Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Fault Tolerance for High-Performance Applications Using Structured Parallelism Models

Carlo Bertolli

Supervisor
Marco Vanneschi

October 7, 2008

Contents

List of Figures

List of Tables

A Soraya

Em demano: què és preferible:
un passament mediocre, alegroi i conformat,
o una obsessió com aquesta, apassionada, tensa, obsessionant?
Josep Pla

Acknowledgments

My main thanks go to my supervisor Marco Vanneschi: his high didactic and research skills both with his patience are the key to understand the ambition behind this thesis, its methodology and its results.

I am indebted with Prof. Massimo Coppola for teaching me the "dark side" of a researcher life: his (very patient) teaching and his comments are included in the main contributions of this thesis.

A special thanks goes to Joaquim Gabarró for teaching me his research approach and methodology, which actually represented an improvement of the quality of this thesis.

Finally, I would like to thank my past and present colleagues (Alessio, Massimiliano, Gianmarco, Francesco, Silvia, Gabriele and Daniele), which contributions are scattered on the whole thesis, my parents and my relatives for supporting me psychologically and economically during my PhD.

Abstract

In the last years parallel computing has increasingly exploited the high-level models of structured parallel programming, an example of which are algorithmic skeletons. This trend has been motivated by the properties featuring structured parallelism models, which can be used to derive several (static and dynamic) optimizations at various implementation levels. In this thesis we study the properties of structured parallel models useful for attacking the issue of providing a fault tolerance support oriented towards High-Performance applications. This issue has been traditionally faced in two ways: (i) in the context of unstructured parallelism models (e.g. MPI), which computation model is essentially based on a distributed set of processes communicating through message-passing, with an approach based on checkpointing and rollback recovery or software replication; (ii) in the context of high-level models, based on a specific parallelism model (e.g. data-flow) and/or an implementation model (e.g. master-slave), by introducing specific techniques based on the properties of the programming and computation models themselves. In this thesis we make a step towards a more abstract viewpoint and we highlight the properties of structured parallel models interesting for fault tolerance purposes. We consider two classes of parallel programs (namely task parallel and data parallel) and we introduce a fault tolerance support based on checkpointing and rollback recovery. The support is derived according to the high-level properties of the parallel models: we call this derivation *specialization* of fault tolerance techniques, highlighting the difference with classical solutions supporting structure-unaware computations. As a consequence of this specialization, the introduced fault tolerance techniques can be configured and optimized to meet specific needs at different implementation levels. That is, the supports we present do not target a single computing platform or a specific class of them. Indeed the specializations are the mechanism to target specific issues of the exploited environment and of the implemented applications, as proper choices of the protocols and their configurations.

Chapter 1

Introduction

High-Performance applications feature intensive computations whose execution on a single computing node can be performed in an unacceptable length of time. In some cases it is also necessary that parts of the computation, or the whole application, is performed according to some timing constraints. The natural choice to meet these demands is to execute such applications on multiple computing nodes. Thus, High-Performance applications are expressed as parallel programs to be mapped on computing platforms featuring many nodes, i.e. parallel computing architectures. Existing parallel architectures span from clusters of off-the-shelf computers or workstations [5] and shared memory architectures [33], featuring ten to hundreds of nodes and interconnected by relatively slow networks, up to highly distributed platforms, like Grids [45]. In the middle of this classification, we can characterize highly integrated architectures, featuring specialized resources and with hundreds/thousands of computing nodes [63] and massively parallel architectures [83, 34], including hundreds of thousands of nodes and interconnected by high-performance networks. As a general trend the demand for computing power of High-Performance applications increases over time. In the development of parallel architectures, the answer to such a demand is to increase the number of computing nodes available to applications.

Independently of the architecture being used one problem is the possibility of computing resources failing, even if we assume that we are executing correct (i.e. error-free) applications. Failures of computing nodes can be sporadic for small architectures, but they increase exponentially with the number of nodes, leading to very high failure rates for architectures with large numbers of computing resources [72]. This is also true if we choose to build parallel architectures with single components featuring a higher level of reliability. Thus, to exploit the whole computing power of such architectures, applications must be supported with fault tolerance mechanisms.

The fault tolerance research field studies the issue of supporting applications (both sequential and parallel) in order to survive[1]different kinds of failures. As

[1]In this thesis, as in the fault tolerance computing field, we say that a process or an application *survives* one or more failures if it continues in its execution in spite of those "catastrophic" events. A process that is not failed is said to be a *survived process*.

fault tolerance computing is a stand-alone research field its results are techniques and mechanisms that exist independently of the context in which they are applied. In some cases, these results cannot be applied to High-Performance computing because:

- they are defined for different computation models, not appropriate for expressing highly parallel computations and/or based on different assumptions or targeting different kinds of applications;

- they can induce overheads that increase with the degree of parallelism of applications according to some super-linear functions.

Several contributions have been given to study and provide solutions to the issue of supporting failures for parallel computations. In some cases High-Performance programming exploits existing solutions of the fault tolerance computing field. In some others, fault tolerance is supported in an *ad-hoc* fashion. In this thesis we face fault tolerance issues in the context of High-Performance computing by focusing on the relationships between computation models, used to support parallel languages, and the fault tolerance logics. Indeed, *the focus is on the properties that the implementer of the parallel language can use to introduce fault tolerance mechanisms.* In the remaining of this thesis we will use the term *specialization* to denote the methodology of definition of fault tolerance techniques based on the exploitation of high-level properties of the structured parallelism models. The specialization highlight the difference with more classical approach, based on unstructured models of parallel computing. As a consequence, we will also say that our support is specialized for structured parallel models.

In this chapter we briefly review recent parallel architectures and the problems of exploiting them due to failures. The review characterizes computing platforms w.r.t. their vulnerability to failures. Next we outline existing contributions towards providing solutions to this issue. Finally, we introduce our research contributions and we state its differences with existing solutions. In the description of the differences we refer to the taxonomy of parallel architectures described here.

1.1 Reliability Issues in High-Performance Architectures

Several taxonomic schemes exist in order to describe High-Performance parallel architectures. These depend on which properties we are interested in highlighting. Here we focus on the reliability ones: we want to characterize parallel architectures w.r.t. some metrics indicating the quantitative and qualitative aspects of failures. There are several factors involved in the definition of the reliability of a parallel computing platform. Examples of such factors are the reliability of the single hardware components or of the system and application software. We focus on the reliability

of hardware components implementing a parallel architecture, which we discuss in relation to a classification of parallel architectures w.r.t. the following aspects:

Size This represents the number of computing resources (node, network links, etc.) of the architecture. We are interested in characterizing parallel architectures w.r.t. the number of resources that can fail, according to some failure model.

Interconnection of Computing Resources This represents the type and the performance of the interconnection network linking computing nodes. Some parallel architectures exploit off-the-shelf networks, while others are supported by specialized and highly optimized networks. In some cases a subset of the computing resources is assigned to the task of implementing node linking facilities, as in [34].

Degree of Integration This represents the degree of distribution in space of the resources in the platform. In some cases parallel architectures are highly integrated (the same chassis, the same room/building), in others they are highly distributed, covering geographic areas that span from metropolitan networks to wide area networks.

We derive some representative classes of the architectures based on these factors. For each class we discuss their reliability in terms of the frequency of failures and how they impact on the whole platform, according to the analysis presented in [72]. This analysis is a mathematical evaluation of the overall Mean Time Between Failures (MTBF)[2] of a system w.r.t. the number of its computing resources, assuming that they can independently fail.

1.1.1 Shared Memory Architectures and Clusters of Workstations

These are architectures featuring a low number of computing nodes (actually not reaching 100 units). As smaller representatives of this class, we can characterize architectures supported at hardware-level shared memory, like in Symmetric Multi-Processors or Non-Uniform Memory Access architectures (see [33]). Interactions between processes are implemented at the lower level with concurrent accesses to shared variables. As a consequence the interconnection networks are mainly assigned to the task of the linking of processing nodes to the shared memory hierarchy. The degree of integration is often limited to a single chassis. As far as concerns reliability, in some cases the failure of a single processing node can induce the failure of the whole platform.

We characterize small clusters of workstations [5] in the same class, whose parallelism degree does not exceed hundreds of units. The interaction model is that

[2]The mean time between failure is a metric denoting the average time that pass between two successive failures.

of distributed systems, i.e. message passing over communication channels, and implemented by off-the-shelf interconnection networks (e.g. fast Ethernet) linking computing nodes. This kind of architecture features a large degree of integration (typically in the same room). The failure of nodes does not have an impact on the availability of the whole platform, if non-failed nodes can still be reached and if system functionalities are still available.

For this class of architectures the frequency of failures does not exceed one hundred of hours, according to [72]. Thus, standard solutions to fault tolerance [19] seem to meet the desired trade-off between the performance overhead they induce w.r.t. the reliability provided.

1.1.2 High-Performance Architectures or Large Systems

In this class, we characterize modern cluster architectures for High-Performance computing. The size of these architectures spans from hundreds up to hundreds of thousands computing nodes. In smaller systems we include architectures composed of workstations in a same building used to support High-Performance computing. They are interconnected through (relatively) slow networks, (e.g. local area networks). These smaller systems can be geographically composed to build a parallel computing platform with larger numbers of nodes, at the cost of slower interconnection networks [50]. In the same class we characterize large systems implemented as specialized architectures, for which the hardware configuration is built *ad-hoc*. The integration degree of this class of architectures is high w.r.t. the number of computational resources. Some of the computational nodes are exploited to implement the interconnection facility, which include several hardware technologies. In [72] this class represents the critical part of the failure rate, which is shown to grow exponentially with the number of nodes. For example, for architectures with size around one hundred a MTBF between 100 to 20 hours is estimated. For the largest architectures, the MTBF decreases to zero, i.e. there is a high probability of failure every instant of time.

For smaller cluster systems, existing fault tolerance solutions are possibly mixed up and properly configured to minimize the performance impact they induce on computations (e.g. [64]). This methodology can be difficult to apply, or can induce unacceptable overheads, depending on the actual structures and quantitative aspects of both the applications and the systems. In some cases, the fault tolerance support requires the introduction of advanced techniques to meet the desired degrees of performance (e.g. [40, 46]).

1.1.3 Grids

In this last class we consider highly distributed and dynamic platforms, composed of heterogeneous computing resources. There is not an upper limit for the size of these systems, which can be supported by Internet-wide interconnection facilities

and can be composed of both small scale networks and/or large scale ones. The exploitation of these kinds of platforms has brought about several research issues and relative solutions, but there is no an accepted methodology for their exploitation for High-Performance computing. Also, the MTBF is theoretically evaluated at zero.

Fault tolerance solutions for these kinds of platforms are necessarily *ad-hoc* w.r.t. the actual and dynamic configuration of the exploited platform. As a general trend, standard techniques are reported to provide poor scalability to applications [41], while specialized ones seem to be much more promising (e.g. [91]).

1.2 Existing Approaches and Contributions of this Thesis

There exist several approaches and techniques to fault tolerance computing. In the context of parallel and distributed platforms the most notable ones are replication [51] and checkpointing with rollback-recovery [39].

Replication has been traditionally introduced to support sequential computations: a single sequential module is replicated and executed on multiple nodes, distributed over the network, to enhance its fault tolerance. The coordination of the replicas is defined in such a way that the sequential computation is emulated. In this model no parallelism is exploited. Research works in the context of high-performance computing highlighted how replication can be extended to cope with parallel computations [14]. Other approaches are specialized for task-parallel computations and they introduce task-replication (re-scheduling or duplication) techniques (e.g. [18, 2]).

In this thesis we focus on checkpointing and rollback recovery-based techniques. These are traditionally associated to the distributed computing research field. The computation model on which they are based is composed of a set of stateful independent processes, each performing its own program. Processes can only interact between themselves according to the message-passing paradigm. The communication model influences the general system model and it can be based on reliable or unreliable (message losing or duplication) protocols. Channels can be FIFO or they can re-order messages. Essentially computations feature nondeterminism: two different runs of the same distributed program on the same input data can generate two different sequences of states. In some cases the nondeterminism is relegated only to the communication semantics, while in some other cases also local sequential computations can be nondeterministic.

Checkpointing and rollback recovery techniques essentially consists in periodically saving the global state of the computation (checkpointing) on some memorization support. In the case of failure one of such states is restored (rollback) and its correctness is possibly re-built (recovery) if this has not been done during checkpointing or rollback. A checkpointing protocol can require the coordination of

all application processes (coordinated checkpointing), either by means of a global synchronization [82] or asynchronously by saving application messages during the protocol execution [27]. More advanced checkpointing protocols do not require process coordination but they record global "correct" states by locally analyzing the process inter-dependencies, due to communications. The information on process inter-dependencies is passed between processes as attached to application messages. It is important to notice that such dependencies are only known at run-time, according to the computation model, and they can change between two different runs of a same program.

Another way of implementing checkpointing without process coordination is to collect also application messages. These can be collected synchronously with the computation (pessimistic logging) or asynchronously with it (optimistic logging). In this former case it is not guaranteed that a correct state different from the initial one can be restored during rollback (incurring in the *domino effect*, see Chapter 2). A further logging technique, causal logging, introduces a trade-off between optimistic and pessimistic logging. It is based on logging messages (or information about it) on the volatile memories of multiple processes (e.g. the senders), which avoids both process blocking during logging operations and the domino effect.

These techniques have been implemented and experimented in several works (e.g. [22]) and they have been also exploited to support high-performance computing applications (e.g. MPICH-V [19, 92]). Experimental results show that they perform well for clusters of workstations [22]. Checkpointing protocols have been also introduced to support large-scale systems [32], Grid Computing applications [12] and in the case in which Grid resources can be mobile [84]. In some cases, for larger computing platforms (large-scale architectures, grids, mobile grids) the experiments are not made on "actual" platforms, but they are emulated on clusters with a relatively small number of nodes [22, 12]. In other cases actual large systems are used for experiments, but showing an high performance impact [32].

It can be observed that these kinds of techniques, applied to the above described computation model, *are difficult to analyze in principle* (that is, statically). A clear example is the one of MPICH-V [22]: several checkpointing and rollback recovery protocols are provided to the programmer and previous experimental results can guide him/her in the selection of the best mechanisms, based on the current application features (if available) and configuration of the exploited platform.

According to a critical view of this research field, it can be noticed that it is lacking of a general methodology. An useful methodology should state, in a formal and analytical way, given a concrete application (or some abstract description of it) and a configuration for the targeted computing platform, (i) which is/are the best protocol(s) to be exploited and (ii) how this/these will impact on the (failure-free) performance of the computation and on the costs of rollback recovery. *In this thesis we make a step towards the definition of this methodology and we give an answer to point (ii) of above by introducing cost models for our checkpointing and rollback recovery techniques.* This is possible because we base our approach on

the state-of-the-art on programming models for parallel computing: we show how the exploitation of the models of the structured parallelism paradigm allows us to introduce several checkpointing and rollback recovery protocols specialized w.r.t. the implemented applications and featuring cost models describing their performance impact. In this study we have characterized the following points as being the keys to understanding the advantages of exploiting structured parallelism models to support fault tolerance:

- The computations that can be expressed with the constructs of structured parallelism are essentially deterministic, in the sense that they generally express the evaluation of some complex functions. In our programming model we compose in a stream-based graph-structure parallel and sequential computations (or nodes). Nondeterminism happens (or is expressed) in the inter-connection between nodes, while parallel and sequential computations inside the nodes are deterministic.

- As stated above, *there is a static knowledge of the structure of the interactions between the parallel entities composing the computations.* This allows us to move at compile-time the majority of the mechanisms to support and analyze fault tolerance.

The first point is a characterizing factor for the computations which High-Performance applications express. It also clearly characterizes the differences between our programming model and the ones studied in the context of fault tolerance computing and in the context of the unstructured models of parallel programming. As stated above, in such models non-determinism is a key factor in the design of fault tolerance mechanisms. Unlike this approach we highly exploit determinism of computations inside program nodes (see above) to design fault tolerance techniques.

The second point allows us to introduce the possibility of *configuring the runtime support of computations depending on application- and platform-specific factors to target optimizations.* In particular, we can extend the models describing the performance behavior of computations to include the overhead induced by fault tolerance mechanisms. This is the key to control and configure our checkpointing and rollback recovery protocol.

Operationally, the deterministic semantics of computations, both with the knowledge of the parallel structure of the interactions, allows us to define *several checkpointing and rollback recovery protocols which target the optimization of different parameters, to meet different kinds of applications and several platform configurations.* For instance (see Section 5.3) we can minimize the failure-free performance induced by checkpointing activities by making processes take checkpoint independently but according to a global coordination (without exploiting any message logging technique). This can be done by exploiting both the properties of above and it comes at the cost of a more heavyweight rollback recovery protocol. In details, we can *control* the performance impact of checkpointing by choosing *different checkpointing*

frequencies to meet an optimization of the rollback depth[3]. In this way, we also obtain a control over the recovery time. This is especially useful for cluster-based platforms for high-performance computing, in which the failure rate is (typically) low.

In another setting, we can require *a minimization of the number of processes involved in a rollback (rollback width) or the number of discarded checkpoints*. For instance, the protocol presented in 5.4 minimizes the rollback width, according to the structural shape of the program. This solution is useful for large-scale systems to limit the synchronizations needed during rollback. The rollback depth can be optimized by introducing global synchronizations in the protocol presented in 5.3. This can be viable for small-scale platforms in the case in which failures are frequent.

All these optimizations and the selection of proper solutions can be done statically, by exploiting the *cost models* describing the performance impact of checkpointing on computations and the performance of rollback recovery protocols.

We shortly discuss two examples of structured parallel models showing how they can be efficiently supported if we consider the information on their structure.

1.2.1 Farm Computations

A farm computation is based on a set of replicated stateless modules which perform all the same function(s) (functional replication). The computation consists in applying that function on a *typed* input stream of tasks to obtain a *typed* output stream of results (one for each task). Task (i.e. function evaluations on each input data) are independent and are performed in parallel on the set of replicated modules. Moreover, they are scheduled to the modules according to several well-defined policies (e.g. on-demand or round-robin).

Structure-Aware Case

For the sake of the discussion, suppose we are in the case in which the application programmer is provided with an high-level parallel language in which a farm computation can be expressed with a specific construct. Our viewpoint is the one of the implementer of the support of these constructs, which can be done, for instance, by exploiting a compiling-based approach. The discussion on fault tolerance techniques which follows is based on this setting. This is done only in this section of the thesis: in the next chapters will not exploit this specific compiler-based case, but we study the models of structured parallel programming independently of their actual realization and provisioning to application programmers. In general, to exploit our results,

[3]In this thesis we use the term "rollback depth" to denote the number of discarded checkpoints during a rollback. We use it, instead of the more common one "rollback extension", to distinguish with the "rollback width" one, which concerns the number of processes involved in a rollback protocol.

it is sufficient to have the knowledge of the properties of the models, independently of the way in which we obtain them.

Farm modules are stateless and the only data of the computation are input tasks and output results. Thus, it seems a reasonable solution to exploit a *message logging technique*: in fact the "history" of the computation is made up of the streams of input and output data of the farm. We can evaluate the time needed to log a message onto a stable storage support (which survives module failures) because we know the types of the elements passed on the input and output streams. This has to be instantiated to the specific stable storage implementation (e.g. local disks of computing nodes, or remote specialized supports).

We can also consider the performance of the context surrounding the farm as the input stream inter-arrival time. Thus, we can exploit the knowledge of the parallel farm structure to model the impact of message-logging on the computation performance (actually, existing performance models do not consider fault tolerance techniques). We can apply this model to the specific application and platform and we can select the best technique between optimistic and pessimistic logging. That is, we can choose the ones which best instantiate the trade-off between failure-free performance and costs of rollback recovery. Then, if we implement input and output streams according to a specific asynchrony property (see Chapter 6), we can also place *an upper limit on the number of lost messages in the case of failure*. This can be done by asynchronously coordinating the behavior of implementation modules accessing a stream. This control over the upper bound on lost messages can be used to control the trade-off between performance and overhead of optimistic and pessimistic message logging.

Structure-Unaware Case

Now suppose that we are at the level of a generic unstructured parallel programming model (e.g. MPI). We want to implement a fault tolerance support for a farm computation implemented on top of this programming model. As a consequence of the constructs of the model, we are not provided with the structural information as in the previous case. In fact, we just know that we have to execute a set of processes which communicate in an arbitrary way. We can select one of the checkpointing and rollback recovery techniques, but we do not know which one is the best. In fact, the best methodology is to select it according to the features of the execution platform we target. This can be done only by analyzing previous experimental results, which are necessarily related to different applications. Thus, we are not guaranteed of the results until we perform experiments and each variation of the experimental conditions is a source of uncertainty. Moreover, this structure-unaware support is not specialized w.r.t. the specific application (that is, w.r.t. its parallel structure).

The situation is even worst because we do not know which is the useful part of the state of the processes implementing the computation. We are also forced to insert checkpoints in casual point in the computation. Unlike the previous case, in which

we could save *only* messages and at *proper* points of the module computation, we have to save also the *whole process states* (data sections, but also the code one and operating system information). In the case we exploit a message logging technique we have also to save message payloads and (possibly) dependency information.

In Chapter 4, after describing our specialized techniques for farm computations, we compare them with the case in which we exploit standard structure-unaware techniques. The comparison is driven by the main issues described in the next section.

1.2.2 Data Parallel Computations

In a data parallel computation a same function is applied to each element of a complex (large) data structure. This state is partitioned over a set of identical modules, which apply in parallel the function to their local partitions. We consider the case in which the function application is iterated for a given (possibly unknown) number of steps. A module can communicate with other modules to obtain their previous results for its local computation. For instance, we can functionally describe the k-th application of a given function $\mathbf{F}$ on the i-th position of an array A in the following way:

$$A^k[i] = F(A^{k-1}[i], A^{k-1}[i+1], A^{k-1}[i+2])$$

In this example, to compute the k-th value of $A[i]$ we need the values of the previous iteration (*k-1*) of the same element and the two next ones. If these "neighbor" values are assigned to another module, a communication is required between the module evaluating F on $A[i]$ and its neighbor.

The structure of the dependencies between function evaluations characterize the computation and they can vary at different iterations of the program or they can be equal for all steps. Indeed, in this thesis we assume that they are completely specified at compile-time for the whole computation. Moreover, the inter-dependencies between parallel modules can be modeled (and implemented) as streams of typed elements. Examples of applications falling in this class are the numerical approximation of the solutions to a system of differential equations (e.g. according to the Jacobi method) or the computation of the shortest path tree of a graph. Also computation and communication patterns, such as reduce operations, are straightforwardly modeled according to this model.

Structure-Aware Case

For data parallel programs we can introduce several checkpointing and rollback recovery protocols. In this thesis we analyze three solutions which are representative of the main classes in which checkpointing and rollback recovery protocols have been characterized [39].

For the sake of the discussion, similarly to the previous case of the farm computations, suppose we are implementing a support for an high-level parallel construct

to express data parallel computations, according to a compiling-based strategy. As a result of the properties of the data parallel programming model we have a graph of dependencies between abstract parallel activities (the modules) which, if properly mapped during compilation, can be used to statically generate a dependency graph between implementation modules. Notice that this graph must be generated at run-time in communication-induced-checkpointing protocols, because of the nature of the computation model. As a consequence of this static knowledge, we can select *different sets of consistent states* (possibly according to different consistency properties) and *we can automatically insert checkpointing procedures in the implementation*. This can be done *to meet different optimizations*. Moreover, we can also limit the asynchrony between implementing modules (which is the difference between the iterations at which they are executing) by supporting interactions with an optimized channel implementation which limits the number of sent but yet unreceived messages.

Structure-Unaware Case

Suppose now that we are provided of a set of processes which implement a specific data parallel application, but we do not have any structural information. For instance, we are supporting fault tolerance for a data parallel computation implemented on top of MPI. As the farm case, we have to checkpoint the whole state of processes and not just the partition of the state assigned to each process. Moreover, we cannot rely on specific patterns of communication between processes and we have to choose between standard checkpointing and rollback recovery protocols, which are not specialized for the data parallel case.

For the case of coordinated synchronous checkpointing [82], we can analyze the cost of a global synchronization but it is difficult to understand the time needed to perform a global checkpointing of the computation. In the case of asynchronous coordinated checkpointing (e.g. [27]) we do not know in advance which will be the amount of messages saved during the checkpointing operations. This variable depends on the relative speed of processes, which is known only at run-time, and on structural information, which is unknown in this programming model.

In the case of communication-induced-checkpointing we have to build the process inter- dependencies during the execution, and to attach this information on application messages. The attached information can be, depending on the actual protocol, very large and induce a degradation of the communication performance. This highlights the difference with the previous case, in which we can *statically* introduce checkpointing operations by analyzing the process inter-dependencies. As a consequence, it is not possible to understand *a priori* the amount of checkpoints taken by each process and, as a consequence, the impact on the performance of the computation.

Suppose now we select a message logging technique. Optimistic and pessimistic message logging cannot enjoy of the types of the data exchanged between processes,

as in the structure-aware case. This cannot be done because MPI channels (as in the general case of distributed systems) are not typed. As a consequence we cannot estimate the overhead due to checkpointing and we have to perform experiments to choose between the pessimistic and optimistic cases. We can just select the best one depending on the targeted execution platform, avoiding optimizations based on application knowledge. Causal message logging techniques can help in improving our support from a performance viewpoint, but it still does not provide any useful a priori information about the final overheads incurred by the computation.

In Chapter 5 we present our optimized mechanisms, both with their cost models to describe their performance impact. We compare them with existing protocols according to the issues described in the next section.

1.2.3 Issues for Fault Tolerance in Structured Parallel Computing Models Considered in this Thesis

As mentioned in the previous subsections, in the thesis we compare classical structure-unaware fault tolerance techniques with the ones we define according to the following list of issues:

Analysis of Performance We will show that, according to the hypotheses of structured parallel programming models, we can derive fault tolerance techniques which allows us to introduce a static definition of the performance overhead of parallel computations. We will also show that this analysis is not possible for unstructured models of parallel programming. For this purpose we provide cost models of the solutions we present and we compare them with the analysis of standard techniques.

Consistency Definitions We will show that under the hypotheses of structured parallel programming models we can derive simple consistency properties for global sets of checkpoints. This is especially important for data parallel programs, for which the tasks that compose the computation are inter-dependent. We will also show that more complex definitions are needed for unstructured models of parallel programming, forcing the definition of more complex and costly fault tolerance techniques.

Determinism of Computations We will see that under the hypotheses of structured parallel programming models we can relegate nondeterminism at the implementation level, by providing program structures to control them (e.g. on-demand scheduling of tasks of the farm computations). This is not true for structure-unaware programming models and we will show the consequences on the fault tolerance protocols. In particular we will show which is the information required to implement protocols of the class of communication-induced-checkpointing (CIC in short), to control the nondeterminism of computations.

We consider CIC protocols because they provide the best trade-off between coordinated and uncoordinated checkpointing protocols.

Modularity and Composability We will see that the modular definition of structured parallel programs allows us to define fault tolerance techniques and mechanisms which are themselves modular. We will also show how to compose them with an optimized message-logging mechanisms. Also this composition mechanism enjoys the high-level definition of the models of structured parallel programming, in particular by exploiting the static knowledge of the data types passed between different parallel modules.

Experimental Results A clear comparison between our solutions and the classical ones should consider experimental cases according to different computing platforms and applications. In this thesis we theoretically compare our techniques with existing ones whenever it is possible to do it statically. We exploit the comparison to prove the actual beneficial of our approach for different applications and computing platforms. We demand to future work experimental experiences.

1.3 List of Contributions of this Thesis

A detailed list of the contributions of this thesis follow:

- We precisely characterize a programming model that is well-suited to study the fault tolerance issue for High-Performance applications.
- We introduce a formal tool that allows us to highlight the properties of the introduced model that can be used to define fault tolerance mechanisms.
- We show how fault tolerance can be derived from such abstract properties for two main constructs of the structured parallelism describing task and data parallel computations.
- We extend the cost models of the performance of task and data parallel constructs to model the overhead given by fault tolerance mechanisms. We also show how these models are instantiated to the actual implementations, which we present at various levels of virtualizations.
- We show how we can derive a study of the performance of task parallel computations in the presence of failures. The performance analysis of data parallel programs in presence of failures has not been faced in this thesis: but several recent research works introduce techniques to study this issue (see, for instance, [87]). We demand to future work this research issue.

- As a feedback of our study, we characterize the main issues of unstructured models of parallel computing, and of the models studied in the context of fault tolerance computing, which introduce complexities in the definition of fault tolerance supports.

1.4 Outline of the Thesis

The outline of the thesis follows:

- Chapter 2 reviews the fault tolerance mechanisms exploited in the context of High-Performance parallel and distributed computing. The mechanisms are presented w.r.t. the class of programming models to which they have been applied. This allows us to provide a description of the dependencies between programming model features and fault tolerance logics.

- Chapter 3 introduces the programming model on which we base our study of fault tolerance, giving details of its characterizing properties. To highlight such properties, we also describe two main constructs to express task and data parallelism. For fault tolerance modeling, we introduce a formal tool inspired from *I-Structures*[7], which is used to highlight the properties which are interesting in this context. We apply this tool to the structured parallel constructs we describe in the model in order to discuss their properties.

- Chapter 4 describes the implementation of the task parallel construct, introducing fault tolerance mechanisms. The logic of the mechanisms is derived from the abstract model, and refined and extended with information related to its implementation. We also show an implementation of the fault tolerance logic, in a specific case in which task parallelism is mapped onto a distributed set of processes which communicate through message passing.

- Chapter 5 shows fault tolerance logics and mechanisms, according to the same methodology of the previous point, for data parallel programs. Also in this case, we show an implementation of fault tolerance mechanisms in the case in which the data parallel construct is implemented as a distributed set of processes which communicate through message passing. We describe three configurable solutions that can be derived from the properties of the high-level model, each targeting different optimizations.

- Chapter 6 describes the implementation in a distributed environment of the stream construct, extended with message logging mechanisms, which we exploit in the implementation of fault tolerance for task and data parallel constructs. It also discusses, in general terms, the overhead impact of message logging on communication latency and the properties of the implemented protocol.

- Chapter 7 introduces a statistical model, based on Markov Chains, which describes the performance impact of failures for the task parallel construct. The model provides an evaluation that can be proved to be an upper bound the actual performance behavior of task parallel computations.
- Chapter 8 presents the conclusions of this thesis and describes the future work.

Chapter 2

Parallel Programming and Fault Tolerance

As we highlighted in Chapter 1 High-Performance computing requires a large degree of computing power, which demand is even increasing as more complex applications are developed. The natural choice to answer this demand is to build parallel and distributed platforms with increasing numbers of computational nodes. This trend has a clear drawback as highlighted by several research works [72]: the failure rate increases with the number of nodes or processing elements according to an exponential distribution. Several research and industrial efforts have been presented in the literature to attack and provide solutions to this issue by supporting high-performance applications with fault tolerance. The main issue in these studies is to limit the performance overhead due to the fault tolerance supports. Typically, this impact is shown by means of experimental results of benchmark applications. In other cases analytical tools are provided in order to study the performance impact before the applications are actually executed. In some cases High-Performance computing inherits fault tolerance techniques from the research context of fault tolerance computing. In some other cases new specialized techniques are introduced. From the fault tolerance computing literature, parallel computing has exploited: (a) checkpointing and rollback recovery protocols, (b) techniques based on replication of modules, and other techniques (e.g. atomic actions). It is worth of notice that some central issues in the context of fault tolerance computing are only of marginal interest in High-Performance computing. One of the most clear case is failure detection in asynchronous systems, which is one of the most studied of the fault tolerance literature. However, in the context of High-Performance computing this issue is typically relegated to the lower layers (e.g. to the communication support) and a very simple failure model is assumed (e.g. the fail-stop failure model [76]).

In this chapter we discuss the fault tolerance techniques exploited in the context of High-Performance computing and we describe the related works presented in the literature. We are mainly interested in the relationship between fault tolerance techniques and parallel programming languages and abstractions. We also review a

system-level fault tolerance technique that exploits a computational model from the parallel computing research field. The discussion is characterized by the programming model on which the studies are based. We characterize *unstructured* models of parallel programming and *high-level* ones. Typically:

- Unstructured models exploit existing fault tolerance techniques.
- High-level models introduce specialized supports by exploiting some knowledge/assumption given by the model itself.

There are clear and interesting cases in which this classification fails. We will highlight also such cases. When needed in the description of the related works we shortly introduce existing fault tolerance techniques and mechanisms.

In the next chapters we will introduce in a formal way some concepts that we just informally describe here. Moreover, in the last chapters of this thesis we have inserted *ad-hoc* sections about related works: these allows us to discuss in a precise way the differences between our contributions and the ones presented in the literature.

2.1 Fault Tolerance for Unstructured Models of Parallel Programming

Unstructured models of parallel computing typically feature a set of processes which implement a distributed program and that interact by means of message passing. The main characteristic of these models is that there is not a static knowledge of the structure of the interactions between processes. Moreover, a process can send/receive a message to/from another one in every phase of the computation. Computations can be nondeterministic in the sense that communications can be performed according to information retrieved externally from the system and/or to partial results of the computation itself. As a consequence, a same program executed multiple times can generate different computations. Message passing is implemented with communication channels linking processes and can be point-to-point or can involve multiple parties (e.g. multicast communication). The main techniques introduced for this kind of computation model are checkpointing and rollback recovery protocols, and replication ones. Below we give an overview of each technique and we describe some notable research studies exploiting them.

2.1.1 Checkpointing and Rollback Recovery Protocols

Checkpointing and rollback recovery protocols are studied in the context of an abstract computation model, which is similar to the one we described above. Moreover they target the fail-stop failure model. A computation consists in a set of processes executing a distributed program, which interact through message passing. Each

process is supported by a local volatile[1]memory. Communication channels can be reliable or unreliable (by duplicating or losing messages) and can be FIFO or can re-order messages (in a nondeterministic fashion). Processes can independently access an abstraction, called *stable storage*, onto which they save their local state. The stable storage, differently from the volatile one, is assumed to survive the failures modeled in the failure model.

One of the main references in the checkpointing and rollback recovery literature is [39], which presents a survey and introduces a taxonomy of the related literature. We exploit the same terminology introduced in this survey. We shortly give a reference list of terms exploited in this thesis and in the literature:

- *Consistent Global State*: a global state is a snapshot of the computation. It is composed of the local state of all processes taken at a given instant of the computation. A global state is said to be consistent if it can be passed by any correct computation. That is, as computations are nondeterministic multiple values for global states can be generated and can be possibly different between different executions. In the fault tolerance literature several research studies have introduced properties for global states to ensure their consistency. These properties are often referred to as *consistency properties*. In Chapter 3 we introduce two consistency properties at different virtualization levels, which we prove to be related between themselves. In [68] it is introduced an elegant theory that is called *zig-zag* theory by the shape of the graphical representation of the consistency property. The zig-zag theory precisely characterizes necessary and sufficient conditions for global states to be consistent.

