on average, once every μ_{NPr} seconds. Just as in Equation (6.5), the corresponding waste is $\frac{1}{\mu_{\text{NPr}}}\left[\frac{T}{2}+ +R\right]$.

(2) Predictions: We now compute the overhead due to a prediction. If the prediction is an actual fault (with probability pr), we lose $C_{pr} + +R$ seconds, but if it is not (with probability $1-pr$), we lose the unnecessary extra checkpoint time C_{pr}. Hence

$$T_{\text{lost}} = pr(C_{pr} + +R) + (1-pr)C_{pr} = C_{pr} + pr(+R)$$

We derive the final value of $\textsc{Waste}_{\text{fault}}$:

$$\begin{aligned}\textsc{Waste}_{\text{fault}} &= \frac{1}{\mu_{\text{NPr}}}\left(\frac{T}{2} + +R\right) + \frac{1}{\mu_{\text{Pr}}}\left(C_{pr} + pr(+R)\right)\\ &= \frac{1-r}{\mu}\left(\frac{T}{2} + +R\right) + \frac{r}{pr\mu}\left(C_{pr} + pr(+R)\right)\\ &= \frac{1}{\mu}\left((1-r)\frac{T}{2} + +R + \frac{rC_{pr}}{pr}\right)\end{aligned}$$

We can now plug this expression back into Equation (6.11):

$$\begin{aligned}\textsc{Waste} &= \textsc{Waste}_{\text{FF}} + \textsc{Waste}_{\text{fault}} - \textsc{Waste}_{\text{FF}}\textsc{Waste}_{\text{fault}}\\ &= \frac{C}{T} + \left(1-\frac{C}{T}\right)\frac{1}{\mu}\left(+R + \frac{rC_{pr}}{pr} + \frac{(1-r)T}{2}\right).\end{aligned}$$

To compute the value of T_{FO}^{pr}, the period that minimizes the total waste, we use the same reasoning as in Section 6.2.5 and obtain:

$$T_{\text{FO}}^{pr} = \sqrt{\frac{2\left(\mu - \left(+R + \frac{rC_{pr}}{pr}\right)\right)C}{1-r}}.$$

We observe the similarity of this result with the value of T_{FO} from Equation (6.8). If μ is large in front of the resilience parameters, we derive that $T_{\text{FO}}^{pr} = \sqrt{\frac{2\mu C}{1-r}}$. This tells us that the recall is more important than the precision. If the predictor is capable of predicting, say, 84% of the faults, then $r = 0.84$ and $\sqrt{1-r} = 0.4$. The optimal period gets 2.5 times larger, and the waste is decreased by 60%. Prediction can help! See Appendix 4 for further information.

6.5 Replication

Another possible way to cope with the numerous faults and their impact on the execution time is to use replication. Replication consists in duplicating all computations. Processors are grouped by pairs, such that each processor has a *buddy* (another processor performing exactly the same computations, receiving the same messages, etc). See Figure 6.10 for an illustration. We say that the two processes in a given pair are *replicas*. When a processor is hit by a fault, its buddy is not impacted. The execution of the application can still progress, until the buddy itself is hit by a fault later on. This sounds quite expensive: by definition, half of the resources are wasted (and this does not include the overhead of maintaining a consistent state between the two processors of each pair). At first sight, the idea of using replication on a large parallel platform is puzzling: who is ready to waste half of these expensive supercomputers?

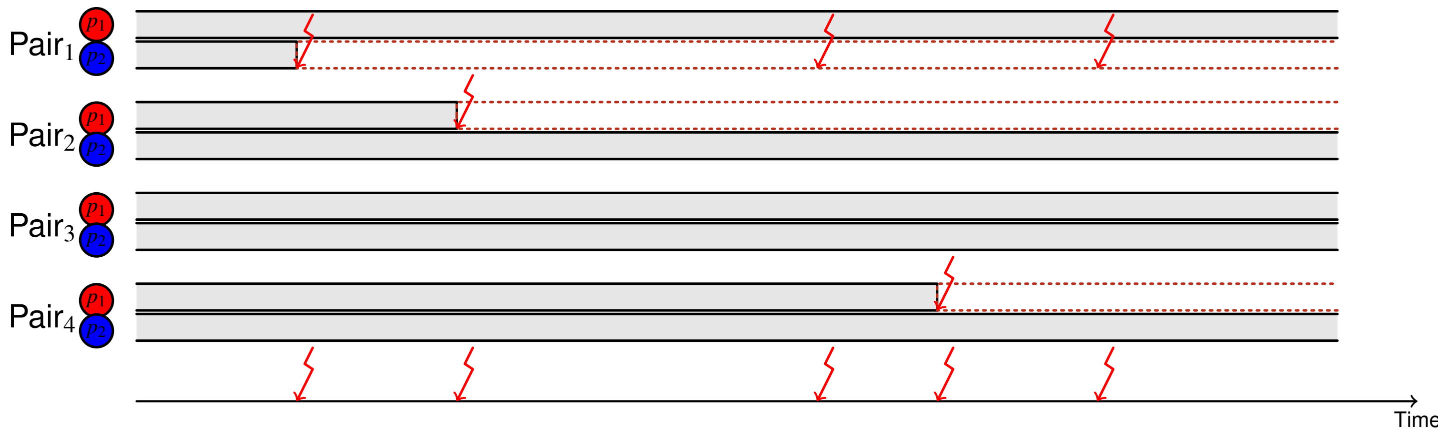


Figure 6.10: Processor pairs for replication: each blue processor is paired with a red processor. In each pair, both processors do the same work.

