Chapter 1
Fault Tolerance Techniques for High-Performance Computing

Jack Dongarra, Thomas Herault and Yves Robert

Abstract This chapter provides an introduction to resilience methods. The emphasis is on checkpointing, the de-facto standard technique for resilience in High Performance Computing. We present the main two protocols, namely coordinated checkpointing and hierarchical checkpointing. Then we introduce performance models and use them to assess the performance of theses protocols. We cover the Young/Daly formula for the optimal period and much more! Next we explain how the efficiency of checkpointing can be improved via fault prediction or replication. Then we move to application-specific methods, such as ABFT. We conclude the chapter by discussing techniques to cope with silent errors (or silent data corruption).

1.1 Introduction

This chapter provides an overview of fault tolerance techniques for High Performance Computing (HPC). We present scheduling algorithms to cope with faults on large-scale parallel platforms. We start with a few general considerations on resilience at scale (Sect. 1.1.1) before introducing standard failure probability distributions (Sect. 1.1.2). The main topic of study is *checkpointing*, the de-facto standard technique for resilience in HPC. We present the main protocols, *coordinated* and *hierarchical*, in Sect. 1.2. We introduce probabilistic performance models

J. Dongarra · T. Herault (✉) · Y. Robert
University of Tennessee, Knoxville, TN, USA
e-mail: herault@icl.utk.edu

J. Dongarra
Oak Ridge National Laboratory, Oak Ridge, USA
e-mail: dongarra@eecs.utk.edu

J. Dongarra
University of Manchester, Manchester, UK

Y. Robert
Ecole Normale Supérieure de Lyon, Lyon, France
e-mail: yves.robert@inria.fr

T. Herault and Y. Robert (eds.), *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks,
DOI 10.1007/978-3-319-20943-2_1

to assess these protocols in Sect. 1.3. In particular, we show how to compute the optimal checkpointing period (the famous Young/Daly formula [25, 69]) and derive several extensions. Then Sect. 1.4 explains how to combine checkpointing with *fault prediction*, and discuss how the optimal period is modified when this combination is used (Sect. 1.4.1). We follow the very same approach for the combination of checkpointing with *replication* (Sect. 1.4.2).

While checkpointing (possibly coupled with fault prediction or replication) is a general-purpose method, there exist many application-specific methods. In Sect. 1.5, we present middleware adaptations to enable application-specific fault tolerance, and illustrate their use on one of the most important one, ABFT, which stands for *Algorithm based Fault Tolerance*, in Sect. 1.5.

The last technical section of this chapter (Sect. 1.6) is devoted to techniques to cope with silent errors (or silent data corruption). Section 1.7 concludes the chapter with final remarks.

1.1.1 Resilience at Scale

For HPC applications, scale is a major opportunity. Massive parallelism with 100,000+ nodes is the most viable path to achieving sustained Petascale performance. Future platforms will enroll even more computing resources to enter the Exascale era. Current plans refer to systems either with 100,000 nodes, each equipped with 10,000 cores (the *fat node* scenario), or with 1,000,000 nodes, each equipped with 1,000 cores (the *slim node* scenario) [27].

Unfortunately, scale is also a major threat, because resilience becomes a big challenge. Even if each node provides an individual MTBF (Mean Time Between Failures) of, say, one century, a machine with 100,000 such nodes will encounter a failure every 9 hours in average, which is larger than the execution time of many HPC applications. Worse, a machine with 1,000,000 nodes (also with a one-century MTBF) will encounter a failure every 53 min in average.[1] Note that a one-century MTBF per node is an optimistic figure, given that each node is composed of several hundreds or thousands of cores.

To further darken the picture, several types of errors need to be considered when computing at scale. In addition to classical fail-stop errors (such as hardware failures), silent errors (a.k.a silent data corruptions) must be taken into account. Contrary to fail-stop failures, silent errors are not detected immediately, but instead after some arbitrary *detection latency*, which complicates methods to cope with them. See Sect. 1.6 for more details.

[1]See Sect. 1.3.2.1 for a detailed explanation on how these values (9 h or 53 min) are computed.

1.1.2 Faults and Failures

There are many types of errors, faults, or failures. Some are transient, others are unrecoverable. Some cause a fatal interruption of the application as soon as they strike, others may corrupt the data in a silent way and will manifest only after an arbitrarily long delay. We refer to Chap. 2 for a detailed classification and analysis of error sources.

In this chapter, we mainly deal with *fail-stop failures*, which are unrecoverable failures that interrupt the execution of the application. These include all hardware faults, and some software ones. We use the terms *fault* and *failure* interchangeably. Again, silent errors are addressed at the end of the chapter, in Sect. 1.6.

Regardless of the fault type, the first question is to quantify the rate or frequency at which these faults strike. For 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 nondeterministic, meaning that they vary from one execution to another. To model this, we use I.I.D. (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 faults, X_3 is the delay between the second and third faults, 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 $Exp(\lambda)$.

In particular, each X_i has the same mean $\mathbb{E}(X_i) = \mu$. This amounts to say that, 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*: one can show (see Sect. 1.3.2.1 for a proof) that the expected number of faults $N_{\text{faults}}(T)$ that will strike during T seconds is such that

$$\lim_{T \to \infty} \frac{N_{\text{faults}}(T)}{T} = \frac{1}{\mu} \tag{1.1}$$

Why are Exponential distribution laws so important? This is because of their *memoryless* property, which writes: if $X \sim Exp(\lambda)$, then $\mathbb{P}(X \geq t + s \mid X \geq s) = \mathbb{P}(X \geq t)$ 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, $\text{RATE}(t)$, is defined as the (instantaneous) rate of fault for the survivors to time t, during the next instant of time:

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

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 *aging* 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 aging, 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 $\mu = \frac{1}{\lambda}$.

Weibull distributions are a good example of probability distributions that account for infant mortality, and they are widely used to model failures on computer platforms [39, 42, 43, 54, 67]. The definition of $Weibull(\lambda)$, the Weibull distribution law of shape parameter k and scale parameter λ, goes as follows:

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

If $k = 1$, we retrieve an Exponential distribution $Exp(\lambda)$ and the failure rate is constant. But if $k < 1$, the failure rate decreases with time, and the smaller k, the more important the decreasing. Values used in the literature are $k = 0.7$ or $k = 0.5$ [39, 54, 67].

1.2 Checkpoint and Rollback Recovery

Designing a fault-tolerant system can be done at different levels of the software stack. We call *general-purpose* the approaches that detect and correct the failures at a given level of that stack, masking them entirely to the higher levels (and ultimately to the end-user, who eventually see a correct result, despite the occurrence of failures). General-purpose approaches can target specific types of failures (e.g., message loss, or message corruption), and let other types of failures hit higher levels of the software stack. In this section, we discuss a set of well-known and recently developed protocols to provide general-purpose fault tolerance for a large set of failure types, at different levels of the software stack, but always below the application level.

These techniques are designed to work in spite of the application behavior. When developing a general-purpose fault-tolerant protocol, two adversaries must be taken into account: the occurrence of failures, that hit the system at unpredictable moments, and the behavior of the application, that is designed without taking into account the risk of failure, or the fault-tolerant protocol. All general-purpose fault tolerance technique rely on the same idea: introduce automatically computed redundant information, and use this redundancy to mask the occurrence of failures to the higher level application.

The general-purpose technique most widely used in HPC relies on checkpointing and rollback recovery: parts of the execution are lost when processes are subject to failures (either because the corresponding data is lost when the failure is a crash, or because it is corrupted due to a silent error), and the fault-tolerant protocol, when catching such errors, uses past checkpoints to restore the application in a consistent state, and recomputes the missing parts of the execution. We first discuss the techniques available to build and store process checkpoints, and then give an overview of the most common protocols using these checkpoints in a parallel application.

1.2.1 Process Checkpointing

The goal of process checkpointing is to save the current state of a process. In current HPC applications, a process consists of many user-level or system-level threads, making it a parallel application by itself. Process checkpointing techniques generally use a coarse-grain locking mechanism to interrupt momentarily the execution of all the threads of the process, giving them a global view of its current state, and reducing the problem of saving the process state to a sequential problem.

Independently of the tool used to create the checkpoint, we distinguish three parameters to characterize a process checkpoint:

- At what level of the software stack it is created;
- How it is generated;
- How it is stored.

Level of the software stack. Many process checkpointing frameworks are available: they can rely on an operating system extension [41], on dynamic libraries,[2] on compilers [50, 53, 62, 63], on a user-level API [5], or on a user-defined routine that will create an application-specific checkpoint [47]. The different approaches provide different levels of transparency and efficiency. At the lowest level, operating system extensions, like BLCR [41], provide a completely transparent checkpoint of the whole process. Such a checkpoint can be restored on the same hardware, with the same software environment (operating system, dynamic libraries, etc.). Since the entire state is saved (from CPU registers to the virtual memory map), the function call stack is also saved and restored automatically. From a programmatic point of view, the checkpoint routine returns with a different error code, to let the caller know if this calls returns from a successful checkpoint or from a successful restart.

[2] See https://code.google.com/p/cryopid/.

System-level checkpointing requires to save the entire memory (although an API allows to explicitly exclude pages from being saved into the checkpoint, in which case the pages are reallocated at restore time, but filled with 0), and thus the cost of checkpointing is directly proportional to the memory footprint of the process. The checkpoint routine is entirely preemptive: it can be called at any point during the execution, from any thread of the application (as long as another thread is not inside the checkpoint routine already).

At the highest level, user-defined application-specific routines are functions that a fault-tolerant protocol can call, to create a serialized view of the application, that another user-defined application-specific routine can load to restore a meaningful state of the process. Such an approach does not belong to general-purpose techniques, since it is application dependent. It is worth noting, however, that some resilient communication middleware propose this option to implement an efficient generic rollback-recovery protocol at the parallel application level. Indeed, as we will see later in the chapter, time to checkpoint is a critical parameter of the overall efficiency of a rollback-recovery technique. User-defined process checkpoints are often orders of magnitude smaller than the process memory footprint, because intermediary data, or data that is easily reconstructed from other critical data, do not need to be saved. User-defined checkpoints also benefit from a more diverse use than solely fault tolerance: they allow to do a post-mortem analysis of the application progress; they permit to restart the computation at intermediary steps, and change the behavior of the application. For these reasons, many applications provide such routines, which is the reason why fault-tolerant protocols try to also benefit from them. It is however difficult to implement a preemptive user-defined routine, capable of saving the process state at any time during the execution, which makes the use of such approach sometimes incompatible with some parallel application resilient protocols that demand to take process checkpoints at arbitrary times.

A note should be made about opened files: most existing tools to checkpoint a process do not provide an automatic way to save the content of the files opened for writing at the time of checkpoint. Files that are opened in read mode are usually reopened at the same position during the restoration routine; files that are opened in append mode can be easily truncated where the file pointer was located at the time of checkpoint during the restore; files that are opened in read/write mode, however, or files that are accessed through a memory map in read/write, must be copied at the time of checkpoint, and restored at the time of rollback. Among the frameworks that are cited above, none of them provide an automatic way of restoring the files, which remains the responsibility of the resilient protocol implementation.

How checkpoints are generated. The checkpoint routine, provided by the checkpointing framework, is usually a blocking call that terminates once the serial file representing the process checkpoint is complete. It is often beneficial, however, to be able to save the checkpoint in memory, or to allow the application to continue its progress in parallel with the I/O intensive part of the checkpoint routine. To do so, generic techniques, like process duplication at checkpoint time can be used, if enough memory is available on the node: the checkpoint can be made asynchronous by duplicating the entire process, and letting the parent process continue its execution, while the child process checkpoints and exits. This technique relies on the copy-on-write pages duplication capability of modern operating systems to ensure that if the parent process modifies a page, the child will get its own private copy, keeping the state of the process at the time of entering the checkpoint routine. Depending on the rate at which the parent process modifies its memory, and depending on the amount of available physical memory on the machine, overlapping between the checkpoint creation and the application progress can thus be achieved, or not.

How checkpoints are stored. A process checkpoint can be considered as completed once it is stored in a non-corruptible space. Depending on the type of failures considered, the available hardware, and the risk taken, this non-corruptible space can be located close to the original process, or very remote. For example, when dealing with low probability memory corruption, a reasonable risk consists of simply keeping a copy of the process checkpoint in the same physical memory; at the other extreme, the process checkpoint can be stored in a remote redundant file system, allowing any other node compatible with such a checkpoint to restart the process, even in case of machine shutdown. Current state-of-the-art libraries provide transparent multiple storage points, along a hierarchy of memory: [57], or [5], implement in-memory double-checkpointing strategies at the closest level, disk-less checkpointing, NVRAM checkpointing, and remote file system checkpointing, to feature a complete collection of storage techniques. Checkpoint transfers happen asynchronously in the background, making the checkpoints more reliable as transfers progress.

1.2.2 Coordinated Checkpointing

Distributed checkpointing protocols use process checkpointing and message passing to design rollback-recovery procedures at the parallel application level. Among them the first approach was proposed in 1984 by Chandy and Lamport, to build a possible global state of a distributed system [20]. The goal of this protocol is to build a consistent distributed snapshot of the distributed system. A distributed snapshot is a collection of process checkpoints (one per process), and a collection of in-flight messages (an ordered list of messages for each point to point channel). The protocol assumes ordered loss-less communication channel; for a given application, messages can be sent or received after or before a process took its checkpoint. A message from process p to process q that is sent by the application after the checkpoint of process p but received before process q checkpointed is said to be an *orphan* message. Orphan messages must be avoided by the protocol, because they are going to be regenerated by the application, if it were to restart in that snapshot. Similarly, a message from process p to process q that is sent by the application before the checkpoint of process p but received after the checkpoint of process q is said to be *missing*. That message must belong to the list of messages in channel p to q, or the snapshot is inconsistent. A snapshot that includes no orphan message, and for which all the saved channel

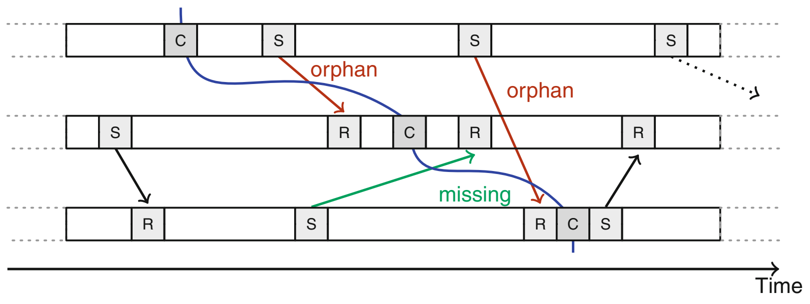


Fig. 1.1 Orphan and missing messages

messages are missing messages is consistent, since the application can be started from that state and pursue its computation correctly.

To build such snapshots, the protocol of Chandy and Lamport works as follows (see Fig. 1.1): any process may decide to trigger a checkpoint wave by taking its local process checkpoint (we say the process entered the checkpoint wave), and by notifying all other processes to participate to this wave (it sends them a notification message). Because channels are ordered, when a process receives a checkpoint wave notification, it can separate what messages belong to the previous checkpoint wave (messages received before the notification in that channel), and what belong to the new one (messages received after the notification). Messages that belong to the current checkpoint wave are appended to the process checkpoint, to complete the state of the distributed application with the content of the in-flight messages, during the checkpoint. Upon reception of a checkpoint wave notification for the first time, a process takes it local checkpoint, entering the checkpoint wave, and notifies all others that it did so. Once a notification per channel is received, the local checkpoint is complete, since no message can be left in flight, and the checkpoint wave is locally complete. Once all processes have completed their checkpoint wave, the checkpoint is consistent, and can be used to restart the application in a state that is consistent with its normal behavior.

Different approaches have been used to implement this protocol. They are discussed in detail in the case of the Message Passing Interface (MPI) in Chap. 3. The main difference is on how the content of the (virtual) communication channels is saved. A simple approach, called Blocking Coordinated Checkpointing, consists in delaying the emission of application messages after entering the checkpointing wave, and moving the process checkpointing at the end of that wave, when the process is ready to leave it (see Fig. 1.3). That way, the state of communication channels is saved within the process checkpoint itself, at the cost of delaying the execution of the application. The other approach, called Non-Blocking Coordinated Checkpointing, is a more straightforward implementation of the algorithm by Chandy and Lamport: in-flight messages are added, as they are discovered, in the process checkpoint of the receiver, and reinjected in order in the "unexpected" messages queues, when loading the checkpoint (see Fig. 1.2).

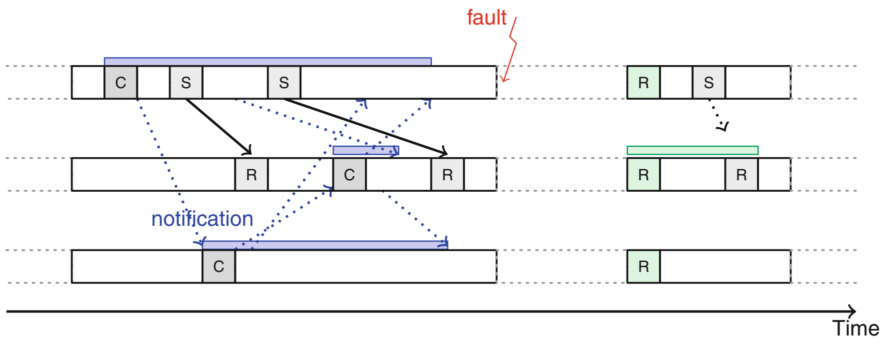


Fig. 1.2 Non-blocking coordinated rollback recovery protocol

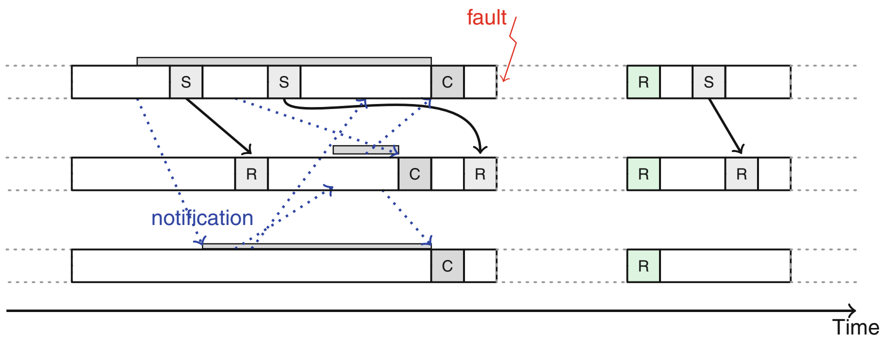


Fig. 1.3 Blocking coordinated rollback recovery protocol

At the application level, resilient application developers have often taken a very simple approach to ensure the consistency of the snapshot: since the protocol is designed knowing the application, a couple of synchronizing barriers can be used, before and after taking the process checkpoints, to guarantee that no application in-flight messages are present at the time of triggering the checkpoint wave, and thus the causal ordering of communications inside the application is used to avoid the issue entirely.

1.2.3 Uncoordinated Checkpointing

Blocking or non-blocking, the coordinated checkpointing protocols require that all processes rollback to the last valid checkpoint wave, when a failure occurs. This ensures a global consistency, at the cost of scalability: as the size of the system grows, the probability of failures increase, and the minimal cost to handle such failures also increase. Indeed, consider only the simple issue of notifying all processes that a rollback is necessary: this can hardly be achieved in constant time, independent of the number of living processes in the system. Chapter 3 will present in further details how uncoordinated checkpointing can be implemented in an MPI library (see Sect. 3.2), but we present here the general approach to compare it with the other protocols.

To reduce the inherent costs of coordinated checkpointing, uncoordinated checkpointing protocols have thus been proposed. On the failure-free part of the execution, the main idea is to remove the coordination of checkpointing, targeting a reduction of the I/O pressure when checkpoints are stored on shared space, and the reduction of delays or increased network usage when coordinating the checkpoints. Furthermore, uncoordinated protocols aim at forcing the restart of a minimal set of processes when a failure happens. Ideally, only the processes subject to a failure should be restarted. However, this requires additional steps.

Consider, for example, a naive protocol, that will let processes checkpoint their local state at any time, without coordination, and in case of failures will try to find a consistent checkpoint wave (in the sense of the Chandy-Lamport algorithm) from a set of checkpoints taken at random times. Even if we assume that all checkpoints are kept until the completion of the execution (which is unrealistic from a storage point of view), finding a consistent wave from random checkpoints might prove impossible, as illustrated by Fig. 1.4. Starting from the last checkpoint (C_1) of process p, all possible waves that include checkpoint C_2 of process q will cross the message m, thus creating another missing message. It is thus necessary to consider a previous checkpoint for p. But all waves including the checkpoint C_3 for p and the checkpoint C_2 for q will cross the message m', creating a missing message. A previous checkpoint must thus be considered for q. This effect, that will invalidate all checkpoint taken randomly, forcing the application to restart from scratch, is called the *domino* effect. To avoid it, multiple protocols have been considered, taking additional assumptions about the application into account.

1.2.3.1 Piecewise Deterministic Assumption

One such assumption is the Piecewise Deterministic Assumption (PWD). It states that a sequential process is an alternate sequence of a nondeterministic choices followed by a set of deterministic steps. As such, the PWD is not really an assumption: it

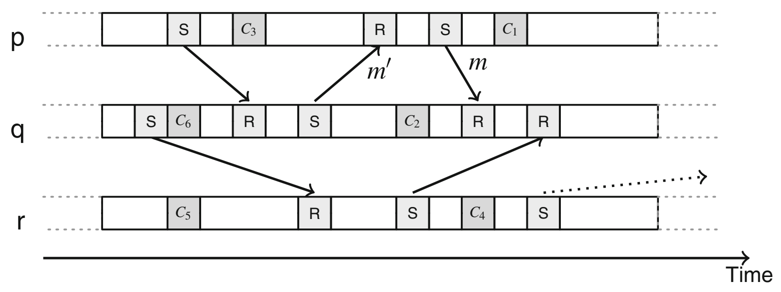


Fig. 1.4 Optimistic uncoordinated protocol: Illustration of the *domino* effect

is a way to describe a possible execution of a sequential program. The assumption resides in the fact that these nondeterministic choices can be captured and their effect replayed. Thus, under the PWD assumption, the behavior of a sequential process can be entirely guided from a given state to another deterministic state by forcing each nondeterministic choice between these two states.

Translated in the HPC world, and especially under the Message Passing Interface (MPI) paradigm, the sources of nondeterminism are rather small. Indeed, all actions that depend upon the input data (environment or user options) are not nondeterministic only in the sense of the PWD assumption: starting from the same state, the same action will follow. Pseudo-random generators fall also in this category of deterministic actions. So, in an MPI application, the only source of nondeterminism comes from time-sensitive decisions, point-to-point message reception order (and request completion order), and related actions (like probe). All these actions are captured by the MPI library (assuming the program relies only on MPI routines to measure time, if its state is time dependent), that is also capable of replaying any value that was returned by a previous call.

In most modern architectures, processes whose state depend on timing have nondeterministic actions, since with modern CPUs and network, an instruction can take a varying time, depending on the actions of other processes sharing the machine, or the operating system, and a misplaced message reception can change significantly this timing measurement. Many MPI operations have a deterministic behavior (e.g., sending a message does not change the state of the sending process; participating to a broadcast operation, seen as an atomic operation, will have a deterministic effect on the state of all processes participating to it, etc...). However, MPI allows the programmer to reorder message receptions, or to not specify an order on the messages reception (using wildcard reception tags, like MPI_ANY_TAG, or MPI_ANY_SOURCE), that enables the library to deliver the messages in an order that is the most efficient, and thus execution-dependent. These actions are then necessarily nondeterministic, since the state of the process between such two receptions depends on what reception actually happened.

Then, consider a parallel application built of sequential processes that use MPI to communicate and synchronize. In case of failure, by replaying the sequence of messages and test/probe with the same result that the process that failed obtained in the initial execution (from the last checkpoint), one can guide the execution of a process to its exact state just before the failure.

1.2.3.2 Message Logging

This leads to the concept of Message Logging (ML). The goal of message logging, in this context, is to provide a tool to capture and replay the most frequent of nondeterministic events: message receptions. To be able to reproduce a message reception, one needs to deliver it in the right order, and with the appropriate content. Message logging thus features two essential parts: a log of the event itself, and a log of the content of the message.

Events Identifiers. Events are usually identified by a few counters: based on the same idea as logical clocks of Lamport [52], these identifiers define a partial order that is sufficient to ensure the consistency of the distributed system by capturing the causality of the nondeterministic events. In most implementations, a nondeterministic message identifier consists of a 4-tuple: identifier of the message emitter, sequence number of emission in that channel, identifier of the message receiver, sequence number of delivery of that message.

The first two counters uniquely identify an outgoing message at the sender. They are used to connect that event identifier with the corresponding payload log. The second two counters make the delivery deterministic. They can only be assigned once the message is delivered by the receiver during the first execution.

A collection of event logs builds the history of a distributed application. If all event logs with the same message receiver identifier are considered, the execution of the receiver is made deterministic up to the end of the log: that process knows exactly what messages it must receive, and in which order they must be delivered.

In some applications, other nondeterministic events may be interleaved between message receptions, and the global ordering of these events on that process must be kept (as well as all information needed to replay these events). For example, in the MPI case, the evaluation of a routine like `MPI_Probe()` is nondeterministic: the routine will return `true` or `false` depending upon the internal state of the library, that depends itself upon the reception of messages. A simple event logging strategy is to remember the return value of each `MPI_Probe()`, associated with an internal event sequence number, to augment the message log with the same internal event sequence number to remember the global ordering of process-specific internal events, and to store these events in the same place as the message logs. To replay the execution, one then needs to have these routines return the same value as during the initial execution, whatever the internal state of the library, and deliver the messages in the order specified by the history. As a result, the library may have to introduce delays, reorder messages, or wait for the arrival of messages that were supposed to be delivered but are not available yet. But the process will be guided to the exact state it reached when the log was interrupted, which is the goal of message logging.