- *Checkpointing*: this denotes an algorithm or a protocol performed by a distributed set of processes to characterize a snapshot of the computation. The checkpointing for a global computation is composed of a checkpoint from each process. The expression "a process takes a checkpoint" means that: (1) the process characterizes its local state that is significant to build a snapshot and (2) it saves it in some memorization support (e.g. onto the stable storage). In the literature several tools have been presented supporting system-level checkpointing for a single process. Checkpointing of a process state depends on the definition of the process abstraction. For instance, in off-the-shelf operating systems checkpointing is complex because information related to processes is not included in the same unit/space but it is sub-divided in multiple parts (often corresponding to user and system spaces) [74]. This issue can be solved by re-defining the process abstraction in a proper way [37]. A modular approach to the previous ones consists in characterizing checkpointing at multiple levels of abstractions. In this way complex tasks in checkpointing can be made simple by exploiting the information on the application structure and/or logics.

[1]In the context of fault tolerance computing the adjective *volatile* associated to a memorization support denotes that, in the case of failure, the content of the support is lost.

In this thesis we exploit this approach, and we assume to be supported of proper process abstractions (e.g. [37]).

- *Rollback*: this denotes an algorithm or a protocol which is performed by one or more processes in the case of failure(s). It consists in retrieving a passed state of the computation (the rollback target) previously saved on some memorization support. A rollback protocol can involve all processes or a subset of them. Rollback targets can be also inconsistent or can generate inconsistencies in the global state of the computation. For instance, this can happen if the checkpointing algorithm builds inconsistent global states. In some cases a rollback protocol iterates the retrieving of older states until consistency for the global state is re-built. The *domino effect* is the case in which the iteration lasts until the beginning of the computation.

- *Recovery*: this denotes an algorithm or a protocol performed by a set of processes after a rollback. It consists in re-executing the lost computation after the failure and the rollback. In the case in which the rollback protocol builds an inconsistent state (see above), it is task of the recovery one to re-build its consistency.

- *Message logging*: this denotes the actions performed by a process to characterize the information related to a message sending or receipt event. The information can include its content (the payload), its source or destination process and the ordering of its sending or reception events w.r.t. other events (e.g. other communications). In the terminology of the rollback recovery literature this information is called message *determinant*. The purpose of characterizing message determinants is to save them on stable storage. They are used during rollback recovery to re-build consistency for global states.

We give a description of the taxonomy exploited in the context of fault tolerance computing, presented in [39]. Checkpointing protocols can be characterized in three main classes each one supported itself by a class of rollback recovery protocols:

- **Uncoordinated checkpointing**. The processes of the system perform checkpointing operations without any interactions among themselves. In the case of rollback recovery, the uncoordination of local checkpointing procedures can lead to *inconsistencies* between process states in recovery lines (i.e. set including a checkpoint from each process). If some property about the determinism of the computation can be assumed (piecewise determinism, PWD in short), uncoordinated checkpointing is extended with message logging (see below). In this setting the recovery protocol re-builds consistency of global states after rollback.

- **Coordinated checkpointing**. The checkpointing procedures performed on each process are globally coordinated. For instance, a global synchronization

of processes can be exploited to implement a coordinated checkpointing protocol. The synchronization allows processes to collect local checkpoints that are consistent between themselves. Existing protocols either block the computation during the checkpointing operation or operate in concurrency with it.

- **Communication-Induced-Checkpointing**. Protocols in this class represent a trade-off between: (a) the cost of coordinating processes in coordinated checkpointing and (b) the building of inconsistent states of uncoordinated checkpointing. Fundamentally, processes can take consistent checkpoints between themselves by analyzing process inter-dependencies given by message exchanging. The knowledge of process inter-dependencies is exchanged attached to application messages themselves (*piggybacked* in the rollback recovery terminology). Typically processes exchange graphs of dependencies, which model the dynamic relations between their execution and message exchanging events. The analysis of a dependency graph received as piggybacked to an application message can induce a forced checkpoint on a process. For this motivation Communication-Induced-Checkpointing protocols require that processes can perform checkpointing at any instants of the computation. This requirement is not always simple or viable to satisfy.

As stated above, uncoordinated checkpointing protocols can be supported with message logging. In this way the recovery protocol can re-build consistency for global states. Several message logging techniques have been introduced in the literature. We shortly review the taxonomy given in [39]:

- Messages are logged synchronously w.r.t. their receiving operation. Before a process receives a message its determinant is logged to stable storage. It is straightforward to notice that, for this kind of protocols, we pay a stable storage access for each communication operation. This case is called **pessimistic message logging** because it assumes that failures are frequent. As a consequence, the high overhead incurred because of message logging is justified by the high failure frequency.

- Messages are logged asynchronously w.r.t. the receiving operations. A received message can be used also if its determinant is not saved on stable storage. In the case of failure of the sender or the receiver of an unlogged and received message we can obtain inconsistent states. As a consequence, the rollback recovery protocol must deal with this situation. This kind of protocols is called **optimistic message logging** because it is based on the "optimistic" assumption that failures are infrequent. This motivates the possible creation of inconsistent states and the higher costs of recovery protocols.

- Message contents are logged asynchronously w.r.t. their receiving operation. Their determinants include the functional dependency they induced, between

the sending and the receiving processes. The determinants are logged synchronously to stable storage, w.r.t. the receive operations. In the case of failure, the logged functional dependencies are used in the recovery procedure to reproduce lost messages, which content was not yet saved on stable storage. This case is called **causal message logging**, as it exploits the causal (or functional) dependencies between processes.

For message logging techniques to be used in a correct way a determinism property for computations must be assumed: message exchanging is the only source of nondeterminism. In other terms a part of sequential computation performed by a process is completely characterized from the messages received before this part.

From this review of checkpointing and rollback recovery techniques it should be clear that one of the main disadvantages of supporting an unstructured model of parallel programming is that the fault tolerance support must deal with unknown patterns of interactions between processes.

We now review some notable works presented in literature on parallel computing exploiting checkpointing and rollback recovery protocols. The presented works represent a cross section of the contributions given in this context, which range from the most lightweight support (failure detection and process restart) to supports that are transparent to the programmer, or based on some knowledge about the parallel structure of programs. These works are based on the Message Passing Interface (MPI)[44], which is implemented to support fault-tolerant computations. In MPI a computation consists in a set of processes interacting through message passing. There is no shared memory abstraction to support processes and programs are typically expressed in the Single Program Multiple Data (SPMD) style.

FT-MPI [42] provides a minimal support to fault tolerance for parallel applications (failure detection and process restart). It extends the MPI semantics with concepts related to failures. At the linguistic level the syntactic extensions allow the programmer to specify:

- The behavior of the run-time support in the case a failure is detected. In general, when a failure affect the computation the communication support stops to work. The programmer can specify if the failed processes should be restarted and which is the behavior of survived ones. It can also specify if the communication support should be restarted and which is its configuration after restart.

- Procedures performed in the case of failure detection. These allow the programmer to implement the fault tolerance logics at the application-level. For this reason the support provided by FT-MPI is said to be semi-transparent.

At the level of implementation FT-MPI provides a failure detection sub-system, together with process restart and run-time recovering. The run-time recovering algorithm involves all processes of the application and it is leader-based [41]. Its

behavior can be configured by the programmer by exploiting the support described above. At the end of the recovery algorithm the user-implemented procedures (if specified) are called back from the support. Because of the small support it provides FT-MPI has been exploited to study several fault tolerance techniques at the application level. For instance, [28] describes a fault tolerance support for a specific class of parallel applications. It shows several strategies of implementation of stable storage supports and their properties (e.g. failure resiliency, and costs).

The MPICH-V project provides fully transparent fault tolerance to MPI applications. Thus, it does not require an extension of the semantics of MPI. The project provides several checkpointing and rollback recovery protocols which are inherited from the fault tolerance literature. They range from a coordinated checkpointing protocol to an uncoordinated checkpointing one associated with message logging protocols. An assumption to enable fault tolerance transparency is made: the only way in which processes can interact is through message passing. This is not required to the FT-MPI programs because the implementation of the application-level fault tolerance logic is relegated to the user. The checkpointing of process states is automatically performed by the run-time by exploiting an existing checkpointing library [59]. The protocols available to the programmers are:

- Uncoordinated checkpointing associated to a pessimistic message logging protocol [19]. The implementation exploits an abstraction called*channel memories* associated to each MPI process. A channel memory is responsible of storing all the messages received by a process onto a stable storage. Performance results show that this approach is effective only in the case of high node volatility[20].

- An optimization of the previous protocol exploiting sender-based message logging [20]. In sender-based message logging the determinant of messages is saved to the volatile memory of the sender process. Experimental results show that this approach performs much more better for general configurations of the execution environment. However, for some applications high communication latencies were monitored due to message logging.

- Uncoordinated checkpointing associated with causal message logging protocol to avoid the high latencies incurred by the previous protocol. As shown by experimental results the drawback of this approach resides in an higher re-execution time w.r.t. the previous cases. [24].

- Also a coordinated checkpointing protocol is supported to be compared with the previous ones from a quantitative viewpoint.

Some experimental results, showing the performance of the support without performing any checkpointing protocol, and the performance of various causal logging protocols are presented in [23]. In general the results show that some of the protocols provide better performances, depending on the failure rate, on the kind of

application and on the configuration of the execution environment. No support is provided to dynamically change the protocol used to obtain fault tolerance. A problem associated with the approach of MPICH-V is that recovery algorithms are not configurable. It is not possible to introduce optimizations based on the knowledge of the parallel structure of programs: the only choice that the programmer can take is related to the rollback/recovery algorithm to be used.

The MPI/FT [11, 10] approach represents the link between unstructured model of parallel computing and high-level ones. It is based on the intuition that fault tolerance strategies can be easily introduced when there is some knowledge of the structure of the computation. The fault tolerance support is restricted to application programmed as master-slave or data parallel expressed in the Bulk-Synchronous Model [80] (BSP). The implementation is based on a coordinator that monitors the application, logs the exchanged messages, restarts failed process and implements all the control needed for redundancy implementation (e.g. in master-slave computations it controls the task redundancy). The coordinator is the more critical point w.r.t. reliability: for this reason it can be replicated. The fault tolerance support for data parallel programs is based on the BSP model of implementation: the processes take coordinated and synchronized checkpoints at the beginning of each loop iteration. Experiments show that the drawback of this approach resides in the poor scalability that the coordinator can support (order of 10 computational nodes).

Other fault-tolerant implementations of MPI provides supports similar to the ones of the above projects. LAM/MPI [92] provides a coordinated-checkpointing support to applications. The starfish project [1] provides both coordinated and uncoordinated checkpointing. FTC-Charm++[93] provides a support based on coordinated checkpointing and rollback recovery performed by all processes implementing the applications. A notable work on MPI has been developed in the context of the LA-MPI project [48]: failures addressed are related to the network, instead of the computational nodes. LA-MPI replaces the TCP/IP protocol with a specialized one, which includes a fault tolerance support for communications and that targets high-performance networks, which are typically exploited as interconnection facility of massively parallel systems.

2.1.2 Replication Techniques

In replication techniques processes are typically characterized in *servers* and *clients*. Server processes provide a set of services that can be invoked as remote procedures. Fault tolerance is achieved by replicating the processes implementing servers. The support to the management of replicated processes can exploit group membership services [29] and/or group communication primitives [36] (see below). In [51] two main replication schemes are characterized and related to the mechanisms needed to implement them:

Primary/Backup one of the replicas of a process (the *primary*) receives all the

requests from the clients. For each request it forwards the input data to all other replicas (the *backups*). Next, all replicas (both primary and backups) perform the same request in parallel. When the primary terminates to perform the request it waits for all other replicas to acknowledge their termination and it returns the results to the client [25]. The failure of a backup does not require a special support: the primary replica stops to forward requests to it. Some additional support is needed to survive the primary failure: one of the backups is elected as new primary replica. In the case of failure during the service of a request the client must re-send its request. The support needed to implement the leader elections is traditionally encapsulated in a group membership protocol (see below). Fundamentally, it is required to all replicas to have the same view of the failed and survived processes. The group membership protocol is included in the communication support of replicas.

Active all replicas are identical and perform the same requests at the same time. The clients issue their requests to the whole set of replicas and receive an identical answer from all of them. No specific support is required in the case of failure of replicas: the clients avoid to send their request to failed replicas. In the active replication scheme the replicas of a same process are required to behave deterministically w.r.t. each others. This can be obtained by making them perform the very same sequence of requests. Thus requests incoming from different clients must be totally ordered on the replicas. This behavior is traditionally encapsulated in a communication primitive, which is called *total ordering broadcast* (see below). To implement determinism it is also required to replica to perform deterministic computations w.r.t. each others. For this reason replicas are described as state machines [77].

In the literature other techniques have been introduced as hybrids between the primary/backup and active ones. The semi-active replication technique [71] is based on the same interaction scheme of the active replication. The nondeterministic part of the computation is performed by a single replica. The information related to this part of the computation is periodically propagated to other replicas. These ones perform only the deterministic part of the computation. Semi-active replication is based on more relaxed assumptions related to process determinism w.r.t. the case of active replication. This is the main motivation behind its introduction. In the semi-passive replication technique [35] all replicas receive the requests from clients but only one of them actually performs the computation in absence of failures. Also in this case failure resiliency is supported with checkpointing of the actually active replica.

2.1.3 Group Membership Services and Group Communication Primitives

We briefly review two main research fields in the context of fault tolerance computing that have been exploited to support replication techniques.

A group membership service is a mechanism that allows a set of processes belonging to a same logical group (e.g. a set of replicas) to interact between themselves in a correct way. Each interaction can involve multiple parties and it is required that all processes have the same knowledge of the participants of the group (in the fault tolerance terminology this is called the same *view*). The uniform vision of the participants is made difficult in the case of failures if failure detection is subject to some kind of asynchrony (see [26]). Sources of asynchrony characterized in the context of fault tolerance computing are:

- Asynchrony of process computations: in general it cannot be assumed which is the time needed for a process to perform a part of its computation. Moreover, it is not possible to relate the relative speeds of different processes during the computation.
- Asynchrony of communications: processes cannot rely on any timing assumptions about the latency of communications.

Group communication primitives [36] allow a set of processes to communicate between themselves. A typical case is that of interactions involving multiple producers and/or consumers. The primitives are characterized w.r.t. their properties. A partial list of properties that can be provided follows:

Agreement If a process receives a message, then all other processes eventually receive the same message. This property is said to be *uniform* if it applies both to failed and survived processes.

Total Ordering Suppose that multiple clients perform multiple requests to the same set of receivers. The communications are said to be totally ordered if the ordering of different client requests to receivers is equal for all recipients. The uniformity feature is the same of above. Uniform total ordering is a critical operation also for data base management systems to implement atomic commitment at the end of an atomic action (see Section 2.2.3 below).

Causal Ordering It is defined exploiting a transitive ordering relation between the nondeterministic events in a distributed system (see [57]). In this ordering of events a send operation precedes the corresponding receive one. As processes are assumed to be internally sequential, successive operations on a same process are trivially ordered. Causal ordered communications ensure that communication orderings reflect the ordering of events: if the sending operation for a message (to multiple recipients) causally precedes the sending of another message then they are accordingly ordered on the receivers.

Ensuring these properties is trivial for point-to-point communications, whereas it is an issue for multiple processes if the asynchrony issues described above arise. The relationship between the synchrony assumptions for a system and the possibility to implement the above mechanisms is discussed in [38].

It is straightforward to notice that group membership services and group communication primitives are strictly related between each others. For instance, an algorithm implementing total ordered multicast can be exploited to implement a group membership service. The reduction is possible because the problems at the core of both mechanisms can be reduced to the (solution of the) consensus problem. In the fault tolerance literature this problem has been chosen to characterize several research issues. For instance, an earlier work state that it is impossible to solve consensus in totally asynchronous systems in spite of even a single failure [43]. This problem is partially solved by introducing failure detection systems with some weak property [26], which extend the system model with some kinds of synchrony properties.

2.1.4 Distributed Objects and Advanced Techniques

Replication techniques have been also employed to support fault tolerance for more abstract programming models like multi-agent systems and distributed objects. In these models there is no knowledge of the structure of interactions between distributed/parallel entities and local computations can be nondeterministic. For this reason, we review these contributions in this section.

In the context of a multi-agent system (see [62]) standard replication techniques are introduced. Also several hybrids of the above techniques have been provided to support agent replication. In addition it is possible to organize a same group of replicas according to different schemes (e.g. active and semi-passive) which are applied at the same time (i.e. to replicas of a same agent). The choice on the exploited technique can be done to optimize a *criticity* factor, which describes the resiliency to faults of the system of replicas.

In the context of distributed objects several CORBA implementations exploit replication techniques to implement the fault tolerant CORBA specifications [69]. As services are replicated, a single request consists in a set of replicated invocations. The main issue faced in these studies is the way in which to handle these replicated invocations. The solutions can be characterized according to the way in which the management of this handling is introduced in the CORBA architecture:

- Integrative: the Object Request Broker (ORB) is modified to provide specific mechanisms for managing requests issued to replicated objects.

- Interception-based: invocations of clients to the ORB are captured externally to it. Requests are mapped in calls to a group communication library if a replicated service is invoked.

- Service-based: a set of CORBA objects implement a service for managing the replication of application-defined objects.

The Eternal system [66] belongs to the class of interception-based implementations. It provides a support for active and passives replication schemes. Some additional support is provided for checking the possibility of nondeterministic actions in the implementation of objects. The implementation of replicated invocations exploits the Totem [65] system that provides group communication primitives with ordering constraints. The DOORS project [67] provides an implementation of the fault tolerance CORBA specifications following the service-based strategy. Two main components are provided for applications: a replication manager, which provides active and passive schemes, and a failure detector system. AQuA [55] is a framework belonging to the class of interception-based strategies.

The Co-Replication technique [14] applies to a more abstract level of description of the computation (w.r.t. the process level). It represents the connection link between unstructured models of parallel programming and high-level ones in the context of replication. The computation is composed of a set of stateful modules that are replicated. Each module deterministically modifies its local state during the computation. The Co-Replication technique is introduced by observing that in classical replication techniques all replicas of a same process perform the very same computation, i.e. they simulate in parallel a sequential computation. Unlike this behavior co-replicated modules perform the same program on different input data, i.e. they perform different computations. As a consequence the local state of each replica assumes different values. To survive failures the co-replicas periodically exchange information related to their local state, in a way that the contributions to the whole computation of each replica is disseminated on all other replicas. Operationally when a replica receives the local state of another replica it merges up its state with the received one. To support this kind of information sharing co-replicated modules are required to perform a computation that, at some abstraction level, can be modeled as a Complete Partial Ordering (CPO in short). The implementation of state merging exploits the CPO definition of the local state of replicas. Co-Replication has been proved to be well-suited to support the master-slave strategy. It can be also applied to data parallel computations.

2.2 Fault Tolerance for High-Level Models of Parallel Programming

High-level programming models do not expose the process level directly to the programmer, but they provide abstractions to express parallel and distributed computations. In some cases, programs are defined as a set of inter-dependent tasks (or functions) to be performed on some input provided at run-time. Task dependencies can be expressed as direct acyclic graphs (DAG in short) [78], data-flow graphs [2], or

inter-dependent functions [18]. The implementation is typically based on a master-slave strategy or on a set of identical executors. For instance, in [2] a data-flow graph is encapsulated in a single function and replicated on a set of independent slave processes. The master is responsible of providing input data to slaves. Similarly, [78] implements the task inter-dependencies on a (logically) centralized scheduler, which assign works to a set of identical executors. In [18] a set of parallel workers exchange tasks according to the dependencies between the functions in evaluation. In some other cases the programmer is provided with constructs expressing well-known parallel structures (e.g. *skeletons*) for which the interactions between the parallel activities follow some known pattern. In [31] it is presented a set of library functions expressing skeletons as extension of the Message Passing Interface. In [56] skeleton constructs are provided as library functions for C/C++ programs. A different approach based on a coordination language is described in [89]. The implementation of this kind of models is not fixed (e.g. based on master-slave) but it is *ad-hoc* w.r.t. the parallel structure.

2.2.1 Fault Tolerance in Task Parallelism

We review the fault tolerance support provided by high-level models of parallel programming. In general fault tolerance is achieved by exploiting the functional replication expressed at some level of virtualization (e.g. implementation or programming model levels).

Cilk [18] extends the C language with abstractions to express parallel procedures implemented with multiple threads. Programs are represented as nested functions and parallelism is achieved by parallelizing the evaluation of functions and sub-functions on multiple identical workers. In practice, the programmer defines a set of *threads* which perform subparts of the computation and that can spawn new threads, as sub-procedures. Unlike sub-procedure calls, spawned threads can be executed in parallel with their caller. In [18] it is presented an example of Cilk program computing Fibonacci numbers according to a recursive style. The Cilk control-flow is implemented by means of a task dissemination technique called *work stealing*, which also implements load balancing between workers. Fault tolerance is based on the parallel structure of the implementation: in the case of failure of one of the workers, the sub-part of the computation it was evaluating is assigned to another worker. The failure of a worker that previously scheduled sub-computations to other workers is obtained by checkpointing its local state and by recovering it in the case of failure.

Satin [91] enjoys the programming model and implementation of Cilk, but it targets Grid environments. It aims at providing the programmer with a construct to express Divide-and-Conquer computations. The computation consists in performing

a set of interdependent tasks organized in a dynamically generated tree structure. A Satin program includes a set of Java methods, which the programmer specify as executable in parallel and which are opportunistically performed in parallel by the run-time system. The programming style is similar to the Cilk one, except that it extends an objective programming language, instead of an imperative one. In [88] it is shown a simple program computing the Fibonacci numbers according to a recursive style. The implementation of Satin is based on a set of identical workers, and shares the *work stealing* techniques of Cilk. Fault tolerance is achieved by exploiting a global result table replicated over workers. The result table is locally updated by each worker and kept consistent in an asynchronous fashion between them. In the case of failure and recovery of a worker the result table is analyzed to schedule the lost tasks. Clearly, as result tables on different workers can be inconsistent w.r.t. each others tasks execution can be replicated.

Muskel As described above, a muskel [2] program is defined as a data-flow graph. As example, in Figure 2.1 we show the graph part of a `muskel` program that specifies the graph of the program. Two farms, defined at lines 1 and 3, are composed in a

```
1  Compute worker1 = new Inc();
2  Compute stage1 = new Farm(worker1);
3  Compute worker2 = new Inc();
4  Compute stage2 = new Farm(worker2);
5  Compute main = new Pipeline(stage1,stage2);
```

Figure 2.1: Example of the definition of the macro data-flow graph in a `muskel` program.

pipeline of two stages, defined at line 5. The implementation of muskel is based on a master-slave scheme and it maps a whole graph into a same function replicated on a set of identical slaves. The master is responsible of assigning tasks and obtaining results from slaves. The kinds of failures that are supported are restricted to (the nodes executing) slave processes. The master process is assumed to be executed on some robust node or implemented in some failure-resiliency fashion (e.g. replicated over several nodes). The failure of a slave induce the loss of the partial computation of the task it was performing. The fault tolerance strategy is based on re-scheduling the tasks that were performed by failed slaves. Failure detection is based on the one provided by the communication support which links the master to the slaves.

Charlotte [9] provides the programmer with constructs to express parallel and sequential routines. Parallel routines are expressed as blocks delimited by two Charlotte primitives: *ParBegin* and *ParEnd*. The commands inside parallel blocks

spawn parallel computations. The implementation of Charlotte exploits a master-slave strategy. Fault tolerance is achieved by task re-scheduling in the case of slave failure, while the master failure is not addressed.

Master-Worker [47] expose to the programmer constructs to express a master-slave computations. Fault tolerance of slave processes is implemented with task re-scheduling by the master. The programmer is also provided with checkpointing facilities that can be inserted in the master code to support master fault tolerance.

Comparison of Fault Tolerance Techniques for Task Parallelism As it should be clear from this review, *fault tolerance is based on the definition of a computational unit (the task) and on their replicated execution*. The replication can be both a kind of hot redundancy and cold redundancy. In the former case a same task can be performed in parallel on different executors. In the latter one a task execution is replicated only if it fails. The main contribution of these kind of projects resides in the intuition that the replication of tasks can be managed in a proper way if there is a global identification of tasks. This point is clear in [91] in which the fault tolerance support exploits a replicated result table.

2.2.2 Specializing Checkpointing and Rollback Recovery for High-Level Models

Checkpointing and rollback recovery techniques have been also used to support fault tolerance to high-level models of parallel programming. The task of defining the mechanisms and the protocols implementing the support is made easier by exploiting the assumptions on which the programming model is based. In this section we review a notable contribution in the context of High-Performance computing. We describe in details this contribution because it is representative of a class of complex research studies that highlight the relationships between abstraction properties and fault tolerance mechanisms.

The ProActive environment [12] features a well-defined programming and computation model based on a process abstraction called *activity*. Fundamentally, an activity is made up of a main thread and an active object along with a local private state. An active object is a unit of computation that provides methods that can be remotely invoked with explicit requests. Activities can send requests to active objects and they can possibly receive back replies. Multiple requests to a same active object are sequentialized in a local queue w.r.t. their arrival order. When an activity requests for a method execution on an active object, if there is a result, its actual value is replaced by a placeholder of it (the *future*). The placeholder will be replaced by the result value when it is sent back from the active object.

As stated above, the fault tolerance support of ProActive is based on checkpointing and rollback recovery, which is defined by exploiting properties of the computa-

tion model. These properties define a relation between the ordering of communication events and the computation of active objects. To support fault tolerance two properties are derived:

- The ordering of requests made to an active object does not influence the general semantics of the computation.
- An active object computation can be fully characterized by the ordering of requests it receives. To characterize a request it is sufficient to use the identity its sender.

The implemented checkpointing protocol belongs to the class of Communication-Induced-Checkpointing (or CIC, see Section 2.1.1), which require processes the ability of taking checkpoints at any instants of the computation. Active objects cannot take a checkpoint at any instants of the execution because their state is not stable while serving a request. As a consequence the CIC protocol builds inconsistent recovery lines (see Chapter 3) because some checkpoints can include orphan messages. Application messages are piggybacked with checkpointing indexes. Local checkpoints are labeled with progressive indexes: the triggering of a checkpointing operation on an active object is given by the receiving of a higher checkpoint index on an application message. To support correct recovery to consistent states after a failure a message logging technique is introduced. The logging protocol is defined as following: suppose that an activity requests an active object to execute a method. When the object receives the request it synchronously saves on stable storage its determinant (i.e. the identity of the requesting activity). In the case of failure all activities rollback to a same checkpointing line, i.e. they all select the checkpoint labeled with the same index. Next each active object retrieves from stable storage the identity list of other activities that previously sent a request. Active objects recover to a consistent state by serving requests according to the recovered list. In fact, the same ordering of events is enforced in the re-computation.

As a rationale of this approach notice that the computation model of ProActive does not include any structuring of the interactions between distributed activities. This induces the exploitation of existing checkpointing and rollback recovery techniques, without giving any chances of optimizations to the programmers. However, the high-level model of computation is used to define in a simple way a checkpointing protocol and to prove its correctness in an elegant fashion.

2.2.3 Expressing Parallel Computations as Atomic Actions

An atomic action is a portion of a (possibly parallel) program which consists in a set of operations that modify the state of the computation. The operations included in an atomic action are performed in an indivisible way, i.e. the operations are performed "all or none". The consequences of this property, called *recoverability*, are that, if one or more failures occur during the execution of the operations in an

action, the value assumed by the state before the first operation(s) of the action is restored. Otherwise, if no failures occur, all performed state modifications are made persistent (i.e. successive failures cannot affect them). Another property of atomic actions, which is more interesting from the concurrency control viewpoint, is their *serializability*: the execution of a set of atomic actions, which are concurrently executed on the same state, semantically corresponds to an execution performed according to some serial scheduling of them. In the description we give here we will focus only on the recoverability property of atomic actions because it is strictly related to the fault tolerance context.

We are interested in the case in which the operations of an atomic action are performed on different processes, which are possibly executed on different computational resources. In this case the atomic action is said to be *concurrent*. From the syntactic viewpoint programmers are (at least) provided with two primitives to define an atomic action:

- A primitive to define the beginning of an atomic action, which ensures that the state of the computation can be recovered at the end of the action.
- A primitive to terminate the execution of an atomic action, which ensures that the state modifications have been made persistent to failures (i.e. successive failures will not induce the loss of information).

In the context of data base management systems several research issue have been faced: [13] reviews some earlier research issues and the provided solutions, which are still the basis of current solutions.

Atomic actions are the base mechanism to support fault tolerance in the Pact parallel programming environment [60]. In Pact the parallelism at the level of the programming model is based on a task parallelization scheme over a shared memory abstraction. An *action* is defined as a set of tasks and it is started and terminated by two specific primitives of the language. These primitives implements the atomicity of the actions inside the parallel construct that modify shared variables. The implementation is based on a set of identical parallel processes and it exploits a two-phase commit protocol [49] to ensure the correct termination of actions. To support fast recovery of long-running atomics, the programmer can also exploit a checkpointing facility for parallel executors. Also in [81] atomic actions are exploited to support task parallel computations.

2.2.4 System-Level Fault Tolerance Exploiting High-Level Models of Parallel Programming

In the previous sections we have focused on fault tolerance supports to parallel and distributed programming. Fault tolerance have been defined also at the level of operating systems for massively parallel architectures, in a way in which several mechanisms are transparently provided to processes. In this section we highlight a

contribution in this field that is based on a high-level model of computation, which has been developed in the context of parallel programming.

The intuition at the base of this contribution is that the system software of a (massively) parallel system is actually a parallel application like any other user applications. An example of a system software task that is a parallel computation is the scheduling of processes to multiple processors. In this approach existing solutions to parallel programming are also applied to system tasks. In [46] the Bulk Synchronous Model (BSP in short) is chosen as computation model to implement a minimal set of system primitives. In a BSP [80] computation parallel activities proceed in synchronized *supersteps* each one including:

- A computation phase, in which the parallel activities locally perform the computation. In this model interactions are assumed to be poor w.r.t. computation operations and are delayed to the communication phase.
- A communication phase, in which all pending interactions are performed.

In the context of this kind of operating system all activities proceed according to a global synchronization signal, which is issued every amount of microseconds. All communication operations are delayed until the next signal is received. After the signal is issued all pending communications are performed. Next a new superstep is performed.

The characterization of the computation in supersteps gives some interesting properties that can be exploited to efficiently support communications and high-level tasks. We are mainly interested in the determinism that this model induces on the computations. In general, communications can be characterized w.r.t. the source of nondeterminism for computations, which induce the creation of inconsistencies in the case of failure and rollback (see Section 2.1.1). By forcing communications to be performed in a BSP-like behavior the state of the computation can be characterized w.r.t. communication operations. Actually, after a synchronization, all pending communications are performed, and the applications (both user and system ones) are in a consistent state (orphan message avoidance, see Section 3). As a consequence of this consistency property it is simpler to define coordinated checkpointing protocols than in totally asynchronous and nondeterministic computational models.

In the context of unstructured models of parallel computing the BSP model has been exploited to implement the MPI interface [70]. Applications exploiting the communication support of this implementation of MPI enjoy the BSP model of computation. Thus checkpointing protocols can be easily defined in a coordinated, efficient and user-transparent fashion.

Chapter 3

A Structured Approach to Fault Tolerance

In this chapter we introduce the programming model on which we based our study of fault tolerance for High-Performance computing. We give a detailed description of two constructs, expressing two main classes of parallel computations, namely task and data parallel. In this thesis we fully develop fault tolerance aspects for these constructs to show our research methodology. Next, we introduce a formal tool to highlight the properties of the model that are interesting for reasoning about fault tolerance. We show how to apply this to these two highlighted classes of parallel programs.

3.1 Programming Model

The structured parallelism programming model is based on *high-level* constructs that express *specific* classes of parallel computations. Each class features a specific structure (or pattern of structures) of the interactions between the parallel entities defining the computation. We name these kinds of computations *structured parallel computations* (for brevity, we sometimes omit the "parallel" adjective), or *structured constructs*, in the case in which we are referring to their linguistic shape. Instances of structured computations are *algorithmic skeletons* [31], that express well-known parallel computations. Skeletons have been defined because:

- They can be specialized to implement specific parallel algorithms. The application programmer is provided with constructs expressing parallel structures that must be completed by specifying which are the functions implementing the wanted behavior. Thus, the programmer tasks are: (1) to choose a proper parallel structure for its application, or part of it; (2) to specify the actual semantics of the computation, by providing sequential functions that implement the application. Syntactically, a structured computation can be seen as a

higher-order general function [30] to which the application programmer passes the user-defined functions which specialize its behavior.

- They relieve the programmer of dealing with implementation and architectural specific issues. For instance, the programmer is unaware of the mechanisms used to implement the interactions between the parallel activities of the application (e.g. message-passing, or shared memory). These tasks, which are highly error-prone, are delegated to the programmer of the support to the structured construct.

- Each of them has an associated cost model, describing their performance behavior w.r.t. the user provided functions and the platform on which they are executed. These models, usually called *performance models*, have been employed to support skeletons with adaptive behaviors [90], and to ensure performance portability [89].

From the expressiveness viewpoint, the *structured parallel computations* are more general constructs w.r.t. skeletons, since a single structured construct can express a class of skeletons. We will see below two examples of structured computations which express two different classes of algorithmic skeletons.

To overcome the clear expressive limits of stand-alone skeletons, we compose structured computations in *generic* graphs. Thus, a parallel application can be viewed as a graph, where nodes are internally parallel or sequential (corresponding to structured computations), and edges are streams, i.e. possibly unlimited sequences of typed elements. Also graphs feature a high-level and partially structured definition because:

- The types of stream elements and of interfaces of the linked parallel and sequential nodes are specified in the program.

- The streams are statically defined, i.e. the programmer specifies which are the interfaces that each stream binds.

In general we assume that structured computations, which we compose in generic graphs, are stream-based. That is, a structured computation has a set of input streams, from which it obtains its input data, and a set of output streams, on which it provides the results. The results of a structured computation, provided on an output stream, can be consumed, according to the application graph, as input data for other computations (i.e. other nodes of the application graph).

In this chapter we study fault tolerance for structured parallel computations by exploiting their high-level definition and properties. The high-level definitions allows us to separate the concerns of:

- Description of the computation from an abstract viewpoint, including its structure and semantic.

- Description of its implementation.

As a consequence of this separation, we can describe in abstract terms the fault tolerance mechanisms and protocols. Next, we can map them in the implementation. Moreover, the properties featuring structured computations are used to optimize the fault tolerance support. The impact of such optimizations on the fault tolerance overhead, and consequently on the whole application performance, are modeled by extending the performance models of structured computations.