In this section, we explain how replication can be used in conjunction with checkpointing and under which conditions it becomes profitable. In order to do this, we compare the checkpointing technique introduced earlier to the replication technique.

A *perfectly parallel application* is an application such that in a fault-free, checkpoint-free environment, the time to execute the application ($\text{TIME}_{\text{Base}}$) decreases linearly with the number of processors. More precisely:

$$\text{TIME}_{\text{base}}(p) = \frac{\text{TIME}_{\text{base}}(1)}{p}.$$

Consider the execution of a perfectly parallel application on a platform with $p = 2P$ processors, each with individual MTBF μ_{ind}. As in the previous sections, the optimization problem is to find the strategy minimizing $\text{TIME}_{\text{final}}$. Because we compare two approaches using a different number of processors, we introduce the THROUGHPUT, which is defined as the total number of useful flops per second:

$$\text{THROUGHPUT} = \frac{\text{TIME}_{\text{base}}(1)}{\text{TIME}_{\text{final}}}$$

Note that for an application executing on p processors, $\text{THROUGHPUT} = p\,(1 - \text{WASTE})$.

The *standard* approach, as seen before, is to use all $2P$ processors to fully parallelize the execution of the application on the platform. This would be optimal in a fault-free environment, but we are required to checkpoint frequently because faults repeatedly strike the p processors. According to Proposition 6.3, the platform MTBF is $\mu = \frac{\mu_{\text{ind}}}{p}$. According to Theorem 6.2.5, the waste is (approximately) $\text{WASTE} = \sqrt{\frac{2C}{\mu}} = \sqrt{\frac{2Cp}{\mu_{\text{ind}}}}$. We have:

$$\text{THROUGHPUT}_{\text{Std}} = p\left(1 - \sqrt{\frac{2Cp}{\mu_{\text{ind}}}}\right) \tag{6.12}$$

The second approach uses *replication*. There are P pairs of processors, all computations are executed twice, hence only half the processors produce useful flops. One way to see the replication technique is as if there were half the processors using only the checkpoint technique, with a different (potentially higher) mean time between faults, μ_{rep}. Hence, the throughput $\text{THROUGHPUT}_{\text{Rep}}$ of this approach writes:

$$\text{THROUGHPUT}_{\text{Rep}} = \frac{P}{2}\left(1 - \sqrt{\frac{2C}{\mu_{\text{rep}}}}\right) \tag{6.13}$$

In fact, rather than MTBF, we should say MTTI, for *Mean Time To Interruption*. As already mentioned, a single fault on the platform does not interrupt the application, because the replica of the faulty processor is still alive. What is the value of *MNFTI*, the *Mean Number of Faults To Interruption*, i.e., the mean number of faults that should strike the platform until there is a replica pair whose processors have both been hit? If we find how to compute *MNFTI*, we are done, because we know that

$$\mu_{\text{rep}} = \mathit{MNFTI} \times \mu = \mathit{MNFTI} \times \frac{\mu_{\text{ind}}}{p}$$

We make an analogy with a balls-into-bins problem to compute *MNFTI*. The classical problem is the following: what is the expected number of balls that you will need, if you throw these balls randomly into P bins, until one bins gets two balls? The answer to this question is given by Ramanujan's Q-Function (see Flajolet [9]), and is equal to $\lceil q(P) \rceil$ where $q(P) = \frac{2}{3} + \sqrt{\frac{\pi P}{2}} + \sqrt{\frac{\pi}{288P}} - \frac{4}{135P} + \dots$. When $P = 365$, this is the birthday problem where balls are persons and bins are calendar dates; in the best case, one needs two persons; in the worst case, one needs $P + 1 = 366$ persons; on average, one needs $\lceil q(P) \rceil = 25$ persons.*

In the replication problem, the bins are the processor pairs, and the balls are the faults. However, the analogy stops here. The problem is more complicated, see Figure 6.11 to see why. Each processor pair is composed of a blue processor and of a red processor. Faults are (randomly) colored blue or red too. When a fault strikes a processor pair, we need to know which processor inside that pair: we decide that it is the one of the same color as the fault. Blue faults strike blue processors, and red faults strike red processors. We now understand that we may need more than

*As a side note, one needs only 23 persons for the probability of a common birthday to reach 0.5 (a question often asked in geek evenings).