Payload Logging. To deliver messages in replay mode, the receiving process needs to have access to the message payload: its event log is not sufficient. The most widely used approach to provide this payload is to keep a copy at the sender. This is called sender-based message logging (although this is a slight abuse of language, as events can be stored at a separate place different from the sender).

The advantage of sender-based payload logging is that the local copy can be made in parallel with the network transfer, trying to minimize the impact on a failure-free execution. Its main drawback is its usage of node memory. The amount of message payload log is a function of the message throughput of the application, and memory can be exhausted quickly, so, a sender-based payload logging protocol must feature mechanisms for control flow and garbage collection.

To understand how the garbage collection mechanism works, one needs to understand first that the sender-based payload log belongs to the state of the sender process:

at any point, a receiver process may request to send back the content of previously sent messages. If the sender process was subject to a failure, and restarted somewhere in its past, it still may need to provide the payload of messages that were sent even further back in its history. Hence, when taking independent checkpoints, the message payload log must be included in the process checkpoint, as any other element of the process state.

Checkpoints, however, provide a guarantee to senders: when a receiver checkpoints, all the processes that sent it messages have the guarantee that the payload of messages delivered before that checkpoint will never be requested again. They can thus be removed from the state of the process, creating a trade-off between processes: taking a checkpoint of a process will relieve memory of processes that sent messages to it, while imposing to save all the data sent by it. In the worst case, memory can become exhausted, and remote checkpoints of sender processes must be triggered before more messages can be sent and logged by processes.

Event Logging. The last element of a Message Logging strategy has probably been the most studied: how to log the events. As described above, to replay its execution, a process needs to collect the history of all events between the restore point and the last nondeterministic event that happened during the initial execution. Since the memory of the process is lost when it is hit by a failure, this history must be saved somewhere. There are three main strategies to save the events log, called optimistic, pessimistic, and causal.

Optimistic message logging consists in sending the history to a remote event logger. That event logger must be a reliable process, either by assumption (the risk that the failure hits that specific process is inversely proportional to the number of processes in the system), or through replication. The protocol is said optimistic because while event logs are in transfer between the receiver process (that completed the event identifier when it delivered the message to the application) and the event logger, the application may send messages, and be subject to a failure.

If a failure hits the application precisely at this time, the event log might be lost. However, the message that was just sent by the application might be correctly received and delivered anyway. That message, its content, its existence, might depend on the reception whose log was lost. During a replay, the process will not find the event log, and if that reception was nondeterministic, might make a different choice, sending out a message (or doing another action), inconsistent with the rest of the application state.

The natural extension to optimistic message logging is pessimistic message logging: when a process does a nondeterministic action (like a reception), it sends the event log to the event logger, and waits for an acknowledge of logging from the event logger before it is allowed to take any action that may impact the state of the application. This removes the race condition found in optimistic message logging protocols, to the cost of introducing delays in the failure-free execution, as the latency of logging safely the event and waiting for the acknowledge must be added to every nondeterministic event.

To mitigate this issue, causal event logging protocols were designed: in a causal event logging protocol, messages carry part of the history of events that lead to their emission. When a process does a nondeterministic action, it sends the event log to the event logger, appends it to a local history slice, and without waiting for an acknowledge, continues its execution. If an acknowledge comes before any message is sent, that event log is removed from the local history slice. If the process sends a message, however, the local history slice is piggybacked to the outgoing message. That way, at least the receiving process knows of the events that may not be logged and that lead to the emission of this message.

The history slice coming with a message must be added to the history slice of a receiver process, since it is part of the history to bring the receiving process in its current state. This leads to a snowballing effect, where the local history slice of processes grows with messages, and the overhead on messages also grows with time. Multiple strategies have been devised to bound that increase, by garbage collecting events that are safely logged in the event logger from all history slices, and by detecting cycles in causality to trim redundant information from these slices.

Uncoordinated Checkpointing with Message Logging and Replay. Putting all the pieces together, all uncoordinated checkpointing with message logging and replay protocols behave similarly: processes log nondeterministic events and messages payload as they proceed along the initial execution; without strong coordination, they checkpoint their state independently; in case of failure, the failed process restarts from its last checkpoint, it collects all its log history, and enters the replay mode. Replay consists in following the log history, enforcing all nondeterministic events to produce the same effect they had during the initial execution. Message payload must be re-provided to this process for this purpose. If multiple failures happen, the multiple replaying processes may have to reproduce the messages to provide the payload for other replaying processes, but since they follow the path determined by the log history, these messages, and their contents, will be regenerated as any deterministic action. Once the history has been entirely replayed, by the piecewise deterministic assumption, the process reaches a state that is compatible with the state of the distributed application, that can continue its progress from this point on.

1.2.4 Hierarchical Checkpointing

Over modern architectures, that feature many cores on the same computing node, message logging becomes an unpractical solution. Indeed, any interaction between two threads introduces the potential for a nondeterministic event that must be logged. Shared memory also provides an efficient way to implement zero copy communication, and logging the payload of such "messages" introduces a high overhead that make this solution intractable.

In fact, if a thread fails, current operating systems will abort the entire process. If the computing node is subject to a hardware failure, all processes running on that machine fail together. Failures are then often tightly correlated, forcing all processes / threads running on that node to restart together because they crashed together. These two observations lead to the development of Hierarchical Checkpointing Protocols. Hierarchical Checkpointing tries to combine coordinated checkpoint and rollback together with uncoordinated checkpointing with message logging, keeping the best of both approaches.

The idea of Hierarchical Checkpointing is rather simple: processes are distributed in groups; processes belonging to the same group coordinate their checkpoints and rollbacks; uncoordinated checkpointing with message logging is used between groups. However, the state of a single process depends upon the interactions between groups, but also upon the interactions with other processes inside the group. Coordinated rollback guarantees that the application restarts in a consistent state; it does not guarantee that the application, if restarting from that consistent state, will reach the same state as in the initial execution, which is a condition for uncoordinated checkpointing to work. A nondeterministic group (a group of processes whose state depend upon the reception order of messages exchanged inside the group for example) cannot simply restart from the last group-coordinated checkpoint and hope that it will maintain its state globally consistent with the rest of the application.

Thus, Hierarchical Checkpointing Protocols remain uncoordinated checkpointing protocols with message logging: nondeterministic interactions between processes of the same group must be saved, but the message payload can be spared, because all processes of that group will restart and regenerate the missing message payloads, if a failure happens. Section 3.6 presents in deeper details how a specific hierarchical protocol works. In this overview, we introduce a general description of hierarchical protocols to allow for a model-based comparison of the different approaches.

Reducing the logging. There are many reasons to reduce the logging (events and payload): intragroup interactions are numerous, and treating all of them as nondeterministic introduces significant computing slowdown if using a pessimistic protocol, or memory consumption and message slowdown if using a causal protocol; intergroup interactions are less sensitive to event logging, but payload logging augments the checkpoint size, and consumes user memory.

Over the years, many works have proposed to integrate more application knowledge in the fault-tolerant middleware: few HPC applications use message ordering or timing information to take decisions; many receptions in MPI are in fact deterministic, since the source, tag, type and size, and the assumption of ordered transmission in the virtual channel make the matching of messages unique from the application level. In all these cases, logging can be avoided entirely. For other applications, although the reception is nondeterministic, the ordering of receptions will temporarily influence the state of the receiving process, but not its emissions. For example, this happens in a reduce operation written over point to point communications: if a node in the reduction receives first from its left child then from its right one, or in the other order, the state of the process after two receptions stays the same, and the message it sends up to its parent is always the same. Based on this observation, the concept of send determinism has been introduced [36], in which many events may be avoided to log.

MPI provides also a large set of collective operations. Treating these operations at the point-to-point level introduces a lot of nondeterminism, while the high-level operation itself remains deterministic. This fact is used in [14] to reduce the amount of events log.

Hierarchical Checkpointing reduces the need for coordination, allowing a load balancing policy to store the checkpoints; size of the checkpoints, however are dependent on the application message throughput and checkpointing policy (if using sender-based payload logging, as in most cases); the speed of replay, the overhead of logging the events (in message size or in latency) are other critical parameters to decide when a checkpoint must done.

In the following section, we discuss how the different checkpointing protocols can be optimized by carefully selecting the interval between checkpoints. To implement this optimization, it is first necessary to provide a model of performance for these protocols.

1.3 Probabilistic Models for Checkpointing

This section deals with probabilistic models to assess the performance of various checkpointing protocols. We start with the simplest scenario, with a single resource, in Sect. 1.3.1, and we show how to compute the optimal checkpointing *period*. Section 1.3.2 shows that dealing with a single resource and dealing with coordinated checkpointing on a parallel platform are similar problems, provided that we can compute the MTBF of the platform from that of its individual components. Section 1.3.3 deals with hierarchical checkpointing. Things get more complicated, because many parameters must be introduced in the model to account for this complex checkpointing protocol. Finally, Sect. 1.3.4 provides a model for in-memory checkpointing, a variant of coordinated checkpointing where checkpoints are kept in the memory of other processors rather than on stable storage, in order to reduce the cost of checkpointing.

1.3.1 Checkpointing with a Single Resource

We state the problem formally as follows. Let $\text{TIME}_{\text{base}}$ be the base time of the application, without any overhead (neither checkpoints nor faults). Assume that the resource is subject to faults with MTBF μ. Note that we deal with arbitrary failure distributions here, and only assume knowledge of the MTBF.

The time to take a checkpoint is C seconds (the time to upload the checkpoint file onto stable storage). We say that the checkpointing period is T seconds when a checkpoint is done each time the application has completed $T - C$ seconds of work. When a fault occurs, the time between the last checkpoint and the fault is lost. This includes useful work as well as potential fault tolerance techniques. After the fault,

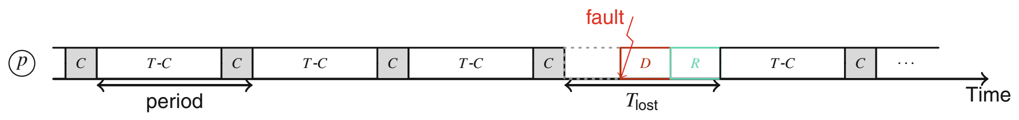


Fig. 1.5 An execution

there is a *downtime* of D seconds to account for the temporary unavailability of the resource (for example rebooting, or migrating to a spare). 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 (*e.g.*, the checkpoint file is read from stable storage). 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 Fig. 1.5.

To avoid introducing several conversion parameters, all model parameters are expressed in seconds. The failure inter-arrival times, the duration of a downtime, checkpoint, or recovery are all expressed in seconds. Furthermore, we assume (without loss of generality) that one work unit is executed in one second. One work-unit may correspond to any relevant application-specific quantity.

The difficulty of the problem is to trade-off between the time spent checkpointing, and the time lost in case of a fault. Let $\text{TIME}_{\text{final}}(T)$ be the expectation of the total execution time of an application of size $\text{TIME}_{\text{base}}$ 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* because 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 ratio 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). There are two sources of waste, which we analyze below.

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

$$\text{TIME}_{\text{FF}} = \frac{T}{T-C} \text{TIME}_{\text{base}}. \tag{1.2}$$

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

$$\text{WASTE}_{\text{FF}} = \frac{\text{TIME}_{\text{FF}} - \text{TIME}_{\text{base}}}{\text{TIME}_{\text{FF}}} \Leftrightarrow \left(1 - \text{WASTE}_{\text{FF}}\right)\text{TIME}_{\text{FF}} = \text{TIME}_{\text{base}}. \tag{1.3}$$

Combining Eqs. (1.2) and (1.3), we get:

$$\text{WASTE}_{\text{FF}} = \frac{C}{T}. \tag{1.4}$$

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 $\text{TIME}_{\text{final}}$ 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 Fig. 1.6. The first part of the execution time is equal to TIME_{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,

$$\text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} + N_{\text{faults}} T_{\text{lost}}. \tag{1.5}$$

In average, during a time $\text{TIME}_{\text{final}}$, $N_{\text{faults}} = \frac{\text{TIME}_{\text{final}}}{\mu}$ faults happen (recall Eq. (1.1)). We need to estimate T_{lost}. A natural estimation for the moment when the fault strikes in the period is $\frac{T}{2}$ (see Fig. 1.5). Intuitively, faults strike anywhere in the period, hence in average they strike in the middle of the period. The proof of this result for Exponential distribution laws can be found in [25]. We conclude that

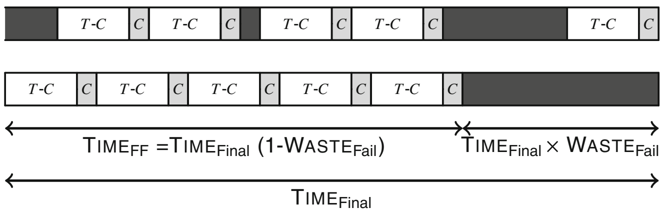


Fig. 1.6 An execution (*top*), and its reordering (*bottom*), to illustrate both sources of waste. Blackened intervals correspond to time lost due to faults: downtime, recoveries, and re-execution of work that has been lost

$T_{\text{lost}} = \frac{T}{2} + D + R$, because after each fault there is a downtime and a recovery. This leads to:

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

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

$$\text{WASTE}_{\text{fault}} = \frac{\text{TIME}_{\text{final}} - \text{TIME}_{\text{FF}}}{\text{TIME}_{\text{final}}} \Leftrightarrow \left(1 - \text{WASTE}_{\text{fault}}\right)\text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}}$$

We derive:

$$\text{WASTE}_{\text{fault}} = \frac{1}{\mu}\left(D + R + \frac{T}{2}\right). \tag{1.6}$$

Equations (1.4) and (1.6) show that each source of waste calls for a different period: a large period for WASTE_{FF}, as already discussed, but a small period for $\text{WASTE}_{\text{fault}}$, 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

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

Altogether, we derive the final result:

$$\begin{aligned}\text{WASTE} &= \text{WASTE}_{\text{FF}} + \text{WASTE}_{\text{fault}} - \text{WASTE}_{\text{FF}}\text{WASTE}_{\text{fault}} \quad (1.7)\\ &= \frac{C}{T} + \left(1 - \frac{C}{T}\right)\frac{1}{\mu}\left(D + R + \frac{T}{2}\right). \quad (1.8)\end{aligned}$$

The two sources of waste do not add up, but we have:

$$(1 - \text{WASTE}) = (1 - \text{WASTE}_{\text{FF}})(1 - \text{WASTE}_{\text{fault}}),$$

just as for discount percentages in a sale: two successive 50 % rebates do not make the product free, but the final price reduction is the product of the two successive ones.

We obtain $\textsc{Waste} = \frac{u}{T} + v + wT$, where $u = C\left(1 - \frac{D+R}{\mu}\right)$, $v = \frac{D+R-C/2}{\mu}$, 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:

$$T_{\text{FO}} = \sqrt{2(\mu - (D+R))C}. \tag{1.9}$$

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

$$\textsc{Waste}_{\text{FO}} = \sqrt{\frac{2C}{\mu}\left(1 - \frac{D+R}{\mu}\right)} + \frac{D+R-C/2}{\mu}. \tag{1.10}$$

In 1974, Young [69] obtained a different formula, namely $T_{\text{FO}} = \sqrt{2\mu C} + C$. Thirty years later, Daly [25] refined Young's formula and obtained $T_{\text{FO}} = \sqrt{2(\mu + R)C} + C$. Equation (1.9) 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 $T_{\text{FO}} = \sqrt{2\mu C}$ when μ is large in front of the resilience parameters D, C and R. Below, we show that this latter condition is the key to the accuracy of the approximation.

First-order approximation of T_{FO}. It is interesting to point out why the value of T_{FO} given by Eq. (1.9) 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 $T_{\text{lost}} = \frac{T}{2} + D + R$. 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 Eq. (1.4), we have to enforce $C \leq T$ in order to have $\textsc{Waste}_{\text{FF}} \leq 1$.
- In Eq. (1.6), we have to enforce $D + R \leq \mu$ in order to have $\textsc{Waste}_{\text{fault}} \leq 1$. In addition, we must cap the period to enforce this latter constraint. Intuitively, we need μ to be large enough for Eq. (1.6) to make sense (see the word of caution at the end of Sect. 1.3.2.1).
- Equation (1.6) 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 $T \leq \eta\mu$, where η 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 $\pi = P(X \geq 2) = 1 - (P(X = 0) + P(X = 1)) = 1 - (1 + \beta)e^{-\beta}$. To get $\pi \leq 0.03$, we can choose $\eta = 0.27$, providing a valid approximation when bounding the period range accordingly. Indeed, with such a conservative value for η, 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: $C \leq \eta\mu$ and $D + R \leq \eta\mu$. However, enforcing these constraints may lead to use a suboptimal period: it may well be the case that the optimal period $\sqrt{2(\mu - (D + R))C}$ of Eq. (1.9) does not belong to the admissible interval $[C, \eta\mu]$. In that case, the waste is minimized for one of the bounds of the admissible interval. This is because, as seen from Eq. (1.8), 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 [4] show that actual job executions can always use the value from Eq. (1.9), accounting for multiple faults whenever they occur by re-executing the work until success. The first-order model turns out to be surprisingly robust!

Let us formulate our main result as a theorem:

Theorem 1.1 *The optimal checkpointing period is* $T_{\text{FO}} = \sqrt{2\mu C} + o(\sqrt{\mu})$ *and the corresponding waste is* $\text{WASTE}_{\text{FO}} = \sqrt{\frac{2C}{\mu}} + o(\sqrt{\frac{1}{\mu}})$.

Theorem 1.1 has a wide range of applications. We discuss several of them in the following sections. Before that, we explain how to compute the optimal period accurately, in the special case where failures follow an Exponential distribution law.

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

Proposition 1.1

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

Proof For simplification, we write TIME instead of $\text{TIME}(T - C, C, D, R, \lambda)$ 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 T_{lost}), and the time spent for downtime and recovery (accounted for by variable T_{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:

$$\text{TIME} = \begin{cases} T & \text{if there is no fault} \\ T_{\text{lost}} + T_{\text{rec}} + \text{TIME} & \text{otherwise} \end{cases} \tag{1.11}$$

T_{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 Fig. 1.5 is $T_{\text{lost}} - D - R$, because for simplification T_{lost} and T_{rec} are not distinguished in that figure).

T_{rec} represents the amount of time needed by the system to recover from the fault (the corresponding value in Fig. 1.5 is $D + R$).

The expectation of TIME can be computed from Eq. (1.11) by weighting each case by its probability to occur:

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

which simplifies into:

$$\mathbb{E}(T) = T + (e^{\lambda T} - 1)\,(E(T_{\text{lost}}) + E(T_{\text{rec}})) \tag{1.12}$$

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

$$\mathbb{E}(T_{\text{lost}}) = \frac{1}{\lambda} - \frac{T}{e^{\lambda T} - 1} \tag{1.13}$$

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

$$\mathbb{E}(T_{\text{rec}}) = e^{-\lambda R}(D + R) + (1 - e^{-\lambda R})(D + \mathbb{E}(R_{lost}) + \mathbb{E}(T_{\text{rec}}))$$

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

Eq. (1.13), we obtain $\mathbb{E}(R_{lost}) = \frac{1}{\lambda} - \frac{R}{e^{\lambda R}-1}$. The expression for $\mathbb{E}(T_{\text{rec}})$ simplifies to

$$\mathbb{E}(T_{\text{rec}}) = De^{\lambda R} + \frac{1}{\lambda}(e^{\lambda R} - 1)$$

Plugging the values of $\mathbb{E}(T_{\text{lost}})$ and $\mathbb{E}(T_{\text{rec}})$ into Eq. (1.12) leads to the desired value:

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

Proposition 1.1 is the key to proving that the optimal checkpointing strategy (with an Exponential distribution of faults) 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 at the end. The expectation of the total execution time is the sum of the expectations of the time needed for each period. Proposition 1.1 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. Assuming k rational, one can find the optimal value k_{opt} by differentiation (and prove uniqueness using another differentiation). Unfortunately, we have to use the (implicit) Lambert function $\mathbb{L}$, defined as $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$), to express the value of k_{opt}, but we can always compute this value numerically. In the end, the optimal number of periods is either $\lfloor k_{opt} \rfloor$ or $\lceil k_{opt} \rceil$, thereby determining the optimal period T_{opt}. As a sanity check, the first-order term in the Taylor expansion of T_{opt} is indeed T_{FO}, which is kind of comforting. See [12] for all details.

1.3.2 *Coordinated Checkpointing*

In this section we introduce a simple model for coordinated checkpointing. Consider an application executing on a parallel platform with N processors, and using coordinated checkpointing for resilience. What is the optimal checkpointing period? We show how to reduce the optimization problem with N 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. When using coordinated checkpointing, 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

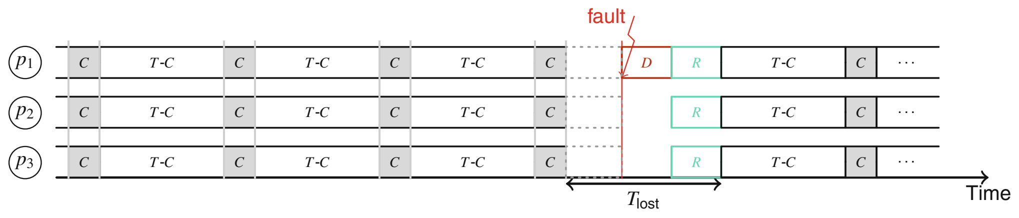


Fig. 1.7 Behavior for a tightly coupled application with coordinated checkpointing

messages are no longer available. This is why all processors have to recover from the last checkpoint and re-execute the work in parallel. On the contrary, with hierarchical checkpointing, only the group of the faulty processor must recover and re-execute (see Sect. 1.3.3 for a model of this complicated protocol).

Figure 1.7 provides an illustration of coordinated checkpointing. 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 is exactly the same pattern as with a single resource. We can see the whole platform as a single *super-processor*, very powerful (its speed is N times that of individual processors) but also very prone to faults: all the faults strike this super-processor! We can apply Theorem 1.1 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? The answer is given in the next section.

1.3.2.1 Platform MTBF

With Fig. 1.8, we see that the super-processor is hit by faults N times more frequently than the individual processors. We should then conclude that its MTBF is N times smaller than that of each processor. We state this result formally:

Proposition 1.2 *Consider a platform with N identical processors, each with MTBF μ_{ind}. Let μ be the MTBF of the platform. Then*

$$\mu = \frac{\mu_{ind}}{N} \tag{1.14}$$

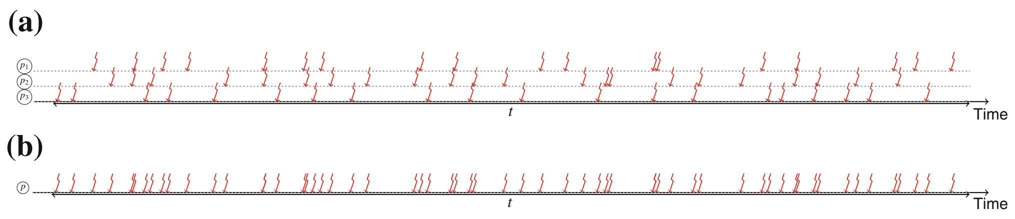


Fig. 1.8 Intuition of the proof of Proposition 1.2. **a** If three processors have around 20 faults during a time t ($\mu_{\text{ind}} = \frac{t}{20}$)... **b**...during the same time, the equivalent processor has around 60 faults ($\mu = \frac{t}{60}$)

Proof We first prove the proposition when the inter-arrival times of the faults on each individual processor are I.I.D. random variables with distribution $Exp(\lambda)$, where $\lambda = \frac{1}{\mu_{\text{ind}}}$. Recall that I.I.D. means *Independent and Identically Distributed*. In that simple case, the inter-arrival times of the faults on the super-processor are I.I.D. random variables with distribution $Exp(N\lambda)$, which proves that its MTBF is $\mu = \frac{\mu_{\text{ind}}}{N}$. To see this, the reasoning is the following:

- 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 $X_1^{(1)}$, $X_1^{(2)}$ …, $X_1^{(N)}$, 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 N random variables following an Exponential distribution $Exp(\lambda_i)$ is a random variable following an Exponential distribution $Exp(\sum_{i=1}^{N} \lambda_i)$ (see [64, 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 distribution of random variable for the occurrence of the first fault on processor P_2? The only new information if that P_2 has been alive for t seconds. The memoryless property states that the distribution of the arrival time of the first fault on P_2 is not changed at all 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 for exponential distributions.