To illustrate our approach, we consider two examples of structured computations, and we describe the modeling and implementation of a fault tolerance support, based on checkpointing and rollback recovery techniques. Concerning the two classes:

- The first class is characterized by functional replication: a stateless module, which implements the evaluation of a "pure" function, is replicated on a set of identical executors.

- The second class features state partitioning and replication of functions: the same functions are applied to different partitions of a same data structure. The evaluation happens in parallel and it is applied to the whole state. We also assume that evaluations are iterated.

These two classes represent a large set of parallel programs, typically named *task parallel* (the former), or *farm*, and *data parallel* (the latter). Below we give a formal model of these classes, that is used in the definition of the fault tolerance support. We leave the description of the fault tolerance support for generic graphs to future work, which will be a composition of the solutions presented here.

3.1.1 Farm

We define farm computations as the parallel evaluation of a (user-provided) function **F** (without side-effects) on a stream of input elements generating a stream of output elements. The applicability of this structure of parallelism is independent of the actual semantics of **F**. We define the task of the computation as the evaluation of **F** on an input element. Tasks are independent w.r.t. each other. The entity that executes tasks is called *worker*, and represents the unit of replication for parallelism. Different workers perform different tasks and generate different results. The farm model includes a notion of management of workers for the distribution of input elements and the collection of results. That is, it can be characterized by the strategy of scheduling it employs: typical strategies are round-robin, and on-demand. The collection of results can be performed according to the FIFO policy. Otherwise, it can be performed by re-ordering output results w.r.t. the sequence of arrival of the corresponding input elements.

We can now clarify the difference between algorithmic skeletons and structured parallel computations: the farm parallel paradigm is a structured computation while

its instances, based on specific scheduling and collection strategies, are specific algorithmic skeletons.

Cost Model

We identify the three main stages composing a farm support as:

Stage 1 or Input Stage: collection of input elements and their scheduling to workers.

Stage 2 or Computation Stage: application of the function **F** to an input element, producing an output result.

Stage 3 or Output Stage: collection of results from workers, and propagation to the output stream.

Clearly, these stages are performed in parallel, according to a pipeline semantics. We can associate a cost to each phase:

- The first stage requires the time needed to apply the scheduling strategy, and to obtain an input element. We denote this cost with T_{IN}.
- The second stage requires the time to evaluate **F** once. We denote this cost with T_F.
- The last stage requires the time to apply the collection strategy and to provide a result. We denote this cost with T_{OUT}.

Starting from these quantities we can obtain some interesting metrics, like the output stream bandwidth, and the optimal number of workers:

Service Time the bandwidth of the output stream can be computed as the inverse of the farm service time which is: $T_{farm} = max(T_F, T_{IN}, T_{OUT})$. This is derived from the pipeline semantics of the three stages composing the farm.

Optimal Number of Workers The optimal number of workers, i.e. the maximum number of workers after which no performance gain is obtained, is limited by the performance of the IN and OUT stages. It can be computed as:

$$n_{opt} = \frac{T_F}{max(T_{IN}, T_{OUT})}$$

The performance model for a specific application executed on a specific platform can be obtained by instantiating the three variables to the actual values, given by the execution platform (static or dynamic) configuration, and the application-specific costs (i.e. the cost of **F**).

3.1.2 Data Parallel

Data parallel programs are based on a strong notion of state, typically represented as a collection of elements. The computation consists in iteratively applying a given function **F** to each element, until some termination condition is eventually satisfied. A virtual processor (VP) is the abstract unit of parallelism and it is defined as the iteration of the evaluation of **F** on a *single* state element. We also assume that each VP is the only one that can modify the value of its assigned element (Owner-Computes rule). For modeling purposes, we uniquely identify VPs. In some cases, we will map names onto integer numbers reflecting, when possible, the assigned element indexes. Data parallel programs are characterized by the functional dependencies between the VPs, commonly referred to as *stencil*. The application of **F** to a state element can require the old values of other elements. A stencil defines for each step of the computation which are the elements needed to evaluate **F** on each element of the state. Stencils can be modeled as graphs of dependencies between VPs during execution. We can classify data parallel programs w.r.t. the moment in the compilation/execution phase in which functional dependencies are known:

Static stencil The subset of elements needed at each iteration for each evaluation is completely specified in the program, i.e. it is known at compile-time.

Dynamic stencil The subset of elements needed at each iteration for each evaluation is known *only* at run-time. For instance, it depends on the values assumed by the state.

The first class can be further characterized depending on the variability of the stencil w.r.t evaluation steps:

Map the application of **F** to an element at each step requires only the value of the same element computed at the previous step.

Fixed Stencil the evaluation of **F** for a given element needs a subset of other elements (along with the element itself), which is equal for all iterations. Also in this case, the values used are those computed at the previous step.

Variable Stencil the evaluation of F for an element needs, at each iteration, different subsets of other elements, whose values are computed at the previous step (along with the element itself).

In the programming model we target, we indeed assume that the elements needed for each evaluation are referred to the same step, i.e. the previous one: for evaluating F at the i-th iteration the values of the needed elements are obtained from the $i-1$ step. In this thesis, we will study the fault tolerance support for static stencil data parallel programs.

From this description it is important to notice that the computations expressed by data parallel programs are deterministic in two ways:

- Multiple applications of **F** to the same input data always produce the same output result.

- For static stencils, dependencies between VPs does not change between different runs of the same computation, as they do not depend on the state values, or on external factors.

We show the difference between the concept of algorithmic skeleton and structured parallel computation for the data parallel case. We consider the generic class of data parallel programs as a structured parallel computation, where the stencil of the program is still to be instantiated. Each data parallel instance, given by a specific stencil rule, is an algorithmic skeleton.

Cost Model

Typically, the performance model of a data parallel program describes the completion time of the computation, and is based on:

- The time needed by VPs to exchange the elements between themselves, to satisfy the functional dependencies given by the stencil. We will denote this quantity $T_{INT}(l)$[1], where l is the size of an element.

- The time needed to apply **F** to the local element. We will denote this quantity with T_F.

Notice that the first quantity depends on the kind of stencil required to evaluate **F**. In a *Map* program this cost is null because no interactions are needed to perform the local computations. For static stencils, T_{INT} varies only according to the element size, and it is instantiated, at the lower virtualization level, with the costs of interactions, which depends on their implementation. For fixed stencil programs, T_{INT} is equal at all steps, unless there are dynamic variations of the performance of the support implementing the interactions between VPs. This is not true for variable and dynamic stencils, for which the variation at each step of the pattern of interactions also induces a variation of T_{INT}.

We define $T^i_{Step} = T_F + T_{INT}(l)$ for a generic $\mathbf{VP}_i$ with an element of l size, and we compute the data parallel completion time for M VPs as: $T_{comp} = N \times \max_{0 \leq i \leq (M-1)}(T^i_{Step})$, where **N** is the number of times that **F** is evaluated. Notice that we assume to know the number of iterations that the computation will take. If this is not the case, we can still obtain the step performance of the computation. We also assume that the slowest VP (we take the maximum step time) is always the same. If this is not true, we have to compute, for each step, the slowest one.

[1]The **INT** subscript stands for "interaction".

3.2 A Tool to Address Fault Tolerance Aspects

We introduce a tool to describe structured computations that allows us to highlight the properties used in the definition of fault tolerance at various implementation levels. Below, we also show how the tool can be applied to the chosen classes of structured computations. The tool is inspired by an existing data structure, defined for data-flow languages: the *Incomplete Structures*[7], or I-Structures. An I-Structure can be defined as a collection of typed elements, possibly of unlimited size. Elements in an incomplete structure have a unique position, or an index. Two operations are defined in order to access an I-Structure:

- *put*: stores a given element in a given position. The prototype of this operation, which will be used in the next section, is $put(position, element)$, where both arguments are input.
- *get*: given a position, it returns the element that is contained, or will be contained, in that position. The prototype of this operation, which will be used in the next section, is: $get(position, element)$, where the first argument is an input, and the second one is an output.

These operations are characterized by two main properties, which give the actual semantics of interactions that can be expressed with I-Structures:

Property 1 *No more than one put can be performed on the same position.*

Property 2 *The get operation on a given position blocks until a put operation has been performed on that position.*

We exploit I-Structures to model the streams, the state variables, and the computation of structured constructs. The general idea behind this approach is that we introduce *sequence identifiers* to model control- and data-flow, which can be used to introduce consistency definitions, to design checkpointing and rollback recovery logic, and to design specialized stable storage supports. In particular, the sequence identifiers relate the actions and events of the computations (the control-flow), to the modifications on the state, or to the interactions between parallel activities (the data-flow). The sequence identifiers of computations can be derived by exploiting the property 1:

Definition 1 *The control-flow of each parallel activity (either worker, or VP in the examples of above) is modeled as a sequence of logical steps. For a generic parallel activity, each step is defined as its activation, exploiting the blocking semantics on its input streams. Computation steps can be defined with sequence identifiers locally on each parallel activity.*

We will see below how farm and data parallel control-flow can be modeled exploiting this lemma. Sequence identifiers for state and streams can be derived by modeling them with I-Structures. In particular, we can derive two lemmas:

Lemma 3.2.1 *Suppose we model a stream with an I-Structure. The elements are assigned a unique position, that cannot be modified, i.e. elements cannot be overwritten in the abstraction as a consequence of property 2. A position of an element represents its sequence identifier.*

Lemma 3.2.2 *Suppose we model a state variable with an I-Structure. Each successive value assumed by the variable is mapped onto a new I-Structure position, which cannot be modified, according to property 2. A position of a value represents its sequence identifier.*

We exploit the relationship between computation sequence identifiers and state or stream sequence identifiers to define stable storage and consistency definition, to support checkpointing and rollback recovery. Figure 3.1 shows the relationships

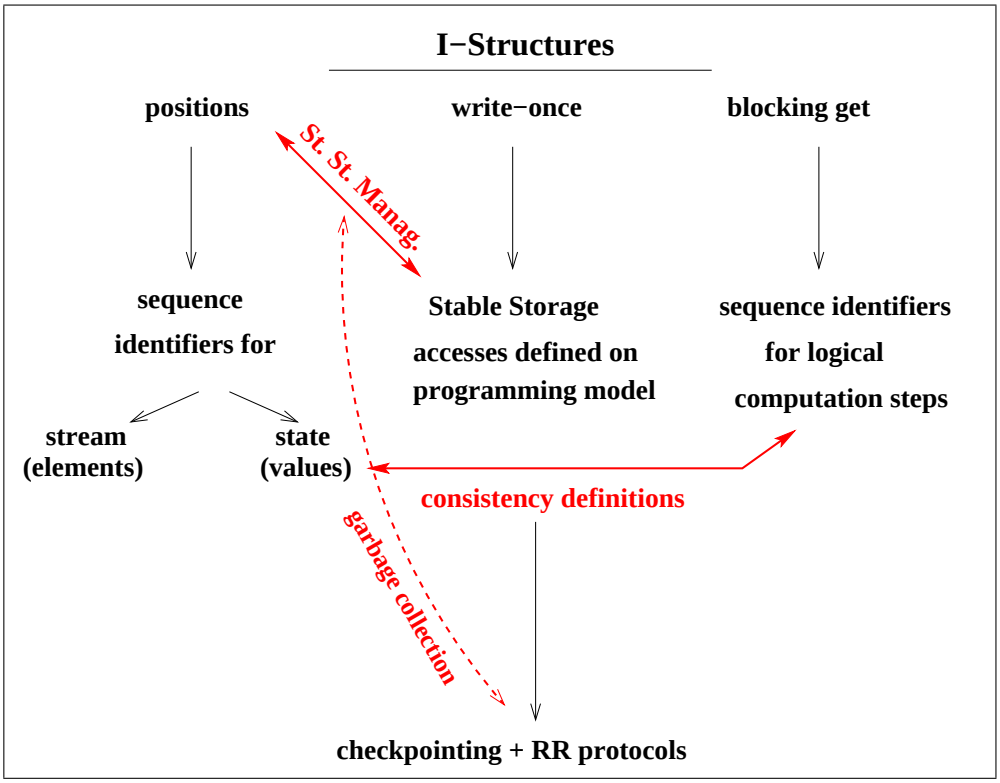


Figure 3.1: Representation of the I-Structure properties and their relationships with FT aspects.

between I-Structures and FT aspects. The I-Structure properties are (top of the figure): (1) elements have a (unique) position; (2) the put is write-once; (3) the get is blocking. We have seen from lemmas 3.2.1, and 3.2.2 that element positions can be used to introduce sequence identifiers for stream elements and the values of state variables. On the other hand, definition 1 allows us to assign sequence identifiers

to the parallel steps performed by each parallel activity composing the computation. We can exploit the relationship between sequence identifiers of stream/state and computational steps to define abstract consistency in the data-flow of parallel computations. The consistency definition we give is abstract in the sense that it does not involve implementation abstractions. Rather, it is placed at the level of definition of the parallel activities making up the applications. Consistency definitions are then exploited at the lower level to implement checkpointing algorithms, and rollback recovery protocols. We also exploit the write-once property to model stable storage at the level of programming model abstractions. That is, at the level of programming model we define the interactions between parallel activities as operations on I-Structures. These are used as *hooks* at the implementation level to insert stable storage accesses, and also to exploit the knowledge of the stream/state types. Clearly, the accesses to the stable storage must be inserted according to some logic: for instance, the logic of checkpointing (e.g. consistency) is derived from the analysis of the parallel structure and semantics, and from the exploitation of the sequence identifiers for stream/state and computational steps. Finally, garbage collection of checkpoints can be based on general properties of the semantics and structure of the parallel computations, and the checkpointing and rollback recovery tasks.

Below we exemplify the exploitation of incomplete structures by focusing our attention on the two specific structured computations described in Section 3.1. We highlight how the above lemmas can be exploited to define consistent states (i.e. states that can be used as rollback recovery targets), and how this information is propagated to the lower implementation levels to define rollback recovery.

Finally, for the data parallel model, we have to add a parameter to the put and get operations, in order to address a specific field of a data structure value at a given position:

- **put(pos, field, el)** denotes that we write the element **el** in the field **field** of the modeled data structure at the position **pos**.
- **get(pos, field, el)** denotes that we want to store in **el** the field **field** of the position **pos** of the modeled data structure.

Motivations for Introducing I-Structures and Other Data Structures In this thesis we have selected I-Structure to model the properties of structured parallel programs useful for fault tolerance purposes. I-Structures seem to capture the essential idea behind checkpointing techniques, which are the basis of rollback recovery protocols. The idea is to describe, at some level of abstraction, the history of a parallel computation (by means of I-Structure positions), which must not be overwritten (write-once property), as it happens in actual levels of implementation. That is, the data-flow semantics abstract in a proper way the idea of checkpointing. Checkpointing procedures are also introduced by exploiting the high-level definition of parallel programs, *on the hooks naturally introduced by blocking points of the computation*, defined according to the get blocking property. In these points, which are

characterized by such hooks, the state of the computation is completely specified, as in the case of reconf-safe points in the modeling of dynamic supports for structured parallel programming [90].

Other data structures introduced in the context of data-flow languages have been also considered. M-Structures [6] have been introduced to model atomic accesses to variables. Unlike I-Structures, the elements stored in a M-Structure are removed when read by some consumer. This is in contrast with the interesting logic behind the I-Structure for checkpointing, for which elements cannot be removed from the abstraction. Such elements could be useful as later rollback targets. The semantics of accesses to J-Structures [58] is similar to the one of I-Structures, in such a way that the former can be used to implement the latter [58]. Finally, L-Structures [58] are as I-Structures, but they also support nonlocking read accesses. In our programming model we have expressed this nonlocking semantics inside the alternative constructs which allow a module to select a ready stream amongst a list of input streams.

3.2.1 A Model for Farm based on I-Structures

We present an abstract model of farm structured computations, highlighting fault tolerance aspects, that we use to implement checkpointing and rollback recovery. Recall that, in farm computations, workers access in mutual exclusion to an input stream, to obtain input data, and to an output stream, to provide output results of local computations. We manage worker accesses in terms of proper accesses to incomplete structures that model the input and output streams. That is, we map the two streams in two different I-Structures, as shown in Fig. 3.2. In more details,

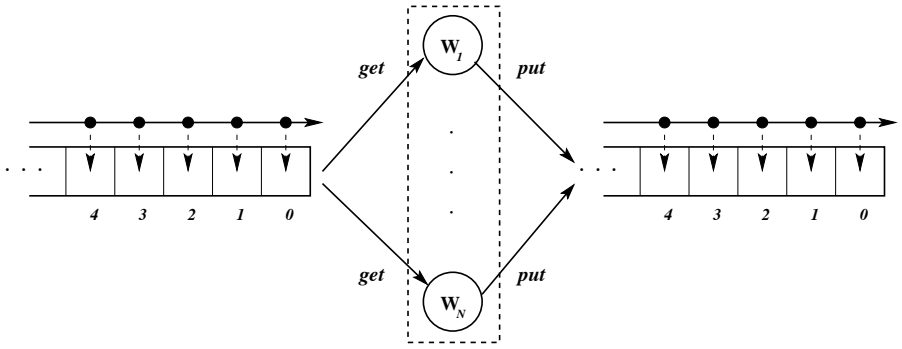


Figure 3.2: Abstract model of a generic farm computation. Two incomplete structures model input and output streams. Workers independently access them. The scheduling of task to workers, and the collection of results is modeled in the choice of positions made by workers in the accesses.

the behavior of each *worker* is shown in Figure 3.3.

The first parameter of the *put* and *get* operations are encapsulated in two special purpose functions, named *compute-next-position* and *compute-free-position*. This is

```
1  while(true) {
2    get(compute-next-position(in_stream), element);
3    result = F(element);
4    put(compute-free-position(out_stream), result);
5  }
```

Figure 3.3: Pseudo-code of the abstract behavior of workers in a farm computation.

motivated by the fact that, at this level of description, we are not interested in modeling the way in which such positions are actually generated. We only exploit the concept of worker coordination, encapsulated in the get and put semantics, and we add all possible policies as part of the language.

The point highlighted in this model is that we dynamically associate a sequence identifier to each stream element. This is also associated to the corresponding evaluation of the function $\mathbf{F}$ on the input data, to the worker performing it, to the corresponding output result, and its sequence identifier on the output stream. Notice that, in the most general case, the index of an element on the input stream is not necessarily equal to that of its corresponding result on the output stream. That is, input data and output results can be un-ordered. The relationship between sequence identifiers is exploited at the implementation level to define rollback recovery protocols (see Chapter 4). We also argue that this relationship can be used to express garbage collection strategies. We will study this problem in details in future work (see Chapter 8). Also for garbage collection strategies, the knowledge on the parallel structure of computations can be exploited to introduce structure-driven optimizations.

To clarify the relationship between sequence identifiers, we show an example of execution of three workers. We denote with $\mathbf{T}_i$ the input element with position i on the input stream; with $\mathbf{R}_j$ the result of the evaluation of the function $\mathbf{F}$, copied in the j-th position of the output stream. Each input element is assigned to a different worker, that, after having evaluated $\mathbf{F}$, stores the result in a different position of the output stream (w.r.t. other results). Function evaluations are identified by the index of the worker performing them, and with the application number on the same worker: $\mathbf{F}_x^y$ denotes that the worker W_x performs the y-th evaluation of $\mathbf{F}$. In the first Figure 3.4(a) we show the assignment of the first three input elements. In this example, the scheduling is performed depending on the order in which worker requests arrive. In the example the order is W_1, W_0 and W_2. That is, accesses and computations of different workers are completely asynchronous w.r.t. each other. After having completed the computation each worker stores the computed result in the first free position. Figure 3.4(b) shows the next assignment of the tasks. The next accesses from workers to input and output streams are different, in the positions, from the previous ones. Clearly assignments and collections could proceed totally asynchronously for each worker. This is not shown in this example, as the

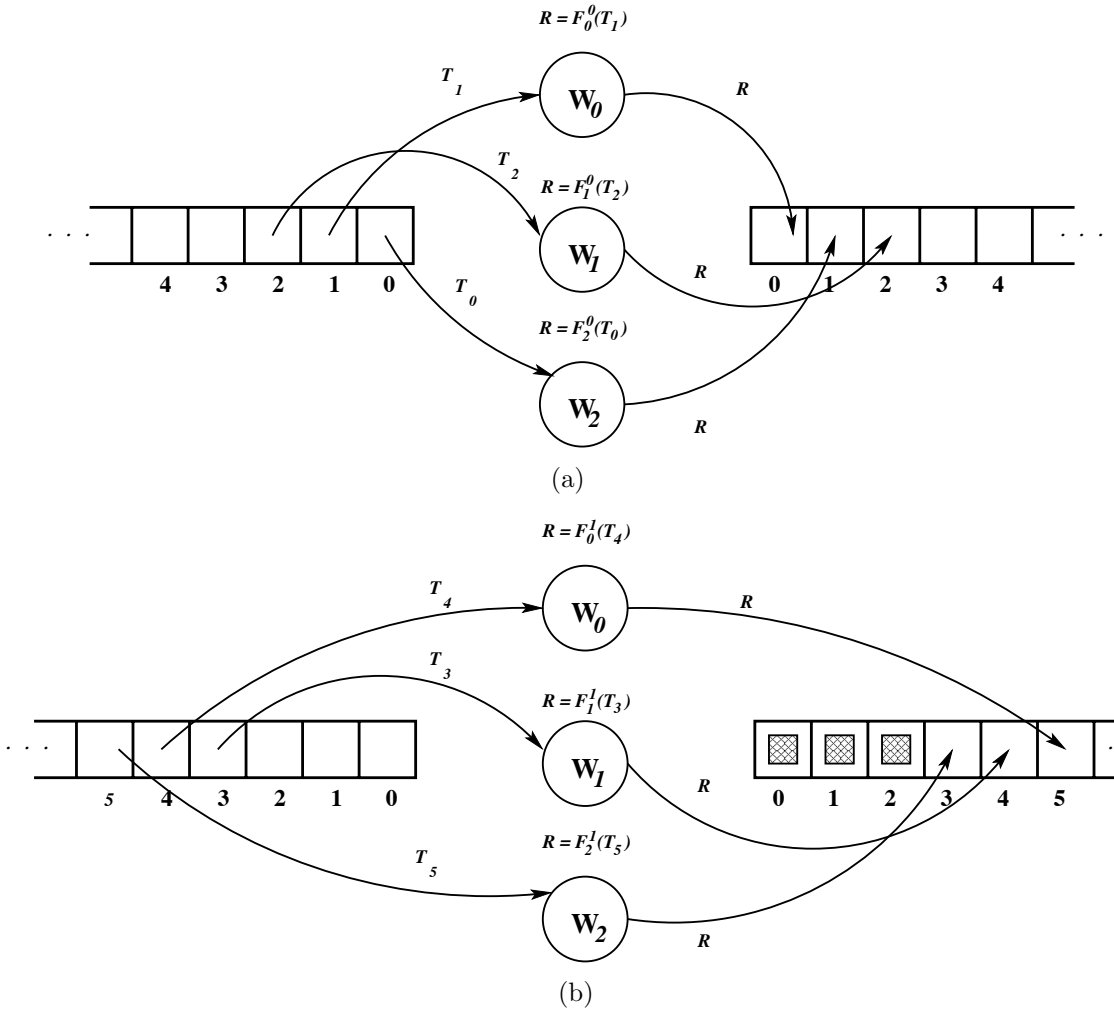


Figure 3.4: Example of computation of a farm, showing the relationship between input task element positions, worker names, and output stream element positions. The notation $R = F_y^x(T_i)$ denotes that the y-th worker is performing its x-th evaluation of $\mathbf{F}$ on the input task obtained from the i-th position on the input I-Structure.

first three results of the first three tasks are copied in the first three positions of the output stream, before any workers could perform other tasks. A clear case could be that of a worker that consumes input elements and produces output elements faster than the others, accessing and occupying contiguous positions, respectively on the input and output streams.

The building of the relation between evaluations and stream elements is done dynamically. For this example, the relation for the first two evaluations on each worker is shown in Table 3.1. This specific relation is a possible way of scheduling input elements to workers, and of collecting output results. As any workers can perform any tasks, any other mappings are allowed by the semantics parallel structure.

$$\begin{array}{ccccc} T_0 & \rightarrow & F_2^0 & \rightarrow & R_2 \\ T_1 & \rightarrow & F_0^0 & \rightarrow & R_0 \\ T_2 & \rightarrow & F_1^0 & \rightarrow & R_1 \\ T_3 & \rightarrow & F_1^1 & \rightarrow & R_4 \\ T_4 & \rightarrow & F_0^1 & \rightarrow & R_5 \\ T_5 & \rightarrow & F_2^1 & \rightarrow & R_3 \end{array}$$

Table 3.1: Dependencies between input elements, function evaluations and output elements of example 3.2.1. The indexes of T and R denote the positions of tasks and results on the input and output streams respectively.

Consequently, from the fault tolerance viewpoint, *it is not strictly required that the same relation is re-built in the case of failure and rollback recovery.* We will see that we do not want to enforce the same scheduling of a previous execution, in the case of failure, but we just need to avoid the re-computation of results.

It is useful to show how the consistency property for global states, defined in the context of distributed systems of processes [27], can be mapped at this abstraction level. In simple terms, the property states that, for a set of checkpoints,:

Property 3 *If the state of a process A includes the event "receive(B,m)", then the state of the process B must include the corresponding event "send(A,m)".*

We map this property at this level as:

Property 4 *If the state of a module A, consumer of a stream (with name str), includes the event "str.get(position,element)", then the state of the module B, producer on the same stream, must include the corresponding event "str.put(position,element)". Notice that the two positions specified by the two modules are equal.*

In the case of farm computations, at this level of abstraction, the producers and consumers, respectively on the farm input and output streams, are beyond the scope of the computation. The set including all the possible above relations is void. As a consequence we can derive the following lemma:

Lemma 3.2.3 *A snapshot of the execution of a farm program, including a local state from each worker, is always consistent during the whole computation.*

According to this lemma uncoordinated checkpointing (see Chapter 2) for farm computations can be applied without incurring in domino-effects. However, at the level of implementation, we will have to deal with lost and duplicated messages.

Below we introduce the fault tolerance strategy which we selected in this thesis to support farm computations and we provide motivations of this choice.

3.2.2 A Model for Data Parallel based on I-Structures

We introduce a model of data parallel programs to characterize the structured parallelism properties that we exploit in the definition of the fault tolerance support. We have seen above that these properties are modeled by the I-Structures, as they allow us to characterize the relationship between state modifications and computational steps. In particular,

- The blocking property of the get operation allows us to model the computations of a VP as a sequence of logical steps. The global computation of VPs is coordinated but asynchronous[2]. The labeling of steps allows us to define the relation between computation and interactions between VPs. This relation resembles the one defined for events in [57], which, differently from the one defined here, is defined at process level.

- Modeling the state and the VP computation under the requirements of the put property forces the design to avoid the overwriting of old state values that, in principle, can be used as rollback targets. The avoidance of overwriting of old state values, applied at all virtualization levels, is not sufficient to survive failures. We also have to "stabilize" all or some of the state values, to avoid loosing them in the case of failures. To do so we can insert periodic checkpointing operations of the state values, saving them on stable storage, on the *hooks* provided by the I-Structure operations.

It should be remarked that the write-once property of the put operation does not include saving the state values on stable storage, but it only requires that positions are not overwritten, independently of I-Structure implementation. In the case of failure, the values included in some positions that have not been stabilized are lost and are to be re-computed again. Theoretically, after rollback a new instance of I-Structure is used, where older "stable" values for the state are in their previous positions. The positions of the lost values are empty in the new instance of I-Structure. Recovery is equal to the lost computation, in the sense that it re-generated the same state values. Thus, the re-computed state values will be re-written in the same positions during recovery.

The I-Structure model of data parallel programs follows: a set of Virtual Processors (VPs) iterate the evaluation of a function **F** on their assigned elements. We have seen that **F** applied on a single element can be dependent, along with the element value itself, on the value of other elements assumed after the previous evaluation. In the abstraction, we model the state S with a single incomplete structure, where each position includes a set of elements $e_0, \ldots, e_{N-1}$. *Each position of the incomplete structure represents a value assumed by S at each parallel evaluation of F.* Each

[2]The logical coordination of a set of parallel activities is not obtained only by means of synchronizations, but it can be derived from the program/computation structure, as we show for data parallel programs. In data parallel programs, the coordination is a consequence of the fact that all virtual processors (see below) actually perform the same number of logical steps.

VP can access with *put* and *get* operations its own element, but only with get ones can it access its neighbor elements. The dependency rules between evaluations on elements, representing the program stencil (see above), are implemented by making VPs access the I-Structure modeling the state in a proper way. In Figure 3.5 we show the pseudo-code of a generic VP. The *step* variable is used to count logical steps:

```
1    int step = 0;
2    while(!termCondition) {
3      get(step, .., element-1);
4      get(step, .., element-N);
5      result = F(element-1, .., element-N);
6      put(step+1, .., result);
7      step++;
8    }
```

Figure 3.5: Pseudo-code of the generic virtual processor.

without loss of generality, we map sequence identifiers of incomplete structures into integer numbers. At each iteration of the evaluation of **F**, each VP collects the needed elements (according to the stencil, but we do not show the addressed fields), it applies **F** to the obtained elements, and it puts the result in the proper field of the next I-Structure position.

In Figure 3.6 we show an example of execution of a static stencil data parallel program: at each step, each VP accesses the elements of an array data structure at its north and south fields. Border VPs accesses only the element at south (upper VP), and at north (lower VP). The snapshot of execution is taken at the i-th application of **F**. For instance, the border $\mathbf{VP}_0$ accesses with get operations the fields with indexes 0 and 1 whose values have been computed at the previous step, i.e. $i-1$. The expression **get(i-1, 0, A[0])** means that we access the $i-1$ position of the I-Structure, at the field 0, and we store the obtained value in the first position of a local array A (A[0]). $\mathbf{VP}_0$ computes the next value for its assigned field 0, and it stores it in the next I-Structure position, i, with the operation **put(i, 0, res)**, where **res** is the value computed by applying **F**. In the figure, we also show that at step i, the I-Structure has "grown" by one position.

We formalize the description given by the model based on I-Structures, to define a consistency property exploited by the fault tolerance support. We denote with $\leftarrow\!\!-$ the causal dependency between a state element value and the other older elements needed to be passed to **F** for its evaluation. We denote with $S^i[k]$ the field k of S computed at the i-th step. We denote the $i-th$ evaluation of **F** as $F^i(\ldots)$. For the above example, for no border VPs we have that:

$$S^i[k] = F^{i-1}(S^{i-1}[k], S^{i-1}[k-1], S^{i-1}[k+1])$$

Thus,

$$S^i[k] \leftarrow\!\!- S^{i-1}[k], S^{i-1}[k-1], S^{i-1}[k+1]$$

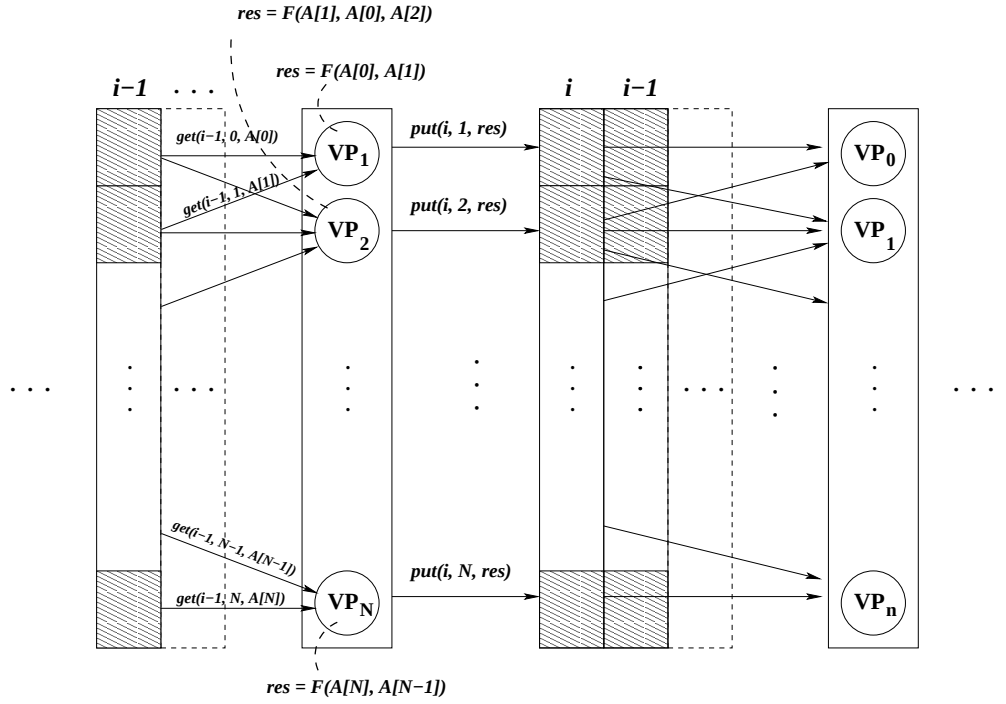


Figure 3.6: Example of execution of a data parallel model: each VP accesses the state values computed at the previous step, at north and south (not for the border VPs).

The $\leftarrow$-- definition depends on the kind of stencil. For static stencil, $\leftarrow$-- is known at compile-time, while this is not true for dynamic stencil. For fixed stencil, $\leftarrow$-- is equal at all steps, while this is not true for the variable stencil. For the special case of the *Map*, the $\leftarrow$-- relation is void. Recall that we target static stencil data parallel programs: we exploit the compile-time knowledge of $\leftarrow$-- to support fault tolerance, at the implementation level.

For fault tolerance purposes, we can define the consistency property for states by exploiting the $\leftarrow$-- definition. If this definition is given at compile-time (the static stencil case) the support can optimize the checkpointing and rollback recovery tasks to target some cost metrics. We will illustrate an example of such optimizations in Chapter 5

In the description of fault tolerance support, we will use two kinds of graphs describing (1) the state values and their relations, (2) the VP computations and their interactions. Figure 3.7 shows the dependency between state values for the above data parallel program. In the figure, we show the $i-1$-th to $i+1$-th values assumed by the fields $S[k-1], S[k], S[k+1]$. The arrow connects two functionally dependent values (see the $\leftarrow$-- example above).