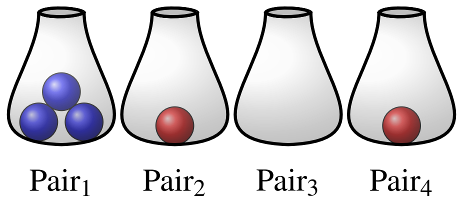


Figure 6.11: Modeling the state of the platform of Figure 6.10 as a balls-into-bins problem. We put a red ball in bin Pair$_i$ when there is a fault on its red processor p_1, and a blue ball when there is a fault on its blue processor p_2. As long as no bin has received a ball of each color, the game is on.

two faults hitting the same pair to interrupt the application: we need one fault of each color. The balls-and-bins problem to compute *MNFTI* is now clear: what is the expected number of red and blue balls that you will need, if you throw these balls randomly into P bins, until one bins gets both one red ball and one blue ball? To the best of our knowledge, there is no closed-form solution to answer this question, but a recursive computation does the job:

$MNFTI = (NFTI|0)$ where

$$(NFTI|n_f) = \begin{cases} 2 & \textit{if } n_f = P, \\ \frac{2P}{2P-n_f} + \frac{2P-2n_f}{2P-n_f}\left(NFTI|n_f+1\right) & \textit{otherwise.} \end{cases}$$

Let $(NFTI|n_f)$ be the expectation of the number of faults needed to interrupt the application, knowing that the application is still running and that faults have already hit n_f different processor pairs. Because each pair initially has 2 replicas, this means that n_f different pairs are no longer replicated, and that $P - n_f$ are still replicated. Overall, there are $n_f + 2(P - n_f) = 2P - n_f$ processors still running.

The case $n_f = P$ is simple. In this case, all pairs have already been hit, and all pairs have only one of their two initial replicas still running. A new fault will hit such a pair. Two cases are then possible:

1. The fault hits the running processor. This leads to an application interruption, and in this case $(NFTI|P) = 1$.
2. The fault hits the processor that has already been hit. Then the fault has no impact on the application. The *MNFTI* of this case is then: $(NFTI|P) = 1 + (NFTI\,|P)$.

The probability of fault is uniformly distributed between the two replicas, and thus between these two cases. Weighting the values by their probabilities of occurrence yields:

$$(NFTI\,|P) = \frac{1}{2} \times 1 + \frac{1}{2} \times (1 + (NFTI\,|P)),$$

hence $(NFTI\,|P) = 2$.

For the general case $0 \le n_f \le P - 1$, either the next fault hits a new pair, i.e., a pair whose 2 processors are still running, or it hits a pair that has already been hit, hence with a single processor running. The latter case leads to the same sub-cases as the $n_f = P$ case studied above. The fault probability is uniformly distributed among the $2P$ processors, including the ones already hit. Hence the probability that the next fault hits a new pair is $\frac{2P-2n_f}{2P}$. In this case, the expected number of faults needed to interrupt the application fail is one (the considered fault) plus $\left(NFTI|n_f+1\right)$. Altogether we have:

$$\left(NFTI|n_f\right) = \frac{2P-2n_f}{2P} \times \left(1+\left(NFTI|n_f+1\right)\right) + \frac{2n_f}{2P} \times \left(\frac{1}{2}\times 1 + \frac{1}{2}\left(1+\left(NFTI|n_f\right)\right)\right).$$

Therefore,

$$\left(NFTI|n_f\right) = \frac{2P}{2P-n_f} + \frac{2P-2n_f}{2P-n_f}\left(NFTI|n_f+1\right).$$

Let us compare the throughput of each approach with an example. From Equations (6.12) and (6.13), we have

$$\text{THROUGHPUT}_{\text{Rep}} \geq \text{THROUGHPUT}_{\text{Std}} \Leftrightarrow (1-\sqrt{\frac{2Cp}{MNFTI\ \mu_{\text{ind}}}}) \geq 2(1-\sqrt{\frac{2Cp}{\mu_{\text{ind}}}})$$

which we rewrite into

$$C \geq \frac{\mu_{\text{ind}}}{2p}\frac{1}{(2-\frac{1}{\sqrt{MNFTI}})^2} \tag{6.14}$$

Take a parallel machine with $p = 2^{20}$ processors. This is a little more than one million processors, but this corresponds to the size of the largest platforms today. Using Proposition 6.5, we compute $MNFTI = 1284.4$. Assume that the individual MTBF is 10 years, or in seconds $\mu_{\text{ind}} = 10 \times 365 \times 24 \times 3600$. After some painful computations, we derive that replication is more efficient if the checkpoint time is greater than 293 seconds (around 6 minutes). This sets a target both for architects and checkpoint protocol designers.

Maybe you can say that $\mu_{\text{ind}} = 10$ years is pessimistic, because one would observe that $\mu_{\text{ind}} = 100$ years in current supercomputers. Because $\mu_{\text{ind}} = 100$ years allows to checkpoint up to one hour, you would decide that replication is not worth it. But maybe you can also say that $\mu_{\text{ind}} = 10$ years is optimistic for processors equipped with thousands of cores and rather take $\mu_{\text{ind}} = 1$ year. In that case, unless you checkpoint in less than 30 seconds, better be prepared for replication. The beauty of performance models is that you can decide which approach is better *without bias nor a-priori*, simply by plugging your own parameters into Equation (6.14).