We now give another proof of Proposition 1.2 that applies to any continuous probability distribution with bounded (nonzero) expectation, not just Exponential laws. Consider a single processor, say processor P_q. Let $X_i, i \geq 0$ denote the I.I.D. 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 $(n_q(F) - 1)$-th fault is the last one to happen strictly before time F, and the $n_q(F)$-th fault is the first to happen at time F or after. By definition of $n_q(F)$, we have

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

Using Wald's equation [64, p. 420], with $n_q(F)$ as a stopping criterion, we derive:

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

and we obtain:

$$\lim_{F \to +\infty} \frac{\mathbb{E}\left(n_q(F)\right)}{F} = \frac{1}{\mu_{\text{ind}}}. \tag{1.15}$$

Now consider a platform with N identical processors, whose fault inter-arrival times are I.I.D. 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 introduced above, are no longer I.I.D. The minimum trick used in the proof of Proposition 1.2 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 Eq. (1.15): this value μ must satisfy

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

where $n(F)$ be 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. Let Y_i, $i \geq 1$ denote the random variables for fault inter-arrival times on the super-processor. 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 $n(F) = \sum_{q=1}^{N} m_q(F)$. By definition, except for the component hit by the last fault, $m_q(F) + 1$ is the number of faults on component q until time F is exceeded, hence $n_q(F) = m_q(F) + 1$ (and this number is $m_q(F) = n_q(F)$ on the component hit by the last fault). From Eq. (1.15) 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 $n(F) = \sum_{q=1}^{N} m_q(F)$, we also have:

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

which answers both questions at the same time and concludes the proof.

Note that the random variables Y_i are not I.I.D., and they do not necessarily have the same expectation, which explains why we resort to Eq. (1.15) to define the MTBF of the super-processor. Another possible asymptotic definition of the MTBF μ of the platform could be given by the equation

$$\mu = \lim_{n\to+\infty} \frac{\sum_{i=1}^{n} \mathbb{E}\left(Y_i\right)}{n}.$$

Kella and Stadje (Theorem 4, [49]) prove that this limit indeed exists and that is also equal to $\frac{\mu_{\text{ind}}}{N}$, if in addition the distribution function of the X_i is continuous (a requirement always met in practice).

Proposition 1.2 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 Eq. (1.10)). 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 requirement.

We conclude this section with a word of caution: the formula $\mu = \frac{\mu_{\text{ind}}}{N}$ expresses the fact that the MTBF of a parallel platform will inexorably decrease as the number of its components increases, regardless how reliable each individual component could be. Mathematically, the expression of the waste in Eq. (1.8) is a valid approximation only if μ is large in front of the other resilience parameters. This will obviously be no longer true when the number of resources gets beyond some threshold.

1.3.2.2 Execution Time for a Parallel Application

In this section, we explain how to use Proposition 1.2 to compute the expected execution time of a parallel application using N processors. We consider the following relevant scenarios for checkpoint/recovery overheads and for parallel execution times.

Checkpoint/recovery overheads—With coordinated checkpointing, checkpoints are synchronized over all processors. We use $C(N)$ and $R(N)$ to denote the time for saving a checkpoint and for recovering from a checkpoint on N processors, respectively (we assume that the downtime D does not depend on N). Assume that the application's memory footprint is Mem, and b_{io} represents the available I/O bandwidth. bytes, with each processor holding $\frac{\text{Mem}}{N}$ bytes. We envision two scenarios:

- Proportional overhead: $C(N) = R(N) = \frac{\text{Mem}}{N\mathrm{b}_{io}}$. This is representative of cases in which the bandwidth of the network card/link at each processor is the I/O bottleneck. In such cases, processors checkpoint their data in parallel.
- Constant overhead: $C(N) = R(N) = \frac{\text{Mem}}{\mathrm{b}_{io}}$, which is representative of cases in which the bandwidth to/from the resilient storage system is the I/O bottleneck. In such cases, processors checkpoint their data in sequence.

Parallel work—Let $W(N)$ be the time required for a failure-free execution on N processors. We use three models:

- Embarrassingly parallel jobs: $W(N) = W/N$. Here W represents the sequential execution time of the application.
- Generic parallel jobs: $W(N) = W/N + \gamma W$. As in Amdahl's law [1], $\gamma < 1$ is the fraction of the work that is inherently sequential.
- Numerical kernels: $W(N) = W/N + \gamma W^{2/3}/\sqrt{N}$. This is representative of a matrix product (or LU/QR factorization) of size n on a 2D-processor grid, where $W = O(n^3)$. In the algorithm in [7], $N = p^2$ and each processor receives $2p$ matrix blocks of size n/p. Here γ is the communication-to-computation ratio of the platform.

We assume that the parallel job is tightly coupled, meaning that all N processors operate synchronously throughout the job execution. These processors execute the same amount of work $W(N)$ in parallel, period by period. Inter-processor messages are exchanged throughout the computation, which can only progress if all processors are available. When a failure strikes a processor, the application is missing one resource for a certain period of time of length D, the *downtime*. Then the application recovers from the last checkpoint (*recovery* time of length $R(N)$) before it re-executes the work done since that checkpoint and up to the failure. Therefore, we can compute the optimal period and the optimal waste $\textsc{Waste}$ as in Theorem 1.1 with $\mu = \frac{\mu_{\text{ind}}}{N}$ and $C = C(N)$. The (expected) parallel execution time is $\mathit{Time}[\mathit{final}] = \frac{\textsc{Time}_{\text{base}}}{1-\textsc{Waste}}$, where $\textsc{Time}_{\text{base}} = W(N)$.

Altogether, we have designed a variety of scenarios, some more optimistic than others, to model the performance of a parallel tightly-coupled application with coordinated checkpointing. We point out that many scientific applications are tightly-coupled, such as iterative applications with a global synchronization point at the end of each iteration. However, the fact that inter-processor information is exchanged continuously or at given synchronization steps (as in BSP-like models) is irrelevant: in steady-state mode, all processors must be available concurrently for the execution to actually progress. While the tightly-coupled assumption may seem very constraining, it captures the fact that processes in the application depend on each other and exchange messages at a rate exceeding the periodicity of checkpoints, preventing independent progress.

1.3.3 Hierarchical Checkpointing

As discussed in Sect. 1.2.4, and presented in deeper details in Sect. 3.6 later in this book, hierarchical checkpointing algorithms are capable of partial coordination of checkpoints to decrease the cost of logging, while retaining message logging capabilities to remove the need for a global restart. These hierarchical schemes partition the application processes in groups. Each group checkpoints independently, but processes belonging to the same group coordinate their checkpoints and recovery. Communications between groups continue to incur payload logging. However, because processes belonging to a same group follow a coordinated checkpointing protocol, the payload of messages exchanged between processes within the same group is not required to be logged.

The optimizations driving the choice of the size and shape of groups are varied. A simple heuristic is to checkpoint as many processes as possible, simultaneously, without exceeding the capacity of the I/O system. In this case, groups do not checkpoint in parallel. Groups can also be formed according to hardware proximity or communication patterns. In such approaches, there may be opportunity for several groups to checkpoint concurrently.

The design and analysis of a refined model for hierarchical checkpointing requires to introduce many new parameters. First, we have to account for non-blocking checkpointing, i.e., the possibility to continue execution (albeit at a reduced rate) while checkpointing. Then message logging has three consequences, two negative and one positive:

- performance degradation in a fault-free execution (negative effect)
- re-execution speed-up after a failure (positive effect)
- checkpoint size increase to store logged messages (negative effect)

The last item is the most important, because intergroup messages may rapidly increase the total size of the checkpoint as the execution progresses, thereby imposing to cap the length of the checkpointing period (see Sect. 1.2.4). The model proposed in this section captures all these additional parameters for a variety of platforms and applications, and provides formulas to compute (and compare) the waste of each checkpointing protocol and application/platform scenario. However, the curious reader must be advised that derivation of the waste becomes much more complicated than in Sects. 1.3.1 and 1.3.2.

1.3.3.1 Instantiating the Model

In this section, we detail the main parameters of the model. We consider a tightly-coupled application that executes on N processors. As before, all model parameters are expressed in seconds. However, in the previous models, one work unit was executed in one second, because we assumed that processors were always computing at full rate. However, with hierarchical checkpointing, when a processor is slowed-down by another activity related to fault tolerance (writing checkpoints to stable storage, logging messages, etc.), one work-unit takes longer than a second to complete. Also, recall that after the striking of a failure under a hierarchical scenario, the useful work resumes only when the faulty group catches up with the overall state of the application at failure time.

Blocking or non-blocking checkpoint. There are various scenarios to model the cost of checkpointing in hierarchical checkpointing protocols, so we use a flexible model, with several parameters to specify. The first question is whether checkpoints are blocking or not. On some architectures, we may have to stop executing the application before writing to the stable storage where the checkpoint data is saved; in that case checkpoint is fully blocking. On other architectures, checkpoint data can be saved on the fly into a local memory before the checkpoint is sent to the stable storage, while computation can resume progress; in that case, checkpoints can be fully overlapped with computations. To deal with all situations, we introduce a slow-down factor α: during a checkpoint of duration C, the work that is performed is αC work units, instead of C work-units if only computation takes place. In other words, $(1 - \alpha)C$ work-units are wasted due to checkpoint jitters perturbing the progress of computation. Here, $0 \leq \alpha \leq 1$ is an arbitrary parameter. The case $\alpha = 0$ corresponds to a fully blocking checkpoint, while $\alpha = 1$ corresponds to a fully overlapped checkpoint, and all intermediate situations can be represented. Note that we have resorted to fully blocking models in Sects. 1.3.1 and 1.3.2.

Periodic checkpointing strategies. Just as before, we focus on periodic scheduling strategies where checkpoints are taken at regular intervals, after some fixed amount of work-units have been performed. The execution is partitioned into periods of duration $T = W + C$, where W is the amount of time where only computations take place, while C corresponds to the amount of time where checkpoints are taken. If not slowed down for other reasons by the fault-tolerant protocol (see Sect. 1.3.3.4), the total amount of work units that are executed during a period of length T is thus $\textsc{Work} = W + \alpha C$ (recall that there is a slow-down due to the overlap).

The equations that define the waste are the same as in Sect. 1.3.1. We reproduce them below for convenience:

$$\begin{aligned}&(1 - \textsc{Waste}_{\text{FF}})\textsc{Time}_{\text{FF}} = \textsc{Time}_{\text{base}}\\ &(1 - \textsc{Waste}_{\text{fail}})\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}}\\ &\textsc{Waste} = 1 - (1 - \textsc{Waste}_{\text{FF}})(1 - \textsc{Waste}_{\text{fail}})\end{aligned} \tag{1.16}$$

We derive easily that

$$\textsc{Waste}_{\text{FF}} = \frac{T - \textsc{Work}}{T} = \frac{(1-\alpha)C}{T} \tag{1.17}$$

As expected, if $\alpha = 1$ there is no overhead, but if $\alpha < 1$ (actual slowdown, or even blocking if $\alpha = 0$), we retrieve a fault-free overhead similar to that of coordinated checkpointing. For the time being, we do not further quantify the length of a checkpoint, which is a function of several parameters. Instead, we proceed with the abstract model. We envision several scenarios in Sect. 1.3.3.5, only after setting up the formula for the waste in a general context.

Processor groups. As mentioned above, we assume that the platform is partitioned into G groups of the same size. Each group contains q processors, hence $N = Gq$. When $G = 1$, we speak of a *coordinated* scenario, and we simply write C, D and R for the duration of a checkpoint, downtime and recovery. When $G \geq 1$, we speak of a *hierarchical* scenario. Each group of q processors checkpoints independently and sequentially in time $C(q)$. Similarly, we use $D(q)$ and $R(q)$ for the duration of the downtime and recovery. Of course, if we set $G = 1$ in the (more general) *hierarchical* scenario, we retrieve the value of the waste for the coordinated scenario. As already mentioned, we derive a general expression for the waste for both scenarios, before further specifying the values of $C(q)$, $D(q)$, and $R(q)$ as a function of q and the various architectural parameters under study.

1.3.3.2 Waste for the Coordinated Scenario ($G = 1$)

The goal of this section is to quantify the expected waste in the coordinated scenario where $G = 1$. Recall that we write C, D, and R for the checkpoint, downtime, and recovery using a single group of N processors. The platform MTBF is μ. We obtain

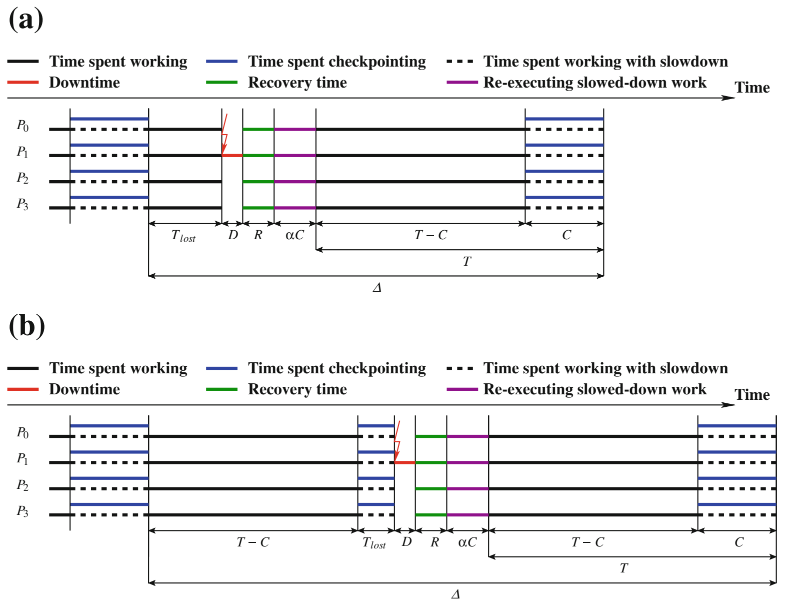


Fig. 1.9 Coordinated checkpoint: illustrating the waste when a failure occurs. **a** during the work phase; and **b** during the checkpoint phase

the following equation for the waste, which we explain briefly below and illustrate with Fig. 1.9:

$$\text{WASTE}_{\text{FF}} = \frac{(1-\alpha)C}{T} \tag{1.18}$$

$$\text{WASTE}_{\text{fail}} = \frac{1}{\mu}\Bigg(R + D + \frac{T-C}{T}\left[\alpha C + \frac{T-C}{2}\right] + \frac{C}{T}\left[\alpha C + T - C + \frac{C}{2}\right]\Bigg) \tag{1.19}$$

- Equation (1.18) is the portion of the execution lost in checkpointing, even during a fault-free execution, see Eq. (1.17).
- The second part of Eq. (1.19) is the overhead of the execution time due to a failure during work interval $T - C$ (see Fig. 1.9a).
- The last part of Eq. (1.19) is the overhead due to a failure during a checkpoint (see Fig. 1.9b).

After simplification of Eqs. (1.18) and (1.19), we get:

$$\text{WASTE}_{\text{fail}} = \frac{1}{\mu}\left(D + R + \frac{T}{2} + \alpha C\right) \tag{1.20}$$

Plugging this value back into Eq. (1.16) leads to:

$$\text{WASTE}_{\text{coord}} = 1 - \left(1 - \frac{(1-\alpha)C}{T}\right)\left(1 - \frac{1}{\mu}\left(D + R + \frac{T}{2} + \alpha C\right)\right) \quad (1.21)$$

The optimal checkpointing period T_{opt} that minimizes the expected waste in Eq. (1.21) is

$$T_{\text{opt}} = \sqrt{2(1-\alpha)(\mu - (D + R + \alpha C))C} \quad (1.22)$$

This value is in accordance with the first-order expression of T_{FO} in Eq. (1.9) when $\alpha = 0$ and, by construction, must be greater than C. Of course, just as before, this expression is valid only if all resilience parameters are small in front of μ.

1.3.3.3 Waste for the Hierarchical Scenario ($G \geq 1$)

In this section, we compute the expected waste for the hierarchical scenario. We have G groups of q processors, and we let $C(q)$, $D(q)$, and $R(q)$ be the duration of the checkpoint, downtime, and recovery for each group. We assume that the checkpoints of the G groups take place in sequence within a period (see Fig. 1.10a). We start by generalizing the formula obtained for the coordinated scenario before introducing several new parameters to the model.

Generalizing previous scenario with $G \geq 1$: We obtain the following intricate formula for the waste, which we illustrate with Fig. 1.10 and the discussion below:

$$\text{WASTE}_{\text{hier}} = 1 - \left(1 - \frac{T - \text{WORK}}{T}\right)\left(1 - \frac{1}{\mu}\left(D(q) + R(q) + \text{RE-EXEC}\right)\right) \quad (1.23)$$

$$\text{WORK} = T - (1-\alpha)GC(q) \quad (1.24)$$

$$\begin{aligned}
\text{RE-EXEC} = {} & \\
& \frac{T - GC(q)}{T}\frac{1}{G}\sum_{g=1}^{G}\left[(G-g+1)\alpha C(q) + \frac{T - GC(q)}{2}\right] \\
& + \frac{GC(q)}{T}\frac{1}{G^2}\sum_{g=1}^{G}\Bigg[\\
& \quad \sum_{s=0}^{g-2}(G - g + s + 2)\alpha C(q) + T - GC(q) \\
& \quad + G\alpha C(q) + T - GC(q) + \frac{C(q)}{2} \\
& \quad + \sum_{s=1}^{G-g}(s+1)\alpha C(q)\Bigg]
\end{aligned} \quad (1.25)$$

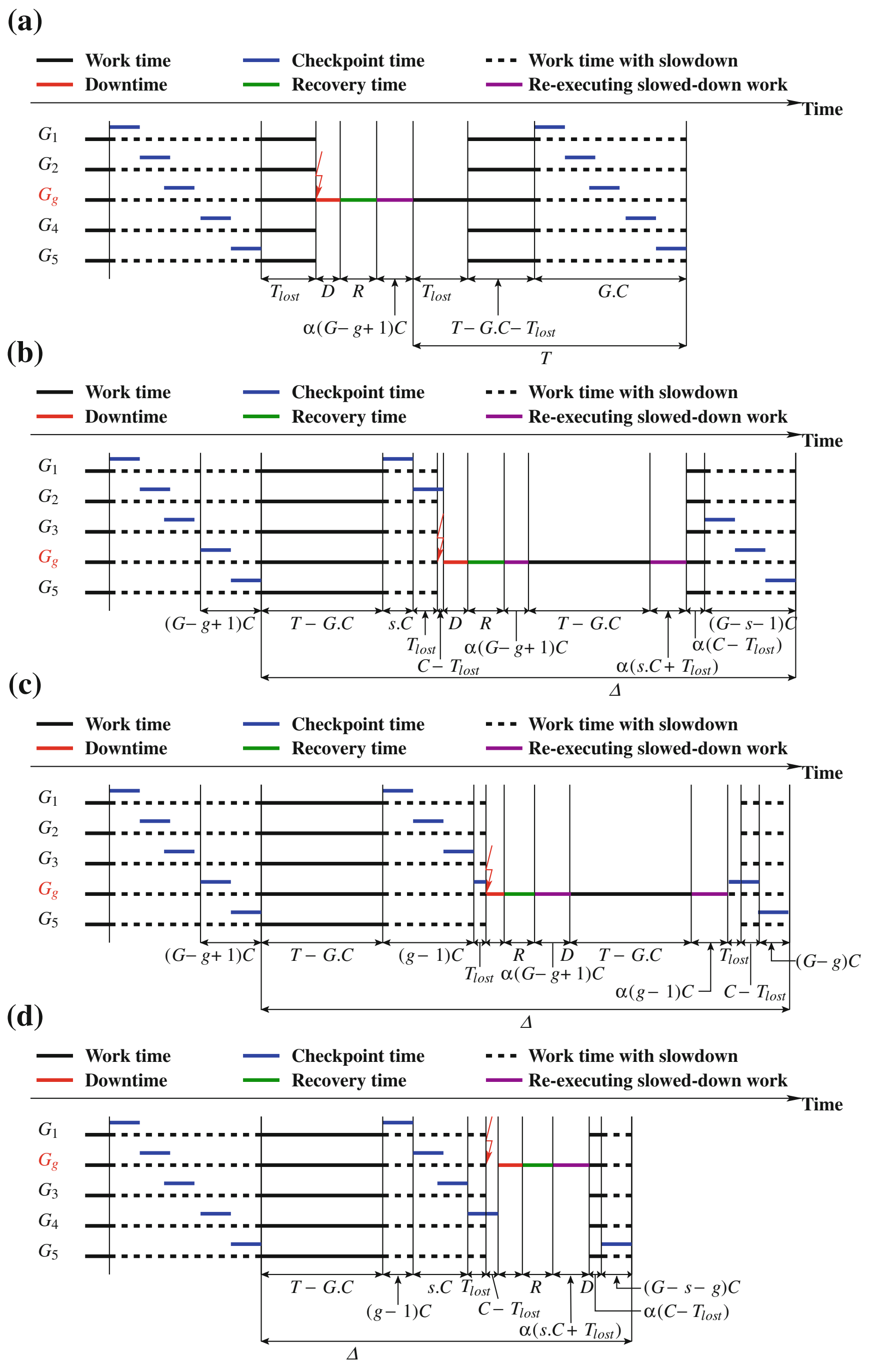


Fig. 1.10 Hierarchical checkpoint: illustrating the waste when a failure occurs. **a** during the work phase (first part of Eq. (1.25)); and during the checkpoint phase (last three parts of Eq. (1.25)), with three sub-cases: **b** before the checkpoint of the failing group (second part of Eq. (1.25)), **c** during the checkpoint of the failing group (third part of Eq. (1.25)), or **d** after the checkpoint of the failing group (last part of Eq. (1.25))

- The first term in Eq. (1.23) represents the overhead due to checkpointing during a fault-free execution (same reasoning as in Eq. (1.17)), and the second term the overhead incurred in case of failure.
- Equation (1.24) provides the amount of work units executed within a period of length T.
- The first part of Eq. (1.25) represents the time needed for re-executing the work when the failure happens in a work-only area, i.e., during the first $T - GC(q)$ seconds of the period (see Fig. 1.10a).
- The second part of Eq. (1.25) deals with the case where the fault happens during a checkpoint, i.e., during the last $GC(q)$ seconds of the period (hence the first term that represents the probability of this event).

  We distinguish three cases, depending upon what group was checkpointing at the time of the failure:

  – The third part of Eq. (1.25) is for the case when the fault happens before the checkpoint of group g (see Fig. 1.10b).
  – The fourth part of Eq. (1.25) is for the case when the fault happens during the checkpoint of group g (see Fig. 1.10c).
  – The fifth part of Eq. (1.25) is the case when the fault happens after the checkpoint of group g, during the checkpoint of group $g + s$, where $g + 1 \leq g + s \leq G$ (See Fig. 1.10d).

Of course this expression reduces to Eq. (1.21) when $G = 1$. Just as for the coordinated scenario, we enforce the constraint

$$GC(q) \leq T \tag{1.26}$$

by construction of the periodic checkpointing policy.

1.3.3.4 Refining the Model

We now introduce three new parameters to refine the model when the processors have been partitioned into several groups. These parameters are related to the impact of message logging on execution, re-execution, and checkpoint image size, respectively.

Impact of message logging on execution and re-execution. With several groups, intergroup messages need to be stored in local memory as the execution progresses, and event logs must be stored in reliable storage, so that the recovery of a given group, after a failure, can be done independently of the other groups. This induces an overhead, which we express as a slowdown of the execution rate: instead of executing one work-unit per second, the application executes only λ work-units, where $0 < \lambda < 1$. Typical values for λ are said to be $\lambda \approx 0.98$, meaning that the overhead due to payload messages is only a small percentage [14, 36].

On the contrary, message logging has a positive effect on re-execution after a failure, because intergroup messages are stored in memory and directly accessible after the recovery. Our model accounts for this by introducing a speedup factor ρ during the re-execution. Typical values for ρ lie in the interval [1; 2], meaning that re-execution time can be reduced by up to half for some applications [13].

Fortunately, the introduction of λ and ρ is not difficult to account for in the expression of the expected waste: in Eq. (1.23), we replace WORK by λWORK and RE- EXEC by $\frac{\text{RE- EXEC}}{\rho}$ and obtain

$$\text{WASTE}_{\text{hier}} = 1 - \left(1 - \frac{T - \lambda\text{WORK}}{T}\right)\left(1 - \frac{1}{\mu}\left(D(q) + R(q) + \frac{\text{RE- EXEC}}{\rho}\right)\right) \quad (1.27)$$

where the values of WORK and RE- EXEC are unchanged, and given by Eqs. (1.24) and (1.25) respectively.

Impact of message logging on checkpoint size. Message logging has an impact on the execution and re-execution rates, but also on the size of the checkpoint. Because intergroup messages are logged, the size of the checkpoint increases with the amount of work per unit. Consider the hierarchical scenario with G groups of q processors. Without message logging, the checkpoint time of each group is $C_0(q)$, and to account for the increase in checkpoint size due to message logging, we write the equation

$$C(q) = C_0(q)(1 + \beta\lambda\text{WORK}) \Leftrightarrow \beta = \frac{C(q) - C_0(q)}{C_0(q)\lambda\text{WORK}} \quad (1.28)$$

As before, $\lambda\text{WORK} = \lambda(T - (1 - \alpha)GC(q))$ (see Eq. (1.24)) is the number of work units, or application iterations, completed during the period of duration T, and the parameter β quantifies the increase in the checkpoint image size per work unit, as a proportion of the application footprint. Typical values of β are given in the examples of Sect. 1.3.3.5. Combining with Eq. (1.28), we derive the value of $C(q)$ as

$$C(q) = \frac{C_0(q)(1 + \beta\lambda T)}{1 + GC_0(q)\beta\lambda(1 - \alpha)} \quad (1.29)$$

The constraint in Eq. (1.26), namely $GC(q) \le T$, now translates into $\frac{GC_0(q)(1+\beta\lambda T)}{1+GC_0(q)\beta\lambda(1-\alpha)} \le T$, hence

$$GC_0(q)\beta\lambda\alpha \le 1 \text{ and } T \ge \frac{GC_0(q)}{1 - GC_0(q)\beta\lambda\alpha} \quad (1.30)$$

1.3.3.5 Case Studies

In this section, we use the previous model to evaluate different case studies. We propose three generic scenarios for the checkpoint protocols, and three application examples with different values for the parameter β.