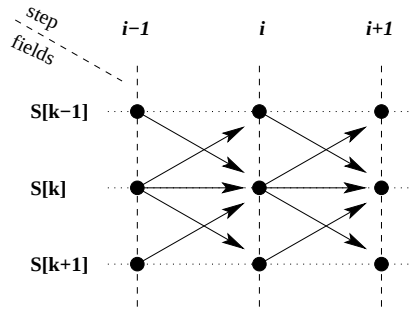
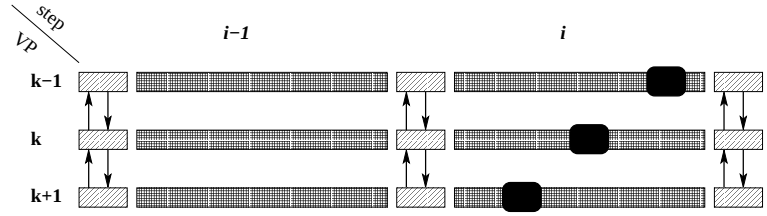
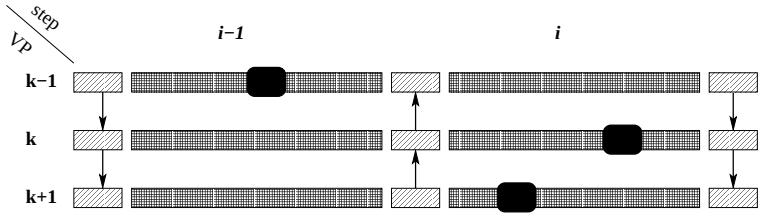


Figure 3.7: Dependency graph of elements w.r.t. computation step for the example of Figure 3.6.

Figure 3.8 shows two graphs describing the interactions between VPs, reflecting the ←-- relation). The first graph (see Figure 3.8(a)) shows the interactions between VPs for the above example (Figure 3.6), for two steps ($i-1$ and i) for three VPs ($\mathbf{VP}_{k-1}$, $\mathbf{VP}_k$, and $\mathbf{VP}_{k+1}$). VP executions are the horizontal lines, which are characterized in: (1) interaction phases, in which the VP performs the put and get on the state I-Structure, represented as short darker rectangles; (2) computation phases, in which the VP locally applies **F**, represented as long lighter rectangles. The black rectangles on computation lines represent the execution points at which VPs are. In this example, VPs are all at step i. The second graph (see Figure 3.8(b))



(a) Example of a graph representing the interactions between VPs implementing the stencil of a data parallel program.



(b) Example of a graph as 3.8(a), but for a different kind of stencil, to highlight the possible asynchrony of VPs execution.

Figure 3.8: Example of interaction graphs for data parallel programs.

refers to another stencil, for which each VP sends to its south neighbor at each odd

step, and to its north neighbor at each even step (we assume $i-1$ to be odd). With this example we highlight the asynchrony that can hold between VPs according to the asynchrony degree that can be supported by the implementation of interactions, and the stencil. Notice that $\mathbf{VP}_{k-1}$ is computing the $i-1$-th evaluation of $\mathbf{F}$, while the other two VPs are at the next step. We highlight this point to clarify the difference between this model, based on I-Structures, and the Bulk-Synchronous one [80], in which each interaction phase, performed every "super-step", forces the VPs to perform a global synchronization (i.e. a barrier).

Consistency Definition for State

We now give a definition of *consistent state* for data parallel programs which we will exploit, at lower virtualization levels, to define the checkpointing algorithms and the rollback recovery protocols.

The consistency definition is based on the ones introduced in [39, 27]. It is worth of re-writing these definition before introducing our property. The definition of [39] follows:

Definition 2 *In a consistent state there are no orphan messages, i.e. messages whose send event is not included in the sender process state, but the corresponding receiving event is included in the receiver process state.*

The definition of [27] states that

Definition 3 *For each application channel c, if m is the number of messages sent along c and m' is the number of messages received along c then a consistent state is one for which* $m = m'$.

In our model we denote with $C^i = \{S^i_k\}_{0 \leq k < N}$ the set of values assumed by the state S at the step i, where N is the number of elements composing the state. We formalize the property of consistency for data parallel states:

Definition 4 *A consistent state for a data parallel program is one for which all its element values are results of the same number of computing steps. In other words, all its element values are result of the same number of evaluations of the* $\mathbf{F}$ *function.*

A consistent state is one of those possible that can be passed during the computation. Clearly there are other states that are actually passed during the computations, but we choose the states of the definition to support fault tolerance. We can show how this model reflects, at an more abstract level, the definition given in literature [39, 27], by proving that:

Theorem 3.2.1 *There do not exists two elements* $S[k], S[j] \in C^i$ *for which* $S[k] \dashleftarrow S[j]$ *or* $S[j] \dashleftarrow S[k]$.

Proof 3.2.1 *Recall from our definition of data parallel programs (see 3.1.2) that, to apply* **F** *on a given element, we need a sub-set of other elements, whose values are taken at the previous step. In a consistent state we consider elements all taken at the same step (i.e. the values they assume at the same step). Consequently, it is not possible that* $S[k] \leftarrow\!\!- S[j]$ *or* $S[j] \leftarrow\!\!- S[k]$*, for any possible values of* k *and* j*.*

In the implementation of data parallel programs, we will show how this property is *sufficient* to prove that C^i states are consistent, according to the definition of [39, 27].

Below we introduce three fault tolerance techniques which we selected in this thesis to support data parallel programs. We also given motivations of this choice.

3.2.3 Methodology of Study of Fault Tolerance for Farm and Data Parallel

In the next two chapters we present a fault tolerance support for farm and data parallel structured computations based on checkpointing and rollback recovery. As previously stated, that these two tasks can be generally defined as:

Checkpointing An algorithm or a protocol that copies a global state of a computation (or a snapshot) onto a stable storage support, i.e. a support that survives the failures targeted from the defined fault tolerance. Usually, checkpointing is a procedure local to each module implementing the application. Checkpointing is based on a higher-level definition (w.r.t. the implementation one) of *consistency* for global system states, that we introduce in Chapters 4 and 5.

Rollback and Recovery These are protocols applied in the event of failure by a (sub)set of modules implementing the application. The rollback protocol takes a set of modules, in which some of them have failed, back to an older state. Clearly some portions of the older state can be retrieved from stable storage (at least the states of the failed modules). A rollback target, i.e. a global state to which the rollback has taken back a computation, can be computed iteratively by trying to re-build consistency. That is, in the case that the rollback target is not consistent, the rollback recursive course can re-select older states. In the case that the rollback iterates to the beginning of the computation, we have the *domino-effect*. This effect can be avoided by building consistent states, and select them during rollback. If the rollback protocol selects an inconsistent state, the recovery protocol re-builds consistency by re-executing part of the computation, or by recovering specific information from stable storage. The necessary and sufficient conditions to define and apply rollback recovery in a correct way are shown in an elegant framework in [68].

Typically checkpointing and rollback recovery techniques are based on the fail-stop failure model [76], for which, the behavior of a software module in the case of failure is to stop its execution. Some issues related to the performance of failure detection, that can induce delays in the application of the rollback recovery protocols (or that can render it impossible) have been studied in [26] (see Chapter 2). We can support checkpointing techniques by considering the high-level definition of the state, and its data-flow, and by mapping this information at the implementation level. In the next chapters we consider an implementation based on distributed processes communicating through message-passing of farm computations and data parallel programs. Methodologically, we define checkpointing strategies by exploiting different abstraction levels:

At the level of programming model We characterize the computation state as the composition of the states of the entities of the structured computation. For instance, in the farm computations we define the checkpointing of stream elements by exploiting the knowledge on their abstract properties as the static knowledge of their data types (recall that streams are typed). In data parallel computations the checkpointing strategies we introduce are based on the local state of VPs, and (possibly) on the information they exchange to implement the stencil. Also in this case the information on types are used to optimize the implementation, and to describe the overhead introduced by the checkpointing.

At the level of implementation We define checkpointing techniques for the processes and communication channels implementing the parallel computations by exploiting the high-level information. We consider several typical configurations of existing computing platforms, and fault tolerance supports (e.g. stable storage implementations). The goal is to obtain cost models featuring the communication, computation, and fault tolerance overheads, which can be instantiated to the actual configurations.

Rollback recovery protocols are defined at the process level, by exploiting the abstract properties of consistency built at the abstraction level. To simplify the discussion, and to model abstract performance models of failure recovery overhead, we define rollback recovery exploiting the high-level properties, and, in some cases, by introducing a simplified model of the structured computation.

Another approach to fault tolerance, that we do not show here, is to exploit the natural functional replication of the two structures to introduce computational replication. Computational replication is mapped on the implementation by supporting software module replication [14, 51], and/or re-scheduling techniques [18]. Anyway, we can introduce replication as extension of the introduced model and support, based on checkpointing and rollback recovery, by highlighting the I-Structure properties that can be used to define replication. We will address this issue in future work.

We now give details on the selected fault tolerance strategies for farm and data parallel structures.

Selected Fault Tolerance Technique for Farm

In the next chapter we show a fault tolerance support for farm computations based on task re-scheduling and message logging [39].

During the computation we log synchronously or asynchronously messages onto stable storage. We also keep some data structures, which reflect the functional dependencies between input and output elements, to enable correct recovery after a failure. In the case of the failure of one of the modules implementing a worker we re-schedule to another worker the task it was executing before the failure. We avoid checkpointing of workers. This is motivated by the absence of a *high-level* state in the definition of workers, at the abstract description level of farm. At the implementation level, the modules feature a local state which reflects the execution of a task. Thus, checkpointing of worker states has a sense only if considered at implementation level and it could be introduced only at that level. Anyway, in this thesis, we avoided to exploit this feature to make clear which are the fault tolerance techniques that we can define by exploiting high-level definitions. Optimizations due to implementation choices are only introduced when they come to support an high-level defined technique. For instance, this is the case of the control over the asynchrony degree of FT-Streams, see Chapter 6.

We have also avoided to exploit task duplication techniques, for which a same task is evaluated in parallel by different workers. This technique can be easily introduced in our farm model, by exploiting unique identifiers of stream elements. By exploiting task duplication, the parallelism degree of the farm depends on the degree of duplication of the evaluation of each task. In this thesis we have chosen to avoid task duplication to enable the highest possible parallelism degree. That is, we have chosen that farm computations incur in performance overheads only in the case of failure. We demand to future work the integration of task duplication techniques in our model and implementation.

The techniques we introduce (message logging and re-scheduling) can be properly used and configured to target specific optimizations. These can be done by exploiting the cost models that can be derived for each configuration to meet the application and platform needs. In Chapter 4 we only show two possible configurations, leaving to future work the complete description of all configurations and the characterization of their costs.

At the end of Chapter 4 we argue that, at the level of implementation, the fault tolerance technique we introduce can be seen as a causal message logging protocol [39]. Unlike classical protocols [3], the technique we introduce minimizes the information collected to enable recovery by exploiting the knowledge of the farm semantics and structure.

Selected Fault Tolerance Techniques for Data Parallel

Chapter 5 is devoted to the introduction and description of three kinds of checkpointing and rollback recovery protocols.

The selected checkpointing algorithms have been chosen to show how to control the trade-off between failure-free performance and recovery costs. The first protocol we describe implements coordinated checkpointing without requiring any kinds of synchronizations nor communications during its execution. This comes at the cost of a global rollback recovery in the case of failure. That is, all implementation modules must participate to it and they possibly lose performed computation steps.

With the second protocol we show how checkpointing can be extended to record on stable storage also the relative positions of implementation modules. This incurs in higher failure-free overheads (w.r.t. the previous one), but we show how it can be exploited to minimize the number of participants to rollback recovery, depending of the defined data parallel stencil.

The last protocol we introduce provides the worst failure-free performance w.r.t. the previous two, but it enables local rollback recovery. It is based on a message logging technique: in this thesis, for brevity, we have shown the support only in the case of pessimistic message logging. Further solutions, possibly based on optimistic message logging, can be thought as a composition of this solution with the previous two. We demand this study to future work.

These three protocols should be intended as a snapshot of all possible solutions which can be selected by exploiting our structure-aware methodology.

Chapter 4

Checkpointing and Rollback Recovery for Farm

In Chapter 3 we have described the abstract farm model that allows us to define a relation between input elements, workers and output elements. We have also seen that tasks (i.e. function evaluations on different input elements) are independent w.r.t. each other. Thus, consistency can be derived by considering independently all the elements processed in the farm when they are on the interaction points (i.e. on put and get). That is, we logically characterize elements when they are passed to put and get operations. We consider "inconsistent" an element while it is processed in a worker (i.e. when the worker is applying it **F**). This choice sets the minimum scope of recovery to the average time needed to perform a task, for the fault tolerance support we present.

Farm computations can be implemented according to several strategies. In Figure 4.1 we show two possible strategies:

Master-Slave Strategy We can implement the farm by mapping stage 1, and 3 on the same software module, the *master* (**m**). The *master* is responsible for: (1) collecting input elements; (2) selecting a worker for scheduling; (3) delivering the input element to the selected worker; (4) selecting a worker to collect a result; (5) collecting a result from the selected worker; (6) delivering the collected result to the output stream. In a concurrent language-like formalism, we could encapsulate the two tasks as different guards of an alternative command. Stage 2 is replicated on the *slave* (**s**) modules.

Emitter-Worker-Collector (E-W-C) We map each stage in a different software module: an emitter (**E**) on which we map the stage 1; a set of workers (**W**) on which we map the stage 2; and a collector (**C**) on which we map the last stage 3. This implementation targets the maximization of the input/output stream bandwidth, when it is possible to decouple them.

To define checkpointing and rollback recovery we exploit the **E-W-C** implementation strategy, because it allow us to study separately the fault tolerance issues for

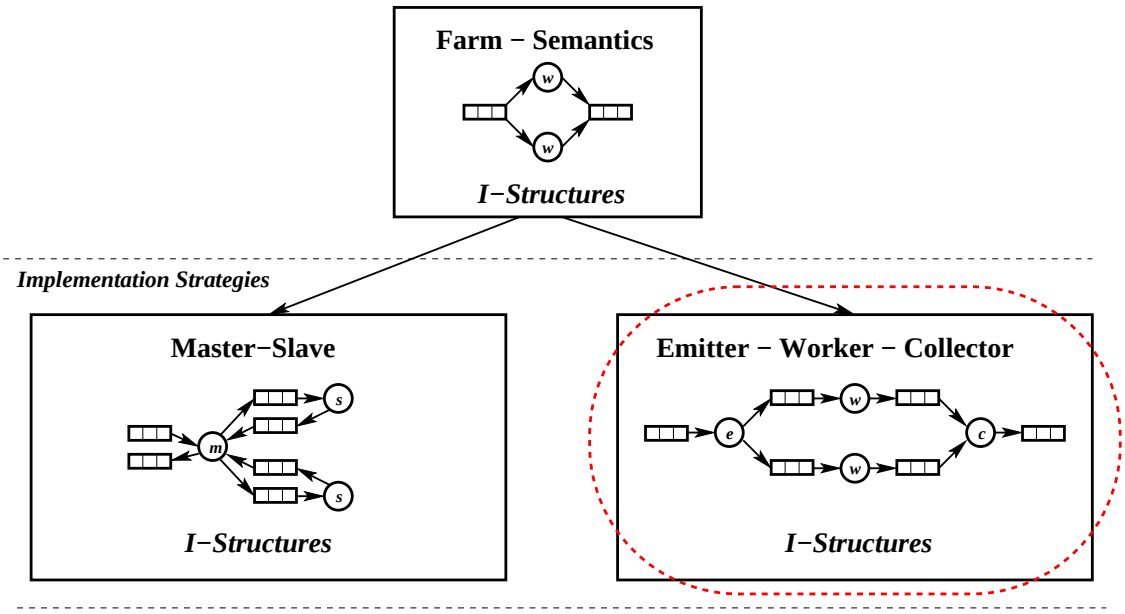


Figure 4.1: Implementation strategies of the farm: the first one (left) is based on a master-slave strategy, in which the master (**m**) is responsible for performing stages 1 and 3, while stage 2 is replicated on slaves (**s**). The second strategy decouples stages 1 and 2 in two implementation modules, the emitter **e**, and the collector **c**.

the first and last stages of the farm. We exploit the **master-slave** strategy to study the impact of slave failures on the overall performance of a farm computation (see Chapter 7).

We discuss the concept of nondeterminism for farm computations. As stated in Chapter 3, in the abstract farm model the generation of positions to access the input and output I-Structures is encapsulated in the *put* and *get* semantics. This generation should reflect the asynchronous and nondeterministic computation of workers because we are assuming that their executions are totally asynchronous between each other. At the next level of abstraction we encapsulate the control of nondeterminism in the emitter and collector (or master) behaviors. As a consequence we can state that:

- Workers computation is composed of deterministic steps each corresponding to an evaluation of **F**. Thus, we assume the Piecewise Determinism (PWD in short) property [39] for worker computations: each step is completely characterized, from the determinism viewpoint, by its input value (the one obtained from the input stream).

- Also the emitter and the collector can be modeled according to the PWD property: given the scheduling decision and the input values, the nondeterministic

choice is controlled according to the semantic of an alternative command of a concurrent language.

As a consequence we can exploit message logging techniques [79, 4, 3] in a sound fashion.

4.1 Farm E-W-C Implementation Strategy

We describe the farm implementation strategy based on the **Emitter-Worker-Collector** scheme, in two versions. The first one features a round-robin strategy for the scheduling of input elements to workers. The second one features an on-demand scheduling strategy. In more general terms, the first version models the case in which the emitter exploits just local knowledge to decide which is the next worker to schedule. The second one also exploits information provided by workers. The strategies describe in which way we implement farms, but they are still abstract enough to target different implementation environments (e.g. distributed processes, or threads on a same node). Also at this level of description, we exploit Incomplete Structures: this allows us to highlight the relations between module computations and their interactions.

4.1.1 Round-Robin Scheduling for Farm

In Figure 4.2 we show a graphical representation of the first farm implementation model based on incomplete structures. Suppose, in this implementation, that the emitter schedules tasks to workers exploiting just local knowledge. A typical policy in this case is the round-robin. Along with input and output streams, we define two streams for each worker, one for the emitter-worker interactions, the other for the worker-collector ones. The semantics of the emitter process in the case of round-

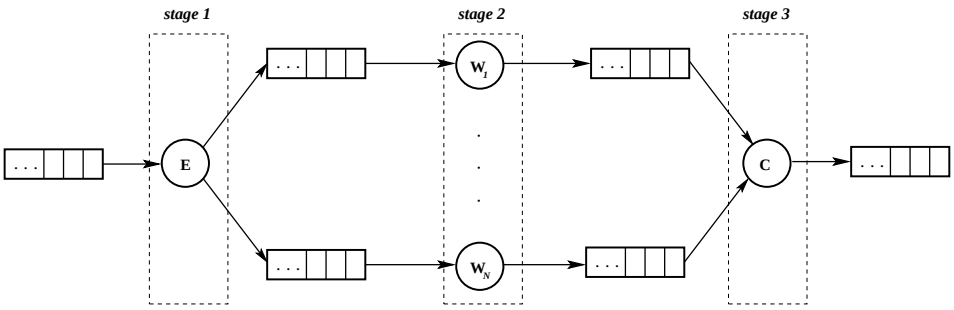


Figure 4.2: E-W-C implementation strategy for farm computations. The emitter exploits just local knowledge to implement the scheduling strategy.

robin scheduling policy is described in Figure 4.3. The emitter receives all the

```
1  int nextWorker = 0;
2  int position = 0;
3  while(true) {
4    in-stream.get(position, element);
5    worker-in-streams[nextWorker].I-St.put(worker-in-streams[
       nextWorker].nextPosition++,element);
6    nextWorker = nextWorker + 1 % N;
7    position++;
8  }
```

Figure 4.3: Pseudo-code of the emitter module, in the case of farm with round-robin scheduling policy. N is the number of workers.

elements passed on the input stream, generating incremental positions in the input I-Structure. The round-robin scheduling is implemented with an integer variable, incremented at each scheduling action. Consider the abstract farm model presented in Section 3.2.1. We have implemented the decision process to access the input I-Structure, made in a distributed fashion on workers, as part of the emitter behavior. The **worker-in-stream** array contains an item for each worker. Each item has two fields: (a) a reference to the worker input stream **I-St**; (b) an integer variable **nextPosition** that contains the next I-Structure position to be accessed for the corresponding I-Structure. At each loop, the **nextWorker** variable is incremented for the next scheduling.

Worker modules do not need to make any special decision on the positions to be addressed in the input streams, but they just receive all the input elements scheduled by the emitter to them. This is implemented by making workers access their input stream with incremental positions. The same behavior is exploited to access the I-Structure modeling their output streams. The worker pseudo-code is represented in Figure 4.4. In the pseudo-code, the names **in-stream** and **out-stream** are unique

```
1  int position = 0;
2  while(true) {
3    in-stream.get(position, element)
4    result = F(element)
5    out-stream.put(position, result)
6    position++;
7  }
```

Figure 4.4: Pseudo-code of the worker module, in the case of farm with round-robin scheduling policy.

for each worker.

The collector module, whose behavior is given as pseudo-code in Figure 4.5, is

responsible for: (a) deciding from which worker to collect the result, (b) to collect a result, (c) and to deliver it to the output stream. In this case, a typical policy for

```
1  int position = 0;
2  while(true) {
3    worker = decide(worker-list);
4    worker-out-streams[worker].I-st.get(workers-out-streams[
         worker].nextPosition++, result);
5    put(position++, result);
6  }
```

Figure 4.5: Pseudo-code of the collector module, in the case of farm with round-robin scheduling policy, and FIFO collection strategy.

this decision process is the First In First Out (FIFO), which can be implemented by exploiting just local information. We encapsulate the collection policy in the *decide* function, which returns a worker index. This is used to obtain a reference to the output stream of the selected worker, from a local array of I-Structures (the *workers-out-stream* variable). Each item in the array of worker output streams includes two fields: (a) a reference **I-st** to the corresponding I-Structure; (b) the next position on that I-Structure to be accessed. The collector properly increments the single position fields when it accesses the I-Structures modeling the worker output streams. The collected results are put in the output stream. As above, the generation of positions on such a stream is implemented by incrementing at each put operation a local integer variable (*position*).

4.1.2 On-Demand Scheduling for Farm

We introduce a second implementation to express scheduling policies of input elements to workers, based on information related to worker execution. We consider the *on-demand* policy as a typical example of this kind of scheduling: each worker notifies the emitter that it is ready to perform a new task. Unlike the previous model, we add a stream for each worker, on which it makes such notifications. Also these streams, which we call *free input/output streams*, are modeled exploiting Incomplete Structures. However, we will not exploit the properties of I-Structures to model the fault tolerance mechanisms for this kind of streams. The motivation to use the same abstraction is to obtain an uniform vision of this level of description. Further abstraction defined "ad-hoc" for these kinds of interactions could have broken the model uniformity.

As for the first implementation we implement the scheduling logic on the emitter. In Figure 4.6 we show a graphical representation of this implementation.

We do not give a detailed implementation of this version of the farm, but we only describe the behavior of the modules:

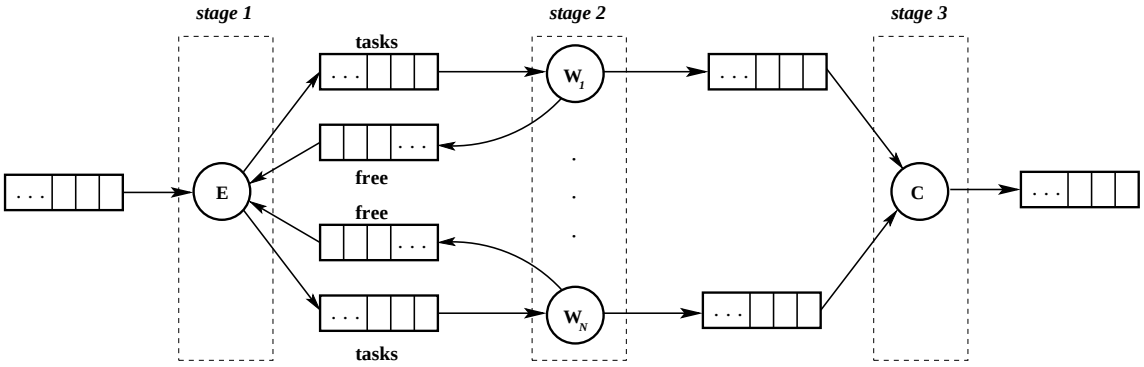


Figure 4.6: E-W-C implementation strategy for farm computations. The emitter exploits local knowledge and worker information (passed on **free** streams) to implement the scheduling strategy.

Emitter It *alternatively* waits for a new element on the input stream by performing a get on the next position *and* for a worker to ask for a new task on the free input streams. Notice that the worker notifications are consumed only if the emitter can assign a task. *This behavior can be expressed by means of an alternative command, to avoid the emitter to block waiting for a worker notification message* (see [53] for a description of alternative commands).

Worker The behavior of a worker is similar to the previous case (round-robin implementation), but, after having performed a task, it also puts a "free bit" on the next free position of its output stream towards the emitter.

Collector It performs the same behavior as the previous implementation.

Notice that, unlike input and output elements, notifications from workers to the emitter do not need to be distinguished between each other. This is also true in the case in which we assume that modules can fail and restart. Below, we will see that the information on worker availability is automatically re-obtained, after the emitter failure and restart, as a re-notification performed by free workers.

4.2 Farm E-W-C Implementations

We characterize two environments in which we map the **E-W-C** farm implementation strategy. Each execution environment is characterized by the way in which interactions between modules are expressed. In Figure 4.7 we show the two possible implementations. The first one is based on a shared memory environment in which modules are implemented as threads. In the other one we map modules in distributed processes, and interactions between modules in communication channels. We give details of the implementations:

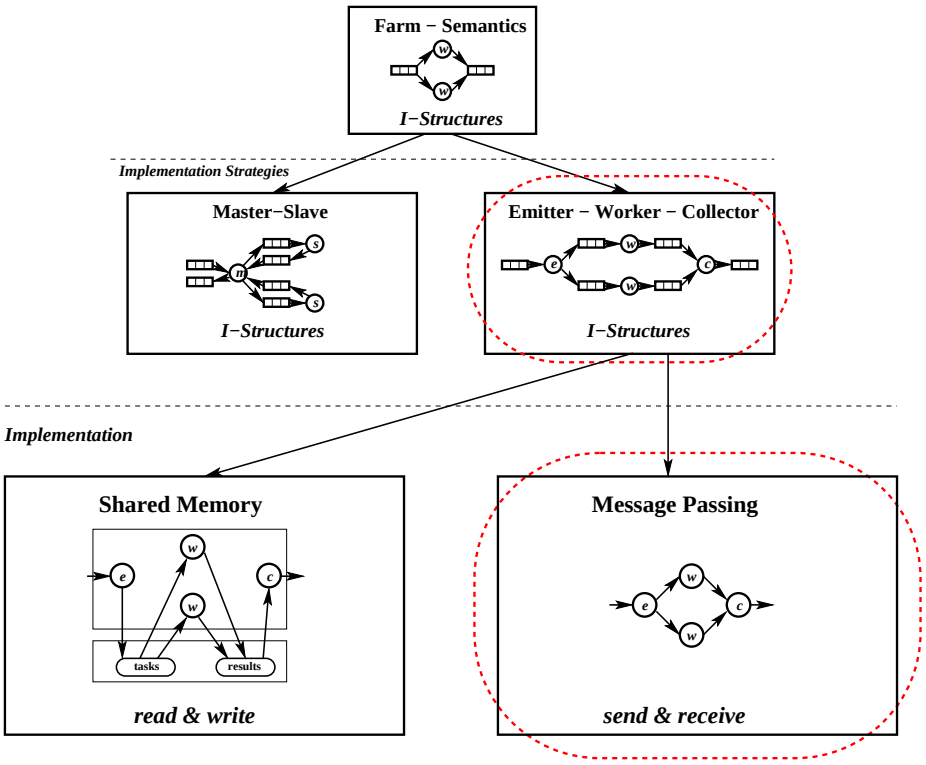


Figure 4.7: Two implementations, based on shared memory and message passing, of the Farm **E-W-C** strategy.

Shared Memory In this case, the emitter, the workers, and the collector modules interact through accesses to variables mapped in a shared memory. The modules can be both threads or processes, depending on the support provided by the exploited platform. We ensure mutual exclusion for the accesses on shared memory, which are denoted with *read* and *write*. The emitter and the workers interact through an input element queue (the **tasks**) mapped onto the shared memory. The workers and the collector interact through a queue of result elements (the **results**). We do not show how the emitter and the collector interact through the input and output streams, respectively. This could be done either through the shared memory, or other supports (e.g. communication channels, or memory-mapped I/O).

Message Passing Modules are mapped in processes, and their interactions are implemented as *send* and *receive* operations on communication channels. Thus, input elements and output results are passed as messages in communication channels.

For fault tolerance purposes, the first model requires the shared memory to implement some kind of stable storage support, or it must be supported by a further

level of stable memory. In this setting, it is possible to move the checkpointing and rollback recovery logic to the level of the memory hierarchy. This has been proved to be fruitful for general parallel computations (e.g. see [8]). However, in this thesis we target the second model to exploit the structure compile-time information: we derive stable storage accesses from I-Structure accesses, which are performed according to the parallel implementation of the farm. This allows us to fully exploit the knowledge of the computation structure in the design and implementation of the stable storage. Consequently we can introduce structure-driven optimizations. This point is also claimed for general parallel computations (e.g. see [28]). In Section 4.4, we detail the implementation of farm computations as a set of distributed processes interacting through message passing.

4.3 Fault Tolerance at Levels of Abstraction

We consider the three description levels we introduced above to discuss fault tolerance issues for farm computations. This study is based on the following steps:

1. Definition of a failure model on the abstract farm model, i.e. semantics of failures. This definition includes also the concept of detectability of faults.

2. Mapping of the failure model in the implementation strategy, and in the implementation of farms as distributed programs.

3. Definition of fault tolerance techniques. As we introduced in Chapter 1, in this thesis we target checkpointing and rollback recovery techniques.

4. Implementation of the fault tolerance techniques for the message-passing **E-W-C** farm. This point is discussed in the next section.

4.3.1 Failure Model and Fault Tolerance Logics for the Abstract Model

We target the *fail-stop* failure model [76]: at any level of abstraction, whenever a module fails, it stops executing its tasks. At the level of the abstract farm model, we can distinguish the following cases (we also show the mapping in the implementation targeted strategy):

1. A worker fails by stopping its execution during the evaluation of the function **F**. In the implementation strategies, this corresponds to the failure of a module that performs the tasks (i.e. the worker).

2. A worker fails during the execution of a *put* or *get* operation. We assume that failures are atomic w.r.t. put and get operations. At the level of implementation strategy, this corresponds either to the failure of a put or get on

an I-Structure linking two modules, or the failure of the module on which we map the task 1 and 3 (the master, or the emitter and the collector).

3. Both points 1 and 2 hold.

As far as concerns the detectability issue, in the event of a failure all entities participating in the computation are assumed to be informed of this event after an unbounded but finite amount of time [26]. This assumption does force all entities to participate in the rollback recovery protocol. We do not enforce any constraints on the frequency and concurrency of failures and we will consider both transient and permanent failures. In this failure model we do not include the possibility for modules to restart. However, in the general model we assume that new workers can join the computation.

At the level of the abstract model, concerning the failures of type 1, the computation performed by the failed worker is lost. We assume that partial computations are not saved and the evaluation must be re-executed from its very beginning. This choice has been motivated in Chapter 1. As a consequence of this choice, another worker has to obtain the input element of the lost computation and re-evaluate the function on it. We can model this by making the next available worker access the position of the input I-Structure of the element whose computation has been lost. This implements task re-scheduling for farm, as in [2, 18].

Thus, the *compute-next-position* function is designed according to this requirement, i.e. to take into account failures. In more general terms, this means that:

- The actions to tolerate a worker failure involve an adaptation of the task scheduling strategy.

- The adaptation is possible because we assume that input tasks are not lost.

We ensure the latter point in the implementation, by exploiting the information derived from the I-Structure model.

If a *get* or *put* operation returns an error (failures falling in the case 2) neither computation nor tasks are lost. First consider the failure of the operations on the *whole* set of workers: depending on the duration of the failure, either the access from external entities to the results produced by the farm, or the access to input elements by the workers could be impossible. Both such situations can be associated to the failure of the whole farm program.

An interesting case is to consider the failures of *put* and *get* operations on a *single* worker. If a worker cannot execute a put operation anymore (permanent failure), the result of its computation is lost, and another worker should re-execute the evaluation of the function on the same input element. The actions to tolerate this kind of situations (i.e. the recovery procedure) are equal to those provided in the case 1. If a worker cannot execute anymore a get operation, it cannot obtain input elements from the input stream, and it is excluded from the computation.

In both cases, a failure corresponds to a reduction of the the parallelism degree. Otherwise, if the duration of the failure is limited in time (transient), when the failure is recovered, the worker can return the result of a computation or obtain a new input element. In this case, the duration of the failure affects the performance of the farm, by lowering the amount of parallelism for the whole duration of the failure. The recovery procedure can be based on some timing constraint (on a given amount of time K):

- if the failure lasts more than K seconds, the worker is considered to be failed, and its task is re-scheduled to another worker. Eventually, when the worker re-joins the computation, it discards its partial computation and re-starts the execution of a new task.

- Otherwise, if the failure lasts less than K seconds, the result is obtained from the same worker.

Finally, case 3 allows a worker to fail both during the function F evaluation or during the *put* and *get* operations. The consequences of these failures and the actions introduced to tolerate them can be provided independently for the first two cases, and combined together. This simply means that the failure handling is composed in the same worker program.

We exploit these fault recovery logics at the lower levels of implementation.

4.3.2 Fault Tolerance for the Farm E-W-C Implementation Strategy

In this section, we define rollback recovery for the **E-W-C** implementation strategy. Our goal is to define rollback recovery protocols for each module composing the computation, by exploiting the I-Structure properties. As stated above, the failure model we consider is a possible mapping from the abstract one:

- Failures during the evaluation of **F** are mapped onto failures of worker modules.

- Failures during the get operations on the (abstract) workers are mapped onto the failure of the emitter module.

- Failures during the put operations on the (abstract) workers are mapped onto failures of the collector modules.

- The last case, as composition of the first two abstract cases, is a composition of the three above cases.

We provide a fault tolerance support for the two versions of the farm computations, with round-robin scheduling policy, and with the on-demand one. In the presented

support, we exploit the I-Structure properties: for instance, we re-access input elements without worrying about where they are stored. This will be implemented at the lower layer by proper mechanisms, as described in Section 4.4.

We describe a fault recovery protocol for both scheduling strategies. When needed we characterize the discussion w.r.t. each strategy.

Supporting Data Structures

We introduce three data structures to model the transition of elements in the farm modules, and their relationships. These data structures reflect the functional dependencies of farm described in Chapter 3. Next we state which is the minimal information included in such data structures required to support the failure of the modules implementing the farm.