6.6 Conclusion

In this chapter, we have dealt with fail-stop faults, i.e. faults that cause the application to crash and require to repair the resource or to find a spare one, and to re-execute work from some state of the application that had been previously saved. Other techniques involve to reconstruct the data lost by the failing processor from redundant information (e.g., checksums) maintained by the other processors. While unrecoverable, a fail-stop error has the nice characteristic that it can be detected immediately. On the contrary, a *silent error*, a.k.a. *silent data corruption*, gets unnoticed until it manifests after some random delay, e.g. because corrupted data is activated. Silent errors come from many sources, from errors in the arithmetic unit (due to low voltages) to bit flips in cache (due to cosmic radiation). Silent errors are difficult to detect, and because of the detection latency, they are even more difficult to correct. We refer the interested reader to studies such as Cappello et al. [4] or Gainaru et al. [11] to know more about the fascinating problems and solution techniques in the area of fault-tolerant computing at very large scale. See also the monograph [13] for a recent survey of fault-tolerant techniques for High Performance Computing.

Exascale computing (10^{18} operations per second, which require either one million processors, each with one thousand cores, or one hundred thousand processors, each with ten thousand cores) is a very large scale, but it is the scale of future-generation machines that will be with us in less than 10 years. Thus the area of resilience at scale is extremely important, and clever scheduling techniques are needed to help solve all the problems. Alice needs more help[†].

[†]By the way, there is a nice little exercise in Appendix 6 if you are motivated to help.

Bibliography

[1] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni. Checkpointing strategies with prediction windows. In *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, pages 1–10. IEEE, 2013.

[2] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni. Checkpointing algorithms and fault prediction. *Journal of Parallel and Distributed Computing*, 74(2):2048–2064, 2014.

[3] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Proceedings of SC'11*, 2011.

[4] F. Cappello, A. Geist, B. Gropp, L. V. Kalé, B. Kramer, and M. Snir. Toward Exascale Resilience. *Int. Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[5] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Combining process replication and checkpointing for resilience on exascale systems. Research report RR-7951, INRIA, May 2012.

[6] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *FGCS*, 22(3):303–312, 2004.

[7] C. Engelmann, H. H. Ong, and S. L. Scorr. The case for modular redundancy in large-scale highh performance computing systems. In *Proc. of the 8th IASTED Infernational Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 189–194, 2009.

[8] K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proc. of the ACM/IEEE SC Conf.*, 2011.

[9] P. Flajolet, P. J. Grabner, P. Kirschenhofer, and H. Prodinger. On Ramanujan's Q-Function. *J. Computational and Applied Mathematics*, 58:103–116, 1995.

[10] A. Gainaru, F. Cappello, and W. Kramer. Taming of the shrew: Modeling the normal and faulty behavior of large-scale hpc systems. In *Proc. IPDPS'12*, 2012.

[11] A. Gainaru, F. Cappello, M. Snir, and W. Kramer. Failure prediction for hpc systems and applications: Current situation and open issues. *Int. J. High Perform. Comput. Appl.*, 27(3):273–282, 2013.

[12] F. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1), 1999.

[13] T. Hérault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag, 2015.

[14] O. Kella and W. Stadje. Superposition of renewal processes and an application to multi-server queues. *Statistics & probability letters*, 76(17):1914–1924, 2006.

[15] D. Kondo, A. Chien, and H. Casanova. Scheduling Task Parallel Applications for Rapid Application Turnaround on Enterprise Desktop Grids. *J. Grid Computing*, 5(4):379–405, 2007.

[16] S. M. Ross. *Introduction to Probability Models, Eleventh Edition*. Academic Press, 2009.

[17] B. Schroeder and G. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[18] J. W. Young. A first order approximation to the optimum checkpoint interval. *Comm. of the ACM*, 17(9):530–531, 1974.

[19] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan. Practical online failure prediction for blue gene/p: Period-based vs event-driven. In *Dependable Systems and Networks Workshops (DSN-W)*, pages 259–264, 2011.

[20] Z. Zheng and Z. Lan. Reliability-aware scalability models for high performance computing. In *Proc. of the IEEE Conference on Cluster Computing*, 2009.

[21] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman. A practical failure prediction with location and lead time for blue gene/p. In *Dependable Systems and Networks Workshops (DSN-W)*, pages 15–22, 2010.