Checkpointing algorithm scenarios.

Coord- IO —The first scenario considers a coordinated approach, where the duration of a checkpoint is the time needed for the N processors to write the memory footprint of the application onto stable storage. Let Mem denote this memory, and b_{io} represents the available I/O bandwidth. Then

$$C = C_{\mathrm{Mem}} = \frac{\mathrm{Mem}}{\mathrm{b}_{io}} \tag{1.31}$$

(see the discussion on checkpoint/recovery overheads in Sect. 1.3.2.2 for a similar scenario). In most cases we have equal write and read speed access to stable storage, and we let $R = C = C_{\mathrm{Mem}}$, but in some cases we could have different values. Recall that a constant value $D(q) = D$ is used for the downtime.

Hierarch- IO —The second scenario uses a number of relatively large groups. Typically, these groups are composed to take advantage of the application communication pattern [32, 36]. For instance, if the application executes on a 2D-grid of processors, a natural way to create processor groups is to have one group per row (or column) of the grid. If all processors of a given row belong to the same group, horizontal communications are intragroup communications and need not to be logged. Only vertical communications are intergroup communications and need to be logged.

With large groups, there are enough processors within each group to saturate the available I/O bandwidth, and the G groups checkpoint sequentially. Hence the total checkpoint time without message logging, namely $GC_0(q)$, is equal to that of the coordinated approach. This leads to the simple equation

$$C_0(q) = \frac{C_{\mathrm{Mem}}}{G} = \frac{\mathrm{Mem}}{G\mathrm{b}_{io}} \tag{1.32}$$

where Mem denotes the memory footprint of the application, and b_{io} the available I/O bandwidth. Similarly as before, we use $R(q)$ for the recovery (either equal to $C(q)$ or not), and a constant value $D(q) = D$ for the downtime.

Hierarch- Port —The third scenario investigates the possibility of having a large number of very small groups, a strategy proposed to take advantage of hardware proximity and failure probability correlations [15]. However, if groups are reduced to a single processor, a single checkpointing group is not sufficient to saturate the available I/O bandwidth. In this strategy, multiple groups of q processors are allowed to checkpoint simultaneously in order to saturate the I/O bandwidth. We define $\mathrm{q}_{\min}$ as the smallest value such that $\mathrm{q}_{\min}\mathrm{b}_{port} \geq \mathrm{b}_{io}$, where b_{port} is the network bandwidth of a single processor. In other words, $\mathrm{q}_{\min}$ is the minimal size of groups so that Eq. (1.32) holds.

Small groups typically imply logging more messages (hence a larger growth factor of the checkpoint per work unit β, and possibly a larger impact on computation slowdown λ). For an application executing on a 2D-grid of processors, twice as many communications will be logged (assuming a symmetrical communication pattern along each grid direction). However, let us compare recovery times in the Hierarch- Port and Hierarch- IO strategies; assume that $R_0(q) = C_0(q)$ for simplicity. In both cases Eq. (1.32) holds, but the number of groups is significantly larger for HIERARCH- PORT, thereby ensuring a much shorter recovery time.

Application examples: We study the increase in checkpoint size due to message logging by detailing three application examples that are typical scientific applications executing on 2D-or 3D-processor grids, but this exhibits a different checkpoint increase rate parameter β.

2D- STENCIL– We first consider a 2D-stencil computation: a real matrix of size $n \times n$ is partitioned across a $p \times p$ processor grid, where $p^2 = N$. At each iteration, each element is averaged with its 8 closest neighbors, requiring rows and columns that lie at the boundary of the partition to be exchanged (it is easy to generalize to larger update masks). Each processor holds a matrix block of size $b = n/p$, and sends four messages of size b (one in each grid direction). Then each element is updated, at the cost of 9 double floating-point operations. The (parallel) work for one iteration is thus WORK $= \frac{9b^2}{s_p}$, where s_p is the speed of one processor.

Here Mem $= 8n^2$ (in bytes), since there is a single (double real) matrix to store. As already mentioned, a natural (application-aware) group partition is with one group per row (or column) of the grid, which leads to $G = q = p$. Such large groups correspond to the HIERARCH- IO scenario, with $C_0(q) = \frac{C_{\text{Mem}}}{G}$. At each iteration, vertical (intergroup) communications are logged, but horizontal (intragroup) communications are not logged. The size of logged messages is thus $2pb = 2n$ for each group. If we checkpoint after each iteration, $C(q) - C_0(q) = \frac{2n}{b_{io}}$, and we derive from Eq. (1.28) that $\beta = \frac{2nps_p}{n^2 9b^2} = \frac{2s_p}{9b^3}$. We stress that the value of β is unchanged if groups checkpoint every k iterations, because both $C(q) - C_0(q)$ and WORK are multiplied by a factor k. Finally, if we use small groups of size q_{min}, we have the HIERARCH- PORT scenario. We still have $C_0(q) = \frac{C_{\text{Mem}}}{G}$, but now the value of β has doubled since we log twice as many communications.

MATRIX- PRODUCT —Consider now a typical linear-algebra kernel involving matrix products. For each matrix-product, there are three matrices involved, so Mem $= 24n^2$ (in bytes). The matrix partition is similar to previous scenario, but now each processor holds three matrix blocks of size $b = n/p$. Consider Cannon's algorithm [18] which has p steps to compute a product. At each step, each processor shifts one block vertically and one block horizontally, and WORK $= \frac{2b^3}{s_p}$. In the HIERARCH- IO scenario with one group per grid row, only vertical messages are logged: $\beta = \frac{s_p}{6b^3}$. Again, β is unchanged if groups checkpoint every k steps, or every matrix product ($k = p$). In the COORD- PORT scenario with groups of size q_{min}, the value of β is doubled.

3D- STENCIL —This application is similar to 2D- STENCIL, but with a 3D matrix of size n partitioned across a 3D-grid of size p, where $8n^3 =$ Mem and $p^3 = N$. Each processor holds a cube of size $b = n/p$. At each iteration, each pixel is averaged with its 26 closest neighbors, and WORK $= \frac{27b^3}{s_p}$. Each processor sends the six faces of its cube, one in each direction. In addition to COORD- IO, there are now three hierarchical

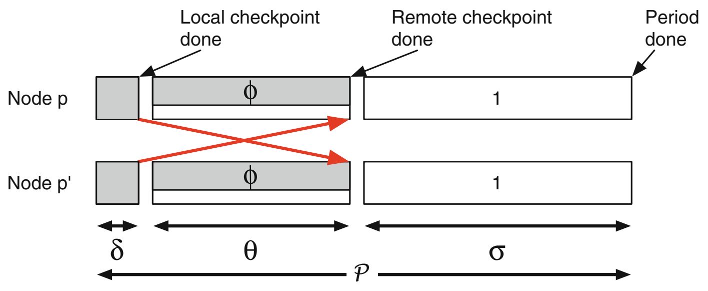


Fig. 1.11 Double checkpoint algorithm

scenarios: (A) HIERARCH- IO- PLANE where groups are horizontal planes, of size p^2. Only vertical communications are logged, which represents two faces per processor: $\beta = \frac{2s_p}{27b^3}$; (B) HIERARCH- IO- LINE where groups are lines, of size p. Twice as many communications are logged, which represents four faces per processor: $\beta = \frac{4s_p}{27b^3}$; (C) HIERARCH- PORT (groups of size $q_{\min}$). All communications are logged, which represents six faces per processor: $\beta = \frac{6s_p}{27b^3}$. The order of magnitude of b is the cubic root of the memory per processor for 3D- STENCIL, while it was its square root for 2D- STENCIL and MATRIX- PRODUCT, so β will be larger for 3D- STENCIL.

Wrap-up. We have shown how to instantiate all the resilience parameters of the model. Now, to assess the performance of a given scenario for hierarchical checkpointing, there only remain to instantiate the platform parameters: individual MTBF μ_{ind}, number of nodes N (from which we deduce the platform MTBF μ), number of cores per node, speed of each core s_p, memory per node, fraction of that memory used for the application memory footprint Mem, I/O network and node bandwidths b_{io} and b_{port}. Then we can use the model to predict the waste when varying the number of groups and the assumptions on checkpoint time. The interested reader will find several examples in [10].

1.3.4 In-Memory Checkpointing

In this section, we briefly survey a recent protocol that has been designed to reduce the time needed to checkpoint an application. The approach to reduce checkpoint time is to avoid using any kind of stable, but slow-to-access, storage. Rather than using a remote disk system, *in-memory checkpointing* uses the main memory of the processors. This will provide faster access and greater scalability, at the price of the risk of a fatal failure in some (unlikely) scenarios.

Figure 1.11 depicts the double checkpoint algorithm of [59, 71]. Processors are arranged into pairs. Within a pair, checkpoints are replicated: each processor stores its own checkpoint and that of its *buddy* in its local memory. We use the notations of [59, 71] in Fig. 1.11, which shows the following:

- The execution is divided into periods of length $\mathscr{P}$

- At the beginning of the period, each node writes its own checkpoint in its local memory, which takes a time δ. This writing is done in blocking mode, and the execution is stopped.
- Then each node send its checkpoint to its buddy. This exchange takes a time θ. The exchange is non-blocking, and the execution can progress, albeit with a slowdown factor Φ
- During the rest of the period, for a time σ, the execution progresses at full (unit) speed

The idea of the non-blocking exchange is to use those time-steps where the application is not performing inter-processor communications to send/receive the checkpoint files, thereby reducing the overhead incurred by the application.

Let us see what happens when a failure strikes one processor, as illustrated in Fig. 1.12a. Node p is hit by a failure, and a spare node will take over. After a downtime D, the spare node starts by recovering the checkpoint file of node p, in time R. The spare receives this file from node p', the buddy of node p, most likely as fast as possible (in blocking mode) so that it can resume working. Then the spare receives the checkpoint file of node p', to ensure that the application is protected if a failure hits p' later on. As before, receiving the checkpoint file can be overlapped with the execution and takes a time Θ, but there is a trade-off to make now: as shown in Fig. 1.12b, the application is at risk until both checkpoint receptions are completed. If a failure strikes p' before that, then it is a critical failure that cannot be recovered from. Hence it might be a good idea to receive the second checkpoint (that of p') as fast as possible too, at the price of a performance degradation of the whole application: when one processor is blocked, the whole application cannot progress. A detailed

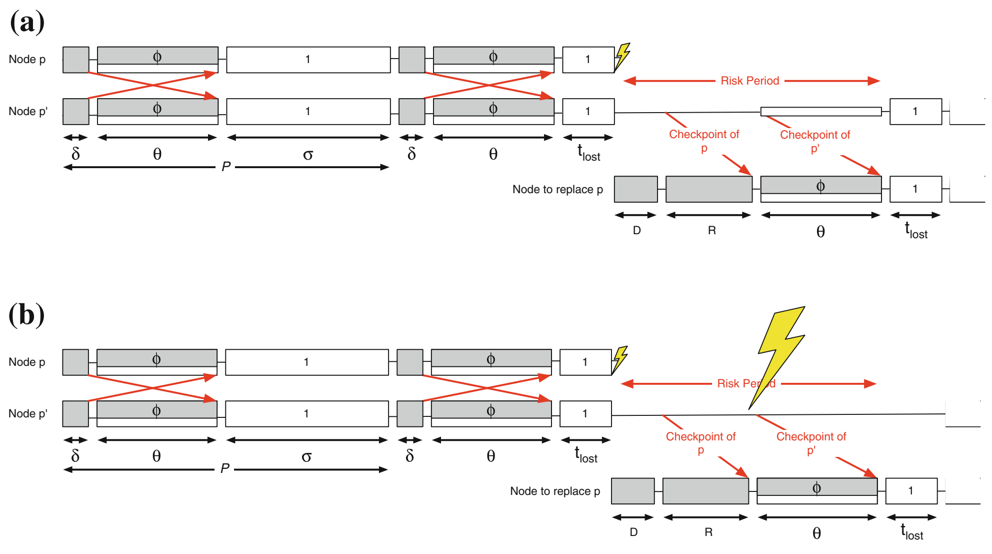


Fig. 1.12 Handling failures in the double checkpoint algorithm. **a** A failure hits node p. **b** A second failure hits node p', the buddy of node p, before the spare node had finished to receive the checkpoint file of p'. This is a fatal failure for the application

analysis is available in [28], together with extensions to a triple-checkpoint algorithm where each node has two buddies instead of one, thereby dramatically decreasing the risk of a fatal failure.

Finally, we mention that the risk of a fatal failure can be eliminated when using a *multi-level* checkpointing protocol, such as FTI. [5] or SCR. [57]. Such protocols allow to set different levels/types of checkpoints during the execution. Different checkpoint levels correspond to different recovery abilities, and also suffer from different checkpoint/recovery overheads. See [5, 57] for further details.

1.4 Probabilistic Models for Advanced Methods

In this section, we present two extensions of checkpointing performance models. Section 1.4.1 explains how to combine checkpointing with *fault prediction*, and discuss how the optimal period is modified when this combination is used. Section 1.4.2 explains how to combine checkpointing with *replication*, and discuss how the optimal period is modified when this combination is used.

1.4.1 Fault Prediction

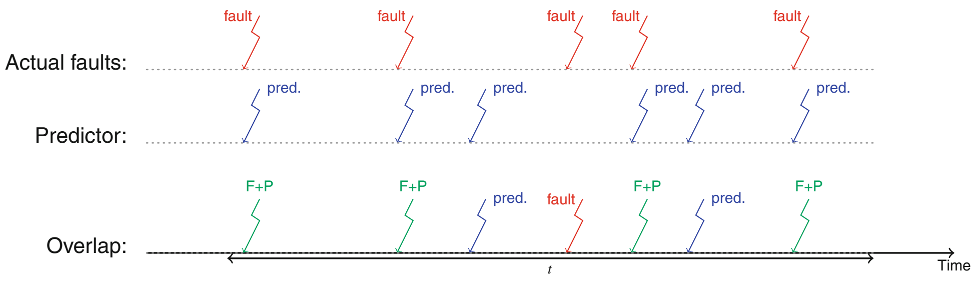
A possible way to cope with the numerous faults and their impact on the execution time is to try and predict them. In this section we do not explain how this is done, although the interested reader will find some answers in Chap. 2 and in [35, 70, 73].

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 p, 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_{P} be the mean time between predicted events (both true positive and false positive), and μ_{NP}be the mean time between unpredicted faults (false negative). The relations between μ_{P}, μ_{NP}, μ, r and p are as follows:

- Rate of unpredicted faults: $\frac{1}{\mu_{\text{NP}}} = \frac{1-r}{\mu}$, since $1 - r$ is the fraction of faults that are unpredicted;
- Rate of predicted faults: $\frac{r}{\mu} = \frac{p}{\mu_{\text{P}}}$, since r is the fraction of faults that are predicted, and p 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 approximately: $\mu = \frac{t}{5}$, $\mu_P = \frac{t}{6}$, and $\mu_{NP} = t$. For this predictor, the recall is $r = \frac{4}{5}$ (green arrows over red arrows), and its precision is $p = \frac{4}{6}$ (green arrows over blue arrows).

Now, given a fault predictor of parameters p and r, can we improve the waste? More specifically, how to modify the periodic checkpointing algorithm to get better results? In order to answer these questions, 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_p (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_p) 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 Fig. 1.13b, c);

We compute the expected waste as before. We reproduce Eq. (1.7) below:

$$\text{WASTE} = \text{WASTE}_{\text{FF}} + \text{WASTE}_{\text{fault}} - \text{WASTE}_{\text{FF}}\text{WASTE}_{\text{fault}} \tag{1.33}$$

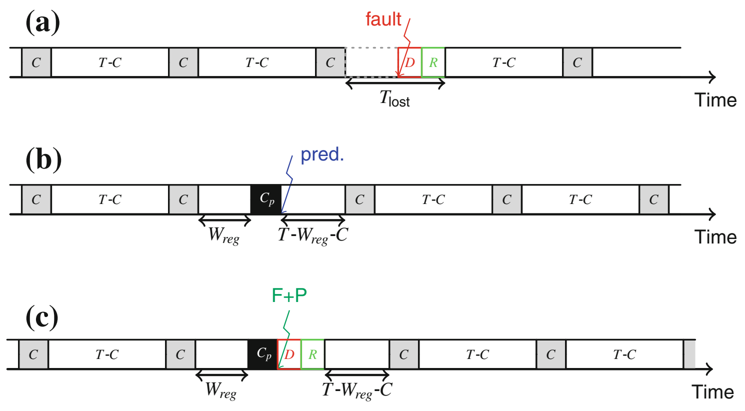


Fig. 1.13 Actions taken for the different event types. **a** Unpredicted fault, **b** Prediction taken into account—no actual fault, **c** Prediction taken into account—with actual fault

While the value of WASTE_{FF} is unchanged ($\text{WASTE}_{\text{FF}} = \frac{C}{T}$), the value of $\text{WASTE}_{\text{fault}}$ is modified because of predictions. As illustrated in Fig. 1.13, there are different scenarios that contribute to $\text{WASTE}_{\text{fault}}$. We classify them as follows:

(1) Unpredicted faults: This overhead occurs each time an unpredicted fault strikes, that is, on average, once every μ_{NP} seconds. Just as in Eq. (1.6), the corresponding waste is $\frac{1}{\mu_{\text{NP}}}\left[\frac{T}{2} + D + R\right]$.

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

$$T_{\text{lost}} = p(C_p + D + R) + (1 - p)C_p = C_p + p(D + R)$$

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

$$\begin{aligned}
\text{WASTE}_{\text{fault}} &= \frac{1}{\mu_{\text{NP}}}\left(\frac{T}{2} + D + R\right) + \frac{1}{\mu_{\text{P}}}\left(C_p + p(D + R)\right) \\
&= \frac{1-r}{\mu}\left(\frac{T}{2} + D + R\right) + \frac{r}{p\mu}\left(C_p + p(D + R)\right) \\
&= \frac{1}{\mu}\left((1-r)\frac{T}{2} + D + R + \frac{rC_p}{p}\right)
\end{aligned}$$

We can now plug this expression back into Eq. (1.33):

$$\begin{aligned}
\text{WASTE} &= \text{WASTE}_{\text{FF}} + \text{WASTE}_{\text{fault}} - \text{WASTE}_{\text{FF}}\text{WASTE}_{\text{fault}} \\
&= \frac{C}{T} + \left(1 - \frac{C}{T}\right)\frac{1}{\mu}\left(D + R + \frac{rC_p}{p} + \frac{(1-r)T}{2}\right).
\end{aligned}$$

To compute the value of T_{FO}^{p}, the period that minimizes the total waste, we use the same reasoning as in Sect. 1.3.1 and obtain:

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

We observe the similarity of this result with the value of T_{FO} from Eq. (1.9). If μ is large in front of the resilience parameters, we derive that $T_{\text{FO}}^{p} = \sqrt{\frac{2\mu C}{1-r}}$. 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 is increased by 40 %, and the waste is decreased by the same factor. Prediction can help!

Going further. The discussion above 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 as we progress within the period. On the contrary, if the prediction happens in the beginning of the period, we have to trade-off the possibility that the proactive checkpoint may be useless (if we indeed take a proactive action) with the small amount of work that may be lost in the case where a fault would actually happen. The optimal approach is to never trust the predictor in the beginning of a period, and to always trust it in the end; the crossover point $\frac{C_p}{p}$ depends on the time to take a proactive checkpoint and on the precision of the predictor. See [4] 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 for that event, a.k.a. a prediction window. More information can be found in [2].

1.4.2 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 as each processor has a *replica* (another processor performing exactly the same computations, receiving the same messages, etc.). See Fig. 1.14 for an illustration. We say that the two processes in a given pair

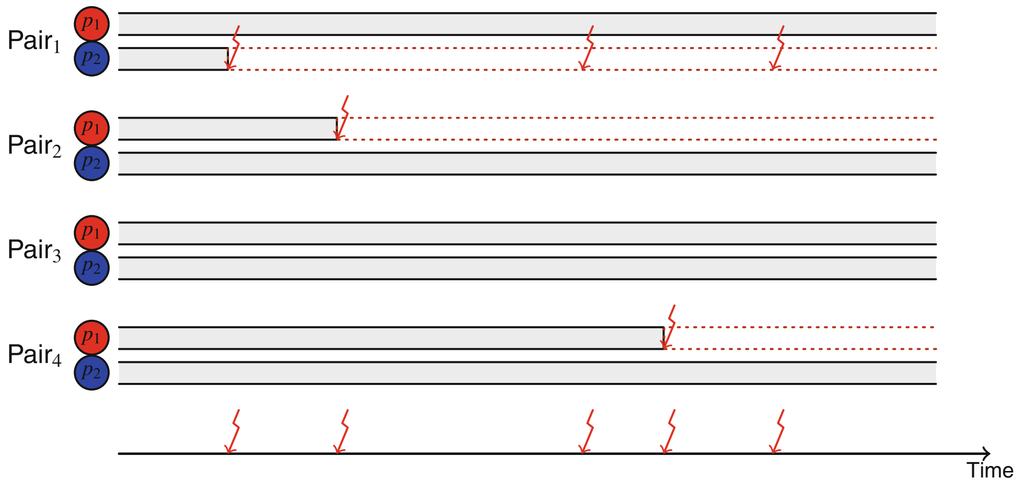


Fig. 1.14 Processor pairs for replication: each *blue* processor is paired with a *red* processor. In each pair, both processors do the same work

are *replicas*. When a processor is hit by a fault, its replica is not impacted. The execution of the application can still progress, until the replica 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?

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 failure-free, checkpoint-free environment, the time to execute the application ($\text{TIME}_{\text{Base}}$) decreases linearly with the number of processors. More precisely:

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

Consider the execution of a perfectly parallel application on a platform with $N = 2n$ 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 N processors,

$$\text{THROUGHPUT} = N\left(1 - \text{WASTE}\right)$$

The *standard* approach, as seen before, is to use all $2n$ processors so the execution of the application benefits from the maximal parallelism of the platform. This would be optimal in a fault-free environment, but we are required to checkpoint frequently because faults repeatedly strike the N processors. According to Proposition 1.2, the platform MTBF is $\mu = \frac{\mu_{\text{ind}}}{N}$. According to Theorem 1.1, the waste is (approximately) $\text{WASTE} = \sqrt{\frac{2C}{\mu}} = \sqrt{\frac{2CN}{\mu_{\text{ind}}}}$. We have:

$$\text{THROUGHPUT}_{\text{Std}} = N\left(1 - \sqrt{\frac{2CN}{\mu_{\text{ind}}}}\right) \qquad (1.34)$$

The second approach uses *replication*. There are n 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:

$$\textsc{Throughput}_{\text{Rep}} = \frac{N}{2}\left(1 - \sqrt{\frac{2C}{\mu_{\text{rep}}}}\right) \tag{1.35}$$

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

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

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 n bins, until one bins gets two balls? The answer to this question is given by Ramanujans Q-Function [34], and is equal to $\lceil q(n) \rceil$ where $q(n) = \frac{2}{3} + \sqrt{\frac{\pi n}{2}} + \sqrt{\frac{\pi}{288n}} - \frac{4}{135n} + \ldots$. When $n = 365$, 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 $n + 1 = 366$ persons; on average, one needs $\lceil q(n) \rceil = 25$ persons.[3]

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 Fig. 1.15 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 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 n bins, until one bins gets 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:

Proposition 1.3 *MNFTI* $= \mathbb{E}(\mathit{NFTI}|0)$ *where*

$$\mathbb{E}(\mathit{NFTI}|n_f) = \begin{cases} 2 & \text{if } n_f = N, \\ \frac{2N}{2N-n_f} + \frac{2N-2n_f}{2N-n_f}\mathbb{E}\left(\mathit{NFTI}|n_f + 1\right) & \text{otherwise.} \end{cases}$$

Proof Let $\mathbb{E}(\mathit{NFTI}|n_f)$ be the expectation of the number of faults needed to interrupt the application, knowing that the application is still running and that faults have

[3] 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).

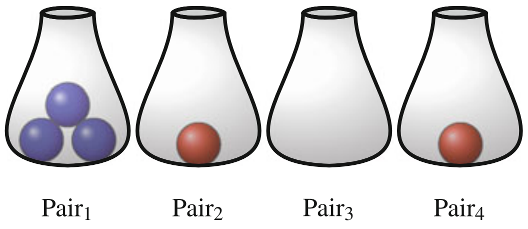


Fig. 1.15 Modeling the state of the platform of Fig. 1.14 as a balls-into-bins problem. We put a *red* ball in bin Pair$_i$ when there is a fault on its *red* processor p_1, and a *blue* ball when there is a fault on its *blue* processor p_2. As long as no bin has received a ball of each *color*, the game is on

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 $N - n_f$ are still replicated. Overall, there are $n_f + 2(N - n_f) = 2N - n_f$ processors still running.

The case $n_f = N$ 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 $\mathbb{E}(NFTI|N) = 1$.
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: $\mathbb{E}(NFTI|N) = 1 + \mathbb{E}(NFTI\,|N\,)$.

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:

$$\mathbb{E}\,(NFTI\,|N\,) = \frac{1}{2} \times 1 + \frac{1}{2} \times (1 + \mathbb{E}\,(NFTI\,|N\,))\,,$$

hence $\mathbb{E}\,(NFTI\,|N\,) = 2$.