We denote with $\mathcal{T}_i$ the data structure that relates the position of the worker W_i input stream to the global input stream positions. We denote the whole set of these data structures with $\mathcal{T}_0, \ldots, \mathcal{T}_{N-1}$, where N is the number of workers. For each worker, its $\mathcal{T}$ data structure can be characterized in two sub-sets: (a) the input positions of the set of elements whose results have been passed to the collector (for W_i we denote this set with $\mathcal{T}_i^{out}$); (b) the input position of the set of elements not yet evaluated, plus the element being evaluated (for W_i we denote this set with $\mathcal{T}_i^{in}$). Similarly, we can define a data structure modeling the relation between worker output stream positions (towards the collector) and global output stream positions. We denote such data structures as: $\mathcal{R}_0, \ldots, \mathcal{R}_{N-1}$. Finally, we define the $\mathcal{O}$ data structure that models the relation between output stream positions and input stream positions. Notice that we can exploit $\mathcal{T}$ and $\mathcal{R}$ structures to obtain the mapping between output stream positions, and input stream ones. This is easy to see by pointing out that the worker behavior is locally sequential: the result of the i-th element obtained (with a get operation) from the worker input stream is put on the i-th position of the worker output stream position. As a consequence the $\mathcal{O}$ structure is redundant w.r.t. the other: it can be defined by merging the $\mathcal{T}$ and $\mathcal{R}$ structures.

All the data structures described above are built dynamically during the computation by the put and get operations. At this level of abstraction we do not enforce a mapping of them on the implementation modules. In this way, at this level of description the failure of a modules does not cause the loss of information.

Fault Recovery Protocol

We characterize which is the information required for restoring the computation in different failure scenarios, by exploiting the data structures introduced above. In the discussion, we will assume that fault tolerance for workers is implemented by a proper re-scheduling of the input elements to survived workers. This behavior reflects the failure model, in which:

- Failed workers are possibly replaced by new workers.
- Failed emitter and collector are replaced by new modules with the same behavior.

The fault recovery protocol is similar for the cases of round-robin and on-demand scheduling strategies, except for the fact that the $\mathcal{T}^{in}$ structures in the latter case include at most one element.

- Suppose that the emitter fails in isolation. Looking at its pseudo-code in Figure 4.3, we can see that it can process at most one element at time. The remaining ones have been copied on the worker input streams, which can include multiple elements to be computed. Thus, at most one element from the input stream is lost. If we lose such an element, we re-obtain it from the input I-Structure. This translates to accessing the next position on the input I-Structure not yet scheduled to any workers. The next position on the input I-Structure to be accessed is computed as $\max_i\{\mathcal{T}^{in}\}_{i\in[0,\ldots,N-1]} + 1$, i.e. we take the maximum value in the $\mathcal{T}$ data structures and we sum 1 to it. The emitter also needs to obtain the next positions to be accessed on all I-Structures towards the workers. Also this information can be retrieved from the $\mathcal{T}$ structures. For the round-robin scheduling, if we do not save the last scheduled worker on a stable storage support, we have to set it to a default value (e.g. 0) at each restart. For this reason, there are failure patterns for which we can obtain a sequential behavior for the farm, by assigning all tasks to the same worker.

- If one or more workers fail: the emitter has to re-obtain and re-schedule all the elements in the $\mathcal{T}^{in}$ sets of all failed workers. If we can access such structures, the emitter directly exploits them to re-access the input stream, and it re-schedules the tasks to survived workers. Otherwise, we have to further distinguish two cases: (a) if all workers failed, it is sufficient to access either the $\mathcal{O}$ structure, or the $\mathcal{T}$ and $\mathcal{R}$ ones (as stated above they include equivalent information). (b) if a subset of the workers failed, we have to avoid the re-scheduling of tasks previously scheduled to survived workers. Unlike the point (a) the $\mathcal{O}$ structure is not sufficient to know which is the task that is to be re-obtained. We also need the $\mathcal{T}$ structures of survived workers, to understand which tasks are to be re-assigned.

- If the collector fails in isolation: the restarted collector has to re-obtain the next position to be accessed on the output stream, and the next positions to be accessed on each worker output stream. For this task, the collector needs both the $\mathcal{O}$ and $\mathcal{R}$ data structures. Operationally: it recovers the identifier of the last position (say x) accessed on the output stream. Next, it re-starts sending out elements whose identifiers are greater than x on each data structure $\mathcal{R}$.

For this behavior to be sound w.r.t. the concurrency semantic of farm, we use the property for which the results produced by a same worker are ordered on the output stream position (i.e. the $\mathcal{R}$ structures of a worker are ordered sets).

- If the emitter and one or more workers fail concurrently: in this case we lose the elements scheduled to failed workers, and (possibly) the one on the emitter. The lost elements can be re-obtained by the new restarted emitter in the same way as in the case of worker(s) failing in isolation. The position of the next element to be accessed is computed as in the case of the emitter failing in isolation.

- If the collector and one or more workers fail concurrently: as in the case of the collector failing in isolation we possibly lose a result. The collector needs also: (1) to re-obtain the last accessed position on the farm output stream; (2) to re-obtain the last accessed positions on the survived worker output streams. This is supported as in the case of collector failing in isolation. For modeling purposes, we assume the most general case, in which we lose also the result elements on the I-Structures between the failed workers, and the collector. Such results are those in the $\mathcal{T}^{out}$ data structures of the failed workers, and must be re-computed. We implement the re-scheduling of such elements from the emitter side, like in the case of workers failing in isolation.

- If the emitter, the collector and one or more workers fail concurrently: we build the recovery protocol by simply merging up the above cases in a modular fashion.

4.4 Fault Tolerance for Message Passing Farms

We show an implementation of the farm **E-W-C** strategy based on distributed processes interacting through message-passing. This level features the following abstractions:

Processes are the unit of execution. We map the emitter, worker and collector modules, at the abstraction level, in corresponding processes. As a working assumption processes are executed on different computational nodes. Each computational node has a main volatile memory, and a local secondary storage (e.g. a hard-disk) that is resilient to faults. We do not distinguish between the failure of a computational node, and the process it is executing. When a node fails, the volatile memory is lost, while the information on the secondary storage can be eventually accessed, when and if the node restarts. We assume the emitter and the collector process to be restarted on the same computational node. Workers are not restarted, but we admit new workers to join the computation.

Communication Channels implement the streams. At this level we will conversely use the terms streams and communication channels. A channel links two processes executed on different nodes, it is either synchronous or asynchronous, and can be supported by message logging. The communication primitives are denoted with **send(ch, msg)**, and **receive(ch,msg)**, whose meaning is obvious. The asynchrony of channels is limited by a known value **K**, that is configured at compile-time, and can possibly be modified at runtime in an atomic fashion, to target optimizations. The meaning of asynchrony **K** is that the message queue between the communicating parties can contain up to **K** messages. If a sender fills up the queue it stops until a receive is performed. For optimization purposes there is a single channel data structure including the message queue mapped onto the receiver volatile memory. This implements the 0-copy technique [73]. We name *Fault-Tolerant Streams* (or *FT-Streams*) the streams supported by message logging (both synchronous or asynchronous [4]).

Chapter 6 describes a full implementation of FT-Streams, which we exploit in the implementations of this chapter, whenever message logging is required. Here we give a short description of FT-Streams to allow the reader to understand the hypotheses on which the fault tolerance support is based.

Message logging on communication channels includes the copy of all messages passed on streams to an abstract stable storage [4]. The stable storage is implemented by a process that receives all messages from the sender (exactly as the receiver does), and that copies them on a secondary storage. There is one such process for each communication channel supported by message logging. We call it **message-logging process** (M-L process or, in short, M-L), it can be executed on both the node executing the sender of the stream, on the one executing the receiver, or on an external node. Each choice corresponds to a different implementation strategy of the stable storage featuring a different support to higher levels, and different costs. This issue is discussed in Chapter 6. The logging of messages can be done synchronously or asynchronously, and is performed as part of a send operation. The synchrony semantics is the following: we ensure that, when the receiver process performs a receive operation, the corresponding message has previously been copied to stable storage (see Chapter 6). The communication between the sender and the M-L processes is made through a communication channel, which is logically associated with the one linking the sender to the receiver. Both channels contains the same messages during the whole computation, but not at the same time. We do not enforce that the receiver process is synchronized with the corresponding M-L. In the case of asynchronous communication channel (asynchronous FT-Stream), also the one connecting a sender to the M-L process features an asynchrony degree that can be possibly different from the application one.

We show the relationships between I-Structures and communication channels:

- Each message passed on a communication channel is uniquely assigned an

identifier, which abstractly corresponds to the *position* on the I-Structure. The sequence identifiers of the last logged messages (before the failure) are provided to failed and restarted (communicating party) processes by the FT-Stream support. Such identifiers are used according to the rollback recovery behavior described in Section 4.3.2, for the $\mathcal{T}$, $\mathcal{R}$ and $\mathcal{O}$ data structures.

- Messages on a channel queue cannot be overwritten until consumed by the receiver. For later re-accesses (i.e. after a message has been received), messages are retrieved from either the sender or the stable storage. This implements the *write-once* property of the put operation.

- The receive operation blocks in the case in which no message is ready on the message queue. This implements the *blocking* property for the get operation.

We exploit sequence identifiers to support rollback recovery protocols. In the implementation based on FT-Streams sequence identifiers are copied into the channel data structures. In the case of failure, these are used to support message re-sending and recovering from stable storage.

According to the description given in Chapter 6 we choose to map the M-L process in the receiver node: the M-L process acts as a hard disk manager on the receiver. The local send support, on the receiver node, copies the message on the channel queues between the sender and the receiver, and the sender and the M-L process.

We now describe the implementation of the supporting data structures. Based on this implementation, we describe the fault recovery protocols for the round robin and on-demand cases, respectively supported by synchronous and asynchronous FT-Streams. There is not any constraints behind this choice: these examples have been chosen to cover a subset of the possible configurations. Another choice could have been to support the round robin version with asynchronous FT-Streams and the on-demand one with synchronous FT-Stream, without any changes in the general idea behind our solution.

4.4.1 Implementation of Supporting Data Structures

We show how we implement the $\mathcal{T}$, $\mathcal{R}$ and $\mathcal{O}$ data structures. As describe above, there is a $\mathcal{T}$ data structure for each worker. For a given worker, this structure relates sequence identifiers of the worker input streams with sequence identifiers of the farm input stream. Messages passing on the FT-Stream abstractions are piggybacked with their sequence identifier on that stream. Thus, we can build $\mathcal{T}$ structures according to the following rules:

- On the emitter, whenever an element is received and assigned to a worker, we can build the relation between the input (from the external world) and output (to a worker) identifiers.

- On the workers we have to piggyback the sequence identifier of the input stream also to the messages between the emitter and the workers. Whenever a worker receives an element it builds the relation.

The $\mathcal{R}$ data structure maps sequence identifiers of the worker output streams to sequence identifiers of the farm output stream. $\mathcal{R}$ structures can be dynamically built:

- By the collector, whenever it receives a result and it sends it to the output channel.
- By the workers, if the collector sends back to each worker the sequence identifiers assigned to each result.

Finally, to build the $\mathcal{O}$ data structure on the collector we have to piggyback each result with the input stream sequence identifier that generated it. This can be done on the workers. The information in the $\mathcal{O}$ data structure can also be propagated from the collector to the emitter.

4.4.2 Synchronous Logging on Input/Output Streams

In this subsection we choose a specific configuration for the logging techniques applied to farm computations: we implement the input and output streams of the farm as FT-Streams; we implement the streams between the emitter and workers and the workers and collector with communication channels without message logging. This configuration is based on the re-scheduling of lost element, in the case of worker failure. This choice allow us to discuss how the supporting data structures described above are exploited in the case of failure of processes *inside* the farm. That is, we can avoid to describe the support to the failure of outer modules. For the same purpose, FT-Streams are supported by synchronous message logging, and messages are stored on the receiver secondary storage. Figure 4.8 shows the architecture of the sender and receiver nodes of an FT-Stream. We assume that:

- All communications are partially implemented by two processes, one for each party, that we name **KP**. The **KPs** can be executed on a specialized support for communications (as in Figure 4.8). We conversely denote with **KP** both the process, and the specialized support executing it.
- A **KP** process interacts with the local Network Interface Card (NIC), implementing the low level networking protocol. The **KP** on the receiver node also interacts with a local secondary storage manager, which copies the received messages on the local secondary storage.
- We exploit two data structures for each communication: one for the communication between the sender (S) and the receiver (R); the other for the

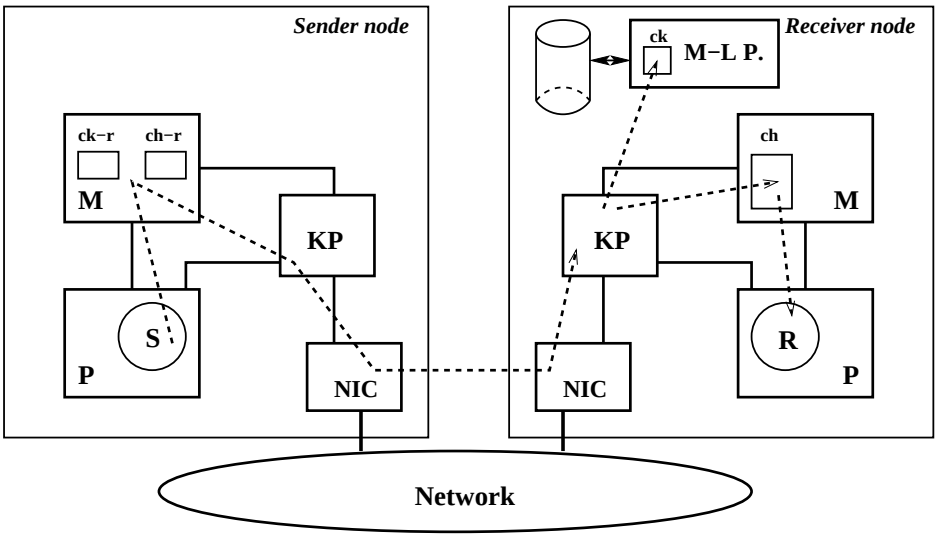


Figure 4.8: Architecture of the nodes executing the sender and the receiver processes in the case of communication through an FT-Stream.

communication between the sender, and the message logger (M-L). Both data structures are associated to the same FT-Stream, and are partially included in both the sender (*ch-r*, and *ck-r*, where "r" stands for "remote") and the receiver (*ch*, and *ck*). The *ch* and *ck* ones are the only ones that include the message queue (0-copy optimization), while the other ones are used to synchronize the sender with the local **KP**.

- The **KP** on the receiver node is responsible for copying each message on both the *ch* and *ck* channels, and for synchronizing the sender with the receiver process and the message logger.

- As describe above, in the case of failure, the emitter and the collector processes are restarted on the same computational node. Workers can be restarted on different computational node, obtained by some resource management subsystem. When the process is restarted we assume that the communication subsystem properly re-generates the communication channels it needs.

As we exploit synchronous message logging at the termination of a send operation messages are copied to the secondary storage of the receiver node. In the case of failure and restart of the receiver, the local **KP** replies the messages from the stable storage. The sender is not involved in the rollback recovery. The computations of the two parties are characterized in logical steps (or computational steps), each one initiated with a communication operation and terminated with the following one. We exploit computational steps to implement the rollback recovery of the sender processes. Messages are assigned the same sequence number as the computational

step that the process is performing. In the case of failure of the sender its local **KP** retrieves the last sequence identifier copied on the channel queue. The sender avoids all the send operations until it reaches the same sequence identifiers, plus one, provided by its **KP**. Operationally, the send operation requires a channel reference, a message reference, and a sequence identifier. In the case of rollback recovery if the sequence identifier is less or equal to the re-set one, the message is not sent but simply discarded. We assume that processes on a communication take independently local checkpoints, which are labeled with the computational step identifier. In the case of failure and restart a process recovers the local state and its sequence identifier. The latter information is used to perform the next communication. We will see that the emitter and the collector do not have a local state to checkpoint. They exploit the sequence identifier provided by their local **KP** for the next communications, both with the supporting data structures.

Implementation of the Fault Recovery Protocol

We characterize the rollback recovery protocols of the emitter and the collector w.r.t. the failure pattern:

Emitter failing in isolation We have synchronously saved on stable storage the messages passed to the emitter, both with their sequence identifiers. The identifier of the message at the top of the receiver message queue corresponds to $\max_i \mathcal{T}^{in} + 1$. If we have saved it on the stable storage, the local **KP** can provide this value to the emitter. Otherwise we have to check for $\mathcal{T}$ structures: if we assume that the emitter does not checkpoint them on stable storage, they can be re-obtained from the workers.

Workers failing in isolation The tasks assigned to failed workers are lost and the emitter must re-obtain them from stable storage and re-assign them to survived workers. As we assume that communication queues are placed on the receiver side, the results in the channels between the failed workers and the collector are not lost.

Collector failing in isolation We lose the result that is possibly in elaboration on the collector, and all the results that were placed in the channel queues between the workers and the collector. The collector can re-obtain the last sequence identifier it used on the output stream from the channel itself: it corresponds to the sequence identifier of the bottom message on the receiver queue. Lost results must be re-computed: we implement it on the emitter by re-scheduling all the elements in the $\mathcal{T}$ data structures, not yet included in the $\mathcal{R}$ data structures. We can also exploit the $\mathcal{O}$ structure if: (a) we keep it replicated on workers, or (b) the collector saves it on stable storage.

Concurrent failure of the emitter and one or more workers In this case, we lose: (a) the element in processing on the emitter; (b) the elements scheduled

to the failed workers, and the elements they were evaluating before the failure. The results, passed from the workers to the collector, are stored in the channel data structure on the main collector memory. If the emitter saves the $\mathcal{T}$ structures on stable storage, it directly computes those elements which are to be re-obtained and it assigns them to survived workers. Otherwise, we lose the $\mathcal{T}$ structures of the failed workers. Thus, the emitter also needs the $\mathcal{O}$ data structure from the collector. The elements can be directly recovered from the local stable storage.

Concurrent failure of the collector and one or more workers In this case we lose the input elements of failed workers and the results of all workers placed in the channels between the workers and the collector. The emitter needs to re-schedule all such tasks: for this purpose it needs both the $\mathcal{T}$ structures of survived workers and the $\mathcal{O}$ structure. We recall that the structure could have been propagated from the collector to the emitter. Otherwise, the collector needs to save it on stable storage and to retrieve it to allow the emitter to implement the recovery protocol.

Concurrent failure of the emitter, one or more workers, and the collector This case is similar to the previous one, except that we cannot assume that the $\mathcal{O}$ data structure can be retrieved form the emitter volatile memory. Thus, it requires the collector to save it on stable storage. If we assume that at least one process implementing the farm does not fail concurrently with the other ones, we can survive this kind of failures by propagating the $\mathcal{O}$ data structure on all processes. The trade-off in this choice is between the time needed to access stable storage and the time needed to propagate the information on all processes.

4.4.3 Asynchronous Logging on Input/Output Streams

We consider the same architecture as above, depicted in Figure 4.8, where the M-L process and the channel queues are implemented on the receiver node. Unlike the previous case, messages are logged asynchronously to stable storage. In the case of failure of the receiver process, some elements are copied onto the stable storage (i.e. the receiver secondary storage), some others must be replayed by the sender. The number of the elements that must be replayed is, at any instant of time, upper bounded by the asynchrony degree of the message logging channel (see Chapter 6). If we assume that the sender and the receiver take checkpoints in an uncoordinated fashion, we can limit the rollback depth (i.e. the number of the lost checkpoints) with the receiver checkpointing frequency, plus the sender checkpointing frequency (in the worst case). In the case of the farm, *we logically map a sending operation of a result on the output stream and its copy on stable storage onto a global farm checkpointing operation.* The task that generated this result is no longer needed, and will not be requested from the external world.

As in the previous case, the streams linking the emitter to the workers, and the workers to the collector are implemented as communication channels without message logging. We need to extend the previous recovery behavior to support the possibility of loosing stream elements:

- In the case of failure of the emitter we lose some elements passed on the channel but not yet copied on stable storage. Suppose the emitter is the only failed process. A set of tasks with contiguous sequence identifiers are to be re-generated from the sender. These tasks are those copied in the channel *ck* but not yet received by the message logger. The emitter obtains the sequence identifier of the last logged message, and it requests the sender of the stream to re-generate all the following elements. In the case in which the emitter fails concurrently with other processes we apply the same scheme as above: we pass the sequence number of the older unlogged message to the sender.

- In the case of failure of the receiving party on the farm output stream, (possibly) the collector is required to regenerate some elements. The receiving side passes the collector the sequence identifier of the last logged message. In this case we have to re-obtain the input tasks, and to re-schedule them to workers to re-compute the lost results. The required results are all the contiguous ones from the received sequence identifier and there on (according to the ordering on the farm output stream). The recovery is similar to the case of failure of the collector for synchronous message logging.

4.4.4 Overhead Analysis

We extend the cost model of the farm to include the overhead due to message logging on the input and output stream. We first instantiate the abstract model, presented in Section 3.1, to the chosen implementation. As stated above, the cost model was based on three quantities: T_{IN} is the time needed to obtain an input element, and to apply the scheduling strategy; T_F is the time needed to evaluate $\mathbf{F}$ on an input element; T_{OUT} is the time needed to apply the collection strategy, and to provide a result on the output stream. In the abstract model, we mapped these quantities in the worker behavior. We now map these quantities onto the farm implementation:

- T_{IN} is mapped onto the service time of the emitter process, that we denote with T_E. In the case of round-robin strategy, this quantity includes the time needed to receive an element, to apply the scheduling strategy (i.e. to get the next worker index), and to send the element to the selected worker. In the case of on-demand scheduling strategy, in the worst case this quantity corresponds to the time needed to receive an element, to check for an available worker, and the time needed to send the element to the selected worker.

- T_F is mapped onto the service time of the worker process, that we denote with T_W. For both scheduling cases, this time includes the time needed to receive

an element, the time needed to evaluate **F**, and the time needed to send an element.

- T_{OUT} is mapped onto the service time of the collector. This consists in the time needed to select a worker, the time needed to receive a result, and the time needed to send it to the output stream.

We have to define the communication costs to instantiate the above quantities. We denote with $T_{send}(K)$ the time needed to send a message of size K on a communication channel, without message logging. We denote with $T_{recv}(K)$ the time needed to receive a message of size K on a communication channel, without message logging. Now consider the case of FT-Streams. We denote with $T_{S-FT-Send}(K)$ the time needed to send a message of size K to a synchronous FT-Stream. This cost corresponds to: $T_{S-FT-Send}(K) = T_{send}(K) + T_{log}(K)$, where the second quantity is the time needed to copy the message to the stable storage, which depends on the chosen implementation. For instance, in the case in which the stable storage is implemented on the receiver secondary storage, $T_{log}(K)$ corresponds to the time needed to locally copy a message of size K on the hard disk. We denote with $T_{S-FT-Recv}(K)$ the time needed to receive a message on a synchronous FT-Stream. If we consider a single receive operation, we have $T_{S-FT-Recv}(K) = T_{rcv}(K)$, i.e. no logging impact is experienced on the receiver. Clearly, as the corresponding send operation is heavier than a send on an unlogged stream, the overall input bandwidth of the receiver will be lower than the corresponding case without message logging. Now consider asynchronous FT-Streams. We denote with $T_{A-FT-Send}(K)$ the time needed to send a message of size K on an asynchronous FT-Stream. In general, $T_{A-FT-Send}(K) = T_{send}(K)$, if the producer has not filled up the communication queues. We denote with $T_{A-FT-Recv}(K)$ the time needed to receive a message of size K from an asynchronous FT-Stream. As above, $T_{A-FT-Recv}(K) = T_{recv}(K)$. Notice that the high-level description of farm computations allows us to know the message size at compile-time, as the definition of streams (and I-Structures) requires the elements to be typed.

We now instantiate the cost model of the farm implementation. The emitter service time for the round robin scheduling can be computed as:

$$\begin{aligned} T_E^{RR} &= T_{S-FT-Recv}(K) + T_{next-w} + T_{send}(K) \\ &= T_{recv}(K) + T_{next-w} + T_{send}(K) \end{aligned}$$

where input elements have size K, and T_{next-w} is the time needed to access an integer variable, and to increment it. For the on-demand scheduling the cost is:

$$\begin{aligned} T_E^{ON-D} &= T_{A-FT-Recv}(K) + T_{next-w} + T_{send}(K) \\ &= T_{recv}(K) + T_{next-w} + T_{send}(K) \end{aligned}$$

In this case, T_{next-w} is the time needed to check the availability of a worker. The worker service time is:

$$T_W = T_{recv}(K) + T_{Eval} + T_{send}(K)$$

where T_{Eval} is the time needed to evaluate once $\mathbf{F}$. Finally, the collector service time for the round-robin strategy is:

$$\begin{aligned} T_C^{RR} &= T_{next-w} + T_{recv}(K) + T_{S-FT-Send}(K) \\ &= T_{next-w} + T_{recv}(K) + (T_{send}(K) + T_{log}(K)) \end{aligned}$$

This quantity, in this case, depends on the logging overhead. In the on-demand case, this is not true (in the best case):

$$\begin{aligned} T_C^{ON-D} &= T_{next-w} + T_{recv}(K) + T_{A-FT-Send}(K) \\ &= T_{next-w} + T_{recv}(K) + T_{send}(K) \end{aligned}$$

Notice that the asynchronous case has a lower overhead, w.r.t. the synchronous one. This property is inverted in the case of rollback recovery, as asynchronous FT-Streams can require processes to re-provide elements, which in the synchronous case are recovered only from the stable storage. Optimizations can be based on choosing the synchrony of exploited FT-Streams, and configure proper asynchrony degrees, in the case of asynchronous FT-Streams.

4.5 Comparison with Structure-Unaware Protocols

In this section we consider checkpointing and rollback recovery protocols defined for structure-unaware programming models. These techniques are applied to farms implemented as distributed processes interacting through message passing, according to the **E-W-C** strategy. Our purpose is to discuss the pros and cons of exploiting these techniques and to compare them with the multiple configurations of the solution we presented above. The comparison is guided by the issues we introduced in Chapter 1, which we re-write here: *analysis of performance*, *consistency definitions*, *determinism of computations*, *modularity and composability* and *experimental results*.

The main issue to be faced in farm computations is represented by its frequent interactions with the provider and consumer modules on the input and output streams. That is, while a data parallel computation performs the larger part of its computation without interacting with the external modules (see Chapter 3), farm computations are defined to perform "frequently" these interactions, also during the same computation. In general terms such external modules cannot be controlled by the farm fault tolerance: they cannot participate to any checkpointing and rollback recovery protocol. In the context of classical checkpointing and rollback recovery protocols this issue is formalized with a special process, called the *Outside World Process* (OWP in short). In rollback recovery the OWP interacts from the outside with the application processes, it cannot fail nor it can rollback and recover. As

a consequence, the messages it sends to the application cannot be replayed and it cannot receive duplicated messages. Checkpointing and rollback recovery protocols face this issue according to different techniques (see [39]). Below we discuss these techniques w.r.t. the one which we introduced for farm computations, based on FT-Streams (see Chapter 6).

4.5.1 Checkpointing Protocols

Suppose we apply existing checkpointing protocols to the farm structure at the level of implementation. These protocols are based on several *consistency definitions* for global states. In [27] it is introduced a consistency property for global states which consider the number of sent and received messages on each application channel, which must be equal. A global state including a state for each process and featuring this property is consistent. In [39] a consistent global state must not include orphan messages (see Chapter 3). At the abstraction level, we have shown that these consistency definitions are useless: the provider and consumer of the input and output streams respectively are not included in the farm model. As these are the *only* communications (more correctly, interactions), we need to apply the solutions to the interactions with the OWP to support farm fault tolerance according to classical checkpointing schemes.

At the implementation level we have to consider also the communication channels between **E-W-C** processes. If we exploit the structure-unaware programming model, the set of processes of the farm must be supported with some checkpointing technique to build or recover consistent states. This can be done by exploiting the above definitions. For instance, suppose that the emitter process fails. One or more workers are performing a previously assigned task. The scheduling messages of these tasks are *orphan* if the emitter has not saved its state after the scheduling actions.

In our structure-aware model we can avoid the message re-sending from the emitter process by exploiting the supporting data structures we introduced in this chapter. During recovery, if the emitter is the only failed module, it re-builds the information on scheduled input elements by interacting with workers and the collector. This support has been defined by exploiting the high-level properties of farm computations, according to the I-Structure model (see Chapter 3).

Further *consistency definitions* can be derived by exploiting the so-called Z-theory [68], to introduce Communication-Induced-Checkpointing (CIC) protocols. These definitions give necessary and sufficient conditions for a checkpoint to be part of a global consistent state. In practice, each checkpoint should not be part of a Z-Cycle (see Chapter 2). If we consider the abstraction level of farm computations, we can see that it is not possible to create Z-Cycles: the dependency graph originates from the external module, which provides input elements, it includes one of the workers and it terminates with another external module, which receives output elements. Thus, at this level of abstraction this theory is useless.

Similar considerations are valid at the implementation level, because functional

dependencies between **E-W-C** processes "flow" in just one sense: from the OWP, passing the emitter, the workers, the collector and terminating to the OWP. As OWP is, by its definition, outside the application, we cannot consider this a Z-Cycle. The only chance of building Z-Cycles is in the on-demand case, because of the *free* streams from worker processes to the emitter one. If we apply a CIC structure-unaware technique (for instance [52]) it is possible that these interactions induce checkpointing operations on some processes. For the purposes of this thesis it is necessary to evaluate in abstract, by applying the Z-theory, if such checkpointing operations are actually forced and how these impacts on the farm performance. We demand to future work this study and, below, we discuss why checkpointing techniques are not well-suited for farm computations from a general viewpoint on performance aspects, aside the considerations on consistency definitions we give here. Anyway, apart of the issue of free streams in the on-demand case, whose solution can be provided by considering the farm semantics (see below), we can deduce that in farm computations no forced checkpoints will be induced. As a result CIC protocols are a mix of uncoordinated (because of local process checkpointing) and globally coordinated (because of interactions with OWP) checkpointing protocols.

In our solution we face with the possible inconsistencies built because of *free* stream dependencies by noticing that, in the case of failure, it is *useless* to manage such messages: the consistency of the farm is automatically guaranteed by the functional stateless definition of the modules implementing farms. The losing of such messages is supported with their re-sending, after a failure. Their duplication does not influences the general semantics of the farm, but it only lowers the performance of load balancing.

We also consider *nondeterminism* issues to compare structure-aware and unaware checkpointing protocols. In our farm model, nondeterminism is relegated to the time at which elements are received on the input stream, to the scheduling decision and to the collection one. This information (on the relegation of nondeterminism) is not available for structure-unaware techniques: any process can be affected by a nondeterministic event. This must be properly supported classical checkpointing techniques. Thus, also for this aspect, it is not possible to relax the hypotheses of checkpointing protocols and provide a simpler technique for farm computations w.r.t. the ones for structure-unaware computation models.

As far as concerns performance and *performance analysis*, two main issue must be considered to compare structure aware and unaware techniques:

- according to any checkpointing technique, the states of the **E-W-C** processes could checkpointed at any instant of time. For instance, checkpointing can happen during the evaluation of the function **F** on the workers and during the scheduling and collection phases on the emitter and the collector respectively (this issue is central in [12]). If we consider the definition of **E-W-C** processes we can see that *their whole states*[1] *are useless* for the purpose of re-building global correctness in the case of failure. The supporting data structures we

exploit in our solution characterize the *necessary and sufficient information needed to re-start from a correct state after a failure*. Again, this information is collected, managed and recovered by exploiting the high-level properties of farm computations; as a consequence checkpointing the state of farm implementation processes is just a source of performance overhead;

- the interactions with the Outside World Process must be supported with a global coordination of the whole set of processes for all flavors of checkpointing protocols [39] (actually, it is a nonsense for uncoordinated checkpointing). On the one hand each message receive event and each message sending event, respectively on the emitter and on the collector, must start a global checkpointing. On the other hand a farm structure should be exploited (and actually it is!) when the parallelism of workers can be fully used. That is, when the frequency of input element arrivals is sufficiently high to allow all workers to be evaluating a task in each moment of the computation. In other terms, there should not be a moment in the time in which a worker is not evaluating a task because there are not available ones. In our techniques the events "receive an element from the input stream" and "send an element onto the output stream", in the worst case, correspond to a synchronous logging operation on stable storage (if we choose synchronous logging for FT-Streams). It is simple to see that a single access to stable storage, in the worst case, is less costly than running a whole coordinated protocol (see Chapter 2) at the end of which we have to perform N accesses to stable storage (with N equal to the number of processes).

From these observations we conclude that the exploitation of checkpointing techniques for farm computations imposes high degradations to its performance.

This evaluation can be also explained by considering the *modularity and composability* of the parallel programs that can be expressed according to the structured parallelism paradigm. The drastic solutions to support the interactions with the OWP in checkpointing protocols, based on global coordination of checkpointing operations, are due to the lack of a modular programming model, typical of structure-unaware models (e.g. it is the case of MPI). In our approach modularity and composition of parallel and sequential modules is exploited to support composition of fault tolerance techniques: these can be easily and soundly composed with the FT-Stream abstraction and the resulting composition can be analyzed by exploiting the farm cost models and the FT-Stream one (see Chapter 6). Fundamentally, it can be argued that this is the result of a different definition of the behavior of the OWP in our model (w.r.t. the ones made for classical checkpointing protocols). *The OWP behavior is encapsulated in the high-level properties of the FT-Stream semantics.*

[1] In general, the state of a process includes its code and data sections and operating system data structures and (in some cases) procedures.

4.5.2 Message Logging Protocols

Message logging protocols typically support uncoordinated checkpointing: inconsistent global states, built because of the uncoordination of checkpointing operations, are recovered to consistent ones by exploiting the logged determinants of the messages. A message logging protocol, to guarantee correct recovery, must implement the so-called *always no-orphan process* conditions: during the execution there cannot be any process which execution depends on nondeterministic events which determinants are not logged on stable storage or cannot be replayed during recovery [4]. Nondeterminism for this kinds of computations is relegated to message receive events. That is, message logging techniques assume the Piecewise Nondeterminism (PWD) hypothesis (see Chapter 2).

In the first part of this Chapter we have seen that the farm computation model satisfy the PWD hypothesis: nondeterminism for farm is relegated in input stream accesses and on scheduling and collection decision. Sequential computations of processes is deterministic, given the same input messages (or elements). Input and output streams are managed by means of FT-Stream, which is the structured way of introducing message logging techniques. Scheduling and collection operations are managed, during recovery by exploiting the knowledge of the structure of the farm and its semantics. In fact, the always no-orphan condition is satisfied by our techniques: by exploiting the high-level properties of the programming model we have characterized the minimal information required to manage nondeterminism and to re-build consistency after a failure. This information is organized in the supporting data structures for farm, whose description is given in the implementation part of this chapter.