Appendix 1: First-order approximation of T_{FO}

It is interesting to point out why the value of T_{FO} given by Equation (6.8) is a first-order approximation, even for large jobs. Indeed, there are several restrictions for the approach to be valid:

- We have stated that the expected number of faults during execution is $N_{\text{faults}} = \frac{\textsc{Time}_{\text{final}}}{\mu}$, and that the expected time lost due to a fault is $T_{\text{lost}} = \frac{T}{2} + +R$. Both statements are true individually, but the expectation of a product is the product of the expectations only if the random variables are independent, which is not the case here because $\textsc{Time}_{\text{final}}$ depends upon the fault inter-arrival times.
- In Equation (6.4), we have to enforce $C \leq T$ in order to have $\textsc{Waste}_{\text{FF}} \leq 1$.

- In Equation (6.5), we have to enforce $+R+\frac{T}{2} \leq \mu$ in order to have $\text{WASTE}_{\text{fault}} \leq 1$. We must cap the period to enforce this latter constraint. Intuitively, we need μ to be large enough for Equation (6.5) to make sense. However, for large-scale platforms, regardless of the value of the individual MTBF μ_{ind}, there is always a threshold in the number of components p above which the platform MTBF, $\mu = \frac{\mu_{\text{ind}}}{p}$, becomes too small for Equation (6.5) to be valid.
- Equation (6.5) is accurate only when two or more faults do not take place within the same period. Although unlikely when μ is large in front of T, the possible occurrence of many faults during the same period cannot be eliminated.

To ensure that the condition of having at most a single fault per period is met with a high probability, we cap the length of the period: we enforce the condition $T \leq \alpha\mu$, where α is some tuning parameter chosen as follows. The number of faults during a period of length T can be modeled as a Poisson process of parameter $\beta = \frac{T}{\mu}$. The probability of having $k \geq 0$ faults is $P(X = k) = \frac{\beta^k}{k!}e^{-\beta}$, where X is the random variable showing the number of faults. Hence the probability of having two or more faults is $\pi = P(X \geq 2) = 1 - (P(X = 0) + P(X = 1)) = 1 - (1+\beta)e^{-\beta}$. If we assume $\alpha = 0.27$ then $\pi \leq 0.03$, hence a valid approximation when bounding the period range accordingly. Indeed, with such a conservative value for α, we have overlapping faults for only 3% of the checkpointing segments in average, so that the model is quite reliable. For consistency, we also enforce the same type of bound on the checkpoint time, and on the downtime and recovery: $C \leq \alpha\mu$ and $+R \leq \alpha\mu$. However, enforcing these constraints may lead to use a sub-optimal period: it may well be the case that the optimal period $\sqrt{2(\mu - (+R))C}$ of Equation (6.8) does not belong to the admissible interval $[C, \alpha\mu]$. In that case, the waste is minimized for one of the bounds of the admissible interval. This is because, as seen from Equation (6.7), the waste is a convex function of the period.

We conclude this discussion on a positive note. While capping the period, and enforcing a lower bound on the MTBF, is mandatory for mathematical rigor, simulations in Aupy et al. [2] show that actual job executions can always use the value from Equation (6.8), accounting for multiple faults whenever they occur by re-executing the work until success. The first-order model turns out to be surprisingly robust!

Appendix 2: Optimal value of T_{FO}

There is a beautiful method to compute the optimal value of T_{FO} accurately. First we show how to compute the expected time $(\text{TIME}(T-C, C, , R, \lambda))$ to execute a work of duration $T-C$ followed by a checkpoint of duration C, given the values of C, , and R, and a fault distribution $Exp(\lambda)$. If a fault interrupts a given trial before success, there is a downtime of duration followed by a recovery of length R. We assume that faults can strike during checkpoint and recovery, but not during downtime.

$$(\text{TIME}(T-C, C, , R, \lambda)) = e^{\lambda R}\left(\tfrac{1}{\lambda}+\right)(e^{\lambda T} - 1).$$

For simplification, we write TIME instead of $\text{TIME}(T-C, C, , R, \lambda)$ in the proof below. Consider the following two cases:

(i) Either there is no fault during the execution of the period, then the time needed is exactly T;

(ii) Or there is one fault before successfully completing the period, then some additional delays are incurred. More specifically, as seen for the first order approximation, there are two sources of delays: the time spent computing by the processors before the fault (accounted for by variable $\text{TIME}_{\text{lost}}$), and the time spent for downtime and recovery (accounted for by variable TIME_{rec}). Once a successful recovery has been completed, there still remain $T-C$ units of work to execute.

Thus TIME obeys the following recursive equation:

$$\text{TIME} = \begin{cases} T & \text{if there is no fault} \\ \text{TIME}_{\text{lost}} + \text{TIME}_{\text{rec}} + \text{TIME} & \text{otherwise} \end{cases} \tag{6.15}$$

$\text{TIME}_{\text{lost}}$ denotes the amount of time spent by the processors before the first fault, knowing that this fault occurs within the next T units of time. In other terms, it is the time that is wasted because computation and checkpoint were not successfully completed (the corresponding value in Figure 6.2 is $T_{\text{lost}} - -R$).

TIME_{rec} represents the amount of time needed by the system to recover from the fault (the corresponding value in Figure 6.2 is $+R$).