For the general case $0 \le n_f \le N - 1$, 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 $n_f = N$ case studied above. The fault probability is uniformly distributed among the $2N$ processors, including the ones already hit. Hence the probability that the next fault hits a new pair is $\frac{2N-2n_f}{2N}$. In this case, the expected number of faults needed to interrupt the application fail is one (the considered fault) plus $\mathbb{E}\left(NFTI|n_f + 1\right)$. Altogether we have:

$$\begin{aligned}\mathbb{E}\left(NFTI|n_f\right) = {} & \tfrac{2N-2n_f}{2N} \times \left(1 + \mathbb{E}\left(NFTI|n_f + 1\right)\right) \\ & + \tfrac{2n_f}{2N} \times \left(\tfrac{1}{2} \times 1 + \tfrac{1}{2}\left(1 + \mathbb{E}\left(NFTI|n_f\right)\right)\right).\end{aligned}$$

Therefore,

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

Let us compare the throughput of each approach with an example. From Eqs. (1.34) and (1.35), we have

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

which we rewrite into

$$C \geq \frac{\mu_{\text{ind}}}{2N}\frac{1}{(2-\frac{1}{\sqrt{MNFTI}})^2} \tag{1.36}$$

Take a parallel machine with $N = 2^{20}$ processors. This is a little more than one million processors, but this corresponds to the size of the largest platforms today. Using Proposition 1.3, 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 would say say that $\mu_{\text{ind}} = 10$ years is pessimistic, because we rather observe that $\mu_{\text{ind}} = 100$ years in current supercomputers. Since $\mu_{\text{ind}} = 100$ years allows us to use a checkpointing period of one hour, you might then decide that replication is not worth it. On the contrary, maybe you would 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 s, 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 Eq. (1.36).

Going further. There are two natural options "counting" faults. The option chosen above is to allow new faults to hit processors that have already been hit. This is the option chosen in [33], 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 executing the application. It turns out that both options are almost equivalent, the values of their *MNFTI* only differ by one [19].

We refer the interested reader to Chap. 4 for a full analysis of replication. For convenience, we provide a few bibliographical notes in the following lines. Replication has long been used as a fault tolerance mechanism in distributed systems [38], and in the context of volunteer computing [51]. Replication has recently received attention in the context of HPC (High Performance Computing) applications [31, 33, 66, 72]. 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* and voting, or TMR [56]. 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).

1.5 Application-Specific Fault Tolerance Techniques

All the techniques presented and evaluated so far are *general* techniques: the assumptions they make on the behavior of the application are as little constraining as possible, and the protocol to tolerate failures considered two adversaries: the occurrence of failures, which can happen at the worst possible time, and also the application itself, which can take the worst possible action at the worst possible moment.

We now examine the case of application-specific fault tolerance techniques in HPC: when the application itself may use redundant information inherent of its coding of the problem, to tolerate misbehavior of the supporting platform. As one can expect, the efficiency of such approaches can be orders of magnitude better than the efficiency of general techniques; their programming, however, becomes a much harder challenge for the final user.

First, the application must be programmed over a middleware that not only tolerates failures for its internal operation, but also exposes them in a manageable way to the application; then, the application must maintain redundant information exploitable in case of failures during its execution. We will present a couple of cases of such applicative scenarios. Finally, we will discuss the portability of such approaches, and present a technique that allows the utilization of application-specific fault tolerance technique inside a more general application, preserving the fault tolerance property while exhibiting performance close to the one expected from application-specific techniques.

1.5.1 Fault-Tolerant Middleware

The first issue to address, to consider application-specific fault tolerance, is how to allow failures to be presented to the application. Even in the case of fail-stop errors, that can be detected easily under the assumption of pseudo-synchronous systems usually made in HPC, the most popular programming middleware, MPI, does not allow to expose failures in a portable way.

The MPI-3 specification has little to say about failures and their exposition to the user:

> It is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.
>
> MPI Standard, v3.0, p. 20, l. 36:39

This fist paragraph would allow implementations to expose the failures, limiting their propagation to the calls that relate to operations that cannot complete because of the occurrence of failures. However, later in the same standard:

> This document does not specify the state of a computation after an erroneous MPI call has occurred.
>
> MPI Standard v3.0, p. 21, l. 24:25

Unfortunately, most Open Source MPI implementations, and the numerous vendor-specific MPI implementations that derive from them, chose, by lack of demand from their users, and by lack of consensus, to interpret these paragraphs in a way that limits the opportunities for the user to tolerate failures: in the worst case, even if all communicators hit by failures are marked to return in case of error, the application is simply shutdown by the runtime system, as a cleanup procedure; in the best case, the control is given back to the user program, but no MPI call that involves a remote peer is guaranteed to perform any meaningful action for the user, leaving the processes of the application as separate entities that have to rely on external communication systems to tolerate failures.

The Fault Tolerance Working Group of the MPI Forum has been constituted to address this issue. With the dawn of extreme scale computing, at levels where failures become expected occurrences in the life of an application, MPI has set a cap to evolve towards more scalability. Capacity for the MPI implementation to continue its service in case of failures, and capacity for the MPI language to present these failures to the application, or to the software components that wish to handle these failures directly, are key among the milestones to remove technological locks towards scalability. Chapter 3 details the User-Level Failures Mitigation (ULFM) proposal of the FTWG of the MPI Forum in its Sect. 3.8. We present here its main features, as an introduction.

There are two main issues to address to allow applications written in MPI to tolerate failures:

- Detect and report failures
- Provide service after the occurrence of failures

ULFM exposes failures to the application through MPI *exceptions*. It introduces a couple of error classes that are returned by pertaining MPI calls if a failure strikes, and prevents their completion (be it because the failure happened before or during the call). As per traditional MPI specification, exceptions are raised only if the user defined a specific error handler for the corresponding communicator, or if it specified to use the predefined error handler that makes exceptions return an error code.

In those cases, the ULFM proposal states that no MPI call should block indefinitely because of the occurrence of failures. Collective calls must return either a success code if they did complete despite the failure, or an error code if their completion was compromised; point to point operations must also return. This raises two issues:

- the same collective call may return with success or fail, depending on the rank. For example, a broadcast operation is often implemented using a broadcast tree to provide logarithmic overheads. If a node low in the broadcast tree is subject to a failure, the root of the tree may not notice the failure and succeed completing all its local operations, while trees under the failed node will not receive the information. In all cases, all processes must enter the broadcast operation, as the meaning of collective is not changed, and all processes must leave the operation, as none could stall forever because of a failure. Nodes under the failed process may raise an exception, while nodes above it may not notice the failure during this call.
- in the case of point to point operations, it may become hard for the implementation to decide whether an operation will complete or not. Take the example of a receive from any source operation: any process in the communicator may be the sender that would, in a failure-free execution, send the message that would match this reception. As a consequence, if a single process failed, the MPI implementation cannot safely decide (unless it finds incoming messages to match the reception) if the reception is going to complete or not. Since the specification does not allow for a process to stall forever because of the occurrence of failures, the implementation should raise an exception. However, the reception operation cannot be marked as failed, since it is possible that the matching send comes later from a different process. The specification thus allows the implementation to delay the notification for as long as seems fit, but for a bounded time, after which the reception must return with a special exception that marks the communication as undecided, thus giving back the control to the application to decide if that message is going to come or not.

To take such decisions, the application has access to a few additional routines. The application can acknowledge the presence of failures in a communicator (using `MPI_Comm_failure_ack`, and resume its operation over the same communicator that holds failed processes. Over such a communicator, any operation that involves a failed process will fail. Thus, collective operations that involve all processes in the communicator will necessarily fail. Point to point communications, on the other hand, may succeed if they are not a specific emission to a failed process or reception from a failed process. Receptions from any source will succeed and wait for a matching message, as the user already acknowledged the presence of some failures. If the user wanted to cancel such a reception, she can decide by requesting the MPI implementation to provide the list of failed processes after an acknowledgment (via `MPI_Comm_get_acked`). If more processes fail after the acknowledgment, more exceptions will be raised and can be acknowledged. Point to point communications will thus continue to work after a failure, as long as they do not directly involve an acknowledged failed process.

The application may also need to fix the communicator, in order to allow for collective operations to succeed. In order to clearly separate communications that happened before or after a set of failures, ULFM does not provide a way to fix the communicator. Instead, it provides a routine that exclude the failed processes from a communicator and creates a new one, suitable for the whole range of MPI routines (the routine `MPI_Comm_shrink`). This communicator creation routine is specified to work despite the occurrence of failures. The communicator that it creates must exclude failures that were acknowledged before entering the routine, but since failures may happen at any time, the newly created communicator may itself include failed processes, for example if a failure happened just after its creation.

The last routine provided by the ULFM proposal is a routine to allow resolution of conflicts after a failure. `MPI_Comm_agree` provides a consensus routine over the surviving ranks of a communicator. It is critical to determine an agreement in the presence of failures, since collective operations have no guarantee of consistent return values if a failure happens during their execution. Its usage is documented more closely in Chap. 3, as it interacts with `MPI_Comm_failure_ack` to enable the user to construct a low cost group membership service, that provides a global view of processes that survived a set of failures.

The leading idea of ULFM was to complement the MPI specification with a small set of routines, and extended specification for the existing routines, in case of process failures, enabling the user application or library to notice failures, react and continue the execution of the application despite the occurrence of these failures. The specification targets a lean set of changes, not promoting any specific model to tolerate failures, but providing the minimal building blocks to implement, through composition of libraries or directly in the application, a large spectrum of application-specific fault tolerance approaches. In the following, we discuss a few typical cases that were implemented over this ULFM proposal.

1.5.2 ABFT for Dense Matrix Factorization

Algorithm-Based Fault Tolerance (ABFT) was introduced by Abraham and Huang in 1984 [45] to tolerate possible memory corruptions during the computation of a dense matrix factorization. It is a good example of application-specific fault tolerance technique that is not simplistic, but provides an extreme boost in performance when used (compared to a general technique, like rollback-recovery). ABFT and disk-less checkpointing have been combined to apply to basic matrix operations like matrix-matrix multiplication [8, 22, 23] and have been implemented on algorithms similar to those of ScaLAPACK [24], which is widely used for dense matrix operations on parallel distributed memory systems, or the High Performance Linpack (HPL) [26] and to the Cholesky factorization [40].

An ABFT scheme for dense matrix factorization was introduced in [16, 29], and we explain it here, because it combines many application-level techniques, including replication, user-level partial checkpointing, and ABFT itself. We illustrate this technique with the LU factorization algorithm, which is the most complex due to its pivoting, but the approach applies to other direct methods of factorization.

Section 3.7.3 of Chap. 3 presents a similar algorithm for the QR factorization in the context of a less supportive communication middleware.

To support fail-stop errors, an ABFT scheme must be built on top of a fault-aware middleware. We assume a failure, defined in this section as a process that *completely* and *definitely* stops responding, triggering the loss of a critical part of the global application state, could occur at any moment and can affect any part of the application's data.

Algorithm Based Fault Tolerance. The general idea of ABFT is to introduce information redundancy in the data, and maintain this redundancy during the computation. Linear algebra operations over matrices are well suited to apply such a scheme: the matrix (original data of the user) can be extended by a number of columns, in which checksums over the rows are stored. The operation applied over the initial matrix can then be extended to apply at the same time over the initial matrix and its extended columns, maintaining the checksum relation between data in a row and the corresponding checksum column(s). Usually, it is sufficient to extend the scope of the operation to the checksum rows, although in some cases the operation must be redefined.

If a failure hits processes during the computation, the data host by these processes is lost. However, in theory, the checksum relation being preserved, if enough information survived the failure between the initial data held by the surviving processes and the checksum columns, a simple inversion of the checksum function is sufficient to reconstruct the missing data and pursue the operation.

No periodical checkpoint is necessary, and more importantly the recovery procedure brings back the missing data at the point of failure, without introducing a period of re-execution as the general techniques seen above impose, and a computational cost that is usually linear with the size of the data. Thus, the overheads due to ABFT are expected to be significantly lower than those due to rollback-recovery.

LU Factorization: The goal of a factorization operation is usually to transform a matrix that represents a set of equations into a form suitable to solve the problem $Ax = b$, where A and b represent the equations, A being a matrix and b a vector of same height. Different transformations are considered depending on the properties of the matrix, and the LU factorization transforms $A = LU$ where L is a lower triangular matrix, and U an upper triangular matrix. This transformation is done by blocks of fixed size inside the matrix to improve the efficiency of the computational kernels. Figure 1.16 represents the basic operations applied to a matrix during a block LU factorization. The GETF2 operation is a panel factorization, applied on a block column. This panel operation factorizes the upper square, and scales the lower rectangle accordingly. The output of that operation is then used to the right of the factored block to scale it accordingly using a triangular solve (TRSM), and the trailing matrix is updated accordingly using a matrix-matrix multiplication (GEMM). The block column and the block row are in their final LU form, and that trailing matrix must be transformed using the same algorithm, until the last block of the matrix is in the LU form. Technically, each of these basic steps is usually performed by applying a parallel Basic Linear Algebra Subroutine (PBLAS).

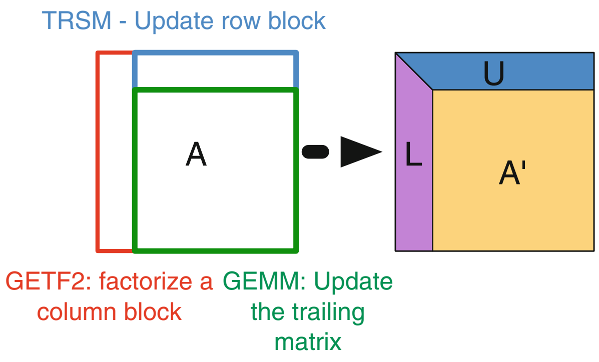

TRSM - Update row block
A
U
L
A'
GETF2: factorize a column block
GEMM: Update the trailing matrix

Fig. 1.16 Operations applied on a matrix, during the LU factorization. A' is the trailing matrix, that needs to be factorized using the same method until the entire initial matrix is in the form LU

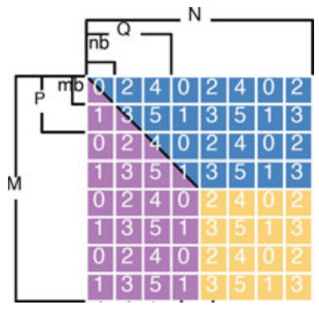


Fig. 1.17 Block cyclic distribution of a $8m_b \times 8n_b$ matrix over a 2×3 process grid

Data Distribution. For a parallel execution, the data of the matrix must be distributed among the different processors. For dense matrix factorization, the data is distributed following a 2D block cyclic distribution: processes are arranged over a 2D cyclic processor grid of size $P \times Q$, the matrix is split in blocks of size $m_b \times n_b$, and the blocks are distributed among the processes cyclically. Figure 1.17 shows how the blocks are distributed in a case of a square matrix of size $8m_b \times 8n_b$, and a process grid of size 2×3.

Reverse Neighboring Scheme: If one of the processes is subject of failure, many blocks are lost. As explained previously, the matrix is extended with checksum columns to introduce information redundancy. Figure 1.18 presents how the matrix is extended with checksum columns following a *reverse neighboring scheme*. The reverse neighboring scheme is a peculiar arrangement of data that simplifies significantly the design of the ABFT part of the algorithm.

The data matrix has 8×8 blocks and therefore the size of checksum is 8×3 blocks with an extra 8×3 blocks copy. Checksum blocks are stored on the right of

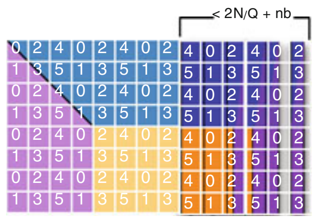


Fig. 1.18 Reverse neighboring scheme of checksum storage

the data matrix. In the example, the first 3 block columns produce the checksum in the last two block columns (hence making 2 duplicate copies of the checksum); the next 3 block columns then produce the next 2 rightmost checksum columns, etc.

Because copies are stored in consecutive columns of the process grid, for any 2D grid $P \times Q$ with $Q > 1$, the checksum duplicates are guaranteed to be stored on different processors. The triangular solve (TRSM) and trailing matrix update (GEMM) are applied to the whole checksum area until the first three columns are factored. In the following factorization steps, the two last block columns of checksum are excluded from the TRSM and GEMM scope. Since TRSM and GEMM claim most of the computation in the LU factorization, shrinking the update scope greatly reduces the overhead of the ABFT mechanism by diminishing the amount of (useless) extra computations; meanwhile, the efficiency of the update operation itself remains optimal as, thanks to the reverse storage scheme, the update still operates on a contiguous memory region and can be performed by a single PBLAS call.

Checksum blocks are duplicated for a reason: since they are stored on the same processes as the matrix and following the same block cyclic scheme, when a process is subject to a failure, blocks of initial data are lost, but also blocks of checksums. Because of the cyclic feature of the data distribution, all checksum blocks must remain available to recover the missing data. Duplicating them guarantees that if a single failure happens, one of the copies will survive. In the example, checksum blocks occupy almost as much memory as the initial matrix once duplicated. However, the number of checksum block column necessary is $2N/(Q \times n_b)$, thus decreases linearly with the width of the process grid.

To simplify the figures, in the following we will represent the checksum blocks over a different process grid, abstracting the duplication of these blocks as if they were hosted by virtual processes that are not subject to failures. We consider here an algorithm that can tolerate only one simultaneous failure (on the same process row), hence at least one of the two checksum blocks will remain available.

***Q*-Panel**: The idea of the ABFT factorization is that by extending the scope of the operation to the checksum blocks, the checksum property is maintained between the matrix and the checksum blocks: a block still represents the sum of the blocks of the initial matrix. This is true for the compute-intensive update operations, like GEMM and TRSM. Unfortunately, this is not true for the GETF2 operation that cannot be extended to span over the corresponding checksum blocks.

To deal with this, a simplistic approach would consist in changing the computational kernel to go update the checksum blocks during the GETF2 operation. We avoid doing this because this would introduce more synchronization, having more processes participate to this operation (as the processes spanning over the corresponding checksum blocks are not necessarily involved in a given GETF2 operation). The GETF2 operation is already a memory-bound operation, that require little computation compared to the update operations. It also sits in the critical path of the execution, and is a major blocker to performance, so introducing more synchronization and more delay is clearly detrimental to the performance.

That is the reason why we introduced the concept of Q-panel update. Instead of maintaining the checksum property at all time for all blocks, we will let some of

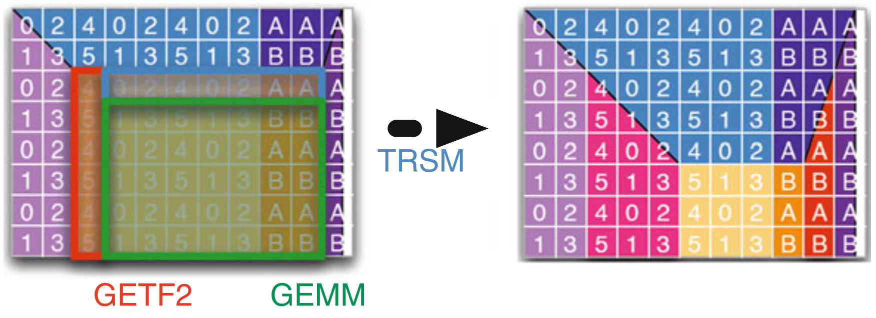


Fig. 1.19 Q-panel update of the ABFT LU factorization

the checksum blocks drift away, for a bounded time, and tolerate the risk for these Q-panel blocks with another approach. Then, when the cost of checksum update can be safely absorbed with maximal parallelism, we will let the algorithm update the checksums of the drifted away blocks, and pursue the computation.

ABFT LU Factorization: We now present the steps of the ABFT LU factorization using Q-panel update :

1. At a beginning of a Q-panel, when process (0, 0) hosts the first block on which GETF2 is going to be applied, processes take a partial checkpoint of the matrix: the first Q-block columns of the trailing matrix are copied, as well as the block column of corresponding checksums.
2. Then, the usual operations of LU are applied, using the first block column of the trailing matrix as a block panel (see Fig. 1.19): GETF2 is applied on that block column, then TRSM extended to the corresponding checksums, and GEMM, also extended on the corresponding checksums, producing a smaller trailing matrix.
   The checksums that correspond to the previously factored part of the matrix are left untouched, as the corresponding data in the matrix, so the checksum property is preserved for them. The checksums that were just updated with TRSM and GEMM also preserve the checksum property, as the update operations preserve the checksum property.
   The part of the checksum represented in red in the figure, however, violates the checksum property: the block column on which GETF2 was just applied hold values that are not represented in the corresponding block column in the reserve neighboring storing scheme.
3. The algorithm iterates, using the second block column of the Q-panel as a panel, until Q panels have been applied. In that case, the checksum property is preserved everywhere, except between the blocks that belong to the Q-panel, and the corresponding checksum block column. A checksum update operation is then executed, to recompute this checksum, the checkpoint saved at the beginning of this Q-panel loop can be discarded, and the next Q-panel loop can start.

Failure Handling. When a failure occurs, it is detected by the communication middleware, and the normal execution of the algorithm is interrupted. The ABFT fac-

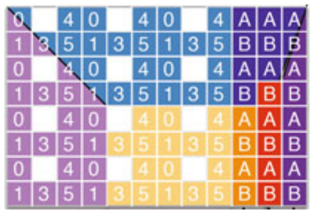


Fig. 1.20 Single failure during a Q-panel update of the ABFT LU factorization

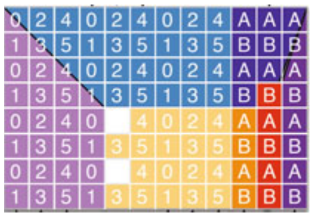


Fig. 1.21 Data restored using valid checksums

torization enters its recovery routine. Failures can occur at any point during the execution. The first step of the recovery routine is to gather the status of all surviving processes, and determine when the failure happened. Spare processes can then be reclaimed to replace the failed ones, or dynamic process management capabilities of the communication middleware are used to start new processes that will replace the missing ones.

In the general case, the failure happened while some blocks have been updated, and others not, during one of the Q-panels (see Fig. 1.20). Since the checksum blocks are replicated on adjacent processes, one copy survived the failure, so they are not missing. For all blocks where the checksum property holds, the checksum blocks are used to reconstruct the missing data.

The checkpoint of the Q-panel at the beginning of the last Q-panel step also lost blocks, since a simple local copy is kept. But because the processes also copied the checksum blocks corresponding to this Q-panel, they can rebuild the missing data for the checkpoint (Fig. 1.21).

The matrix is then overwritten with the restored checkpoint; the corresponding checksum blocks are also restored to their checkpoint. Then, the processes re-execute part of the update and factorization operations, but limiting their scope to the Q-panel section, until they reach the step when the Q-panel factorization was interrupted. At this point, all data has been restored to the time of failure, and the processes continue their execution, and are in a state to tolerate another failure.

If a second failure happens before the restoration is complete (or if multiple failures happen), the application may enter a state where recovery is impossible. This can be mitigated by increasing the number of checksum block columns, and by replacing checksum copies with linearly independent checksum functions. Then, when multiple failures occur, the restoration process consists of solving a small system of equations for each block, to determine the missing values. More importantly, this exhibits one of the features of application-specific fault tolerance: the overheads are a function of the risk the developer or user is ready to take.

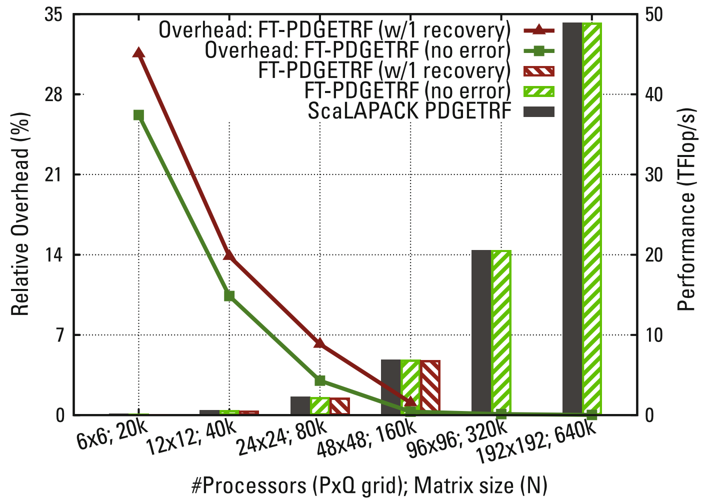


Fig. 1.22 Single failure during a Q-panel update of the ABFT LU factorization

Performance of ABFT LU. Fig. 1.22 (from [16]) shows a weak scalability study of the ABFT scheme that we presented above. On the left axis, the lines show the relative overhead of the ABFT-LU implementation in a failure-free and 1-failure/1-recovery scenario, compared to the non fault-tolerant implementation. On the right axis, the bar graphs show the raw performance of each scenario. This is a weak-scaling experiment, and the matrix size progresses with the process grid size, so that in each case, each processor is responsible for the same amount of data. We denote by $Q \times Q$; N in the x-axis the process grid size ($P \times Q$) and the matrix size (N).

That experiment was conducted on the NSF Kraken supercomputer, hosted at the National Institute for Computational Science (NICS). At the time of the experiment, this machine featured 112,896 2.6 GHz AMD Opteron cores, 12 cores per node, with the Seastar interconnect. At the software level, to serve as a comparison base, we used the non fault-tolerant ScaLAPACK LU in double precision with block size $m_b = n_b = 100$.

The recovery procedure adds a small overhead that also decreases when scaled to large problem size and process grid. For largest setups, only 2–3 percent of the execution time is spent recovering from a failure. Due to the introduction of checksum, operations counts and communication have been increased, as update operation span on a larger matrix comprised of the original trailing matrix and the checksums. During checkpointing and recovery, extra workload is performed and this all together leads to higher computing complexity than the original implementation in ScaLAPACK.