We discuss also performance analysis in the case of exploitation of classic message-logging techniques. While optimistic message logging must be supported with proper operations to solve the Outer World Problem, pessimistic logging guarantee its solution without any effort. The trade-off is clear: while optimistic message logging performs asynchronously or in proper points in the execution the stable storage accesses, pessimistic logging performs them at each message receive. As a consequence, pessimistic logging is (much) more costly than the optimistic one.

Consider to support message logging (both optimistic and pessimistic) at the implementation level. As we do not have any information about the communications or the structural shape of the computation, we cannot know which one is best-suited to meet our needs. That is, we cannot statically analyze their impact on the performance of the computation.

Causal message logging has been introduced to enable an optimization of this trade-off. The always no-orphan condition is satisfied by exploiting the volatile memories of multiple application processes to store the determinants of the same nondeterministic event. Correct recovery requires much more effort than for pessimistic logging, but it can be enabled by registering and storing antecedence information of the dependencies between processes induced by communications. Also this informa-

tion is stored on the volatile memories of multiple processes and scattered by means of application messages piggybacking. Anyway, it is difficult to understand *statically* which is the actual impact on the performance of computations. While in practice causal logging techniques perform well in some experiments (by fixing application and execution environment), in theory there is not a cost model to describe their impact of the performance.

Now consider the case of structure-aware programming model. We can access the structural information of farm computations. In the previous sections we have chosen a specific configuration, by implementing input and output streams with synchronous-logging FT-Streams. Internal farm streams are not supported with any logging technique. Cost models have been derived to statically analyze the impact on the computation performance for this specific solution. Anyway, there are multiple configurations which can be expressed for farm. For instance, we can support synchronous (pessimistic) logging for all streams in the farm. In this case we are implementing a pessimistic logging strategy. Cost models can be simply derived by exploiting the high-level model of farm and FT-streams. In another case we can avoid to support all streams with synchronous message logging. In our solutions, we exploit asynchronous logging version of the FT-Stream and we can guarantee recovery costs by exploiting implementation features (see Chapter 6). Also this second limit case can be analyzed with specific cost models.

While in this thesis we presented the cost models for just one configuration, we leave to future work the description of similar cost models for optimistic and pessimistic logging techniques. In this way we give the application programmer a analytical mechanism to select the most appropriate message logging flavor.

Finally consider causal logging techniques. Our solution is based on the dependencies between input stream elements, workers and output stream elements. These dependencies are mapped at the implementation level and collected, managed and used to support correct recovery after a failure. That is, we have introduced some kind of technique to manage antecedence graphs at the programming model level. Thus, our solutions represent an optimized causal logging technique, based on the structural hypotheses of farm computations. Unlike standard causal logging the different configurations of our solution can be statically modeled by cost models.

As message logging techniques seem to meet more specifically the needs of (fault tolerance for) farm computations, it could be useful to show, by means of experiments, how we can configure message logging for farm to meet optimizations which cannot be introduced in standard structure-unaware techniques. We demand to future work this experimental session.

It has been noticed that message logging techniques are an optimized solution for applications which, for several reasons, enjoy modular design and module compositions [64] (for instance, it is the case of cluster federations). In our model we characterize this aspect in the FT-Stream model, which properties are exploited to optimize the overhead as a proper configuration of the fault tolerance techniques. This can be done because we exploit an high-level model based on structured paral-

lel programming. As a consequence, our techniques can be thought as an high-level definition of the properties of message logging techniques which favor modular applications.

Chapter 5

Checkpointing and Rollback Recovery for Data Parallel

In this chapter we present our study on fault tolerance for data parallel programs based on the computation model and consistency definition introduced in Chapter 3. We describe three different solutions, each addressing different kinds of optimizations, which influence the failure-free execution and the rollback recovery performance:

- The first solution we present exploits just the (uncoordinated) checkpointing of the local module states. The rollback recovery protocol is performed in a synchronized fashion by all modules implementing the computation. For this solution, we minimize the failure-free performance at the cost of a global synchronization during rollback recovery.

- The second solution extends the previous one by checkpointing and copying to stable storage also the computational step reached by a VPM. As a mapping lies between the computational steps and the number that **F** is evaluated on a VPM, this also means that the computational steps are checkpointed at each evaluation of **F** locally on each implementation module. This information is recovered and used in the rollback recovery protocol to minimize the number of participating VPs (also called the rollback recovery *width*). The minimization of the rollback recovery width is obtained at the cost of a local stable storage access at each computational step.

- The last solution exploits the mapping between computational steps and the stencil definition by introducing a logging mechanism: the elements exchanged between VPMs, to implement the stencil, are checkpointed and saved on stable storage. In this case the rollback recovery width is limited to the failed module but at the cost of lower failure-free performance. This is due to larger and more frequent stable storage accesses and to the exploitation of a message logging technique.

Similarly to the farm case, we introduce an implementation strategy for data parallel programs. The strategy is based on I-Structures that allow us to describe the mapping between the local computation, the information exchanged between VPs, and the values assumed by the state. Next, we present an implementation of the strategy based on processes interacting through message-passing. At this level of description, we show the implementation of the three checkpointing algorithms and rollback recovery protocols.

5.1 An Implementation Strategy for Data Parallel Programs

The implementation strategy is represented in Figure 5.1. At this level of description we define a computational unit as a *Virtual Processor Module* (VPM in short) and we map abstract Virtual Processors in VPMs. Each VPM is responsible for executing a subset of VPs and it owns their assigned elements as a local partition. Stencils are implemented as interactions between VPMs, properly resolving the data dependencies, and they consist in passing sub-parts of the local partition between VPMs. Each VPM computation is characterized in steps, reflecting those of the VPs. At each step a VPM evaluates the function **F** on *all* its assigned elements in a sequential fashion. Each VPM is the only entity that can perform put operations on its local partition (Owner-Computes rule). We model each VPM partition with an I-Structure: each value assumed by the local partition of a VPM is placed at a different position on the related incomplete structure.

As a rationale of this implementation strategy it should be noticed that it reflects the abstract programming model by introducing a relationship between VPM computational steps, and the values assumed by their local partitions. This is achieved by modeling each local partition as an I-Structure. Figure 5.2 shows the pseudo-code of a generic VPM_i for a generic data parallel program. In the pseudo-code we avoided to show the initialization phase performed by each VPM and we just show the steady-state behavior. In this implementation strategy we map I-Structure positions onto integer values: the positions are counted by means of an integer variable (**step**) which is incremented at each program loop. The **ghosts** variable represents a set of references to neighbor sub-partitions obtained at each step according to the stencil definition. We do not show how the computation terminates but we just use a variable **term**, which we assume to be properly modified according to the program semantics. At each iteration the program performs the following actions:

1. The VPM obtains the sub-partitions of its neighbors according to the stencil (lines 5 to 10):

   (a) It obtains a list of references to neighbor partitions by applying the **stencilNeighParts** to the current step value and the VPM identifier. The

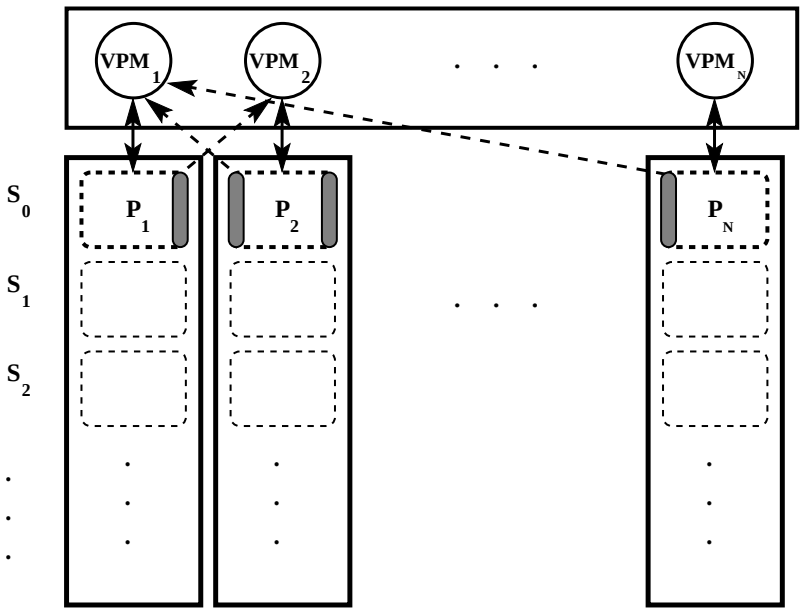


Figure 5.1: Implementation strategy for data parallel programs. Each VPM is assigned a partition of the global state, modeled as a single I-Structure. The model exploits a number of I-Structures equal to the number of VPMs. In this figure we show that each VPM can access its local partition with put and get operations but can access remote partitions, i.e. assigned to other VPMs, only with get ones.

function implements the VP stencil at the level of VPMs and it depends of the mapping of the formers onto the latters.

(b) For each element in the neighbor partitions **pList** it obtains the edges of the sub-partitions it needs to pass and

(c) It performs a get operation with the current step value.

(d) The obtained value is stored in the next free position of the **ghosts** variable.

2. The VPM obtains the value of its partition (line 11). The **myPart** variable contains a reference to the I-Structure that models the local partition of the VPM. Notice that we exploit notation of the get operation used to access the whole content of the I-Structure position. The result is stored on a local variable **localPart**.

3. The VPM applies the function **F** to the local partitions, and the remote ghost sub-partitions, obtaining a result (line 13).

4. The result is put on the next free position of the I-Structure that models the local partition (line 15, 16).

```
1   partition localPart;
2   partitions ghosts;
3   int step = 0;
4
5   while(!term) {
6     pList = stencilNeighParts(step, myid);
7     for each p in pList do {
8       ghostEdges = obtainSubPart(p, step, myid);
9       p.get(step, ghostEdges, &tmp-ghost);
10      ghosts.add(tmp-ghost);
11    }
12    myPart.get(step, localPart);
13
14    res = F(localPart, ghosts);
15
16    step++;
17    myPart.put(step, all, res);
18  }
```

Figure 5.2: VP pseudo-code of a generic data parallel program. The *Partition* data structure implements the local partition of a VPM. The *partitions* data structure implements a container of *Partition* data structures.

Notice that differently from the abstract model presented in Section 3.2.2, which is based on the VP abstraction, in this implementation strategy there is an independent I-Structure for each VPM. We map the sub-set of state elements assigned to a VPM onto this I-Structure, according to the mapping between VPs and VPMs.

We map the relation $\leftarrow\!\!-$ defined between state elements onto state partitions:

Definition 5 *P_k depends of P_j at step i ($P_k \leftarrow\!\!-^i_P P_j$) if there are two elements l, m of S, for which $S^i[l] \leftarrow\!\!- S^{i-1}[m]$, and $S[l] \in P_k$ and $S[m] \in P_j$.*

In other words suppose that the evaluation of **F** on an element of a partition P_k needs the value of at least one element of another partition P_j. Notice that the values of P_j are referred to the previous step. In this case P_k depends on P_j. The definition of consistent states for data parallel computations can be based on the definition of $\leftarrow\!\!-_P$:

Definition 6 *A consistent state is a set composed of the local partitions of all VPMs in which all elements included in all partitions are taken at the same computational step. In other words it is not possible to relate (according to $\leftarrow\!\!-_P$) any pairs of states in the set.*

In this way we simply group the C^i sets defined in 3.2.2 in partitions assigned to VPMs. This consistency rule will be exploited in the implementation that we show in the next sub-sections, which is the basis to implement checkpointing algorithms.

Below we also exploit a $\dashleftarrow_{VPM}$ that relates VPMs instead of their partitions as a mapping of $\dashleftarrow_P$ from partitions to corresponding VPMs.

5.1.1 Cost Model

We can instantiate the cost model of data parallel programs presented in 5.1 to the implementation strategy of above. The step performance is the time needed to perform a computational step. In this context it is referred to a VPM unit and it consists of:

- Evaluating the function **F** on all elements of the local partition. We denote with g the size of a local partition and with $T_F(g)$ the time needed to apply **F** to g elements.
- Implementing the stencil with get operations applied to the I-Structure. We denote this quantity as $T_{get}(g)$, which value depends on the size of the local partition.
- Copying each new computed partition values to the proper I-Structure by means of a put operation. We denote this quantity with $T_{put}(g)$.

The time needed to perform a loop iteration for a generic VPM_i is:

$$T^i_{step} = T_F(g) + T_{get}(g) + T_{put}(g)$$

5.1.2 Implementation

We implement data parallel programs according to the strategy we described above in a distributed environment. Thus, the computation consists of a set of processes communicating through message passing. We map each VPM onto a process, whose local memory includes (at least) the local partition of the mapped VPs. We place a communication channel between each pair of VPMs[1], which interact according to the stencil definition. The messages exchanged between the processes implementing the VPMs are the sub-partitions (or ghosts) exchanged between VPMs. One of the solutions we present exploits the FT-Stream abstraction (see Section 4.4, and Chapter 6) to implement message logging techniques. We also add a communication channel between each pair of VPMs to carry control messages related to the rollback recovery protocols. We denote such channels as *rollback* channels or *failure notification* channels, depending on the way in which the VPMs use them. Clearly, this represents a simplification of optimized implementations of failure notification sub-systems, and rollback recovery communications. We use this simplification to abstract the specific needs of optimized supports: this allows us to make clear the

[1] At the implementation level we will use the term VPM to denote the process implementing an abstract VPM.

description of the checkpointing algorithms and rollback recovery protocols. The actual implementation of data parallel programs is described as a pseudo-code of the behavior of VPMs and it includes the checkpointing algorithms.

5.2 Failure Model and Detection and Stable Storage

We assume that the failure model is the *fail-stop* one [76] which is applied to VPMs: a VPM can fail by stopping its execution. Other VPMs are informed of its failure when they try to communicate with it. After a failure some sub-system restarts the VPM on a computing resource and it provides it with the needed rollback recovery information (see below). We avoid the description of the process restart sub-system. When a VPM restarts it can send, by means of their *failure notification* channels, a message to all other VPMs to indicate its failure and restart. This message is eventually received at all destinations.

We assume *reliability* of communications between processes that have not failed.

We also assume that communications implementing stencils are atomic w.r.t. failures. This means that we can provide a support to communication that makes a set of communications performed at a same computation step atomic. For each solution that we describe we also discuss when this feature is actually exploited and on which sets of communications.

5.3 Coordinated Checkpointing and Global Rollback Recovery

The main feature of the first checkpointing algorithm we present is the low overheads it induces on processes: periodically each VPM checkpoints and saves on stable storage its local partition. No further stable storage accesses are needed. The checkpointing frequency is equal on all VPMs and it can be configured at compile-time or dynamically. This last case can be obtained by implementing an atomic modification of the value on the whole set of VPMs. The resulting checkpointing algorithm is coordinated (according to the taxonomy in [39]), but it is totally asynchronous, as each VPM checks locally for the reached computational step. The coordination of checkpointing operations is used to dynamically build consistent sets of local VPM states. The rollback recovery protocol, exploiting such consistent states as rollback targets, involves the whole set of VPMs. This is a clear drawback of this protocol and it represents the cost to be paid for exploiting a lightweight checkpointing algorithm.

5.3.1 Checkpointing Algorithm

Here we informally describe the algorithm, give the pseudo-code for a generic VPM and discuss its correctness. We do this by proving that it builds consistent states in stable storage.

Informal description

In this algorithm each VPM exploits sequence identifiers to label evaluation steps, reflecting those of the abstract definition of data parallel programs. Labels are mapped onto integer numbers and they are implemented as an integer variable which is local to each VPM. This is incremented at each evaluation of **F**. Checkpoints are taken whenever the variable reach some threshold, which we require to be equal for all VPMs. In the VPMs code we can implement the threshold as a constant value or a variable, which can possibly be modified *atomically* during run-time on all VPMs. Thus, the VPMs *coordinately* but *asynchronously* checkpoint their local state elements on their portion of stable storage at the same step indexes: this means that they independently check for the step value, and (if it is the case) they copy the local partition to the stable storage, without any interactions with the other VPMs. The checkpointed state element is labeled with the step index at which it was taken. We guarantee consistency for dynamically built global states by exploiting the assumptions on the parallel structure of data parallel programs. This statement is proved below.

VPM pseudo-code

In Figure 5.3 we show the pseudo-code for a generic VPM which identifier is **myid** and that performs the checkpointing algorithm. The actions performed by each VPM are:

- Interactions with the neighbors, according to the stencil (lines 6-9). These are implemented with receive operations on proper channels. The channel list is obtained by applying the **stencilInNeighChannels** function, which implements the stencil definition and the mapping of VPs onto VPMs. Notice that there is a mapping between the get operations of the implementation strategy (see Section 5.1). and the receive operations. This is due to the choice of mapping each VPM in a separate process. Notice that in the pseudo-code we also pass the current step value. This supports the elimination of duplicated messages in the case of rollback recovery (see below). The elimination is implemented at the level of the communication support.

- Evaluation of the function **F** on each element of the local partition (line 11). We denote this operation in the same way in which we denote the function **F**, even if we are applying it to multiple elements instead of a single one. The

```
1   partition myPart;
2   partitions ghosts;
3   int step = 0;
4
5   while(!term) {
6     chInList = stencilInNeighChannels(step, myid);
7     int i = 0;
8     for each ch in chInList do
9       receive(ch, &ghosts[i++], step);
10
11    myPart = F(myPart, ghosts);
12
13    step++;
14    if(step % CHK_DELTA == 0) {
15      stst-checkpoint(myPart, step);
16    }
17
18    chOutList = stencilOutNeighChannels(step, myid);
19    for each ch in chOutList do
20      send(ch, ghost(myPart, ch), step);
21  }
```

Figure 5.3: VPM pseudo-code extended with the first version of the checkpointing algorithm. Each VPM periodically saves its local state partition on stable storage. The *partition* and *partitions* data structures implements, respectively, a local partition and a collection of partitions.

evaluation exploits also the neighbor elements (ghosts) that are received at the beginning of the loop.

- Increment of the step variable, check for its value and (possibly) copy of the local partition on the stable storage (lines 13-16). This part implements the checkpointing algorithm. Notice that the computational steps are implemented by means of a local private variable on each VPM. This is locally incremented without exploiting any information provided from other VPMs. We assume that the stable storage access is performed synchronously, i.e. the VPM computation does not proceed until the checkpointing operation is terminated.

- Passage of the local sub-partitions to other VPMs according to the stencil (lines 18-21). The **stencilOutNeighChannels** function returns the channels on which sending sub-parts of the local partition. Clearly, each neighbor VPM can need different partitions of the local partition. The **ghost** function returns the correct sub-partition, by exploiting the channel identifier. In our

implementation the channel identifier on a VPM represents the identifier of its neighbor. We also pass the current step value to send operations to support the elimination of duplicated messages. Notice that differently from the implementation strategy of Section 5.1 a single put of the result, obtained by the application of F, is mapped onto: (1) a copy of the result on the local partition variable (**myPart**) and (b) multiple send operations to implement the stencil.

The variable that implements the local partition of a VPM is overwritten at each iteration, i.e. the variable does not implement an I-Structure. In fact, we exploit the I-Structure model to implement the checkpointing algorithm: the checkpointed state at each step **s** is saved to stable storage with the label **s**. The checkpoint is not overwritten on stable storage, as we could need also old checkpoints (see the rollback recovery protocol below). That is, the checkpointing algorithm is based on the mapping between the sequence identifiers of state values and the computational steps, which logic is given by the abstract I-Structure model.

Correctness of the Checkpointing Algorithm

Proving the correctness of the checkpointing algorithm consists in:

- Proving that it implements the consistency rule defined at the level of implementation strategy.
- Proving that it builds consistent "global" states, according to the literature consistency rule 3. That is, the checkpointed states, which are built according to our abstract rule, do not include orphan messages.

It is trivial to map the $\dashleftarrow_{VPM}$ relation onto a relation between the corresponding processes that implement the VPMs. Next we can prove the following:

Theorem 5.3.1 *The set of checkpoints of VPMs taken at the same iteration of the data parallel loop is consistent.*

Proof 5.3.1 *To prove this theorem we have to prove that all the elements collected from VPMs at the same step are results of the same number of evaluations, i.e. the value assumed at the same computational step.*

We first consider the values of the elements in a same partition. A single VPM is responsible for evaluating the function **F** *on each elements in its partition. At each program loop* **F** *is applied to all elements in the local partition. For some elements the evaluation can be void in the sense that we simply apply the identity function to the element. Consequently, the elements of a partition checkpointed to stable storage at the same evaluation step are all results of the same number of evaluations, corresponding to the value of the* **step** *variable.*

Now consider multiple partitions. We logically label each checkpointed partition with the step value at which the local snapshot is taken. According to the checkpointing algorithm partitions on different VPMs are checkpointed to stable storage at the same computational steps. This is true because the **CHK-DELTA** *value is the same on all VPMs at any instants of time. Consequently, locally to a VPM all elements in the same checkpoint are result of the same number of evaluations, which corresponds to the local partition label. Globally the set of local checkpoints labeled with the same step value includes elements which are results of the same number of evaluations. This proves the consistency of a set of checkpoints which are taken taken at the same step according to the Definition 6.*

As a lemma of this theorem we have the correctness of the algorithm:

Lemma 5.3.1 *The checkpointing algorithm builds global consistent states, according to Definition 6.*

We have now to prove the actual correctness of the protocol, w.r.t. the definition given in literature (Definition 3).

Theorem 5.3.2 *A global consistent state, which is built by the checkpointing algorithm, is consistent w.r.t. definition 3.*

Proof 5.3.2 *We have to prove that for a consistent state built by the checkpointing algorithm there are no orphan messages, i.e. there is no a message* **m** *which send operation is not included in the state but the receive one is. More precisely: if the state of a VPM process A includes the event "receive(ch-from-B,m)", then the state of the process B must include the corresponding event "send(ch-to-A,m)". ch-from-B and ch-to-A are the channel references on A and B which link them.*

Consider the checkpoints taken by all VPMs at step s. If the state is not consistent there exists an orphan message m from VPM_i to VPM_j. The set of checkpoints reflects the reception of m by VPM_i, but not its sending from VPM_j. This means that VPM_j sends m some step after the checkpoint at step s and that VPM_i receives it before s. Denote with t the step at which VPM_i receives m and with p the step at which VPM_j sends it. We know that $t < s$ and $s < p$. For transitivity we have that $t < p$. This means that m is sent at a given step and received some step before. This is in contrast with the assumptions of Sect. 3.1.2, for which a message sent at a given step is received at the very same step. Consequently there cannot be orphan messages in the set of checkpoints.

Here we give an example of how the checkpointing algorithm works, and how it builds global consistent states for a specific data parallel program. Consider the execution of a simple fixed stencil data parallel shown in Figure 5.4. The behavior of each VPM (locally) alternates communication to computation. The end of each computation can include the (independent) checkpointing of the local state, which in

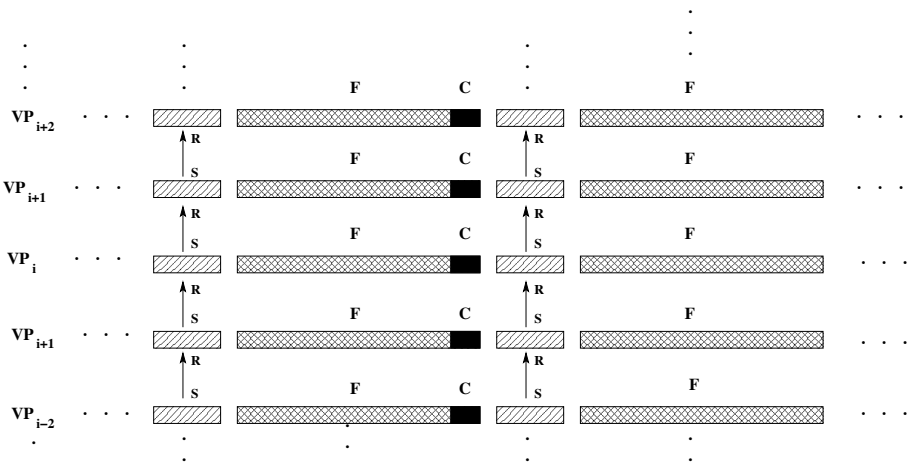


Figure 5.4: Representation of execution of a fixed stencil data parallel program, where each VP passes its local state to the upper VPM.

the figure is represented in black. Again, it is important to notice that the execution of VPMs is asynchronous, and it is only constrained by the program stencil and the asynchrony degree of communications. As we have highlighted in Section 3.1.2 VPMs can execute a different steps of evaluation.

After a checkpoint is taken, each VPM performs a send of the local state to the upper neighbor and it receives a partition of the state of the lower neighbor. Suppose now that VPMs are executing at some (possibly different) steps after the shown checkpoint step s. Suppose also that VPM$_i$ fails at step f (not shown in the figure) and it rollbacks to its last checkpoint at step s. For the sake of discussion, assume that all other VPMs will eventually rollback to the same checkpoint step (as the rollback recovery protocol presented below actually does). The set of checkpoints $\{C_1, \ldots, C_n\}$ represents a consistent state if it does not include orphan messages. The undone send operations are those from the failure step, to s. We have to show that also the corresponding receive operations are undone. All VPMs rollback to the same checkpoint step: all receive operations preceding s are not undone, while all those following s are undone. According to the data parallel programming model, the send operation related to the state computed by VPM$_i$ at the step s will be used at some step *after* s. But all receive operations on all VPMs after s are undone, and there are no orphan messages. This applies to all other VPMs. Consequently, orphan messages are not included in the checkpointed state. From the figure we can observe that this can be derived from the property for interactions between VPMs introduced in Section 3.1.2: each evaluation is related to the local and remote state element values computed at the previous step.

It is important to notice that another class of messages, i.e. lost messages, are not created by the checkpointing algorithm. Lost messages are those whose send operation is not undone, whereas the receive one is undone. Similarly to orphan

messages, we can prove that no lost messages are created:

Theorem 5.3.3 *No lost messages are included in a global consistent state.*

Proof 5.3.3 *Similarly to the proof of the previous theorem, consider a global checkpoint performed at step s. Suppose an orphan message is sent at step t, with $t < s$, and received at step p, with $s < p$. For transitivity we have that $t < p$. This is not possible under the hypothesis of the data parallel model, because a message sent at a step s is received at that very same step.*

5.3.2 Global Rollback Recovery Protocol

The *rollback protocol*, for the checkpointing algorithm described above, involves the whole set of VPMs, and it selects the last consistent global checkpoint that *all* VPMs have passed. The protocol synchronizes all VPMs to avoid inconsistencies due to their asynchronous execution. The *recovery protocol* is performed after the rollback one, from a consistent state: it consists in a re-execution of the lost computation steps, as in a normal execution. The protocols are designed to manage also:

- Concurrent failures, i.e. the concurrent failure of multiple VPMs. In these kinds of situations, multiple VPMs can start the execution of an instance of rollback recovery. Multiple instances are merged together, by selecting one of the failed VPMs as leader.
- Recursive failures, i.e. failures of the same VPM during the rollback protocol. In this case a new instance of the rollback protocol is started.
- Recovery failures, i.e. failures happening during the rollback protocol. Also in this case, a new instance of rollback recovery is started.

During the execution of rollback recovery, the VPMs exchange control messages. Such messages are not copied to stable storage (either synchronously or asynchronously). In the case of one of the above failures, the information related to the rollback recovery protocol in execution is lost. This is a choice, which can possibly be changed with the aim of minimizing the cost of rollback recovery.

Rollback Protocol Description

We assume that whenever a failure affects one of the VPMs, say VPM_f, all the other ones are *eventually* informed of its failure. This behavior reflects the asynchronous execution of VPMs: when VPM_f is restarted it notifies all other VPMs of its failure by sending a control message (see below for the messages used in the protocol). If some of the other VPMs attempt to communicate with a failed VPM during the failure, a fault exception is returned from the communication subsystem. Along with failure detection provided by the communication support, each VPM checks at the

beginning of each step if any of its rollback channels contains a failure notification. We extend the VPM implementation to manage communication failures and notifications in the pseudo-code of Figure 5.5. All communication operations are checked

```
1   partition myPart;
2   partitions ghosts;
3   int step = 0, res = OK;
4
5   while(!term) {
6     chInList = stencilInNeighChannels(step, myid);
7     int i = 0;
8     for each ch in chInList do
9       res = receive(ch, &ghosts[i++], step);
10      if (res == ERROR) rollback(ch);
11
12    myPart = F(myPart, ghosts);
13
14    step++;
15    if(step % CHK_DELTA == 0) {
16      stst-checkpoint(myPart, step);
17    }
18
19    chOutList = stencilOutNeighChannels(step, myid);
20    for each ch in chOutList do
21      res = send(ch, ghost(myPart,ch), step);
22      if (res == ERROR) rollback(ch);
23
24    if(isVoid-failure-ch() == false) rollback(null);
25  }
```

Figure 5.5: VPM pseudo-code extended with the handling of communication primitives. The *rollback()* function is the routine that performs the rollback recovery protocol as non leader VPM. The isVoid-failure-ch() function checks if the failure notification channel is void, and the VPM possibly performs the rollback protocol.

for errors: in the case of detection, the VPM performs the rollback protocol. To simplify, we assume that the error management on receive operations for application channels returns the control to the caller also if any of the rollback channels received a failure notification. Moreover, at the end of each loop, the VPMs check for the failure notification channels if some of the other VPMs have sent a rollback message to denote the request of execution of an instance of the rollback recovery protocol (denoted with the **isVoid-failure-ch** function). During the time between a failure notification is sent and received, other VPMs go on in the computation, possibly communicating to implement the stencil. Some of them can eventually receive a

failure notification because of dependencies on a failed VPMs. They can also stop attempting to communicate with some VPMs that received a failure notification, and are waiting to perform the rollback protocol. During this time, further failures can be incurred by notified VPMs that, generally, lose the notification messages (as explained above). The rollback protocol manages concurrent failures as multiple rounds, and by forcing all VPMs to choose the same rollback step. We exploit the ordering relation between steps, implemented as integer values, to choose from multiple rollback steps: suppose that k VPMs fail concurrently and/or recursively. Suppose that VPM_m is the one executing at the lowest step index, and that its last checkpoint is at step *min*. VPM_m can be any VPMs, i.e. both a failed and not-failed one. The protocol forces all VPMs to choose *min* as rollback target from all other proposed steps. In the case that two failed VPMs propose the same rollback step we exploit the ordering between VPM indexes to select a leader of the rollback protocol.

As far as concern the rollback recovery protocol, the messages used in the protocol are:

- *Rollback messages*, sent by a failed and restarted VPM to all other VPMs. It includes the name of the failed VPM and its rollback step.
- *Acknowledgment messages* (*ack* in short), sent by a notified VPM after it has received a rollback message. It includes its identifier, and its last checkpointing step.
- *Restart messages*, used by one of the concurrently or recursively failed VPMs to restart the computation after a rollback. It includes the minimum rollback step proposed during the rollback.

These messages are those exchanged in the case of failure to implement rollback recovery protocols, exploiting the rollback channels. Their reception, as seen above, takes place on the rollback channels of each VPM. Notice that, due to the asynchrony of the transmission between different processes, we cannot guarantee the ordering of messages sent from the same sender to the same destination on different channels. In other words, we cannot guarantee causal ordering of messages on different channels (see Chapter 2).

We first give a detailed description of the behavior of VPMs performing the rollback protocol. Next, we show the pseudo-code of the protocol.

- A failed and restarted VPM starts a rollback protocol as *leader*, by sending all other VPMs a rollback message, including its selected rollback step, and its name. The send operations are checked for failures. Concurrent failures are discussed below.
- All other VPMs eventually receive the rollback message, and answer the notifying VPM with an acknowledgment message, including their last checkpointing step.

- The leader VPM receives an ack message from all other VPMs, and it selects the minimum checkpointing step *min* from the proposed ones.
- The leader VPM sends a restart message including the *min* rollback step.
- At the end of the protocol, all VPMs know the checkpointing step: they obtain the checkpointed state from the stable storage, and they clear the application channels to avoid duplicated messages. Finally, they restart the execution from that step by properly modifying the local variables.

The clearing of the application channels is not sufficient to avoid all duplicated messages, but only those that have been received until the end of the rollback protocol. There can still be application messages that are sent before the start of the rollback protocol, and received after its termination. This can happen because, as we notice above, we exploit two different channels for rollback control messages and application messages, and their ordering cannot be ensured in the hypothesis of complete communication asynchrony. These messages will be regenerated during the re-execution and as such duplicated. Their elimination is relegated to the communication support, which exploits the step values piggybacked to each application message (provided with the send operations), and the step values provided with the receive operations.

We describe an example of the rollback recovery protocol execution in order to study some interesting aspects. Figure 5.6 shows a simple case of four VPMs exchanging the messages during the rollback recovery protocol. A is the failed and

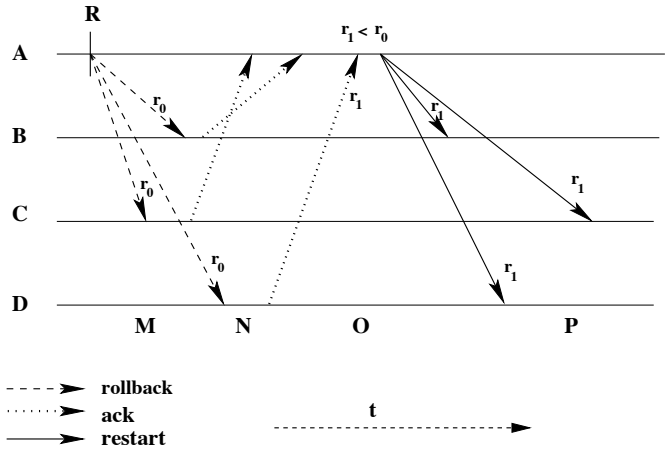


Figure 5.6: Temporal representation of the messages exchanged during the rollback protocol. We only show the answer by D (r_0) because its rollback step is lower than all the other ones.

restarted VPM. After its restart, A sends a rollback message to all other VPMs, indicating its name and the step from which it was restarted, namely r_0. After

sometime all VPMs receive the rollback message. In the most general case, suppose that B and C are executing after r_0 and that D is executing at r_1, with $r_1 < r_0$: the protocol selects r_1 as the rollback target of all VPMs; D answers to A with an ack message, including its last checkpointing step. A decides for r_1, and it sends all other VPMs a restart message including r_1; next, all VPMs recover their r_1 checkpoint from stable storage, and they restart the execution. To avoid inconsistencies in message passing, the communication channels for application messages are cleared.