The expectation of TIME can be computed from Equation (6.15) by weighting each case by its probability to occur:

$$\begin{aligned}(\text{TIME}) &= \mathbb{P}(\text{no fault}) \cdot T + \mathbb{P}(\text{a fault strikes}) \cdot (\text{TIME}_{\text{lost}} + \text{TIME}_{\text{rec}} + \text{TIME}) \\ &= e^{-\lambda T} T + (1-e^{-\lambda T})\left((\text{TIME}_{\text{lost}}) + (\text{TIME}_{\text{rec}}) + (\text{TIME})\right) ,\end{aligned}$$

which simplifies into:

$$(\text{TIME}) = T + (e^{\lambda T} - 1)\left(E(\text{TIME}_{\text{lost}}) + E(\text{TIME}_{\text{rec}})\right) \tag{6.16}$$

We have $(\text{TIME}_{\text{lost}}) = \int_0^\infty x\mathbb{P}(X=x|X<T)dx = \frac{1}{\mathbb{P}(X<T)}\int_0^T xe^{-\lambda x}dx$, and $\mathbb{P}(X<T) = 1-e^{-\lambda T}$. Integrating by parts, we derive that

$$(\text{TIME}_{\text{lost}}) = \frac{1}{\lambda} - \frac{T}{e^{\lambda T}-1} \tag{6.17}$$

Next, the reasoning to compute $(\text{TIME}_{\text{rec}})$, is very similar to (TIME) (note that there can be no fault during but there can be some during R):

$$(\text{TIME}_{\text{rec}}) = e^{-\lambda R}(+R) + (1-e^{-\lambda R})(+(R_{lost}) + (\text{TIME}_{\text{rec}}))$$

Here, R_{lost} is the amount of time lost to executing the recovery before a fault happens, knowing that this fault occurs within the next R units of time. Replacing T by R in Equation (6.17), we obtain $(R_{lost}) = \frac{1}{\lambda} - \frac{R}{e^{\lambda R}-1}$. The expression for $(\text{TIME}_{\text{rec}})$ simplifies to

$$(\text{TIME}_{\text{rec}}) = e^{\lambda R} + \frac{1}{\lambda}(e^{\lambda R} - 1)$$

Plugging the values of $(\text{TIME}_{\text{lost}})$ and $(\text{TIME}_{\text{rec}})$ into Equation (6.16) leads to the desired value:

$$(\text{TIME}(T-C, C, , R, \lambda)) = e^{\lambda R}\left(\frac{1}{\lambda} + \right)(e^{\lambda T} - 1)$$

Proposition 6.6 is the key to proving that the optimal checkpointing strategy is periodic. Indeed, consider an application of duration $\text{TIME}_{\text{base}}$, and divide the execution into periods of different lengths T_i, each with a checkpoint as the end. The expectation of the total execution time is the sum of the expectations of the time needed for each period. Proposition 6.6 shows that the expected time for a period is a convex function of its length, hence all periods must be equal and $T_i = T$ for all i.

There remains to find the best number of periods, or equivalently, the size of each work chunk before checkpointing. With k periods of length $T = \frac{\text{TIME}_{\text{base}}}{k}$, we have to minimize a function that depends on k. This is easy for a skilled mathematician who knows the Lambert function $\mathbb{L}$ (defined as $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$). She would find the optimal rational value k_{opt} of k by differentiation, prove that the objective function is convex, and conclude that the optimal value is either $\lfloor k_{opt} \rfloor$ or $\lceil k_{opt} \rceil$, thereby determining the optimal period T_{opt}. What if you are not a skilled mathematician? No problem, simply use T_{FO} as a first-order approximation, and be comforted that the first-order terms in the Taylor expansion of T_{opt} is ... T_{FO}! See Bougeret et al. [3] for all details.

Appendix 3: MTBF of a platform with p parallel processors

In this section we give another proof of Proposition 6.3. Interestingly, it applies to any continuous probability distribution with bounded (nonzero) expectation, not just Exponential laws.

First we prove that Equation (6.1) does hold true. Consider a single processor, say processor p_q. Let X_i, $i \geq 0$ denote the IID (independent and identically distributed) random variables for the fault inter-arrival times on p_q , and assume that $X_i \sim D_X$, where D_X is a continuous probability distribution with bounded (nonzero) expectation μ_{ind}. In particular, $\mathbb{E}(X_i) = \mu_{\text{ind}}$ for all i. Consider a fixed time bound F. Let $n_q(F)$ be the number of faults on p_q until time F. More precisely, the $(n_q(F))$-th fault is the last one to happen before time F or at time F, and the $(n_q(F)+1)$-st fault is the first to happen after time F. By definition of $n_q(F)$, we have

$$\sum_{i=1}^{n_q(F)} X_i \leq F < \sum_{i=1}^{n_q(F)+1} X_i.$$

Using Wald's equation (see the textbook of Ross [16, p. 420]), with $n_q(F)$ as a stopping criterion, we derive:

$$\mathbb{E}\left(n_q(F)\right)\mu_{\text{ind}} \le F \le (\mathbb{E}\left(n_q(F)\right)+1)\mu_{\text{ind}},$$

and we obtain:

$$\lim_{F\to+\infty}\frac{\mathbb{E}\left(n_q(F)\right)}{F} = \frac{1}{\mu_{\text{ind}}}. \tag{6.18}$$

As promised, Equation (6.18) is exactly Equation (6.1).