For simplicity of description, we consider square data matrices of size $N \times N$ distributed on a square grid $Q \times Q$. The operation count ration for LU factorization without and with checksum is:

$$
\begin{aligned}
R &= \frac{\frac{2}{3}N^3 - \frac{1}{2}N^2 + \frac{5}{6}N}{\frac{2}{3}(N+\frac{N}{Q})^3 - \frac{1}{2}(N+\frac{N}{Q})^2 + \frac{5}{6}(N+\frac{N}{Q})} \\
&= \frac{\frac{2}{3} - \frac{1}{2N} + \frac{5}{6N^2}}{\frac{2}{3}(1+\frac{1}{Q})^3 - \frac{1}{2N}(1+\frac{1}{Q})^2 + \frac{5}{6N^2}(1+\frac{1}{Q})}
\end{aligned} \tag{1.37}
$$

Clearly $\lim_{Q\rightarrow+\infty} R = 1$. Hence for systems with high number of processes, the extra flops for updating checksum columns is negligible with respect to the normal flops realized to compute the result.

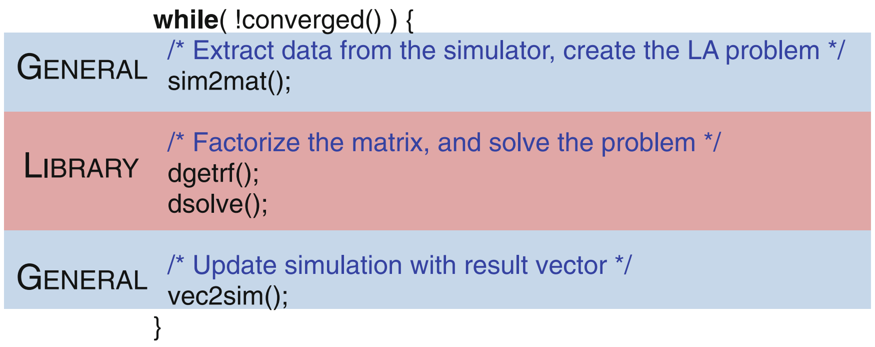
In addition, checksums must be generated, once at the start of the algorithm, the second time at the completion of a Q-wide panel scope. Both these activities account for $O(N^2)$ extra computations, but can be computed at maximal parallelism, since there is no data dependency.

1.5.3 Composite Approach: ABFT and Checkpointing

ABFT is a useful technique for production systems, offering protection to important infrastructure software. As we have seen, ABFT protection and recovery activities are not only inexpensive, but also have a negligible asymptotic overhead when increasing node count, which makes them extremely scalable. This is in sharp contrast with checkpointing, which suffers from increasing overhead with system size. Many HPC applications do spend quite a significant part of their total execution time inside a numerical library, and in many cases, these numerical library calls can be effectively protected by ABFT.

However, typical HPC applications do spend some time where they perform computations and data management that are incompatible with ABFT protection. The ABFT technique, as the name indicates, allows for tolerating failures only during the execution of the algorithm that features the ABFT properties. Moreover, it then protects only the part of the user dataset that is managed by the ABFT algorithm. In case of a failure outside the ABFT-protected operation, all data is lost; in case of a failure during the ABFT-protected operation, only the data covered by the ABFT scheme is restored. Unfortunately, these ABFT-incompatible phases force users to resort to general-purpose (presumably checkpoint based) approaches as their sole protection scheme.

A composition scheme proposed in [9, 11], protects the application partly with general fault tolerance techniques, and partly with application-specific fault tolerance techniques, harnessing the best of each approach. Performance is close to ABFT, as the ABFT-capable routines dominate the execution, but the approach is generic enough to be applied to any application that uses for at least a part of its execution ABFT-capable routines, so generality is not abandoned, and the user is not forced to rely only on generic rollback-recovery. We present this scheme below, because the underlying approach is key to the adoption of application-specific fault tolerance



```
          while( !converged() ) {
GENERAL     /* Extract data from the simulator, create the LA problem */
            sim2mat();

LIBRARY     /* Factorize the matrix, and solve the problem */
            dgetrf();
            dsolve();

GENERAL     /* Update simulation with result vector */
            vec2sim();
          }
```

Fig. 1.23 Pseudo-code of a typical application using Linear Algebra routines

methods in libraries: without a generic composition scheme, simply linking with different libraries that provide internal resilience capabilities to protect their data from a process crash will not make an application capable of resisting such crashes: process failure breaks the separation introduced by library composition in the software stack, and non protected data, as well as the call stack itself, must be protected by another mean.

As an illustration, consider an application that works as the pseudo-code given in Fig. 1.23. The application has two data: a matrix, on which linear algebra operations are performed, and a simulated state. It uses two libraries: a simulation library that changes the simulated state, and formulates a problem as an equation problem, and a linear algebra library that solves the problem presented by the simulator. The first library is not fault-tolerant, while there is an ABFT scheme to tolerate failures in the linear algebra library.

To abstract the reasoning, we distinguish two phases during the execution: during GENERAL phases, we have no information about the application behavior, and an algorithm-agnostic fault tolerance technique, namely checkpoint and rollback recovery, must be used. On the contrary, during LIBRARY phases, we know much more about the behavior of the library, and we can apply ABFT to ensure resiliency.

ABFT&PERIODICCKPT Algorithm. During a GENERAL phase, the application can access the whole memory; during a LIBRARY phase, only the LIBRARY dataset (a subset of the application memory, which is passed as a parameter to the library call) is accessed. The REMAINDER dataset is the part of the application memory that does not belong to the LIBRARY dataset.

The ABFT&PERIODICCKPT composite approach (see Fig. 1.24) consists of alternating between periodic checkpointing and rollback recovery on one side, and ABFT on the other side, at different phases of the execution. Every time the application enters a LIBRARY phase (that can thus be protected by ABFT), a partial checkpoint is taken to protect the REMAINDER dataset. The LIBRARY dataset, accessed by the ABFT algorithm, need not be saved in that partial checkpoint, since it will be reconstructed by the ABFT algorithm inside the library call.

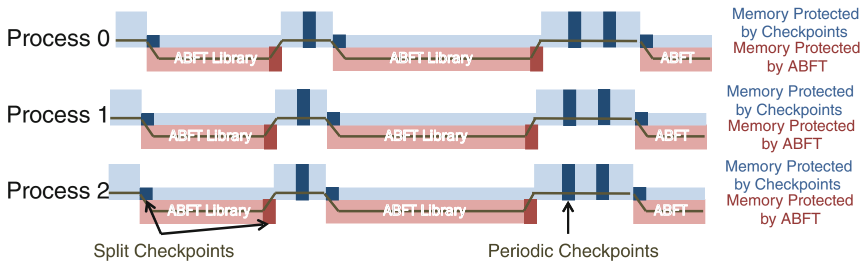


Fig. 1.24 ABFT&PeriodicCkpt composite approach

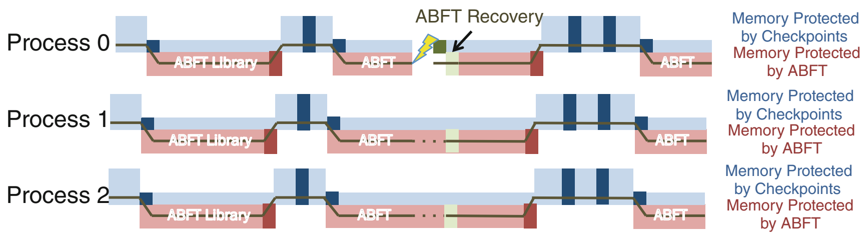


Fig. 1.25 Fault handling during a Library phase

When the call returns, a partial checkpoint covering the modified Library dataset is added to the partial checkpoint taken at the beginning of the call, to complete it and to allow restarting from the end of the terminating library call. In other words, the combination of the partial entry and exit checkpoints forms a split, but complete, coordinated checkpoint covering the entire dataset of the application.

If a failure is detected while processes are inside the library call (Fig. 1.25), the crashed process is recovered using a combination of rollback recovery and ABFT. ABFT recovery is used to restore the Library dataset before all processes can resume the library call, as would happen with a traditional ABFT algorithm. The partial checkpoint is used to recover the Remainder dataset (everything except the data covered by the current ABFT library call) at the time of the call, and the process stack, thus restoring it before quitting the library routine. The idea of this strategy is that ABFT recovery will spare some of the time spent redoing work, while periodic checkpointing can be completely de-activated during the library calls.

During General phases, regular periodic coordinated checkpointing is employed to protect against failures (Fig. 1.26). In case of failure, coordinated rollback recovery brings all processes back to the last checkpoint (at most back to the split checkpoint capturing the end of the previous library call).

ABFT&PeriodicCkpt Algorithm Optimization. Recall from Sect. 1.3.2 that a critical component to the efficiency of periodic checkpointing algorithms is the duration of the checkpointing interval. A short interval increases the algorithm overheads, by introducing many coordinated checkpoints, during which the application experiences slowdown, but also reduces the amount of time lost when there is a failure: the last checkpoint is never long ago, and little time is spent re-executing part of the application. Conversely, a large interval reduces overhead, but increases the

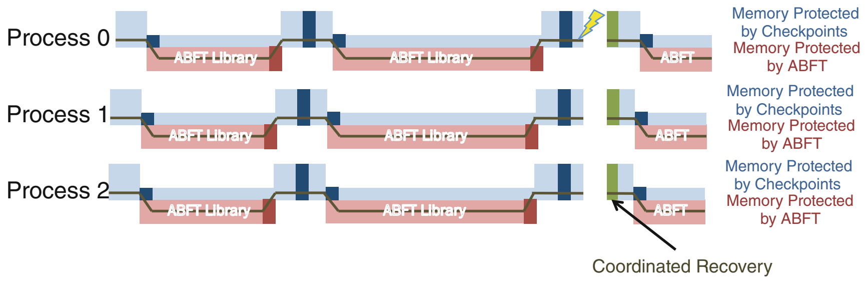


Fig. 1.26 Fault handling during a GENERAL phase

time lost in case of failure. In the ABFT&PERIODICCKPT algorithm, we interleave periodic checkpointing protected phases with ABFT protected phases, during which periodic checkpointing is de-activated. Thus, different cases have to be considered:

- When the time spent in a GENERAL phase is larger than the optimal checkpoint interval, periodic checkpointing is used during these phases in the case of ABFT-&PERIODICCKPT;
- When the time spent in a GENERAL phase is smaller than the optimal checkpoint interval, the ABFT&PERIODICCKPT algorithm already creates a complete valid checkpoint for this phase (formed by combining the entry and exit partial checkpoints), so the algorithm will not introduce additional checkpoints.

Moreover, the ABFT&PERIODICCKPT algorithm forces (partial) checkpoints at the entry and exit of library calls; thus if the time spent in a library call is very small, this approach will introduce more checkpoints than a traditional periodic checkpointing approach. The time complexity of library algorithms usually depends on a few input parameters related to problem size and resource number, and ABFT techniques have deterministic, well known time overhead complexity. Thus, when possible, the ABFT&PERIODICCKPT algorithm features a safeguard mechanism: if the projected duration of a library call with ABFT protection (computed at runtime thanks to the call parameters and the algorithm complexity) is smaller than the optimal periodic checkpointing interval, then ABFT is not activated, and the corresponding LIBRARY phase is protected using the periodic checkpointing technique only.

1.5.3.1 Performance Model of ABFT&PERIODICCKPT

The execution of the application is partitioned into epochs of total duration T_0. Within an epoch, there are two phases: the first phase is spent outside the library (it is a GENERAL phase, of duration T_G), and only periodic checkpointing can be employed to protect from failures during that phase. Then the second phase (a LIBRARY phase of duration T_L) is devoted to a library routine that has the potential to be protected by ABFT. Let α be the fraction of time spent in a LIBRARY phase: then we have $T_L = \alpha \times T_0$ and $T_G = (1-\alpha) \times T_0$.

As mentioned earlier, another important parameter is the amount of memory that is accessed during the LIBRARY phase (the LIBRARY dataset). This parameter is important because the cost of checkpointing in each phase is directly related to the amount of memory that needs to be protected. The total memory footprint is M , and the associated checkpointing cost is C (we assume a finite checkpointing bandwidth, so $C > 0$). We write $M = M_L + M_{\overline{L}}$, where M_L is the size of the LIBRARY dataset, and $M_{\overline{L}}$ is the size of the REMAINDER dataset. Similarly, we write $C = C_L + C_{\overline{L}}$, where C_L is the cost of checkpointing M_L , and $C_{\overline{L}}$ the cost of checkpointing $M_{\overline{L}}$. We can define the parameter ρ that defines the relative fraction of memory accessed during the LIBRARY phase by $M_L = \rho M$, or, equivalently, by $C_L = \rho C$.

Fault-free execution. During the GENERAL phase, we separate two cases. First, if the duration T_G of this phase is short, *i.e.* smaller than $P_G - C_{\overline{L}}$, which is the amount of work during one period of length P_G (and where P_G is determined below), then we simply take a partial checkpoint at the end of this phase, before entering the ABFT-protected mode. This checkpoint is of duration $C_{\overline{L}}$, because we need to save only the REMAINDER dataset in this case. Otherwise, if T_G is larger than $P_G - C_{\overline{L}}$, we rely on periodic checkpointing during the GENERAL phase: more specifically, the regular execution is divided into periods of duration $P_G = W + C$. Here W is the amount of work done per period, and the duration of each periodic checkpoint is $C = C_{\overline{L}} + C_L$, because the whole application footprint must be saved during a GENERAL phase. The last period is different: we execute the remainder of the work, and take a final checkpoint of duration $C_{\overline{L}}$ before switching to ABFT-protected mode. The optimal (approximated) value of P_G will be computed below.

Altogether, the length T_G^{ff} of a fault-free execution of the GENERAL phase is the following:

- If $T_G \le P_G - C_{\overline{L}}$, then $T_G^{\text{ff}} = T_G + C_{\overline{L}}$
- Otherwise, we have $\lfloor \frac{T_G}{\text{WORK}} \rfloor$ periods of length P_G, plus possibly a shorter last period if T_G is not evenly divisible by W. In addition, we need to remember that the last checkpoint taken is of length $C_{\overline{L}}$ instead of C.

This leads to

$$T_G^{\text{ff}} = \begin{cases} T_G + C_{\overline{L}} & \text{if } T_G \le P_G - C_{\overline{L}} \\ \lfloor \frac{T_G}{P_G - C} \times P_G \rfloor + (T_G \bmod W) + C_{\overline{L}} & \text{if } T_G > P_G - C_{\overline{L}} \text{ and } T_G \bmod W \neq 0 \\ \frac{T_G}{P_G - C} \times P_G - C_L & \text{if } T_G > P_G - C_{\overline{L}} \text{ and } T_G \bmod W = 0 \end{cases} \tag{1.38}$$

Now consider the LIBRARY phase: we use the ABFT-protection algorithm, whose cost is modeled as an affine function of the time spent: if the computation time of the library routine is t, its execution with the ABFT-protection algorithm becomes $\phi \times t$. Here, $\phi > 1$ accounts for the overhead paid per time-unit in ABFT-protected mode. This linear model for the ABFT overhead fits the existing algorithms for linear algebra, but other models could be considered. In addition, we pay a checkpoint C_L when exiting the library call (to save the final result of the ABFT phase). Therefore, the fault-tree execution time is

$$T_L^{\text{ff}} = \phi \times T_L + C_L \tag{1.39}$$

Finally, the fault-free execution time of the whole epoch is

$$T^{\text{ff}} = T_G^{\text{ff}} + T_L^{\text{ff}} \tag{1.40}$$

where T_G^{ff} and T_L^{ff} are computed according to the Eqs. (1.38) and (1.39).

Cost of failures. Next we have to account for failures. For each phase, we have a similar equation: the final execution time is the fault-free execution time, plus the number of failures multiplied by the (average) time lost per failure:

$$T_G^{\text{final}} = T_G^{\text{ff}} + \frac{T_G^{\text{final}}}{\mu} \times t_G^{\text{lost}} \tag{1.41}$$

$$T_L^{\text{final}} = T_L^{\text{ff}} + \frac{T_L^{\text{final}}}{\mu} \times t_L^{\text{lost}} \tag{1.42}$$

Equations (1.41) and (1.42) correspond to Eq. (1.5) in Sect. (1.3.1). Equation (1.41) reads as follows: T_G^{ff} is the failure-free execution time, to which we add the time lost due to failures; the expected number of failures is $\frac{T_G^{\text{final}}}{\mu}$, and t_G^{lost} is the average time lost per failure. We have a similar reasoning for Eq. (1.42). Then, t_G^{lost} and t_L^{lost} remain to be computed. For t_G^{lost} (GENERAL phase), we discuss both cases:

- If $T_G \le P_G - C_{\overline{L}}$: since we have no checkpoint until the end of the GENERAL phase, we have to redo the execution from the beginning of the phase. On average, the failure strikes at the middle of the phase, hence the expectation of loss is $\frac{T_G^{\text{ff}}}{2}$ time units. We then add the downtime D (time to reboot the resource or set up a spare) and the recovery R. Here R is the time needed for a complete reload from the checkpoint (and $R = C$ if read/write operations from/to the stable storage have the same speed). We derive that:

$$t_G^{\text{lost}} = D + R + \frac{T_G^{\text{ff}}}{2} \tag{1.43}$$

- If $T_G > P_G - C_{\overline{L}}$: in this case, we have periodic checkpoints, and the amount of execution which needs to be redone after a failure corresponds to half a checkpoint period on average, so that:

$$t_G^{\text{lost}} = D + R + \frac{P_G}{2} \tag{1.44}$$

For t_L^{lost} (LIBRARY phase), we derive that

$$t_L^{\text{lost}} = D + R_{\overline{L}} + \text{Recons}_{\text{ABFT}}$$

Here, $R_{\overline{L}}$ is the time for reloading the checkpoint of the REMAINDER dataset (and in many cases $R_{\overline{L}} = C_{\overline{L}}$). As for the LIBRARY dataset, there is no checkpoint to retrieve, but instead it must be reconstructed from the ABFT checksums, which takes time $\text{Recons}_{\text{ABFT}}$.

Optimization: finding the optimal checkpoint interval in GENERAL phase.

We verify from Eqs. (1.39) and (1.42) that T_L^{final} is always a constant. Indeed, we derive that:

$$T_L^{\text{final}} = \frac{1}{1 - \frac{D + R_{\overline{L}} + \text{Recons}_{\text{ABFT}}}{\mu}} \times (\phi \times T_L + C_L) \qquad (1.45)$$

As for T_G^{final}, it depends on the value of T_G: it is constant when T_G is small. In that case, we derive that:

$$T_G^{\text{final}} = \frac{1}{1 - \frac{D + R + \frac{T_G + C_{\overline{L}}}{2}}{\mu}} \times \left(T_G + C_{\overline{L}}\right) \qquad (1.46)$$

The interesting case is when T_G is large: in that case, we have to determine the optimal value of the checkpointing period P_G which minimizes T_G^{final}. We use an approximation here: we assume that we have an integer number of periods, and the last periodic checkpoint is of size C. Note that the larger T_G, the more accurate the approximation. From Eqs. (1.38), (1.41) and (1.44), we derive the following simplified expression:

$$T_G^{\text{final}} = \frac{T_G}{X} \text{ where } X = \left(1 - \frac{C}{P_G}\right)\left(1 - \frac{D + R + \frac{P_G}{2}}{\mu}\right) \qquad (1.47)$$

We rewrite:

$$X = \left(1 - \frac{C}{2\mu}\right) - \frac{P_G}{2\mu} - \frac{C(\mu - D - R)}{\mu P_G}$$

The maximum of X gives the optimal period P_G^{opt} . Differentiating X as a function of P_G, we find that it is obtained for:

$$P_G^{\text{opt}} = \sqrt{2C(\mu - D - R)} \qquad (1.48)$$

We retrieve Eq. 1.9 of Sect. 1.3.1 (as expected). Plugging the value of P_G^{opt} back into Eq. (1.47) provides the optimal value of T_G^{final} when T_G is large. We conclude this with reminding the word of caution given at the end of Sect. 1.3.2.1): the optimal value of the waste is only a first-order approximation, not an exact value. Just as in [25, 69], the formula only holds when μ, the value of the MTBF, is large with respect to the other resilience parameters. Owing to this hypothesis, we can neglect the probability of several failures occurring during the same checkpointing period.

Comparison of the scalability of approaches. The ABFT&PeriodicCkpt approach is expected to provide better performance when a significant time is spent in the Library phase, *and* when the failure rate implies a small optimal checkpointing period. If the checkpointing period is large (because failures are rare), or if the duration of the Library phase is small, then the optimal checkpointing interval becomes larger than the duration of the Library phase, and the algorithm automatically resorts to the periodic checkpointing protocol. This can also be the case when the epoch itself is smaller than (or of the same order of magnitude as) the optimal checkpointing interval (i.e., when the application does a fast switching between Library and General phases).

However, consider such an application that frequently switches between (relatively short) Library and General phases. When porting that application to a future larger scale machine, the number of nodes that are involved in the execution will increase, and at the same time, the amount of memory on which the ABFT operation is applied will grow (following Gustafson's law [37]). This has a double impact: the time spent in the ABFT routine increases, while at the same time, the MTBF of the machine decreases. As an illustration, we evaluate quantitatively how this scaling factor impacts the relative performance of the ABFT&PeriodicCkpt and a traditional periodic checkpointing approach.

First, we consider the case of an application where the Library and General phases scale at the same rate. We take the example of linear algebra kernels operating on $2D$-arrays (matrices), that scale in $O(n^3)$ of the array order n (in both phases). Following a weak scaling approach, the application uses a fixed amount of memory M_{ind} per node, and when increasing the number x of nodes, the total amount of memory increases linearly as $M = xM_{ind}$. Thus $O(n^2) = O(x)$, and the parallel completion time of the $O(n^3)$ operations, assuming perfect parallelism, scales in $O(\sqrt{x})$.

To instantiate this case, we take an application that would last a thousand minutes at 100,000 nodes (the scaling factor corresponding to an operation in $O(n^3)$ is then applied when varying the number of nodes), and consisting for 80 % of a Library phase, and 20 % of a General phase. We set the duration of the complete checkpoint and rollback (C and R , respectively) to 1 minute when 100,000 nodes are involved, and we scale this value linearly with the total amount of memory, when varying the number of nodes. The MTBF at 100,000 nodes is set to 1 failure every day, and this also scales linearly with the number of components. The ABFT overheads, and the downtime, are set to the same values as in the previous section, and 80 % of the application memory (M_L) is touched by the Library phase.

Given these parameters, Fig. 1.27 shows (i) the relative waste of periodic checkpointing and ABFT&PeriodicCkpt, as a function of the number of nodes, and (ii) the average number of faults that each execution will have to deal with to complete. The expected number of faults is the ratio of the application duration by the platform MTBF (which decreases when the number of nodes increases, generating more fail-

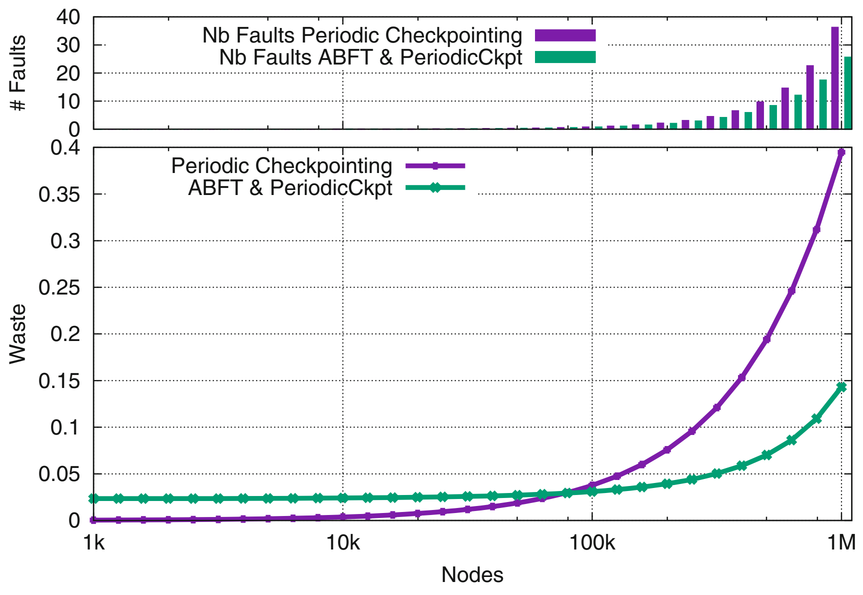


Fig. 1.27 Total waste for periodic checkpointing and ABFT&PeriodicCkpt, when considering the weak scaling of an application with a fixed ratio of 80 % spent in a Library routine

ures). The fault-free execution time increases with the number of nodes (as noted above), and the fault-tolerant execution time is also increased by the waste due to the protocol. Thus, the total execution time of periodic checkpointing is larger at 1 million nodes than the total execution time of ABFT&PeriodicCkpt at the same scale, which explains why more failures happen for these protocols.

Up to approximately 100,000 nodes, the fault-free overhead of ABFT negatively impacts the waste of the ABFT&PeriodicCkpt approach, compared to periodic checkpointing. Because the MTBF on the platform is very large compared to the application execution time (and hence to the duration of each Library phase), the periodic checkpointing approach has a very large checkpointing interval, introducing very few checkpoints, thus a small failure-free overhead. Because failures are rare, the cost due to time lost at rollbacks does not overcome the benefits of a small failure-free overhead, while the ABFT technique must pay the linear overhead of maintaining the redundancy information during the whole computation of the Library phase.