We also have to discuss failures during the communication operations (see the VPM pseudo-code in Figure 5.5, lines 5 to 10, and 19 to 23). For this kind of protocol, all VPMs rollback to a common checkpoint and restart the execution from the same step. Consequently, previously sent messages will be completely replayed during recovery (this corresponds to a full re-execution). Message channels can be cleared at the beginning of the recovery phase. Also partial execution of the communications performed before any failures are cleared from the recipients. Thus, no atomicity is required to support both the set of send and receive operations.

We define the notion of concurrent failure as the failure of a VPM (possibly the same failed before) while it is performing the rollback protocol. We define recovery failures as those happening during a re-execution. In Figure 5.6 consider the failure of D during the rollback protocol. We have different scenarios depending on the concurrency of events.

We define the time intervals during the rollback protocol (M, N, O and P) to describe the actions performed in the case of failures during rollback. Each such interval is referred to each VPM:

- The interval M starts from any point in the execution and it terminates when the rollback message is received.
- The interval N starts from the receiving of the rollback message and it terminates when the acknowledgment is returned to the leader.
- The interval O starts from the sending of the acknowledgment to the leader and it terminates when the restart message from the leader is received.
- The interval P is any point in the execution after the restart message is received.

We now describe the behavior of VPMs in the case of recursive failure. The first two behaviors below are related to concurrent failures, whereas the third one is related to recovery failures:

- if D fails during M: the send operation from A to D fails and an error is returned to A. A knows now that D has failed and it waits for a rollback message instead of the ack one. A postpones the restart message after the decision about the new leader of the rollback will be taken. All other VPMs,

while waiting for the restart message, will eventually receive the rollback message from D. Thus, they will decide a new rollback step and a new leader. The leader VPM is the one proposing the oldest step: in this way we guarantee the same choice is taken by all VPMs. If A is the leader it receives the D rollback message, it checks that its step is older than the one proposed by D and it re-sends the same rollback message to D. This decision is taken locally by all other VPMs. Otherwise, if D proposes the oldest step, A is not anymore the leader and it will answer to D with an ack message. Also this decision is taken locally on all other VPMs. The leader VPM terminates its round of the rollback protocol and a new round, with D as leader, is performed. If equal steps are proposed, A is chosen by all VPMs as the leader of the protocol (exploiting in this case the lexicographic ordering between VPM names). This is the reason behind the introduction of VPM names in the rollback control message. Notice that we do not need to exploit a group membership service to guarantee a same view of active processes. This is due to the fact that we assume that processes are restarted and that the re-start sub-system provides a failed VPM with its identity.

- if D fails during N: A attempts to receive the ack message from D and the communication support notifies it of the failure. The protocol is similar to the above one but it can happen that A receives an ack message from one of the non-failed VPMs indicating a different step than the one it notified. This can happen if one of the VPMs receives the rollback message from D before the one from A. In this case, A knows that at least another failure happened, and it waits for the rollback message from D. This is the motivation behind the introduction of the rollback step in the ack messages. This behavior is not strictly required for the correct execution of the protocol but it optimizes the dissemination of the failure information.

- If D fails during O or P: D has yet sent its checkpointing step to A. A new instance of the rollback protocol will be re-executed with D as leader. As no further computation is performed D will decide the same checkpointing step selected as a result of the previous protocol. This happens in both the case of failure during O and P: the only difference is that A can detect the failure of D when sending the restart message, and it can automatically wait for the D rollback message rather than performing unuseful computation. In the next instance of the protocol, all VPMs propose the same checkpointing step, and D remains the leader of this protocol, if no further failures happen. If other failures affect the rollback protocol, we apply the same scheme we described in these three points.

Notice that recursive failures can be seen as an instance of the above points because survived VPMs receive multiple instances of rollback message from the same VPM. The proposed step is the same at each recursive failure because the failed VPM

cannot execute some steps until the rollback protocol is terminated. The survived VPMs, which receive multiple rollback messages from the same VPM, simply re-start the rollback protocol from its very beginning.

5.3.3 Protocol Pseudo-Code

We give the pseudo-code of VPMs related to the rollback protocol. The pseudo-code is in the style of a generic imperative language.

The procedure described in Figure 5.7 implements the protocol performed by a failed and restarted VPM. The parameters passed to the procedure from the restart sub-system are the VPM identifier (i) and its nearest checkpointing step r. The *getRollbChannels* function returns the list of input and output rollback channels from and to all other VPMs, given a VPM name. The results are stored in two local list. In the first part (lines 9 to 21), the VPM sends to all other VPMs a rollback message which indicates its identity and its rollback step. Each communication is checked for failure: in the case one of the other VPMs fails (case **M** of Figure 5.6) a new leader is elected. The VPM waits for the rollback message from the other failed and restarted VPM. The pairs exchanged between VPMs include the rollback step and a VPM identifier. Two pairs $p_1 = (r_1, i)$ and $p_2 = (r_2, j)$ can be related according to the following rule[2]:

$$p_1 > p_2 \Leftrightarrow r_1 < r_2 \vee r_1 = r_2 \wedge i < j$$

This rule is implemented in the **decideLeader** function (see Figure 5.8). Notice that the two pairs are received from all VPMs before the restart message of the initial protocol is sent: this because the leader of that protocol waits for the rollback message from the new failed VPM. If the VPM is still leader of the rollback protocol, we re-send to the new failed VPM the same rollback message that we previously sent. In this way it can decide the new leader. To re-send rollback messages to concurrently failed and restarted VPMs we have to perform this sequence of actions until all VPMs have decided the same leader. If the VPM is not anymore leader, it performs the rollback protocol as simple participant. From line 23 to 36, the leader VPM receives the acknowledgment messages. In the case of concurrent failures (case **N** of Figure 5.6), we decide a new leader, as above. At each received ack message, we check for the proposed value, and we possibly set a new minimum. Next (lines 38 to 46), the leader VPM sends to all other VPMs the decided minimum. Failures in this phase (case **O** of Figure 5.6) are managed by sequentializing a new instance of the rollback protocol. The leader VPM detecting a failure, does not start a new leader election, but it waits for a new rollback message from the newly failed VPM (or from

[2]There is no apparent reason behind the choice of exploiting the proposed rollback step to decide which is the leader of a protocol. The actual reason behind this choice is that we can simply extend this protocol if we make this assumption. Essentially, in the extension we choose from only the rollback steps of failed VPMs and not from *all* VPMs. We do not show the extension in this thesis.

```
1  failedAndRestarted(name = i, rollback-step = r) {
2    //we assume r is the minimum
3    rollback-step min = r;
4
5    (CH-IN-List,CH-OUT-List) = getRollbChannels(i);
6
7    outTmp = CH-OUT-List;
8
9    while(isEmpty(outTmp) == false) {
10     for each ch in outTmp -> {
11       try {
12         send(ch, ``rollback(i,r)'');
13       } catch(Failure) { //waits for rollb. message
14         ch-in = getInFromOut(ch);
15         bool iAmLeader = decideLeader(i, r, ch-in)
16         if(iAmLeader == false)
17         partRollback(i, r, ch, ch-in);
18       }
19       remove(ch, outTmp);
20     }
21   }
22
23   while(isEmpty(outTmp) == false) {
24     for each ch in CH-IN-List -> {
25       try {
26         receive(ch, ``ack(i,s)'');
27         if(s < min) min = s;
28       } catch(Failure) { //waits for rollb message
29         ch-in = getInFromOut(ch);
30         bool iAmLeader = decideLeader(i, r, ch-in)
31         if(iAmLeader == false)
32           partRollback(i, r, ch, ch-in);
33       }
34       remove(ch, CH-IN-List);
35     }
36   }
37
38   bool immediateRestart = true;
39
40   for each ch in CH-OUT-List -> {
41     try {
42       send(ch, ``restart(i,min)'');
43     } catch(Failure) {
44       immediateRestart = false;
45     }
46   }
47
48   if(immediateRestart == true) {
49      myPart = stst-recover(min);
50      clearApplicationChannels();
51      restart from min;
52   } else {//I will receive a rollb message
53     <wait for rollback message>
54   }
55 }
```

Figure 5.7: Pseudo-code of a failed and restarted VPM for the first rollback recovery protocol.

```
1 bool decideLeader(name = i, rollback step = r, channel = ch)
    {
2   receive(ch, ``rollback(j,s)'');
3   if((r < s) || ((r == s) && (i < j))) { //I am leader
4     return true;
5   }
6   return false;
7 }
```

Figure 5.8: Pseudo-code of the function performed by the leader of a rollback protocol in the case of concurrent failures.

other failed VPMs). If no failures are detected, the VPM restarts its execution from the selected minimum: it recovers the checkpoint that corresponds to the minimum step (line 49) and it clears the communication channels for application messages (**clearApplicationChannels** procedure).

In Figure 5.9 we show the behavior of a VPM participating in the rollback protocol (not as leader). The very same actions described above (see the previous subsection) are performed. Notice that, before receiving a restart message, we check for the presence of rollback messages on the input rollback channel. These are the channels on which any other VPMs sends its rollback message in the case of failure and restart. If another VPM is failed, a new leader can be locally decided with the same rule of above. If the leader changes, the rollback protocol is restarted from its very beginning. Also notice that we check for the leader failure (recursive failures) and that we manage this case by restarting the rollback protocol. When a round terminates, i.e. a restart message is received, a VPM recovers the checkpointed state from stable storage, clears the communication channels for application messages, and restarts the execution from the selected step.

5.3.4 Correctness

The described rollback protocol is based on the well-known notions of *leader election* and *rotating coordinator* [26, 86] (where, in our terminology the coordinator is the leader). A new leader is elected at each concurrent failure, according to the rule described above. The choice is independent of the precedence of failure events and rollback notifications, because it only depends on the proposed rollback step and the VPM identifier. Each new leader election corresponds to a new *round* of the rollback protocol. Each round can terminate by deciding for a rollback step. Otherwise it can fail by incurring in a rotation of the leader due to a concurrent failure. Indeed, the rotation is needed to ensure that all VPMs participating in the protocol actually choose the same leader independently of the local sequence of events.

The correctness proof consists in proving that, in spite of k concurrent failures

```
1  partRollback(name = i, rollback step = r, channel ch-out,
2    channel ch-in) {
3    try {
4      send(ch-out, ``ack(i,r)'');
5    } catch (Failure) { //recursive fail.
6      partRollback(i, r, ch-out, ch-in);
7    }
8
9    //check for my rollback input channel
10   if(isEmpty(mych-in) == false) { //new rollb mess.
11     receive(mych-in, ``rollback(k, s)'');
12     if((k < r) || ((k == r) && (s < i))) { //s new leader
13       partRollback(i, r, getOutChannel(s), getInChannel(r));
14   }
15
16   try {
17     receive(ch-in, ``restart(i,min)'');
18   } catch (Failure) { //recursive fail.
19     partRollback(i, r, ch-out, ch-in);
20   }
21
22    myPart = stst-recover(min);
23    clearApplicationChannels();
24    restart from min;
25 }
```

Figure 5.9: Pseudo-code of the function performed by a VPM participating in the rollback protocol not as leader.

(failures in **M** or **N** of Figure 5.6):

- All VPMs *eventually* elect the *same* leader, and
- After the last concurrent failure all VPMs decide for the same rollback step.

To prove these points, we start from some lemmas:

Lemma 5.3.2 *The rollback protocol is correct in spite of a single failure.*

Proof 5.3.1 *Obvious from the algorithm: the leader is the failed and restarted VPM. It receives the proposed steps appended to ack messages, it locally computes the minimum, and sends all other VPMs the decided step.*

Next, we consider concurrent failures. We start from proving that, in the case of a concurrent failure, all VPMs take the same decision: they either start a new round, or sequentialize the management in a new rollback protocol.

Lemma 5.3.3 *Suppose VPMs are in a round d_i of the rollback protocol and that one of them fails. All other VPMs uniquely decide to either start a new round or sequentialize the management in a new protocol.*

Proof 5.3.2 *We can prove that it is not possible that two VPMs decide differently on the action to be undertaken. Consider four VPMs (named L_1, L_2, S, and T): L_1 is the leader of the current protocol; L_2 is the concurrently failed VPM and we assume that L_2 becomes leader of the new protocol; S and T are participants to the current protocol. We consider the different cases that can happen w.r.t. the concurrency of events.*

Suppose that L_2 fails before it can send the ack message to L_1 (case **M** *and* **N***). To consider a* **new round** *S and T have to receive its new rollback message either before the one from L_1, or after it and before they can send the ack message. To* **sequentialize** *the protocols S and T have to receive the L_2 message after they have sent they ack message. Suppose that T receives the L_2 message in the first case (before sending the ack message to L_2). In this case, it decides for a new round. The current protocol cannot be terminated, as L_1 will not receive the ack from T. S will not receive the restart message from L_1. S and L_1 eventually receive the rollback message from L_2, and they can decide for a new round with its new leader.*

Suppose that T receives the rollback message from L_2 after sending its ack message to L_1. T sequentialize the protocol. The case of S receiving the notification from L_2 before it can send its ack message to L_1 is the same of the previous with S and T inverted. If S receive it after sending the ack message to L_1, then it decides to sequentialize the protocol.

Finally assume that: (a) S decides for a new round, i.e. it receives the L_2 rollback message before sending the ack message to L_1; (b) L_1 decide to sequentialize the rollback protocol for the L_2 failure, i.e. L_2 failed after sending to L_1 the ack message

for the current protocol. For L_1 to send its restart message, it is required to it to have received an ack from all other VPMs. But we have assumed that S does not send its ack message to L_1. Thus, the hypothesis is wrong, and S and L_1 decide for a new round.

Consequently, no two pairs of VPMs decide for a different behavior.

Now we consider the whole set of rounds. The main problem is represented by the asynchrony of communications. In principle each VPM can start a different rollback protocol, each performed for a different failure. In the following theorem we consider multiple concurrent failures and multiple VPMs starting a different rollback protocol with a different leader. We prove that, at the end of the protocol, all VPMs have decided the same leader.

Theorem 5.3.4 *The same leader is elected from all VPMs in spite of k concurrent failures.*

Proof 5.3.4 *Consider k concurrent failures and k VPMs respectively notified of the failures. Suppose that each notified VPM receives the failure notification from a different failed VPM. In other words, we have k leaders each notifying (at least) one VPM. According to the previous lemma, as failures are concurrent, all VPMs decide to make a new round of the protocol, every time they are notified. Denote the k notified VPMs with $\{V_i\}_{i\in 1,\ldots,k}$, each starting in a different round $\{r_i\}_{i\in 1,..K}$. Concurrent failures prevent each round to terminate the rollback protocol in a decision. Each VPM is eventually notified of all other failures before the termination of any previous rounds. We denote the set of decisions taken by V_i at the j-th failure notification as d^i_j. Thus, V_i takes the following set of decisions: $D^i = d^i_1, d^i_2 \ldots, d^i_k$. Each decision consists in applying the decision rule of above, by computing a minimum between two pairs (step and VPM index). The minimum function features commutativity in the sense that its result is independent of the sequence of the factors. In other terms, if we denote with $\oplus$ the rule above, we have that $(a \oplus b) \oplus c = a \oplus (b \oplus c)$. Consider the whole set of decisions taken by all VPMs. We have the sets $D^1, \ldots, D^k$ each corresponding to different sequences of decisions, i.e. application of $\oplus$ to a different sequence of operators. Denote with $r_i = D^i$ the result of the sequence of decisions taken by V_i. As the $\oplus$ operations is commutative we can conclude that $r_1 = r_2 = \ldots = r_k$, i.e. all VPMs eventually decide for the same leader.*

After the last failure a leader is eventually elected, and all other VPMs are required to send to it their proposed step:

Lemma 5.3.4 *After all failures the leader eventually receives an acknowledgment message from all other VPMs.*

Proof 5.3.3 *Obvious from the algorithm: at the last round all VPMs restart the rollback protocol by sending their ack message to the new leader.*

As a consequence of theorem 5.3.4 and lemma 5.3.4, we obtain:

Theorem 5.3.5 *The rollback protocol is correct in spite of concurrent failures.*

Proof 5.3.4 *Consequence of lemmas 5.3.4 and 5.3.4.*

5.3.5 Overhead of Checkpointing and Performance of Rollback

We model the performance of the presented implementation of data parallel programs with two metrics:

- The cost of a computational step. This is a mapping of the abstract cost models presented in the previous sections.
- The cost of performing the rollback protocol.

The cost of a computational step is similar to that of the implementation strategy, in which we replace put and get costs with communication costs. For each loop in which we do not perform checkpointing and stable storage access each VPM pays:

- The cost of evaluating **F** on all elements of the local partition. As above, we denote this quantity with $T_F(g)$, where g is the size of the local partition.
- The cost of receiving the elements from the neighbors. We denote this quantity with $T_{stenc-in}(\overline{g}, i)$, which depends on the cost of receiving a message, the average size of partitions $\overline{g}$ and the step index that we are performing. We instantiate the cost w.r.t. the step index, because of the possible variability of the stencil.
- The cost of sending sub-partitions of the local one to neighbor VPMs. We denote this quantity with $T_{stenc-out,i}(\overline{g}, i)$, which depends on the send communication latency, the average size of partitions $\overline{g}$ and the step index that we are performing.

The quantities $T_{stenc-in}(g)$, and $T_{stenc-out}(g)$ are instantiated with the actual stencil of the program. The time needed to perform a step is:

$$T_{step,i} = T_F(g) + T_{stenc-in}(\overline{g}, i) + T_{stenc-out}(\overline{g}, i)$$

If we denote with $T_{chk}(s)$ the function modeling the cost of checkpointing and copying to stable storage a data of size s we can express the cost of a step in which we perform checkpointing as:

$$T_{step-chk,i} = T_F(g) + T_{stenc-in}(\overline{g}, i) + T_{stenc-out}(\overline{g}, i) + T_{chk}(g)$$

We analyze the cost of the execution of an instance of the rollback protocol in the case of a single failure: we consider the time that passes between the failed VPM starts to send the rollback messages, and the time at which the last VPM receives the restart message. The time is a sum of the following quantities:

- The communication latency for the rollback messages to all VPMs. This cost depends on the available support: if a multicast operation is supported, communications can be performed in overlap w.r.t. each others, and we pay the cost of a single communication $T_{rollback} = T_{send}(\text{rollsize})$, where *rollsize* is the size of the rollback message. If multicast is not supported, the cost is: $T_{rollback} = N \cdot T_{send}(\text{rollsize})$, where N is the number of VPMs.

- The maximum time needed by VPMs to detect the failure. This is upper bounded by the time T_{step} needed to perform a computational step, in the worst case.

- The communication latency for the acknowledgment messages. This costs as N communications: $T_{ack} = N \cdot T_{send}(\text{asize})$, where *asize* is the size of the acknowledgment message.

- The time needed by the failed and restarted VPM to send N restart messages. As for rollback ones, this cost depends on the presence of a multicast support: $T_{restart} = T_{send}(\text{ressize})$, if it is provided, $T_{restart} = N \cdot T_{send}(\text{ressize})$, if it is not, where *ressize* is the size of the restart message.

5.4 Coordinated Checkpointing and Partitioned Rollback Recovery

As stated above, the rollback protocol presented in the previous section includes all VPMs. We extend the previous protocol by introducing a new rollback recovery protocol, which possibly includes a subset of all VPMs. The protocol is strongly based on the assumption of the compile-time knowledge of stencils, which characterizes subsets of VPMs as units of rollback recovery. The protocol is supported by a checkpointing algorithm that extends the one presented in the previous section. The extension consists in checkpointing and saving onto stable storage, *at each loop of the program*, also the computation step reached. This has a stronger impact on the program performance w.r.t. the previous algorithm. Unlike the previous rollback recovery protocol: (a) this one can exploit inconsistent states as rollback targets; (b) the very same previous rollback protocol is applied to a subset of VPMs. Thus, the inconsistency of rolled back states derives from the fact that not all VPMs participate in the rollback protocol: the task of the recovery protocol is to restore consistency after rollback.

5.4.1 Checkpointing Algorithm

The checkpointing algorithm is similar to the previous one, as each VPM periodically saves the checkpoint of its local state onto stable storage. In addition, we also save onto stable storage the value of the computation step at each iteration. This is

done at the beginning of each iteration. The information on the computation step is provided to the VPM in the case of rollback recovery, both in the case of failure and restart, and in the case of rollback due to the failure of another process. In Figure 5.10 we show the pseudo-code of the checkpointing algorithm. Each VPM,

```
1   partition myPart;
2   partitions ghosts;
3   int step = 0, res = OK;
4
5   while(!term) {
6     sts-checkpoint(step)
7
8     chInList = stencilInNeighChannels(step, myid);
9     int i = 0;
10    for each ch in chInList do {
11      res = receive(ch, &ghosts[i++], step);
12      if (res == ERROR) rollback(ch,step);
13    }
14
15    myPart = F(myPart, ghosts);
16
17    step++;
18    if(step % CHK_DELTA == 0) {
19      stst-checkpoint(myPart, step);
20    }
21
22    chOutList = stencilOutNeighChannels(step, myid);
23    i = 0;
24    for each ch in chOutList do {
25      res = send(ch, ghost(myPart,ch), step);
26      if (res == ERROR) rollback(ch, step);
27    }
28
29    if(isvoid-failure-ch() == false) rollback(null,step);
30
31  }
```

Figure 5.10: Checkpointing algorithm for the partitioned rollback recovery protocol. At the beginning of each step (loop iteration) the computation step is checkpointed and saved onto stable storage.

at the beginning of the loop, checkpoints and saves onto stable storage the value of the step variable, i.e. the reached computation step (line 6). As in the previous

algorithm, we support message duplication by passing the current step value to the communication primitives. At the end of the loop we also check if a failure has been notified on the *failure notification* channels. It a failure notification is received, we call the **rollback** procedure passing it the current step value. This is due to the need of supporting partitioned rollback recovery protocols, as we explain below.

5.4.2 Performance Impact of Checkpointing

We extend the cost model of the first protocol (see 5.3.5) with the cost of checkpointing the step value at each iteration of the VPM loop. For steps in which we do not perform the checkpointing of the local state the cost is:

$$T_{step,i} = T_F(g) + T_{stenc-in}(\overline{g}, i) + T_{stenc-out}(\overline{g}, i) + T_{chk}(sizeof(int))$$

We add each step the cost of synchronously accessing the stable storage to copy an integer variable (we implement step identifiers with integer values). For steps in which we perform checkpointing of the local state we have:

$$T_{step-chk,i} = T_F(g) + T_{stenc-in}(\overline{g}, i) + T_{stenc-out}(\overline{g}, i) + T_{chk}(sizeof(int)) + T_{chk}(g)$$

5.4.3 Description of the Partitioned Rollback Recovery Protocol

The protocol we present optimizes the number of participating VPMs by exploiting the knowledge of the step at which the failure happens or a rollback control message is received. As stated in Section 5.3 each VPM has at each computation step two lists of neighbors: one including the VPMs from which it receives the ghost partitions; another including the ones to which it sends sub-parts of its local partition. We implemented the neighbor lists as lists of communication channels, respectively called **stencilInNeghChannels** and **stencilOutNeighChannels**. These correspond to the implementation of the neighbor lists at the programming model level, which are completely specified in the program. The protocol we present is not based on message logging. Consequently, if we consider a rollback protocol of a VPM_i, from failure step f to rollback step r, we need to replay all messages sent to it between r and f, i.e. we have to replay *lost messages*. Suppose now that some of the VPMs, to which VPM_i sends some messages between r and f, are not involved in the rollback protocol. The re-sending of such *orphan messages*, during the recovery of VPM_i, is incorrect: the state of the receiver VPM includes such messages, sent in a previous execution. Thus, we avoid re-sending orphan messages during recovery. The main problem of the recovery protocol is to characterize which messages are orphan or, in other terms, which VPMs are involved in each rollback recovery instance. As a consequence, from the consistency viewpoint, the rollback protocol also exploits as targets inconsistent states. The recovery procedure re-builds a consistent

state starting from the rollback one by replaying lost messages, and avoiding send operations for orphan ones

Duplicated messages are managed in the same way as in the previous protocol, i.e. at the level of communication support. No duplications are created in the case of orphan messages as their sending operations are avoided. The only duplications are related to re-sent messages during recovery, whose management is, also in this case, relegated to the communication support (as described in Section 5.3).

In the protocol we present in this section we target a static knowledge of the participants to any rollback recovery instances on each VPM.

To characterize the set of VPMs participating in a rollback after a failure we define two sets:

$\mathbb{P}$ (for *participating*) is the set of VPMs from which the failed one functionally depends on at all steps and *transitively*, according to the program stencil.

$\mathbb{E}$ (for *excluded*) denotes the set of VPMs not in $\mathbb{P}$.

Notice that we consider functional dependencies at *all* steps, even if they can be different, and we could, in principle, characterize each step with different subsets. We consider all steps as we can:

- Define statically P and E as the list of neighbor VPMs for which a VPM has an input channel.
- Avoid global synchronizations of VPMs needed to compute the actual list of participants in every rollback recovery that depends on their reached step.

Another possibility, which we will study in future work (see Chapter 8), is to precisely characterize the functional dependencies between VPMs between the rollback and failure step of all VPMs. This information is known dynamically, as the VPMs compute in a total asynchronous fashion, only constrained by the stencil and the degree of communication asynchrony.

More precisely, suppose that a generic VPM_i fails at some step. In this section we define the P set, for the rollback recovery protocol executed because of the VPM_i failure, as *the set including all VPMs for which VPM_i and its neighbors have an input channel*. This corresponds to the recursive merge of the **stencilInNeighChannels** lists from each step.

Operationally, for each VPM_i we add all VPMs from which VPM_i receive the partition at some steps. For each such VPMs we recursively add all the ones for which they have an input channel. In a more formal fashion, as we denote with $i \xleftarrow{-}_{VPM} j$ the fact that VPM_i has an input channel for VPM_j (i.e. at some steps VPM_j sends its local state to VPM_j), $\mathbb{P}$ can be defined as the transitive closure of $\xleftarrow{-}_{VPM}$.

Fig. 5.11 shows an inconsistent state that is used for restart after a failure. In this example $\mathbb{P} = \{C, D, E\}$ and $\mathbb{E} = \{A, B\}$. C fails at step r and rolls back at the

end of step p. D and E rollback with C because it has an input channel for D and D has one for E (as stated above, P is defined transitively). A and B do not participate in the protocol. The global state composed of the local states of each VPM includes two orphan messages sent from C to B at steps p and q. The recovered state will include the states of all VPMs at the end of step r. Notice that B will possibly and eventually stop waiting for the message from C until the recovery protocol is terminated. *Possibly* because the recovering VPMs can reach the step r+1 before B terminates the execution of step r. *Eventually* because this happens also in the case of further failures, which force B to participate in some rollback recovery. Both situations can happen because B is not informed of the execution of the recovery protocol of C, D and E.

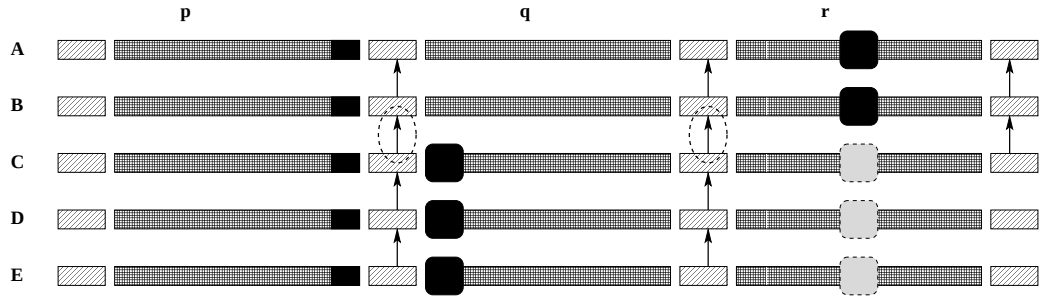


Figure 5.11: Example of inconsistent state obtained after a rollback. The global state includes the checkpoints at the end of step p for C, D and E, and the computation states of A and B while executing step R. In this case $\mathbb{P} = \{C, D, E\}$ and $\mathbb{E} = \{A, B\}$

We discuss a first version of the protocol supporting single failures. Next we extend it to support concurrent and recursive ones. For the second case, as the recovery protocol is different from a normal execution, we express the recursive course of recoveries using *sessions*. Information on sessions must be kept stable during recursive rollback recoveries to avoid the loss of recovery information because of recursive failures.

5.4.4 Surviving Single Failures

The key to understand this protocol is that we exploit the same synchronization scheme of the previous protocol, but we apply it to just the $\mathbb{P}$ set of the (single) failed VPM.

Protocol Pseudo-Code

The pseudo-codes describing the behavior of VPMs during rollback recovery exploit two functions:

P-ch that, given a VPM identifier *id*, returns a pair including (1) the list of rollback channels from VPM_{id} to the VPMs in its $\mathbb{P}$ set (see above) and (2) the list of rollback channels to VPM_{id} from the VPMs in its $\mathbb{P}$ set.

Stencil-Out that, given a VPM identifier *id* and an execution step *s*, returns the list of rollback channels to the VPMs to which the VPM_{id} must send its local partition at step *s*, according to the program stencil.

As stated before, both functions can be defined at compile-time by exploiting the knowledge of the program stencil.

Figure 5.12 shows the behavior of the failed VPM during rollback. VPM_i fails at step *f* and restarts at step *r*. It computes the set of rollback channels of the VPMs participating to the rollback recovery protocol, both in input and output, by applying the *P-ch* function to its identifier. The pair is stored in the list variables *P-IN-List*, and *P-OUT-List*. As in the previous rollback recovery protocol, a synchronization protocol starts, selecting, as a result, the minimum proposed rollback step. All VPMs, which rollback channels are included in the *P-CH-List*, are notified of the failure (with a rollback message). Next they acknowledge the receiving of the rollback message and receive a restart message from the failed VPM. Notice that we avoid to check for failures of communication primitives, because we do not need to support failures during rollback recovery. For this reason this protocol consists in a single round of the previous one (see 5.3). After the synchronization the leader VPM: (1) clears all the application channels from the VPMs which participate in the protocol (**clear-P-ApplicationChannels** function); (2) recovers from stable storage the checkpoint labeled with *min*; (3) performs the recovery protocol from step *min* to step *f*. Finally, it re-starts the normal computation from step *f+1*.

Figure 5.13 shows the behavior of VPM_j that receives a rollback message from VPM_i, which failed at step *f* and rolled back to step *r*. The VPM answers the rollback message with an ack message, including the rollback step it proposes which is obtained by applying the **getMyLastChkStep** function. Next, it receives the minimum step selected by the leader VPM (the failed and restarted one). It retrieves from stable storage the checkpoint that corresponds to the minimum step and it clears the application channels from the VPMs participating in the protocol. Next, it performs the recovery protocol from the minimum step to its failure step. In the pseudo-code the rollback procedure is provided with the input rollback channel from which the rollback message is received. VPM_j obtains the corresponding output rollback channel towards the failed VPM by applying the function *getOutRollbChannel*.

Here we discuss the atomicity (w.r.t. failures) of the set of send and receive operations in the VPM program loop (see Figure 5.10). For a generic VPM we notice that application channels for VPMs in the $\mathbb{P}$ set are cleared (in fact not all channels in the set are cleared, but only the ones to which the VPM is directly connected to) All receive operations (beginning of the loop) are undone and the VPMs in $\mathbb{P}$ participate in the rollback. Thus, receive operations are not required to be atomic.

```
1  failedAndRestarted(name = i, failure step = f, rollback step
       =
2    r) {
3
4      (P-IN-List,P-OUT-List) = P-ch(i);
5
6      step min = r;
7
8      for each ch in P-OUT-List {
9        send(ch, ``rollback(i,r)'');
10     }
11
12     for each ch k in P-IN-List {
13       receive(ch, ``ack(k,s)'');
14       if(s < min) min = s;
15     }
16
17     for each ch k in P-OUT-List {
18       send(ch, ``restart(i,min)'');
19     }
20
21     clear-P-ApplicationChannels();
22
23     myPart = stst-recover(min);
24
25     recover(i,min,f);
26
27     restart from f;
28 }
```

Figure 5.12: Pseudo-code of a failed and restarted VPM for the second rollback recovery protocol in the case of a single failure.

```
1  partRollback(name = j, rollback step = r, failure step = f,
2    channel ch-in) {
3
4      ch-out = getOutRollbChannel(ch);
5
6      p = getMyLastChkStep();
7
8      send(ch-out, ``ack(j,p)'');
9
10     receive(ch, ``restart(i,s)'');
11
12     clear-P-ApplicationChannels();
13
14     myPart = stst-recovery(s);
15     recover(j,s,f);
16
17     restart from f+1;
18 }
```

Figure 5.13: Optimized rollback recovery protocol in the case of a single failure.

This is not true for send operations: if a failure affects the computation during these communications, some messages could have been delivered for the failure step, whereas some others could not. This can induce inconsistencies at the last step of recovery: for simplicity, we assume the set of send operations at the end of the program loop to be atomic w.r.t. failures. This can be implemented by the VPMs by re-executing all send operations when they reach the last step of recovery and attaching the computation step to the messages. Then, the communication support can discard duplicated messages by just checking the attached step value.

In Figure 5.14 we show the pseudo-code for the recovery procedure of a generic VPM_i failed at step f and rolled back to step r. Also in this case the *P-List* variable contains the result of application of the P function to the VPM identifier. The re-computation phase consists in re-executing steps from *r+1* to *f*. To avoid the resending operations of orphan messages the *Stencil-Out* function is exploited at each step: the set resulting of the application of *Stencil-Out* to the VPM identifier and the actual re-computation step is intersected with the set (of channels) of VPMs participating to the recovery protocol. The result is a list of VPM channels to which sending the local state partition. This is stored in the variable *S-CH-List*. *S-CH-List* is successively used as target of the send primitive. As stated before, for all VPMs (which output channel is included) in the *Stencil-Out* list (i.e. the ones participating in the rollback recovery) no orphan messages have been created. This is due to the fact that they rollback to the same step of the failed VPM. This is not necessarily

```
1  recover(i, r, f) {
2    (P-IN-List,P-OUT-List) = P-ch(i);
3
4    for (step = r+1; step <= f; step++) {
5      chInList = stencilInNeighChannels(step, myid);
6      int i = 0;
7      for each ch in chInList do {
8        res = receive(ch, &ghosts[i++]);
9        if (res == ERROR) rollback(ch,step);
10     }
11
12     myPart = it F(myPart, ghosts);
13
14     S-CH-List = intersect(P-OUT-List, Stencil-Out(i,step));
15     for each ch in S-CH-List do
16       send(ch, ghost(local-state));
17   }
18 }
```

Figure 5.14: Recovery protocol performed by VPMs participating in the rollback protocol.

true for the VPMs in *Stencil-Out* that did not rollback. The receive operations for this protocol are performed as in a normal execution, because all VPMs for which VPM_i has an input channel participate in the rollback recovery. The function **F** is applied to the local partition elements and the result is the next value of the local partition. Notice that the saving on stable storage of the local partition is not performed in the recovery protocol: the partition has been saved in the first execution of the recovered steps and we assume that it is not garbage collected. This assumption can be relaxed to minimize the size of occupied stable storage at the cost of slower recovery times. The two solutions are viable: we leave the discussion of garbage collection strategies to future work (see Chapter 8).