Now consider a platform with p identical processors, whose fault inter-arrival times are IID random variables that follow the distribution D_X. Unfortunately, if D_X is not an Exponential law, then the inter-arrival times of the faults of the whole platform, i.e., of the super-processor of Section 6.3, are no longer IID. The minimum trick used in the proof of Proposition 6.3 works only for the first fault. For the following ones, we need to remember the history of the previous faults, and things get too complicated. However, we could still define the MTBF μ of the super-processor. Using Equation (6.18), μ must satisfy:

$$\lim_{F\to+\infty}\frac{\mathbb{E}\left(n(F)\right)}{F} = \frac{1}{\mu},$$

where $n(F)$ is the number of faults on the super-processor until time F. But does the limit always exist? and if yes, what is its value?

The answer to both questions is not difficult. Consider a fixed time bound F as before. Let $n(F)$ be the number of faults on the whole platform until time F, and let $m_q(F)$ be the number of these faults that strike component number q. Of course we have $n(F) = \sum_{q=1}^{p} m_q(F)$. By definition, $m_q(F)$ is the number of faults on component q until time F. From Equation (6.18) again, we have for each component q:

$$\lim_{F\to+\infty}\frac{\mathbb{E}\left(m_q(F)\right)}{F} = \frac{1}{\mu_{\text{ind}}}.$$

Since $n(F) = \sum_{q=1}^{p} m_q(F)$, we also have:

$$\lim_{F\to+\infty}\frac{\mathbb{E}\left(n(F)\right)}{F} = \frac{p}{\mu_{\text{ind}}} \tag{6.19}$$

which answers both questions at the same time!

The curious reader may ask how to extend Equation (6.19) when processors have different fault-rates. Let $X_i^{(q)}$, $i \ge 0$ denote the IID random variables for the fault inter-arrival times on p_q , and assume that $X_i^{(q)} \sim D_X^{(q)}$, where $D_X^{(q)}$ is a continuous probability distribution with bounded (nonzero) expectation $\mu^{(q)}$. For instance if $\mu^{(2)} = 3\mu^{(1)}$, then (in expectation) processor 1 experiences three times more failures than processor 2. As before, consider a fixed time bound F, and let $n_q(F)$ be the number of faults on p_q until time F. Equation (6.18) now writes $\lim_{F\to+\infty}\frac{\mathbb{E}\left(m_q(F)\right)}{F} = \frac{1}{\mu^{(q)}}$. Now let $n(F)$ be the total number of faults on the whole platform until time F. The same proof as above leads to

$$\lim_{F\to+\infty}\frac{\mathbb{E}\left(n(F)\right)}{F} = \sum_{q=1}^{p}\frac{1}{\mu^{(q)}} \tag{6.20}$$

Kella and Stadje [14] provide more results on the superposition of renewal processes (which is the actual mathematical name of the topic discussed here!).

Appendix 4: Going further with prediction.

The discussion on predictions in Section 6.4 has been kept overly simple. For instance when a fault is predicted, sometimes there is not enough time to take proactive actions, because we are already checkpointing. In this case, there is no other choice than ignoring the prediction.

Furthermore, a better strategy should take into account at what point in the period does the prediction occur. After all, there is no reason to always trust the predictor, in particular if it has a bad precision. Intuitively, the later the prediction takes place in the period, the more likely we are inclined to trust the predictor and take proactive actions. This is because the amount of work that we could lose gets larger and larger. On the contrary, if the prediction happens in the beginning of the period, we have to trade-off the probability that the proactive checkpoint may be useless (if we take a proactive action) with the small amount of work that may be lost in the case where a fault would actually happen (if we do not trust the predictor). The optimal approach is to never trust the predictor in the beginning of a period, and to always trust it in the end; the cross-over point $\frac{C_{pr}}{pr}$ depends on the time to take a proactive checkpoint and on the precision of the predictor. Details are provided by Aupy et al. [2] for details.

Finally, it is more realistic to assume that the predictor cannot give the exact moment where the fault is going to strike, but rather will provide an interval of time, a.k.a. a prediction window. Aupy et al. [1] provide more information.

Appendix 5: Going further with replication.

In the context of replication, there are two natural options for "counting" faults. The option chosen in Section 6.5 is to allow new faults to hit processors that have already been hit. This is the option chosen by Ferreira et al. [8], who introduced the problem. Another option is to count only faults that hit *running processors*, and thus effectively kill replica pairs and interrupt the application. This second option may seem more natural as the running processors are the only ones that are important for the application execution. It turns out that both options are almost equivalent, the values of their *MNFTI* only differ by one, as shown by Casanova et al. [5].