Once the number of nodes reaches 100,000, however, two things happen: failures become more frequent, and the time lost due to failures starts to impact rollback recovery approaches. Thus, the optimal checkpointing interval of periodic checkpointing becomes smaller, introducing more checkpointing overheads. During 80 % of the execution, however, the ABFT&PeriodicCkpt approach can avoid these overheads, and when they reach the level of linear overheads due to the ABFT technique, ABFT&PeriodicCkpt starts to scale better than both periodic checkpointing approaches.

All protocols have to resort to checkpointing during the General phase of the application. Thus, if failures hit during this phase (which happens 20 % of the time in

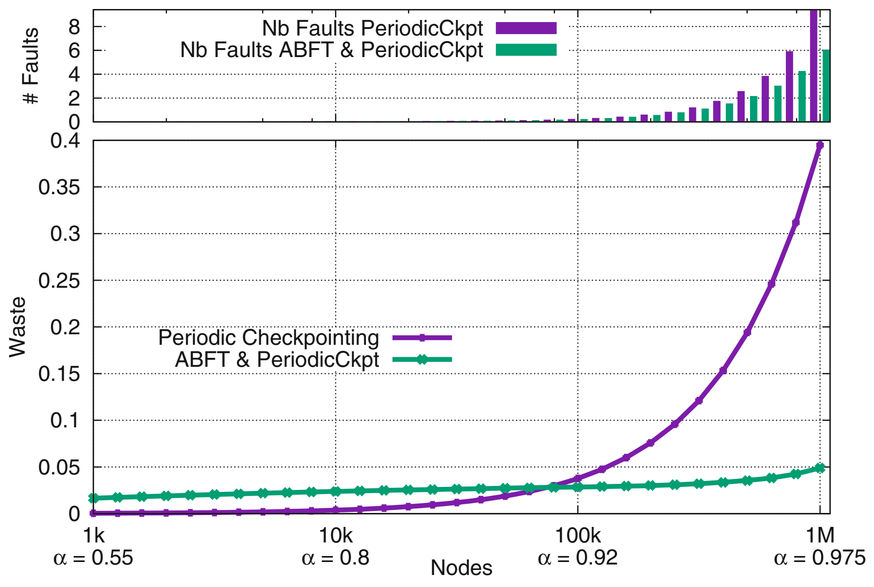


Fig. 1.28 Total waste for ABFT&PeriodicCkpt and periodic checkpointing when considering the weak scaling of an application with variable ratio of time spent in a Library routine

this example), they will all have to resort to rollbacking and lose some computation time. Hence, when the number of nodes increases and the MTBF decreases, eventually, the time spent in rollbacking and recomputing, which is linear in the number of faults, will increase the waste of all algorithms. However, one can see that this part is better controlled by the ABFT&PeriodicCkpt algorithm.

Next, we consider the case of an unbalanced General phase: consider an application where the Library phase has a cost $O(n^3)$ (where n is the problem size), as above, but where the General phase consists of $O(n^2)$ operations. This kind of behavior is reflected in many applications where matrix data is updated or modified between consecutive calls to computation kernels. Then, the time spent in the Library phase will increase faster with the number of nodes than the time spent in the General phase, varying α . This is what is represented in Fig. 1.28. We took the same scenario as above for Fig. 1.27, but α is a function of the number of nodes chosen such that at 100,000 nodes, $\alpha = T_L^{\text{final}}/T^{\text{final}} = 0.8$, and everywhere, $T_L^{\text{final}} = O(n^3) = O(\sqrt{x})$, and $T_{PC}^{\text{final}} = O(n^2) = O(1)$. We give the value of α under the number of nodes, to show how the fraction of time spent in Library phases increases with the number of nodes.

The periodic checkpointing protocol is not impacted by this change, and behaves exactly as in Fig. 1.27. Note, however, that $T^{\text{final}} = T_L^{\text{final}} + T_{PC}^{\text{final}}$ progresses at a lower rate in this scenario than in the previous scenario, because T_{PC}^{final} does not increase with the number of nodes. Thus, the average number of faults observed for all protocols is much smaller in this scenario.

The efficiency on ABFT&PeriodicCkpt, however, is more significant. The latter protocol benefits from the increased α ratio in both cases: since more time is

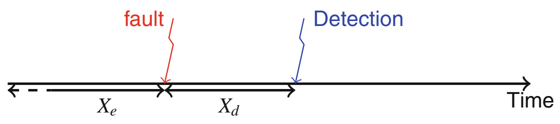


Fig. 1.29 Error and detection latency

spent in the LIBRARY phase, periodic checkpointing is de-activated for relatively longer periods. Moreover, this increases the probability that a failure will happen during the LIBRARY phase, where the recovery cost is greatly reduced using ABFT techniques. Thus, ABFT&PERIODICCKPT is capable of mitigating failures at a much smaller overhead than simple periodic checkpointing, and more importantly with better scalability.

1.6 Silent Errors

This section deals with techniques to cope with silent errors. We focus on a general-purpose approach that combines checkpointing and (abstract) verification mechanisms. Section 1.6.1 provides some background, while Sect. 1.6.2 briefly surveys different approaches form the literature. Then Sect. 1.6.3 details the performance model for the checkpoint/verification approach and explains how to determine the optimal pattern minimizing the waste.

1.6.1 Motivation

Checkpoint and rollback recovery techniques assume reliable error detection, and therefore apply to fail-stop failures, such as for instance the crash of a resource. In this section, we revisit checkpoint protocols in the context of *silent* errors, also called silent data corruption. Such errors must be accounted for when executing HPC applications [58, 61, 74–76]. The cause for silent errors may be for instance soft efforts in L1 cache, or bit flips due to cosmic radiation. The problem is that the detection of a silent error is not immediate, but will only manifest later as a failure, once the corrupted data has impacted the result (see Fig. 1.29). If the error stroke before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted, and cannot be used to restore the application. In the case of fail-stop failures, a checkpoint cannot contain a corrupted state, because a process subject to failure will not create a checkpoint or participate to the application: failures are naturally contained to failed processes; in the case of silent errors, however, faults can propagate to other processes and checkpoints, because processes continue to participate and follow the protocol during the interval that separates the error and its detection.

To alleviate this issue, one may envision to keep several checkpoints in memory, and to restore the application from the last *valid* checkpoint, thereby rolling back to the last *correct* state of the application [55]. This multiple-checkpoint approach has three major drawbacks. First, it is very demanding in terms of stable storage: each checkpoint typically represents a copy of the entire memory footprint of the application, which may well correspond to several terabytes. The second drawback is the possibility of fatal failures. Indeed, if we keep k checkpoints in memory, the approach assumes that the error that is currently detected did not strike before all the checkpoints still kept in memory, which would be fatal: in that latter case, all live checkpoints are corrupted, and one would have to re-execute the entire application from scratch. The probability of a fatal failure is evaluated in [3] for various error distribution laws and values of k. The third drawback of the approach is the most serious, and applies even without memory constraints, i.e., if we could store an infinite number of checkpoints in storage. The critical question is to determine which checkpoint is the last valid one. We need this information to safely recover from that point on. However, because of the detection latency (which is unknown), we do not know when the silent error has indeed occurred, hence we cannot identify the last valid checkpoint, unless some verification system is enforced.

This section introduces algorithms coupling verification and checkpointing, and shows how to analytically determine the best balance of verifications between checkpoints so as to minimize platform waste. In this (realistic) model, silent errors are detected only when some verification mechanism is executed. This approach is agnostic of the nature of this verification mechanism (checksum, error correcting code, coherence tests, etc.). This approach is also fully general-purpose, although application-specific information, if available, can always be used to decrease the cost of verification.

The simplest protocol (see Fig. 1.30) would be to perform a verification just before taking each checkpoint. If the verification succeeds, then one can safely store the checkpoint and mark it as *valid*. If the verification fails, then an error has struck since the last checkpoint, which was duly verified, and one can safely recover from that checkpoint to resume the execution of the application. This protocol with verifications eliminates fatal errors that would corrupt all live checkpoints and cause to restart execution from scratch. However, we still need to assume that both checkpoints and verifications are executed in a reliable mode.

There is room for optimization. Consider the second pattern illustrated in Fig. 1.31 with three verifications per checkpoint. There are three chunks of size w, each followed by a verification. Every third verification is followed by a checkpoint. We assume that $w = W/3$ to ensure that both patterns correspond to the same amount

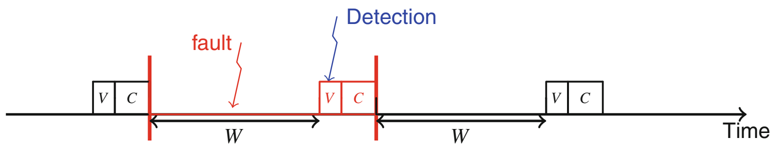


Fig. 1.30 The first pattern with one verification before each checkpoint

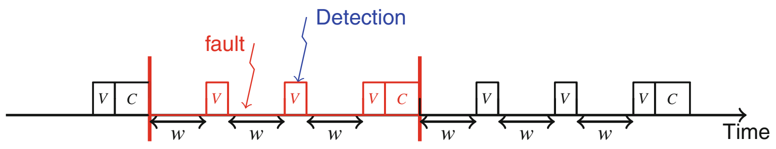


Fig. 1.31 The second pattern with three verifications per checkpoint

of work, W. Just as for the first pattern, a single checkpoint needs to be kept in memory, owing to the verifications. Also, as before, each error leads to re-executing the work since the last checkpoint. But detection occurs much more rapidly in the second pattern, owing to the intermediate verifications. If the error strikes in the first of the three chunks, it is detected by the first verification, and only the first chunk is re-executed. Similarly, if the error strikes in the second chunk (as illustrated in the figure), it is detected by the second verification, and the first two chunks are re-executed. The entire pattern of work needs to be re-executed only if the error strikes during the third chunk. On average, the amount of work to re-execute is $(1 + 2 + 3)w/3 = 2w = 2W/3$. On the contrary, in the first pattern of Fig. 1.30, the amount of work to re-execute always is W, because the error is never detected before the end of the pattern. Hence the second pattern leads to a 33 % gain in re-execution time. However, this comes at the price of three times as many verifications. This overhead is paid in every failure-free execution, and may be an overkill if the verification mechanism is too costly.

This little example shows that the optimization problem looks difficult. It can be stated as follows: given the cost of checkpointing C, recovery R, and verification V, what is the optimal strategy to minimize the (expectation of the) waste? A strategy is a periodic pattern of checkpoints and verifications, interleaved with work segments, that repeats over time. The length of the work segments also depends upon the platform MTBF μ. For example, with a single checkpoint and no verification (which corresponds to the classical approach for fail-stop failures), recall from Theorem 1.1 that the optimal length of the work segment can be approximated as $\sqrt{2\mu C}$. Given a periodic pattern with checkpoints and verifications, can we extend this formula and compute similar approximations?

We conclude this introduction by providing a practical example of the checkpoint and verification mechanisms under study. A nice instance of this approach is given by Chen [21], who deals with sparse iterative solvers. Chen considers a simple method such as the PCG, the Preconditioned Conjugate Gradient method, and aims at protecting the execution from arithmetic errors in the ALU. Chen's approach performs a periodic verification every d iterations, and a periodic checkpoint every $d \times c$ iterations, which is a particular case of the pattern with $p = 1$ and $q = c$. For PCG, the verification amounts to checking the orthogonality of two vectors and to recomputing and checking the residual, while the cost of checkpointing is that of storing three vectors. The cost of a checkpoint is smaller than the cost of the verification, which itself is smaller than the cost of an iteration, especially when the preconditioner requires much more flops than a sparse matrix-vector product. In this context, Chen [21] shows how to numerically estimate the best values of the parameters d and c. The results given in Sect. 1.6.3 show using equidistant verifications, as suggested in [21], is asymptotically optimal when using a pattern with a single checkpoint ($p = 1$), and enable to determine the best pattern with p checkpoints and q verifications as a function of C, R, and V, and the MTBF μ.

1.6.2 Other Approaches

In this section, we briefly survey other approaches to detect and/or correct silent errors. Considerable efforts have been directed at error-checking to reveal silent errors. Error detection is usually very costly. Hardware mechanisms, such as ECC memory, can detect and even correct a fraction of errors, but in practice they are complemented with software techniques. General-purpose techniques are based on replication, which we have already met in Sect. 1.4.2: using replication [31, 33, 66, 72], one can compare the results of both replicas and detect a silent error. Using TMR [56] would allow to correct the error (by voting) after detection. Note that another approach based on checkpointing and replication is proposed in [60], in order to detect and enable fast recovery of applications from both silent errors and hard errors.

Coming back to verification mechanisms, application-specific information can be very useful to enable ad-hoc solutions, that dramatically decrease the cost of detection. Many techniques have been advocated. They include memory scrubbing [48], but also ABFT techniques [8, 46, 68], such as coding for the sparse-matrix vector multiplication kernel [68], and coupling a higher-order with a lower-order scheme for Ordinary Differential Equations [6]. These methods can only detect an error but do not correct it. Self-stabilizing corrections after error detection in the conjugate gradient method are investigated by Sao and Vuduc [65]. Also, Heroux and Hoemmen [44] design a fault-tolerant GMRES capable of converging despite silent errors, and Bronevetsky and de Supinski [17] provide a comparative study of detection costs for iterative methods. Elliot et al. [30] combine partial redundancy and checkpointing, and confirm the benefit of dual and triple redundancy. The drawback is that twice the number of processing resources is required (for dual redundancy).

As already mentioned, the combined checkpoint/verification approach is agnostic of the underlying error-detection technique and takes the cost of verification as an input parameter to the model.

1.6.3 Optimal Pattern

In this section, we detail the performance model to assess the efficiency of any checkpoint/verification pattern. Then we show how to determine the best pattern.

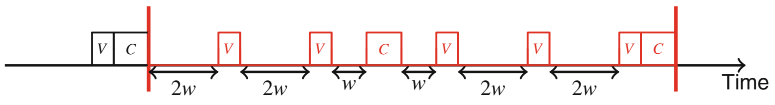


Fig. 1.32 The BALANCEDALGORITHM with five verifications for two checkpoints

1.6.3.1 Model for Patterns

Consider a periodic pattern with p checkpoints and q verifications, and whose total length is $S = pC + qV + W$. Here, W is the work that is executed during the whole pattern, and it is divided into several chunks that are each followed by a verification, or a checkpoint, or both. Checkpoints and verifications are at arbitrary location within the pattern. The only constraint is that the pattern always ends by a verification immediately followed by a checkpoint: this is to enforce that the last checkpoint is always valid, thereby ruling out the risk of a fatal failure. In the example of Fig. 1.31, we have three chunks of same size w, hence $W = 3w$ and $S = C + 3V + 3w$. The example of Fig. 1.32 represents a more complicated pattern, with two checkpoints and five verifications. The two checkpoints are equidistant in the pattern, and so are the five verifications, hence the six chunks of size either w or $2w$, for a total work $W = 10w$, and $S = 2C + 5V + 10w$. The rationale for using such chunk sizes in Fig. 1.32 is given in Sect. 1.6.3.2.

We compute the waste incurred by the use of a pattern similarly to what we did for fail-stop failures in Sect. 1.3.1. We consider a periodic pattern with p checkpoints, q verifications, work W, and total length $S = pC + qV + W$. We assume a a *selective reliability* model where checkpoint, recovery and verification are error-free operations. The input parameters are the following:

- the cost V of the verification mechanism;
- the cost C of a checkpoint;
- the cost R of a recovery;
- the platform MTBF μ.

We aim at deriving the counterpart of Eq. (1.33) for silent errors. We easily derive that the waste in a fault-free execution is $\text{WASTE}_{\text{ff}} = \frac{pC+qV}{S}$, and that the waste due to silent errors striking during execution. is which is the waste due to checkpointing is $\text{WASTE}_{\text{fail}} = \frac{T_{\text{lost}}}{\mu}$, where T_{lost} is the expected time lost due to each error. The value of T_{lost} is more complicated to compute than for fail-stop errors, because it depends upon which pattern is used. Before computing T_{lost} for arbitrary values of p and q in Sect. 1.6.3.2, we give two examples.

The first example is for the simple pattern of Fig. 1.30. We have $p = q = 1$, a single chunk of size $w = W$, and a pattern of size $S = C + V + W$. Computing T_{lost} for this pattern goes as follows: whenever an error strikes, it is detected at the end of the work, during the verification. We first recover from the last checkpoint, then re-execute the entire work, and finally redo the verification. This leads to $T_{\text{lost}} = R + W + V = R + S - C$. From Eq. (1.33), we obtain that

$$\textsc{Waste} = 1 - \left(1 - \frac{R+S-C}{\mu}\right)\left(1 - \frac{C+V}{S}\right) = aS + \frac{b}{S} + c, \qquad (1.49)$$

where $a = \frac{1}{\mu}$, $b = (C+V)(1 + \frac{C-R}{\mu})$ and $c = \frac{R-V-2C}{\mu}$. The value that minimizes the waste is $S = S_{\text{opt}}$, and the optimal waste is $\textsc{Waste}_{\text{opt}}$, where

$$S_{\text{opt}} = \sqrt{\frac{b}{a}} = \sqrt{(C+V)(\mu+C-R)} \quad \text{and} \quad \textsc{Waste}_{\text{opt}} = 2\sqrt{ab} + c. \qquad (1.50)$$

Just as for fail-stop failures, we point out that this approach leads to a first-order approximation of the optimal pattern, not to an optimal value. As always, the approach is valid when μ is large in front of S, and of all parameters R, C and V. When this is the case, we derive that $S_{\text{opt}} \approx \sqrt{(C+V)\mu}$ and $\textsc{Waste}_{\text{opt}} \approx 2\sqrt{\frac{C+V}{\mu}}$. It is very interesting to make a comparison with the optimal checkpointing period T_{FO} (see Eq. (1.9)) when dealing with fatal failures: we had $T_{\text{FO}} \approx \sqrt{2C\mu}$. In essence, the factor 2 comes from the fact that we re-execute only half the period on average with a fatal failure, because the detection is instantaneous. In our case, we always have to re-execute the entire pattern. And of course, we have to replace C by $C+V$, to account for the cost of the verification mechanism.

The second example is for the BALANCEDALGORITHM illustrated in Fig. 1.32. We have $p = 2$, $q = 5$, six chunks of size w or $2w$, $W = 10w$, and a pattern of size $S = 2C + 5V + W$. Note that it may now be the case that we store an invalid checkpoint, if the error strikes during the third chunk (of size w, just before the non-verified checkpoint), and therefore we must keep two checkpoints in memory to avoid the risk of fatal failures. When the verification is done at the end of the fourth chunk, if it is correct, then we can mark the preceding checkpoint as valid and keep only this checkpoint in memory. Because $q > p$, there are never two consecutive checkpoints without a verification between them, and at most two checkpoints need to be kept in memory. The time lost due to an error depends upon where it strikes:

- With probability $2w/W$, the error strikes in the first chunk. It is detected by the first verification, and the time lost is $R + 2w + V$, since we recover, and re-execute the work and the verification.
- With probability $2w/W$, the error strikes in the second chunk. It is detected by the second verification, and the time lost is $R + 4w + 2V$, since we recover, re-execute the work and both verifications.
- With probability w/W, the error strikes in the third chunk. It is detected by the third verification, and we roll back to the last checkpoint, recover and verify it. We find it invalid, because the error struck before taking it. We roll back to the beginning of the pattern and recover from that checkpoint. The time lost is $2R + 6w + C + 4V$, since we recover twice, re-execute the work up to the third verification, redo the checkpoint and the three verifications, and add the verification of the invalid checkpoint.
- With probability w/W, the error strikes in the fourth chunk. It is detected by the third verification. We roll back to the previous checkpoint, recover and verify it.

In this case, it is valid, since the error struck after the checkpoint. The time lost is $R + w + 2V$.

- With probability $2w/W$, the error strikes in the fifth chunk. Because there was a valid verification after the checkpoint, we do not need to verify it again, and the time lost is $R + 3w + 2V$.
- With probability $2w/W$, the error strikes in the sixth and last chunk. A similar reasoning shows that the time lost is $R + 5w + 3V$.

Averaging over all cases, we derive that $T_{\text{lost}} = \frac{11R}{10} + \frac{35w}{10} + \frac{C}{10} + \frac{22V}{10}$. We then proceed as with the first example to derive the optimal size S of the pattern. We obtain $S_{\text{opt}} = \sqrt{\frac{b}{a}}$ and $\text{WASTE}_{\text{opt}} = 2\sqrt{ab} + c$ (see Eq. (1.50)), where $a = \frac{7\mu}{20}$, $b = (2C + 5V)(1 - \frac{1}{20\mu}(22R - 12C + 9V))$ and $c = \frac{1}{20\mu}(22R - 26C - 17V)$. When μ is large, we have $S_{\text{opt}} \approx \sqrt{\frac{20}{7}(2C + 5V)\mu}$ and $\text{WASTE}_{\text{opt}} \approx 2\sqrt{\frac{7(2C+5V)}{20\mu}}$.

1.6.3.2 Optimal Pattern

In this section, we generalize from the examples and provide a generic expression for the waste when the platform MTBF μ is large in front of all resilience parameters R, C and V. Consider a general pattern of size $S = pC + qV + W$, with $p \leq q$. We have $\text{WASTE}_{\text{ff}} = \frac{o_{\text{ff}}}{S}$, where $o_{\text{ff}} = pC + qV$ is the fault-free overhead due to inserting p checkpoints and q verifications within the pattern. We also have $\text{WASTE}_{\text{fail}} = \frac{T_{\text{lost}}}{\mu}$, where T_{lost} is the time lost each time an error strikes and includes two components: re-executing a fraction of the total work W of the pattern, and computing additional verifications, checkpoints and recoveries (see the previous examples). The general form of T_{lost} is thus $T_{\text{lost}} = f_{\text{re}}W + \alpha$ where f_{re} stands for *fraction* of work that is *re-executed* due to failures; α is a constant that is a linear combination of C, V and R. For the first example (Fig. 1.30), we have $f_{\text{re}} = 1$. For the second example (Fig. 1.32), we have $f_{\text{re}} = \frac{7}{20}$ (recall that $w = W/10$). For convenience, we use an equivalent form

$$T_{\text{lost}} = f_{\text{re}}S + \beta, \tag{1.51}$$

where $\beta = \alpha - f_{\text{re}}(pC + qV)$ is another constant. When the platform MTBF μ is large in front of all resilience parameters R, C and V, we can identify the dominant term in the optimal waste $\text{WASTE}_{\text{opt}}$. Indeed, in that case, the constant β becomes negligible in front of μ, and we derive that

$$S_{\text{opt}} = \sqrt{\frac{o_{\text{ff}}}{f_{\text{re}}}} \times \sqrt{\mu} + o(\sqrt{\mu}), \tag{1.52}$$

and that the optimal waste is

$$\text{WASTE}_{\text{opt}} = 2\sqrt{o_{\text{ff}}f_{\text{re}}}\sqrt{\frac{1}{\mu}} + o(\sqrt{\frac{1}{\mu}}). \tag{1.53}$$

This equation shows that the optimal pattern when μ is large is obtained when the product $o_{\mathrm{ff}} f_{\mathrm{re}}$ is minimal. This calls for a trade-off, as a smaller value of o_{ff} with few checkpoints and verifications leads to a larger re-execution time, hence to a larger value of f_{re}. For instance, coming back to the examples of Figs. 1.30 and 1.32, we readily see that the second pattern is better than the first one for large values of μ whenever $V > 2C/5$, which corresponds to the condition $\frac{7}{20} \times (5V + 2C) > 1 \times (V + C)$.

For a general pattern of size $S = pC + qV + W$, with $p \le q$, we always have $o_{\mathrm{ff}} = o_{\mathrm{ff}}(p, q) = pC + qV$ and we aim at (asymptotically) minimizing $f_{\mathrm{re}} = f_{\mathrm{re}}(p, q)$, the expected fraction of the work that is re-executed, by determining the optimal size of each work segment. It turns out that $f_{\mathrm{re}}(p, q)$ is minimized when the pattern has pq same-size intervals and when the checkpoints and verifications are equally spaced among these intervals as in the BalancedAlgorithm, in which case $f_{\mathrm{re}}(p, q) = \frac{p+q}{2pq}$. We first prove this result for $p = 1$ before moving to the general case. Finally, we explain how to choose the optimal pattern given values of C and V.

Theorem 1.2 *The minimal value of $f_{re}(1, q)$ is obtained for same-size chunks and it is $f_{re}(1, q) = \frac{q+1}{2q}$.*

Proof For $q = 1$, we already know from the study of the first example that $f_{\mathrm{re}}(1, 1) = 1$. Consider a pattern with $q \ge 2$ verifications, executing a total work W. Let $\alpha_i W$ be the size of the i-th chunk, where $\sum_{i=1}^{q} \alpha_i = 1$ (see Fig. 1.33). We compute the expected fraction of work that is re-executed when a failure strikes the pattern as follows. With probability α_i, the failure strikes in the i-th chunk. The error is detected by the i-th verification, we roll back to the beginning of the pattern, so we re-execute the first i chunks. Altogether, the amount of work that is re-executed is $\sum_{i=1}^{q} \left(\alpha_i \sum_{j=1}^{i} \alpha_j W\right)$, hence

$$f_{\mathrm{re}}(1, q) = \sum_{i=1}^{q} \left(\alpha_i \sum_{j=1}^{i} \alpha_j\right). \tag{1.54}$$

What is the minimal value of $f_{\mathrm{re}}(1, q)$ in Eq. (1.54) under the constraint $\sum_{i=1}^{q} \alpha_i = 1$? We rewrite

$$f_{\mathrm{re}}(1, q) = \frac{1}{2}\left(\sum_{i=1}^{q} \alpha_i\right)^2 + \frac{1}{2}\sum_{i=1}^{q} \alpha_i^2 = \frac{1}{2}\left(1 + \sum_{i=1}^{q} \alpha_i^2\right),$$

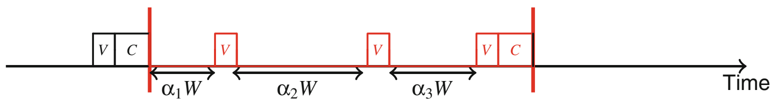


Fig. 1.33 A pattern with different-size chunks, for $p = 1$ and $q = 3$

and by convexity, we see that f_{re} is minimal when all the α_i's have the same value $1/q$. In that case, we derive that $f_{\text{re}}(1, q) = \frac{1}{2}(1 + \sum_{i=1}^{q} \frac{1}{q^2}) = \frac{q+1}{2q}$, which concludes the proof.

When $p = 1$, BALANCEDALGORITHM uses q same-size chunks. Theorem 1.2 shows that this is optimal.

Theorem 1.3 *For a pattern with $p \geq 1$, the minimal value of $f_{re}(p, q)$ is $f_{re}(p, q) = \frac{p+q}{2pq}$, and it is obtained with the* BALANCEDALGORITHM.

Proof Consider an arbitrary pattern with p checkpoints, $q \geq p$ verifications and total work W. The distribution of the checkpoints and verifications is unknown, and different-size chunks can be used. The only assumption is that the pattern ends by a verification followed by a checkpoint.

The main idea of the proof is to compare the gain in re-execution time due to the $p - 1$ intermediate checkpoints. Let $f_{\text{re}}^{(p)}$ be the fraction of work that is re-executed for the pattern, and let $f_{\text{re}}^{(1)}$ be the fraction of work that is re-executed for the same pattern, but where the $p - 1$ first checkpoints have been suppressed. Clearly, $f_{\text{re}}^{(p)}$ is smaller than $f_{\text{re}}^{(1)}$, because the additional checkpoints save some roll-backs, and we aim at maximizing their difference.

In the original pattern, let $\alpha_i W$ be the amount of work before the i-th checkpoint, for $1 \leq i \leq p$ (and with $\sum_{i=1}^{p} \alpha_i = 1$). Figure 1.34 presents an example with $p = 3$. What is the gain due to the presence of the $p - 1$ intermediate checkpoints? If an error strikes before the first checkpoint, which happens with probability α_1, there is no gain, because we always rollback from the beginning of the pattern. This is true regardless of the number and distribution of the q verifications in the pattern. If an error strikes after the first checkpoint and before the second one, which happens with probability α_2, we do have a gain: instead of rolling back to the beginning of the pattern, we rollback only to the first checkpoint, which saves $\alpha_1 W$ units of re-executed work. Again, this is true regardless of the number and distribution of the q verifications in the pattern. For the general case, if an error strikes after the $(i - 1)$-th checkpoint and before the i-th one, which happens with probability α_i, the gain is $\sum_{j=1}^{i-1} \alpha_j W$. We derive that

$$f_{\text{re}}^{(1)} - f_{\text{re}}^{(p)} = \sum_{i=1}^{p} \left(\alpha_i \sum_{j=1}^{i-1} \alpha_j \right).$$

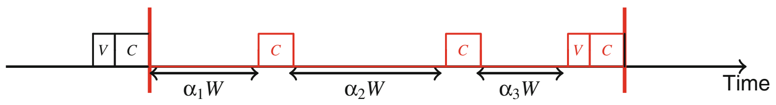


Fig. 1.34 A pattern with different-size chunks, with 3 checkpoints (we do not show where intermediate verifications are located)

Similarly to the proof of Theorem 1.2, we have

$$\sum_{i=1}^{p}\left(\alpha_i \sum_{j=1}^{i-1}\alpha_j\right) = \frac{1}{2}\left(\left(\sum_{i=1}^{p}\alpha_i\right)^2 - \sum_{i=1}^{p}\alpha_i^2\right) = \frac{1}{2}\left(1 - \sum_{i=1}^{p}\alpha_i^2\right)$$

and by convexity, the difference $f_{\text{re}}^{(1)} - f_{\text{re}}^{(p)}$ is maximal when $\alpha_i = 1/p$ for all i. In that latter case, $f_{\text{re}}^{(1)} - f_{\text{re}}^{(p)} = \sum_{i=1}^{p}(i-1)/p^2 = (p-1)/p^2$. This result shows that the checkpoints should be equipartitioned in the pattern, regardless of the location of the verifications.

To conclude the proof, we now use Theorem 1.2: to minimize the value of $f_{\text{re}}^{(1)}$, we should equipartition the verifications too. In that case, we have $f_{\text{re}}^{(1)} = \frac{q+1}{2q}$ and $f_{\text{re}}^{(p)} = \frac{q+1}{2q} - \frac{p-1}{2p} = \frac{q+p}{2pq}$, which concludes the proof.

Theorem 1.3 shows that BALANCEDALGORITHM is the optimal pattern with p checkpoints and q verifications when μ is large. An important consequence of this result is that we never need to keep more than two checkpoints in memory when $p \leq q$, because it is optimal to regularly interleave checkpoints and verifications.

To conclude this study, we outline a simple procedure to determine the best pattern. We start with the following result:

Theorem 1.4 *Assume that μ is large in front of C, R and V, and that $\sqrt{\frac{V}{C}}$ is a rational number $\frac{u}{v}$, where u and v are relatively prime. Then the optimal pattern S_{opt} is obtained with the* BALANCEDALGORITHM, *using $p = u$ checkpoints, $q = v$ verifications, and pq equal-size chunks of total length $\sqrt{\frac{2pq(pC+qV)\mu}{p+q}}$.*

We prove this theorem before discussing the case where $\sqrt{\frac{V}{C}}$ is not a rational number.

Proof Assume that $V = \gamma C$, where $\gamma = \frac{u^2}{v^2}$, with u and v relatively prime integers. Then, the product $o_{\text{ff}} f_{\text{re}}$ can be expressed as

$$o_{\text{ff}} f_{\text{re}} = \frac{p+q}{2pq}(pC + qV) = C \times \frac{p+q}{2}\left(\frac{1}{q} + \frac{\gamma}{p}\right).$$

Therefore, given a value of C and a value of V, i.e., given γ, the goal is to minimize the function $\frac{p+q}{2}\left(\frac{1}{q} + \frac{\gamma}{p}\right)$ with $1 \leq p \leq q$, and p, q taking integer values.

Let $p = \lambda \times q$. Then we aim at minimizing

$$\frac{1+\lambda}{2}\left(1 + \frac{\gamma}{\lambda}\right) = \frac{\lambda}{2} + \frac{\gamma}{2\lambda} + \frac{1+\gamma}{2},$$

and we obtain $\lambda_{opt} = \sqrt{\gamma} = \sqrt{\frac{V}{C}} = \frac{u}{v}$. Hence the best pattern is that returned by the BALANCEDALGORITHM with $p = u$ checkpoints and $q = v$ verifications. This pattern uses pq equal-size chunks whose total length is given by Eq. (1.52), hence the result.

For instance, for $V = 4$ and $C = 9$, we obtain $\lambda_{opt} = \sqrt{\frac{V}{C}} = \frac{2}{3}$, and a balanced pattern with $p = 2$ and $q = 3$ is optimal. This pattern will have 6 equal-size chunks whose total length is $\sqrt{\frac{12(2C+3V)\mu}{5}} = 6\sqrt{2\mu}$. However, if $V = C = 9$, then $\lambda_{opt} = 1$ and the best solution is the base algorithm with $p = q = 1$ and a single chunk of size $\sqrt{(C+V)\mu} = \sqrt{13\mu}$.

In some cases, $\lambda_{opt} = \sqrt{\frac{V}{C}}$ may not be a rational number, and we need to find good approximations of p and q in order to minimize the asymptotic waste. A solution is to try all reasonable values of q, say from 1 to 50, and to compute the asymptotic waste achieved with $p_1 = \lfloor \lambda_{opt} \times q \rfloor$ and $p_2 = \lceil \lambda_{opt} \times q \rceil$, hence testing at most 100 configurations (p, q). Altogether, we can compute the best pattern with $q \leq 50$ in constant time.

1.7 Conclusion

This chapter presented an overview of the fault tolerance techniques most frequently used in HPC. Large-scale machines consist of components that are robust but not perfectly reliable. They combine a number of components that grows exponentially and will suffer from failures at a rate inversely proportional to that number. Thus, to cope with such failures, we presented two sets of approaches:

- On the one hand, middleware, hardware, and libraries can implement general techniques to conceal failures from higher levels of the software stack, enabling the execution of genuine applications not designed for fault tolerance. Behind such approaches, periodic checkpointing with rollback-recovery is the most popular technique used in HPC, because of its multiple uses (fault tolerance, but also post-mortem analysis of behavior, and validation), and potential better usage of resources. We presented many variations on the protocols of these methods, and discussed practical issues, like checkpoint creation and storage.
  At the heart of periodic checkpointing with rollback recovery lays an optimization problem: rollback happens when failures occurs; it induces re-execution, hence resource consumption to tolerate the failure that occurred; frequent checkpointing reduces that resource consumption. However, checkpointing also consumes resources, even when failures do not occur; thus checkpointing too often becomes a source of inefficiency. We presented probabilistic performance models that allow to deduce the optimal trade-off between frequent checkpoints and high failure-free overheads for the large collection of protocols that we presented before.
  The costs of checkpointing and coordinated rollback are the major source of overheads in these protocols: future large scale systems can hope to rely on rollback recovery only if the time spent in checkpointing and rolling-back can be kept orders of magnitude under the time between failures. The last protocol we studied, that

uses the memory of the peers to store checkpoints, aims precisely at this. But since the checkpoints become stored in memory, that storage becomes unreliable, and mitigating the risk of a non-recoverable failure reenters the trade-off. Here again, probabilistic models allow to quantify this risk, to guide the decision of resource usage optimization.

- On the other hand, by letting the hardware and middleware expose the failures to the higher-level libraries and the application (while tolerating failures at their level to continue providing their service), we showed how a much better efficiency can be expected. We presented briefly the current efforts pursued in the MPI standardization body to allow such behavior in high-performance libraries and application. Then, we illustrated over complex realistic examples how some applications can take advantage of failures awareness to provide high efficiency and fault tolerance. Because these techniques are application-specific, many applications may not be capable of using them. To address this issue, we presented a composition technique that enables libraries to mask failures that are exposed to them from a non fault-tolerant application. That composition relies on the general rollback-recovery technique, but allows to disable periodic checkpointing during long phases where the library controls the execution, featuring the high-efficiency of application-specific techniques together with the generality of rollback-recovery.

To conclude, we considered the case of silent errors: silent errors, by definition, do not manifest as a failure at the moment they strike; the application may slowly diverge from a correct behavior, and the data be corrupted before the error is detected. Because of this, they pose a new challenge to fault tolerance techniques. We presented how multiple rollback points may become necessary, and how harder it becomes to decide when to rollback. We also presented how application-specific techniques can mitigate these issue by providing data consistency checkers (validators), allowing to detect the occurrence of a silent error not necessarily when it happens, but before critical steps.

Designing a fault-tolerant system is a complex task that introduces new programming and optimization challenges. However, the combination of the whole spectrum of techniques, from application-specific to general tools, at different levels of the software stack, allows to tolerate a large range of failures with the high efficiency expected in HPC. In the rest of this book, experts in fault tolerance and HPC have contributed with technical chapters, in which they dig deeper into some of the topics that were overviewed in this chapter. Their contributions present the most recent advances in an intellectually buoyant research field. We hope they will inspire innovative solutions and the adoption of sound approaches to tolerate failures at large scale.

Acknowledgments Yves Robert is with the Institut Universitaire de France. The research presented in this chapter was supported in part by the French ANR (*Rescue* project) and by contracts with the DOE through the SUPER-SCIDAC project, and the CREST project of the Japan Science and Technology Agency (JST). This chapter has borrowed material from publications co-authored with many colleagues and PhD students, and the authors would like to thank Guillaume Aupy, Anne Benoit, George Bosilca, Aurélien Bouteiller, Aurélien Cavelan, Franck Cappello, Henri Casanova, Amina Guermouche, Saurabh K. Raina, Hongyang Sun, Frédéric Vivien, and Dounia Zaidouni.

References

1. Amdahl G (1967) The validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS conference proceedings, vol 30. AFIPS Press, pp 483–485
2. Aupy G, Robert Y, Vivien F, Zaidouni D (2013) Checkpointing strategies with prediction windows. In: IEEE 19th Pacific Rim international symposium on dependable computing (PRDC), 2013. IEEE, pp 1–10
3. Aupy G, Benoit A, Herault T, Robert Y, Vivien F, Zaidouni D (2013) On the combination of silent error detection and checkpointing. In: PRDC 2013, the 19th IEEE Pacific Rim international symposium on dependable computing. IEEE Computer Society Press
4. Aupy G, Robert Y, Vivien F, Zaidouni D (2014) Checkpointing algorithms and fault prediction. J Parallel Distrib Comput 74(2):2048–2064
5. Bautista-Gomez L, Tsuboi S, Komatitsch D, Cappello F, Maruyama N, Matsuoka S (2011) FTI: high performance fault tolerance interface for hybrid systems. In: International conference high performance computing, networking, storage and analysis SC'11
6. Benson AR, Schmit S, Schreiber R (2013) Silent error detection in numerical time-stepping schemes. CoRR, abs/1312.2674
7. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK users' guide. SIAM
8. Bosilca G, Delmas R, Dongarra J, Langou J (2009) Algorithm-based fault tolerance applied to high performance computing. J Parallel Distrib Comput 69(4):410–416
9. Bosilca G, Bouteiller A, Herault T, Robert Y, Dongarra JJ (2014) Assessing the impact of ABFT and checkpoint composite strategies. In: 2014 IEEE international parallel and distributed processing symposium workshops, Phoenix, AZ, USA, May 19–23 2014, pp 679–688
10. Bosilca G, Bouteiller A, Brunet E, Cappello F, Dongarra J, Guermouche A, Herault T, Robert Y, Vivien F, Zaidouni D (2014) Unified model for assessing checkpointing protocols at extreme-scale. Concurr Comput Pract Exp 26(17):925–957
11. Bosilca G, Bouteiller A, Herault T, Robert Y, Dongarra JJ (2015) Composing resilience techniques: ABFT, periodic and incremental checkpointing. IJNC 5(1):2–25
12. Bougeret M, Casanova H, Rabie M, Robert Y, Vivien F (2011) Checkpointing strategies for parallel jobs. In: Proceedings of SC'11
13. Bouteiller A, Herault T, Krawezik G, Lemarinier P, Cappello F (2006) MPICH-V: a multiprotocol fault tolerant MPI. IJHPCA 20(3):319–333
14. Bouteiller A, Bosilca G, Dongarra J (2010) Redesigning the message logging model for high performance. Concurr Comput Pract Exp 22(16):2196–2211
15. Bouteiller A, Herault T, Bosilca G, Dongarra JJ (2011) Correlated set coordination in fault tolerant message logging protocols. In: Proceedings of Euro-Par'11 (II). LNCS, vol 6853. Springer, pp 51–64
16. Bouteiller A, Herault T, Bosilca G, Du P, Dongarra J (2015) Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. ACM Trans Parallel Comput 1(2):10:1–10:28
17. Bronevetsky G, de Supinski B (2008) Soft error vulnerability of iterative linear algebra methods. In: Proceedings of 22nd international conference on supercomputing, ICS '08. ACM, pp 155–164
18. Cannon LE (1969) A cellular computer to implement the Kalman filter algorithm. Ph.D. thesis, Montana State University
19. Casanova H, Robert Y, Vivien F, Zaidouni D (2012) Combining process replication and checkpointing for resilience on exascale systems. Research report RR-7951, INRIA
20. Chandy KM, Lamport L (1985) Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst 3(1):63–75
21. Chen Z (2013) Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In: Proceedings of 18th ACM SIGPLAN symposium on principles and practice of parallel programming, PPoPP '13. ACM, pp 167–176

22. Chen Z, Dongarra J (2006) Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In: Proceedings of the 20th international conference on parallel and distributed processing, IPDPS'06, Washington, DC, USA. IEEE Computer Society, pp 97–97
23. Chen Z, Dongarra J (2008) Algorithm-based fault tolerance for fail-stop failures. IEEE TPDS 19(12):1628–1641
24. Choi J, Demmel J, Dhillon I, Dongarra J, Ostrouchov S, Petitet A, Stanley K, Walker D, Whaley R (1996) ScaLAPACK: a portable linear algebra library for distributed memory computers-design issues and performance. Comput Phys Commun 97(1–2):1–15
25. Daly JT (2004) A higher order estimate of the optimum checkpoint interval for restart dumps. FGCS 22(3):303–312
26. Davies T, Karlsson C, Liu H, Ding C, Chen Z (2011) High performance linpack benchmark: a fault tolerant implementation without checkpointing. In: Proceedings of the international conference on supercomputing, ICS '11. ACM, New York, pp 162–171
27. Dongarra J, Beckman P, Aerts P, Cappello F, Lippert T, Matsuoka S, Messina P, Moore T, Stevens R, Trefethen A, Valero M (2009) The international exascale software project: a call to cooperative action by the global high-performance community. Int J High Perform Comput Appl 23(4):309–322
28. Dongarra J, Herault T, Robert Y (2014) Performance and reliability trade-offs for the double checkpointing algorithm. Int J Netw Comput 4(1):23–41
29. Du P, Bouteiller A, Bosilca G, Herault T, Dongarra J (2012) Algorithm-based fault tolerance for dense matrix factorizations. In: Proceedings of the 17th ACM SIGPLAN symposium on principles and practice of parallel programming, PPOPP 2012, New Orleans, LA, USA, 25–29 February 2012, pp 225–234
30. Elliott J, Kharbas K, Fiala D, Mueller F, Ferreira K, Engelmann C (2012) Combining partial redundancy and checkpointing for HPC. In: Proceedings of ICDCS '12. IEEE Computer Society
31. Engelmann C, Ong HH, Scorr SL (2009) The case for modular redundancy in large-scale highh performance computing systems. In: Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks (PDCN), pp 189–194
32. Esteban Meneses CLM, Kalé LV (2010) Team-based message logging: preliminary results. In: Workshop resilience in clusters, clouds, and grids (CCGRID 2010)
33. Ferreira K, Stearley J, Laros JHI, Oldfield R, Pedretti, K, Brightwell R, Riesen R, Bridges, PG, Arnold D (2011) Evaluating the viability of process replication reliability for exascale systems. In: Proceedings of the ACM/IEEE on supercomputing
34. Flajolet P, Grabner PJ, Kirschenhofer P, Prodinger H (1995) On Ramanujan's Q-function. J Comput Appl Math 58:103–116
35. Gainaru A, Cappello F, Kramer W (2012) Taming of the shrew: modeling the normal and faulty behavior of large-scale HPC systems. In: Proceedings of IPDPS'12
36. Guermouche A, Ropars T, Snir M, Cappello F (to appear) HydEE: failure containment without event logging for large scale send-deterministic MPI applications. In: Proceedings of IEEE IPDPS 2012
37. Gustafson JL (1988) Reevaluating Amdahl's law. IBM Syst J 31(5):532–533
38. Gärtner F (1999) Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput Surv 31(1):1–26
39. Hacker TJ, Romero F, Carothers CD (2009) An analysis of clustered failures on large supercomputing systems. J Parallel Distrib Comput 69(7):652–665
40. Hakkarinen D, Chen Z (2010) Algorithmic cholesky factorization fault recovery. In: 2010 IEEE International symposium on parallel distributed processing (IPDPS). IEEE, Atlanta, pp 1–10
41. Hargrove PH, Duell JC (2006) Berkeley lab checkpoint/restart (BLCR) for linux clusters. In: Proceedings of SciDAC 2006
42. Heath T, Martin RP, Nguyen TD (2002) Improving cluster availability using workstation validation. SIGMETRICS Perform Eval Rev 30(1):217–227

43. Heien R, Kondo D, Gainaru A, LaPine D, Kramer B, Cappello F (2011) Modeling and tolerating heterogeneous failures on large parallel system. In: Proceedings of the IEEE/ACM conference on supercomputing (SC)
44. Heroux M, Hoemmen M (2011) Fault-tolerant iterative methods via selective reliability. Research report SAND2011-3915 C, Sandia National Laboratories
45. Huang K-H, Abraham J (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput C-33(6):518–528
46. Huang K-H, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput 33(6):518–528
47. Hursey J, Squyres J, Mattox T, Lumsdaine A (2007) The design and implementation of checkpoint/restart process fault tolerance for open MPI. In: IEEE international parallel and distributed processing symposium, 2007. IPDPS. pp 1–8
48. Hwang AA, Stefanovici IA, Schroeder B (2012) Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design. SIGARCH Comput Archit News 40(1):111–122
49. Kella O, Stadje W (2006) Superposition of renewal processes and an application to multi-server queues. Stat Probab Lett 76(17):1914–1924
50. Kingsley G, Beck M, Plank JS (1995) Compiler-assisted checkpoint optimization using SUIF. In: First SUIF compiler workshop
51. Kondo D, Chien A, Casanova H (2007) Scheduling task parallel applications for rapid application turnaround on enterprise desktop grids. J Grid Comput 5(4):379–405
52. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
53. Li C-C, Fuchs W (1990) Catch-compiler-assisted techniques for checkpointing. In: 20th international symposium fault-tolerant computing, 1990. FTCS-20. Digest of papers, pp 74–81
54. Liu Y, Nassar R, Leangsuksun C, Naksinehaboon N, Paun M, Scott S (2008) An optimal checkpoint/restart model for a large scale high performance computing system. In: IPDPS'08. IEEE
55. Lu G, Zheng Z, Chien AA (2013) When is multi-version checkpointing needed. In: 3rd Workshop for fault-tolerance at extreme scale (FTXS). ACM Press. https://sites.google.com/site/uchicagolssg/lssg/research/gvr
56. Lyons RE, Vanderkulk W (1962) The use of triple-modular redundancy to improve computer reliability. IBM J Res Dev 6(2):200–209
57. Moody A, Bronevetsky G, Mohror K, Supinski B (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: International conference high performance computing, networking, storage and analysis SC'10
58. Moody A, Bronevetsky G, Mohror K, Supinski BR de (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings of the ACM/IEEE conference SC, pp 1–11
59. Ni X, Meneses E, Kalé LV (2012) Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In: Proceedings of IEEE international conference on cluster computing. IEEE Computer Society
60. Ni X, Meneses E, Jain N, Kalé LV (2013) ACR: automatic checkpoint/restart for soft and hard error protection. In: Proceedings of international conference high performance computing, networking, storage and analysis, SC '13. ACM
61. O'Gorman T (1994) The effect of cosmic rays on the soft error rate of a DRAM at ground level. IEEE Trans Electron Devices 41(4):553–557
62. Plank JS, Beck M, Kingsley G (1995) Compiler-assisted memory exclusion for fast checkpointing. IEEE Tech Comm Oper Syst Appl Environ 7:10–14
63. Rodríguez G, Martín MJ, González P, Touriño J, Doallo R (2010) CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. Concurr Comput Pract Exp 22(6):749–766
64. Ross SM (2009) Introduction to probability models, 8th edn. Academic Press, San Diego
65. Sao P, Vuduc R (2013) Self-stabilizing iterative solvers. In: Proceedings of ScalA '13. ACM

66. Schroeder B, Gibson G (2007) Understanding failures in petascale computers. J Phys Conf Ser 78(1):188–198
67. Schroeder B, Gibson GA (2006) A large-scale study of failures in high-performance computing systems. In: Proceedings of DSN, pp 249–258
68. Shantharam M, Srinivasmurthy S, Raghavan P (2012) Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In: Proceedings of ICS '12. ACM
69. Young JW (1974) A first order approximation to the optimum checkpoint interval. Commun ACM 17(9):530–531
70. Yu L, Zheng Z, Lan Z, Coghlan S (2011) Practical online failure prediction for blue gene/p: period-based vs event-driven. In: Dependable systems and networks workshops (DSN-W), pp 259–264
71. Zheng G, Shi L, Kalé LV (2004) FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In: Proceedings of IEEE international conference on cluster computing. IEEE Computer Society
72. Zheng Z, Lan Z (2009) Reliability-aware scalability models for high performance computing. In: Proceedings of the IEEE conference on cluster computing
73. Zheng Z, Lan Z, Gupta R, Coghlan S, Beckman P (2010) A practical failure prediction with location and lead time for blue gene/p. In: Dependable systems and networks workshops (DSN-W), pp 15–22
74. Ziegler J, Muhlfeld H, Montrose C, Curtis H, O'Gorman T, Ross J (1996) Accelerated testing for cosmic soft-error rate. IBM J Res Dev 40(1):51–72
75. Ziegler J, Nelson M, Shell J, Peterson R, Gelderloos C, Muhlfeld H, Montrose C (1998) Cosmic ray soft error rates of 16-Mb DRAM memory chips. IEEE J Solid-State Circuits 33(2):246–252
76. Ziegler JF, Curtis HW, Muhlfeld HP, Montrose CJ, Chin B (1996) IBM experiments in soft fails in computer electronics. IBM J Res Dev 40(1):3–18