Speaking of faults, an important question is: why don't we repair (or rejuvenate) processors on the fly, instead of doing so only when the whole application is forced to stop, recover from the last checkpoint, and restart execution? The answer is technical: current HPC resource management systems assign the user a fixed set of resources for the execution, and do not allow new resources (such as spare nodes) to be dynamically added during the execution. In fact, frequently, a new configuration is assigned to the user at restart time. But nothing prevents us from enhancing the tools! It should then be possible to reserve a few additional nodes in addition to the computing nodes. These nodes would be used to migrate the system image of a replica node as soon as its buddy fails, in order to re-create the failed node on the fly. Of course the surviving node must be isolated from the application while the migration is taking place, in order to maintain a coherent view of both nodes, and this induces some overhead. It would be quite interesting to explore such strategies.

Here a few bibliographical notes about replication. Replication has long been used as a fault-tolerance mechanism in distributed systems (see the survey of Gartner [12]), and in the context of volunteer computing (see the work of Kondo et al. [15]). Replication has recently received attention in the context of HPC (High Performance Computing) applications. Representative papers are those by Schroeder and Gibson [17], Zheng and Lan [20], Engelmann, Ong, and Scorr [7], and Ferreira et al. [8]. While replicating all processors is very expensive, replicating only critical processes, or only a fraction of all processes, is a direction being currently explored under the name *partial replication*.

Speaking of critical processes, we make a final digression. The de-facto standard to enforce fault-tolerance in critical or embedded systems is *Triple Modular Redundancy*, or TMR. Computations are triplicated on three different processors, and if their results differ, a voting mechanism is called. TMR is not used to protect from fail-stop faults, but rather to detect and correct errors in the execution of the application. While we all like, say, safe planes protected by TMR, the cost is tremendous: by definition, two thirds of the resources are wasted (and this does not include the overhead of voting when an error is identified).

Appendix 6: Scheduling a linear chain of tasks.

In this exercise you are asked to help Alice (again). She is still writing her thesis but she does not want to checkpoint at given periods of time. She hates being interrupted in the middle of something because she loses concentration. She now wants to checkpoint only at the end of a chapter. She still has to decide after which chapters it is best to checkpoint.

The difference with the original problem is that the checkpoints can only be taken at given time-steps. If we formulate the problem in a abstract way, we have a linear chain of n tasks (the n chapters in Alice's thesis), $T_1, T_2, \dots, T_n$. Each task T_i has weight w_i (the time it takes to write that chapter). The cost to checkpoint after T_i is C_i. The time to recover from a fault depends upon where the last checkpoint was taken. For example, assumt that T_i was checkpointed, and that T_{i+1}, T_{i+2} were not. If a fault strikes during the execution of T_{i+3}, we need to roll back and read the checkpoint of T_i from stable storage, which costs R_i. Then we start re-executing T_{i+1} and the following tasks. Note that the costs C_i and R_i are likely proportional to the chapter length).

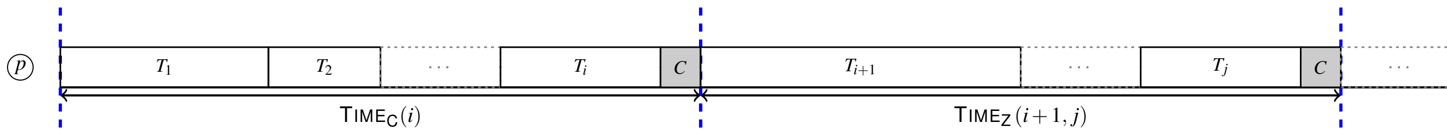


Figure 6.12: Hint for the exercise.

As before, the inter-arrival times of the faults are IID random variables following the Exponential law $Exp(\lambda)$. We must decide after which tasks to checkpoint, in order to minimize the expectation of the total time. Figure 6.12 gives you a hint. $\textsc{Time}_{\mathrm{C}}(i)$ is the optimal solution for the execution of tasks $T_1, T_2, \ldots, T_i$. The solution to the problem is $\textsc{Time}_{\mathrm{C}}(n)$, and we use a dynamic programming algorithm to compute it. In the algorithm, we need to know $\textsc{Time}_{\mathrm{Z}}$ (i+1,j), the expected time to compute a segment of tasks $[T_{i+1}..T_j]$ and to checkpoint the last one T_j, knowing that there is a checkpoint before the first one (hence after T_i) and that no intermediate checkpoint is taken. $\textsc{Time}_{\mathrm{Z}}$ stands for *Zero intermediate checkpoint*. It turns out that we already know the value of $\textsc{Time}_{\mathrm{Z}}(i+1, j)$: check that we have

$$\textsc{Time}_{\mathrm{Z}}(i+1, j) = \left(\textsc{Time} \left(\sum_{k=i+1}^{j} w_k, C_j, , R_i, \lambda \right) \right)$$

and use Proposition 6.6.