Correctness

We want to prove the correctness of the rollback recovery protocols for single failures. From the previous protocol (see 5.3) we inherit the correctness property for VPMs that participate in the protocol: the state of VPMs participating in a rollback recovery protocol is consistent. In general, as stated above, the global state after a rollback, which includes also the VPMs that did not rollback, can be inconsistent. We need to prove that the state is consistent after recovery. We start from some lemmas.

Lemma 5.4.1 *All messages from any VPMs participating in the rollback recovery protocol to any VPMs not participating in the protocol are avoided.*

Proof 5.4.1 *Obvious from the recovery protocol.*

We denote *orphan-avoidance* the property stated by this lemma. We also consider the lost messages:

Lemma 5.4.2 *All messages from any VPMs participating in the rollback recovery protocols to any other VPMs participating in the same protocols are re-executed during recovery.*

Proof 5.4.2 *We have seen that a recovery protocol is equal of a normal execution except for sending events to VPMs that do not participate in the rollback recovery protocol. Message sending operations to VPMs participating in the protocol are not avoided.*

As a consequence of this last lemma (5.4.2), we can state that:

Lemma 5.4.3 *After a recovery, the participating VPMs recover to their initial failure step that they were performing before the rollback recovery.*

Proof 5.4.3 *VPMs participating in a rollback recovery perform the recovery protocol from the globally selected minimum step to their own failure step, i.e. the step at which they detected a failure or they failed. According to 5.4.2 all messages exchanged between participating VPMs are re-executed. Thus, there are no orphan or lost messages at the end of the recovery and the state including all the local states of the participating VPMs is consistent.*

With lemmas 5.4.1 and 5.4.3 we obtain the correctness theorem:

Theorem 5.4.1 *The rollback recovery protocol is correct in spite of single failures.*

Proof 5.4.1 *According to 5.4.3, the state of the VPMs that participate in a rollback recovery is consistent both after rollback and after recovery. According to 5.4.1 each VPM avoids all message sending operations from the rollback step to its failures step towards VPMs not participating in the rollback protocol. Thus, orphan message resending is avoided and the state of all VPMs after a rollback recovery is consistent.*

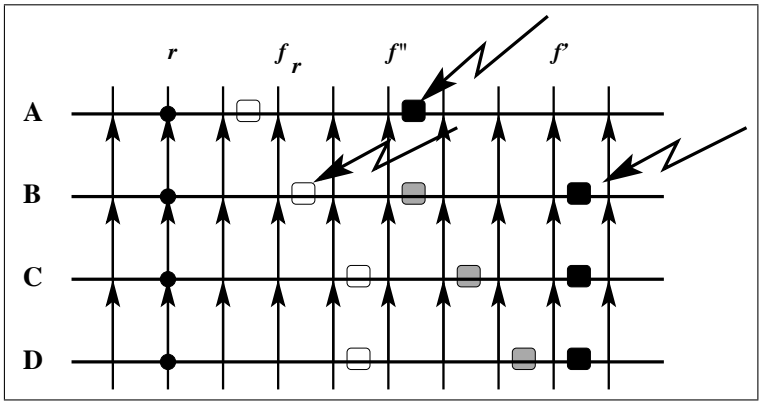


Figure 5.15: Example of concurrent and recursive failures during recovery, to introduce the concept of *recovery session*.

5.4.5 Surviving Concurrent and Recursive Failures

Here we extend the previous protocol to survive concurrent failures. In the protocol described in Section 5.3, failures during the rollback execution were managed as rounds of the same protocol, or sequentialized, depending on the time at which the first VPM detected them. Moreover, the recovery protocol was exactly equal to a full re-execution of the lost computational steps. Thus, failures during the recovery were managed in the same way as failures during normal execution. To extend the protocol for single failures we have to face three main problems: (a) not all VPMs participate to the protocol, (b) the recovery protocol is not a total re-execution of the lost computational steps, i.e. some communication operations are avoided, and; (c) the participants at each rollback recovery can change depending on the identity of the failed VPM. Consequently, each VPM needs to precisely characterize the participants in each protocol it performs, in order to understand which communication operations are needed. Figure 5.15 shows an example of two successive failures: black squares denote execution before any failures, gray ones execution during the first recovery and white ones during the second recovery. The VPM B fails at step f' and recovers to r. According to its $\mathbb{P}$ set, also C and D participate in the rollback recovery. During the recovery, from r to f', it will avoid send operations to A. We characterize this recovery protocol of B in a *recovery session* which includes: (1) the rollback step r ; (2) the failure step f ; (3) the list of VPMs to which perform send operations. For this recovery, this list is empty. Each participating VPM has its own recovery session instance for this recovery. During recovery, A recursively fails at step f'', forcing all other VPMs to rollback again to r, because they are all in its $\mathbb{P}$ set. In this second recovery we require B to perform the communication operations lost, because of the failure of A. To support this behavior we organize recovery sessions on a same VPM in a *stack* of recoveries (denoted with S) and we perform the following actions:

- Each recovery corresponds to a new recovery session that we push onto the top of the stack. In the example, when B rolls back for the second time, it pushes a new recovery session onto S, composed of the failure step f_r, the recovery step r and the list of neighbors on which to perform the send operations (includes just A). Notice that, differently from the previous session, we now have A in the send list.

- When a recovery terminates, we pop up the top of the stack of sessions, and we merge up the send list of the popped element, to the one at the top of S. In the example, when B returns back to f_r, it pops up the recovery session for the recursive failure on the top of S, it merges up the two lists obtaining the list including just A, and it assigns the list to the element at the top of S. From this point to the end of the recovery at step *f'*, B will perform send operations to A.

Consider the case of a recursive failure of the same VPM during recovery. Even if the same rollback step is selected, the failure step changes, and the information related to recovery sessions must be recovered. For instance, suppose in the previous example that B recursively fails during the second recovery protocol. A will not participate in this third rollback recovery. When B returns back to the second failure step it has to restart the recovery session including A. Functionally, we merge up send lists as in the above case, but we have also to support session stacks with stable storage to avoid losing information. Thus, the stack S is kept in stable storage, and its modifications are synchronously saved: (1) at the end of a rollback, the new pushed session is copied onto stable storage, and (2) at the end of a recovery, the pop of the top element, and (if it changes) the merge of the send lists are also copied onto stable storage.

We discuss the cases of failures during rollback recovery and we informally introduce the support to each case. Consider multiple failures during a rollback protocol: we exploit an example to describe the actions performed by VPMs on the stack of recovery sessions and in the rollback protocol. Suppose a VPM C is participant in a rollback protocol started by the failure of VPM A. In the rollback protocol A is the leader. In addition suppose that B fails during the rollback. We assume that $C \in \mathbb{P}(B)$ and we distinguish the cases w.r.t. the $\mathbb{P}$ sets of A and B:

- If $B \in \mathbb{P}(A)$ and $A \in \mathbb{P}(B)$, B was participating to the rollback protocol. We denote this case *rollback failure*. In this case, all participating VPMs decide a new leader by applying the same rule of the first protocol (see Section 5.3).

- Also if $B \in \mathbb{P}(A)$ and $A \notin \mathbb{P}(B)$, B was participating to the rollback protocol. In this case there can be a VPM, say D, for which $D \in \mathbb{P}(A)$ but $D \notin \mathbb{P}(B)$. We choose A as leader of the rollback protocol, because its $\mathbb{P}$ set is larger than the one of B. All VPMs participating in the protocol can select A as leader by checking for its $\mathbb{P}$ set. That is, the rule for deciding a leader change w.r.t. the previous point.

- If $B \notin \mathbb{P}(A)$ and $A \in \mathbb{P}(B)$, we select B as new leader of the protocol and we perform a further round, as we do in the above case.

- If $B \notin \mathbb{P}(A)$ and $A \notin \mathbb{P}(B)$, the management cannot be performed in common for the two events, as rounds of the same protocol. C participates independently to both rollback protocols. It acknowledges the leader of the protocols with its rollback step and it starts recovery from the selected step. Suppose that the rollback step decided in the rollback leaded by A is greater than the one of B. C first pushes a recovery session from the A decided rollback step. Next it pushes another session for the rollback step of B. It recovers according to the sequence of sessions.

The last cases we consider are those featuring $C \notin \mathbb{P}(B)$. We can assume that $A \notin \mathbb{P}(B)$, because, if we assume the contrary, we negate the hypothesis on C ($\mathbb{P}$ is transitive by its definition). We have only two cases:

- If $B \notin \mathbb{P}(A)$, the failure of B does not affect the execution of A and C both during normal execution and rollback recovery.

- If $B \in \mathbb{P}(A)$, C is not aware of its failure. According to the rule of above C continues to see A as its leader. Also A and B selects the former one as leader, because we have that $C \in \mathbb{P}(A)$ but $C \notin \mathbb{P}(B)$ (according to the modified rule as in the second point of the above ones).

Failures during a recovery protocol can induce further rollback and recovery executions. As stated above, each recovery has a different set of participants. We have to properly manage recovery sessions to reflect the correct behavior in different cases. In the same setting of above, first suppose that $C \in \mathbb{P}(B)$:

- If $A \notin \mathbb{P}(B)$ and $B \notin \mathbb{P}(A)$, C performs a new rollback recovery and it pushes a new recovery session onto the top of the stack for the protocol performed with B.

- If $A \notin \mathbb{P}(B)$ and $B \in \mathbb{P}(A)$, all VPMs in $\mathbb{P}(B)$ re-execute rollback recovery by selecting B as leader. A is not involved in this protocol.

- If $A \in \mathbb{P}(B)$ and $B \notin \mathbb{P}(A)$, a new rollback recovery is performed. In this case B is elected as leader of the new protocol.

- Also if $A \in \mathbb{P}(B)$ and $B \in \mathbb{P}(A)$, a new rollback recovery is started but the decision about the leader of the protocol is done according to the rule of the protocol described in Section 5.3.

Suppose now that $C \notin \mathbb{P}(B)$. Also in this case we cannot consider the case in which $A \in \mathbb{P}(B)$ because we negate the hypothesis on C. We have only two cases:

- If $B \in \mathbb{P}(A)$, A is elected leader from B and C independently by locally checking that $\mathbb{P}(C) \subset \mathbb{P}(A)$ and $\mathbb{P}(B) \subset \mathbb{P}(A)$
- $B \notin \mathbb{P}(A)$, no support is needed, as neither A nor C are notified of the failure.

5.4.6 Performance of Rollback Recovery

In this thesis we only give a qualitative description of the costs of the rollback recovery in the case of single failures.

The cost of performing the rollback protocol is similar to the previous protocol (see Section 5.3). In this case we have to instantiate the costs to the actual number of participating VPMs, which depends on the implemented stencil. Thus, it can be computed statically.

In principle, the cost of recovery for participants to the rollback recovery is lower than a complete re-execution of the lost steps (as it happens for the global rollback recovery protocol). This because some sending operations are avoided.

Finally, the cost of rollback recovery for VPMs which do not participate to rollback recovery is void, but they can incur in degradations: some of the VPMs from which they need to receive an element could be involved in (possibly several) rollback recovery protocols, which, in fact, increase the number of computational steps which execution is replicated.

5.5 Uncoordinated Checkpointing and Message Logging

The protocols presented above are based on the replaying of lost messages from their producers in the case of failures. In this section we show a protocol supporting failures with receiver-based message logging, i.e. whenever a VPM receives a message we ensure that it is copied onto stable storage. Message logging techniques can be safely exploited because, as envisioned in Chapter 1, the data parallel programming model features the piecewise determinism property. We discuss synchronous message logging: a VPM that receives a message has to wait for its logging onto stable storage before proceeding in the execution. In this solution a failed VPM performs rollback recovery in complete isolation without any interactions with the other ones. This comes at the cost of a higher overhead to the VPM execution w.r.t. the two previous protocols. During rollback a VPM re-obtains the last checkpointed state and it performs the recovery protocol replaying all messages from the rollback step to the failure one. We label messages with the I-Structure position of the state value from which they have originated, as in the previous protocols (recall that VPMs exchange sub-parts of local partitions). In this case the labels are exploited during recovery to replay messages in the correct order and not to manage duplications.

It should be noticed that this solution is not based on the consistency definition we introduced in Chapter 3 for data parallel programs. The pessimistic logging technique we exploit allow us to introduce uncoordinated checkpointing to save local VPM states. In this section we exploit the following consistency property:

Definition 7 *A consistent global state for a data parallel program includes the local states of VPMs taken at arbitrary steps, where each local state is attached also with the message received from the checkpointing operation to the current execution step.*

This protocol supports concurrent and recursive failures without any specific extensions:

- Recursive failures, i.e failure of the same VPM during rollback recovery, are managed as a re-execution of rollback recovery.
- Concurrent failures do not affect any other VPMs during rollback recovery. Thus, no special support is needed.

5.5.1 Checkpointing Algorithm with Message Logging

In Figure 5.16 we show a checkpointing algorithm with message logging. Unlike the previous algorithms (see 5.3 and 5.10) we do not need any coordinations of checkpointing operations. Each VPM has its own **MY-DELTA-CHK** value which denotes the number of steps between two successive checkpoints. At each loop we guarantee that all messages that a VPM receives are synchronously copied onto stable storage. In the case of rollback the last checkpointing step is passed to the procedures implementing the rollback recovery protocols. Notice also that we have modified the implementation of the stencil at the end of the loop (line 30) w.r.t. the previous solutions. The messages exchanged are labeled with the step value of the sender. Suppose we are at the beginning of step s. We assume that all receive operations are performed only after all messages for step s are copied on stable storage. We can ensure this atomicity property in the implementation of the communication support by providing proper localized functions for rolling back message receipt events. In the pseudo-code we model this support with lines 11 and 12: the **allStable** function checks if, for all channels in **chInList**, the messages for step s are copied on stable storage. Next the VPM can copy on stable storage also the reached computation step, i.e. the VPM is ensured that all messages are copied on stable storage for that step. In the previous recovery protocols the ordering of messages exchanged by VPMs was ensured by the synchronous execution of the rollback protocol: all VPMs participating in the rollback recovery started from the same rollback step. Thus, the messages exchanged between them were in the correct order. In this protocol, we label messages exchanged between VPMs to support message replaying during recovery in the correct order. This task is relegated to the failed and recovering VPM and it is not a consequence of the coordination between

```
1   partition myPart;
2   partition ghosts[];
3   int step = 0;
4
5   while(!term) {
6     chInList = stencilInNeighChannels(step, myid);
7     int i = 0;
8
9     //waits for all message to be copied on stable storage
10    while(allStable(chInList) != true) {}
11    stst-checkpoint(step);
12
13    for each ch in chInList do
14      receive(ch, &ghosts[i++], step);
15
16    stst-checkpoint(ghosts, step);
17
18
19    myPart = F(myPart, ghosts);
20
21    step++;
22    if(step % MY_CHK_DELTA == 0) {
23      stst-checkpoint(myPart, step);
24    }
25
26    chOutList = stencilOutNeighChannels(step, myid);
27    i = 0;
28    for each ch in chOutList do
29      send(ch, pair(ghost(myPart,ch)),step));
30  }
```

Figure 5.16: Checkpointing algorithm with message logging: each VP periodically saves its local state partition on stable storage, and the ghost partitions received from its neighbors.

functionally dependent VPMs. Also notice that we do not check for communication errors because we do not need to consider concurrent failures. In the case a VPM fails we assume that the communication support blocks the sender VPM to perform further send operations to the same destination until the receiver is restarted. Next the communication channels of the restarted VPM are cleared. They can be filled up of new application messages which are still unreceived in the execution performed before the failure. We ensure that no messages are lost because of the synchroniza-

```
1 failedAndRestarted(name = i, failure step = f, rollback step
    =
2   r) {
3     myPart = stst-recover(r);
4     recover(i,r,f);
5     restart from f;
6 }
```

Figure 5.17: Pseudo-code of a failed and restarted VPM for the rollback recovery protocol based on message logging.

```
1 recover(i, r, f) {
2   partitions ghosts;
3   for (step = r+1; step <= f; step++) {
4     ghosts = stst-recover-message(step)
5
6     myPart = F(myPart, ghosts);
7   }
8 }
```

Figure 5.18: Pseudo-code of the recovery protocol, based on message logging.

tion of message logging. When the failed VPM terminates the recovery it re-starts receiving messages from the newly created communication channels.

5.5.2 Rollback Recovery Protocol

The rollback recovery protocol is performed in isolation from failed VPMs, without any coordinations. Consequently, no control messages are exchanged between VPMs and the rollback recovery protocols are implemented as a procedure locally performed on failed VPMs. The failed VPM is provided with its name, the step at which it failed, and the step of its last checkpointed state (see Figure 5.17). The VPM recovers the state at step r, and performs the recovery protocol. Next, it restarts its execution from the failure step.

In the recovery protocol, shown in Figure 5.18, a VPM accesses the stable storage at each re-executed step to recover messages. The **stst-recover-message** functions takes the current step value as input, and it returns the set of messages copied onto stable storage (labeled with the current step). Notice that no communication with other VPMs is performed during recovery.

We discuss the atomicity feature of communication operations in the VPM program loop (see Figure 5.16). After a receive operation we guarantee that the message is copied onto stable storage. Thus, if a failure happens during the set of receive

operations, some messages are copied onto stable storage, some others not (the ones not yet copied on stable storage). We assume the communication protocol guarantees the copy of such messages in the application channels after recovery. At the last step of recovery the VPM will avoid all send operations: we require the set of send operations to be atomic w.r.t. failures (see Chapter 8). This can be implemented by exploiting attached computation steps, in a similar way to the previous protocol (see Section 5.4).

We can support synchronous message logging with synchronous FT-Streams by modifying their support w.r.t. the one described in Section 4.4 and Chapter 6. For synchronous FT-Streams the sender is synchronized with the message logging process, i.e. a send operation terminates after the logging is performed. In the current implementation a message can be copied onto the application channel (the one linking the sender to the receiver), even if the message logger has not yet copied it onto the stable storage. Consequently, the receiver can obtain a message that is still unlogged. We want to avoid this situation, as we assume the receiver can independently re-play messages during recovery. In the modified version of the FT-Streams, the receive operation blocks until the message is copied onto stable storage. This is implemented by the *KP* on the receiver node, which copies the message onto the application channel *only after* the message has been logged onto stable storage.

5.5.3 Correctness of Checkpointing and Rollback Recovery

From the description of the checkpointing algorithm (Figure 5.16) we can notice that each VPM takes checkpoints at a possibly different step. That is, there is not a property that guarantees the consistency of a set of checkpoints. The rollback recovery protocol is performed in isolation, i.e. it is not required to VPMs to coordinate during the execution of the protocols. Informally speaking, the rollback protocol selects as a target a state that is inconsistent w.r.t. the set of all other states of VPMs because it generates orphan and lost messages. Orphan messages are those sent from a failed VPM between the rollback step (plus 1) $r+1$ and the failure step f to any other VPMs. Lost messages are those received from the failed VPM in the same steps. The recovery protocol manages orphan messages to re-build consistency of global states. Lost messages are regenerated by retrieving them from stable storage. If we consider a failure at step f and a recovery from step $r+1$ of a VPM we can assume the followings:

Lemma 5.5.1 *All send operations from $r+1$ to f are avoided. All messages received from step $r+1$ to step f are replayed from the stable storage.*

Proof 5.5.1 *Obvious from the recovery protocol.*

This is also true for recursive failures because the same protocol is re-executed. We can consider the consistency of recovered states with this lemma:

Lemma 5.5.2 *After an arbitrary number of concurrent and recursive failures and successive rollback recovery executions the states of VPMs are consistent w.r.t. each others.*

Proof 5.5.2 *To prove this we have to prove that the states do not include orphan messages. Suppose that an arbitrary number of concurrent and recursive failures affect the computation. After all recoveries, which are performed locally by all failed VPMs, lemma 5.5.1 ensures us that no sending operations previously performed will be performed again. Thus the states of recovered VPMs include the message sending operations performed at the first execution of the recovery steps for orphan messages. Thus no orphan messages are included in the states of VPMs.*

5.5.4 Performance Impact of Message Logging and Checkpointing

We can model the cost of message logging on the performance of VPMs in the same way that we did for the first two solutions (see 5.3.5, and 5.4.2). Unlike the previous checkpointing algorithm we admit uncoordination of checkpointing frequencies to VPMs. Thus, to obtain the general formula for the completion time, the formula for the step performance should be properly multiplied for each VPM checkpointing frequency. The performance of steps if we do not perform the checkpointing of the local state is:

$$T_{step-ml,i} = T_F(g) + T_{stenc-in}(\overline{g}, i) + T_{stenc-out}(\overline{g}, i) + T_{chk}(sizeof(int)) + T_{chk}(ghost(i))$$

In this formula we denote with $ghost(i)$ a function that, given a step identifier, returns the size of the ghost partitions received from neighbors at that step. Notice that, under the hypothesis of synchronous (i.e. pessimistic) logging, the cost of the *allStable* function is included in the $T_{stenc-in}$ quantity.

The step performance in the case of checkpointing is a simple extension of the above one as in the previous solutions:

$$\begin{aligned} T_{step-chk+ml,i} = \quad & T_F(g) + T_{stenc-in}(\overline{g}, i) + T_{stenc-out}(\overline{g}, i) + \\ & + T_{chk}(sizeof(int)) + T_{chk}(ghost(i)) + T_{chk}(g) \end{aligned}$$

5.5.5 Performance of Recovery

For this protocol it is simple to derive a formula describing the cost of a single recovery for a VPM. Suppose we are recovering n steps from step r to step f. Suppose also that the cost of a single access to stable storage for a checkpoint of k words is $T_{st-read}(k)$. We can also suppose to be provided with a function *ste-size* which returns the total size of all messages received from a VP at a given step.

The total time needed to perform the recovery protocol is:

$$T_{recovery}(r, f) = \sum_{i=r}^{f} T_F(g) + T_{st-read}(ste - size(i))$$

where g is the average size of the state partition assigned to a VPM.

5.6 Comparison with Structure-Unaware Protocols

In this section we consider a protocol from each class presented in [39] and we apply it to data parallel computations. When it is possible, we derive a static analysis of the performance of these protocols and we compare it with the ones we introduced in this thesis. While performance comparison between protocols is our main goal in this section, we also consider each specific issue introduced in Chapter 1 to discuss the differences between the protocols. It is worth of re-write these issues: *analysis of performance*, *consistency definitions*, *determinism of computations*, *modularity and composability* and *experimental results*.

In the discussion we take the point of view of the implementer of the fault tolerance support starting from a structure-unaware basis: data parallel programs are implemented as a set of processes, interacting through message-passing. In deriving fault tolerance supports, we do not exploit high-level structural information.

5.6.1 Coordinated Asynchronous Checkpointing

We consider the Chandy-Lamport[27] protocol as a well-studied representative of this class of protocols. In this protocol, messages are logged with local states during the checkpointing operations. The global coordination is obtained in an asynchronous way: processes are not blocked in the computation during this phase. The protocol is based on the exchanging of *markers* on application channels: when a marker is received on a channel proper actions related to checkpointing are performed. In fact, markers are the start signal of the checkpointing protocol. This technique based on markers can be applied only if each pair of processes has some communication path which link them. Otherwise, the protocol cannot terminate. This property is true for a large part of data parallel programs, but not for all of them. For instance, in *map* programs there is no application channel between any processes. This protocol is based on a different definition of consistent states w.r.t. the one that we presented in 3:

Definition 8 *A consistent state for a parallel program includes a checkpoint from each process. Moreover, for each process we have to add to the checkpoint all the messages received during the execution of the checkpointing protocol.*

The protocol we present here is based on a single coordinator and it is a slightly modified version of the one originally presented in [82]. Unlike the original protocol, which assumed a crash-stop failure model, in the one we present here we assume the fail-stop model. The protocol is started from the coordinator with a broadcast of a special control message. Next, it waits for all other processes to answer back to its message. A generic participant notifies to all its neighbors that it is participating to the rollback and it waits for a same message from all of them. During this phase, all the application messages are attached to the checkpoint. When all neighbors have answered back, the participant acknowledges to the coordinator its local termination. After receiving such acknowledgment from all other processes, the coordinator broadcast a control message to restart the computation.

To implement this behavior the coordinator performs the following actions:

1. Send a "checkp-start" message to all other processes.
2. Take a tentative checkpoint.
3. Receive messages on all its input channels and checkpoint the content as part of the current checkpoint until it receives an "ack" message from all other processes on the same channels. Notice that the answer from participant processes is sent along application channels or, in the case no application channel is placed between a process and the coordinator, on the rollback one.
4. Send a "restart" message to all other processes.

For comparison purposes, we have also avoided to manage reincarnation of this protocol due to recursive failures and we have assumed that the coordinator directly contact the whole set of processes of the application, instead of implementing an optimized neighbor-based protocol. The behavior of a generic participant to this protocol follows:

1. Check for the rollback channel at each computation step and eventually receive the "checkp-start" message.
2. Send a "checkp" message to every process for which it has an application channel.
3. Take a tentative checkpoint.
4. Receive messages from application channels and append them to the current checkpoint until a "checkp" message is receive on each channel.
5. Send a "ack" message to the coordinator (choose the rollback channel if no application channel towards the coordinator is available).
6. Receive a "restart" message from the coordinator, confirm the checkpoint and restart the computation.

In this description notice that we have to synchronize each participant process with the neighbor processes for which it has an application channel. This cannot be avoided because it implements the correctness of the checkpointing protocol w.r.t. the consistency definition for global states.

We have discussed above the consistency property on which is based this protocol. We now consider the remaining issues. By looking at the protocol of above we can see that the overhead incurred because of checkpointing on each process depends on their relative speed. The costs of checkpointing can be subsumed as following: the cost of taking a checkpoint (stable storage access) plus the cost of the broadcasting of the "checkp-start" message. Next, specific operations are performed before the termination of the protocol: application messages are logged and, when all processes have answered with an "ack" message, a broadcasting of a "restart" message is needed. The costs of local checkpointing and broadcasting are fixed, while the remaining costs depend on the relative speeds of processes and on their causal dependencies built during checkpointing. As a consequence it is difficult, when not impossible, to estimate the amount of logged messages during global checkpointing because at this level of implementation we do not have any information related to the parallel structure of the computation.

For comparison purposes we describe a coordinated protocol for data parallel programs featuring the same characteristics of this one for which we can analyze its costs. This analysis is possible because of the knowledge of the structure of the parallel computations. Next, we compare this protocol with the ones we described above in this Chapter. In this way we can compare the Chandy-Lamport protocol with the ones that *can* be derived according to our approach.

The protocol exploits the knowledge of computation steps of data parallel programs and it is based on the consistency definition given in Chapter 3. The purpose of this protocol is to select the maximum computation step between all processes and to roll-forward all of them to that point. The execution of the processes is blocked only when they reach the maximum step, while waiting for all other ones to reach it.

The protocol includes an exchanging of control messages between a coordinator and all other processes to decide which is the maximum execution step. This phase requires three control messages. Next all processes reach the maximum step and they all notify to the coordinator that they are ready to restart. The coordinator restarts the execution by broadcasting a proper control message.

The actions of the coordinator follow:

1. Send all other processes a "checkp-start(step)" message indicating its current execution step.

2. Receive a "ack(step)" message from all process.

3. Select the maximum between the received step values.

4. Send the computed maximum to all other VPMs with a "max(step)" message.
5. Proceed in the computation until the selected maximum step.
6. Wait for an "reached" message from all other VPMs.
7. Take a checkpoint.
8. Send a "restart" message to all other VPMs and restart the computation.

The behavior of a generic participant to this protocol is:

1. Check for the rollback channel at each computation step and eventually receive the "checkp-start(step)" message.
2. Select the maximum between the current step and the received one.
3. Send a "ack(step)" message to the coordinator indicating the selected maximum.
4. Receive the "max(step)" message from the coordinator.
5. Execute until the received maximum step (if needed).
6. Take a checkpoint.
7. Send a "reached" message to the coordinator.
8. Receive the "restart" message from the coordinator, confirm the checkpoint and restart the computation.

For performance analysis purposes, notice that the distance, in terms of steps, between the process at the minimum step and the one at the maximum one influences the blocking time for the latter process. Thus, unlike the previous protocol, this one partially blocks the computation. The maximum distance between VPMs can be computed by statically analyzing the program graph: for map programs it cannot be limited, thus the solution presented in 5.4 is more efficient. For step-synchronous programs, the difference between steps is 0, and the cost of synchronization is equal to the time needed to perform a single computation step. For other programs (for instance, see Figure 5.15) it depends on the asynchrony degree of communication channels which, according to our framework (see 6), can be controlled. Thus, we can conclude that structure-based protocols allow us to statically analyze fault tolerance techniques, while this is not possible for structure-unaware protocols.

Finally we discuss the scalability of applying the Chandy-Lamport protocol to a whole application, instead of exploiting it modular definition which is part of the structural information in our study. The Chandy-Lamport protocol does not blocks processes during the computation. Anyway, it requires a global communication,

which in the original work could be done not by a global broadcasting of information, but by exploiting the process communication graph. Thus, the time needed to perform a global checkpoint, which influences the cost paid by each process during the checkpointing phase, depends also on the number of processes in the system. An experimental session should analyze the scalability of this solution w.r.t. the number of processes and compare it with the coordinated protocol we introduced in Section 5.3.

5.6.2 Communication-Induced-Checkpointing

In Chapter 2 we have seen that CIC protocols are based on the Z-theory[68] to define consistency properties for global states. In the model of CIC protocols processes take independent checkpoints and the protocol induces some forced checkpoints which guarantee the building of global consistency. The Z-theory allows the fault tolerance designer to derive useful predicates which can be used by each process to locally decide if it is needed to take a checkpoint (forced ones). This evaluation must be done before any message delivery event and it is applied to local control information of the process and to control information appended (or piggybacked) on application messages by the senders. These checks can be done only at run-time because computations are nondeterministic both in communication and computation [39]. There exist many CIC protocols, from more simple ones [17] to more complex ones [52]. In [52] it is introduced a theoretical framework which can be used to derive a class of CIC protocols.

CIC protocols are characterized between each others w.r.t. the predicate to be evaluated when a message is received. This depends on the kind of information kept on the process and piggybacked to application messages. For instance in [52] a process receiving a message needs to check for maximum values inside received arrays of remote timestamps. The array sizes are equal to the number of processes, powered to two.

Rollback recovery supported by CIC protocols are themselves characterized w.r.t. the specific checkpointing protocol. In some cases it is required a selection of a recovery line different from the last checkpoint taken by each process. In some other cases, the last checkpointing line can be selected as the checkpointing protocol guarantees its consistency [39].

We compare CIC protocols with the ones that we introduced in this thesis, according to the issues introduced in Chapter 1. The kinds of data parallel programs which we considered in this thesis are the ones featuring *static stencil*. In these programs the programmer specifies at the abstraction level the needed interactions between virtual processors. At the implementation level this can be mapped in a graph of interactions between the processes implementing the application (in our terminology, between virtual processor modules implementing the virtual processes). Indeed these graphs of interactions can be properly transformed in graphs of dependencies between implementation processes. This allows us to derive a con-

sistency model for the computations which is exploited by checkpointing algorithms inserted directly at compile-time without the need of sharing information between processes. That is, application processes can decide whenever to take a checkpoint only by using *local information*, without the need of exchanging control messages or to piggyback application messages. As a consequence, we can statically define the costs incurred by the checkpointing algorithm and we can control these costs. On the other hand, for CIC protocols the frequency of forced checkpoints on each process depends on: the frequency of local checkpoints, taken independently; on the interactions between processes. Typically, the design of a CIC protocol requires the minimization of forced checkpoints or, if possible, the control of their frequency. The problem behind the performance characterization of CIC protocols according to this metric is highlighted in [52]: it is not possible to estimate the number of forced checkpoints taken by a CIC protocol because this depends on the causal dependencies built at run-time, which depend on the relative speed of processes and on the local checkpointing frequency. The latter factor can be, in the most general case, a *nondeterministic event*. As a consequence, some literature works tried to analyze the failure-free performance of CIC protocols by means of statistical analysis (e.g. see [61]). The goal of knowing the number of checkpoints taken by a protocol and to possibly control it before the application is executed is obtained in this thesis for the checkpointing and rollback recovery protocols we have defined in the previous sections.

CIC protocols are performed by the whole set of processes. In more complex cases the information piggybacked on application messages includes data structures which size is equal to (or depends on) the number of processes. This is a clear limit to the scalability of these protocols: in our approach a parallel program is composed of a set of parallel and sequential modules in a stream-based graph. Each module is supported by its own protocol, which can be chosen also to optimize its mapping on the execution platform. The different protocols and techniques applied on each module are composed together to target global correctness. In Chapter 6 we have introduced an optimized technique based on message logging to support the composition of fault-tolerance of different modules. They can be also composed together to meet some global constraints (e.g. some QoS) related to the costs of checkpointing and rollback recovery and, as a consequence, to the performance of the application. In some simple cases the information is not a source of overhead, but the related protocols force too many checkpoints [52]: it is the case of the CIC protocol which forces to take a checkpoint before every message is processed.

5.6.3 Message Logging

As we described in Chapter 2, message logging techniques can be classified according to three flavors:

- *optimistic* message logging, in which messages are not synchronously copied onto stable storage whenever received, but some buffering is exploited;
- *pessimistic* message logging, in which message copying onto stable storage is done synchronously, by blocking the computation;
- *casual* message logging, which is a trade-off between the previous two techniques and it is based on partially saving dependency information about processes on the volatile memories of processes.

While pessimistic and casual message logging are domino-effect free techniques, optimistic logging is not: by exploiting this technique it is possible to rollback to the very beginning of the computation. This is due to the possibility of incurring in a failure while one or more messages are not yet logged on stable storage and if such messages cannot be recovered.

We have seen that message logging techniques base their consistency properties on the piecewise determinism assumption (PWD, see Chapter 2). According to this assumption, processes local computation is deterministic: given the same inputs, the same outputs and actions are produced. Nondeterminism happens in input communications: in the case in which multiple messages from multiple senders are ready on an input channel, one of them is nondeterministically selected and delivered to the application [4]. This is valid for any process of the computation

In our model we assume the piecewise nondeterminism property but, unlike what happens in unstructured models of parallel programming, we relegate nondeterminism in message receive events in specific points of the programs. Nondeterminism can be expressed in the choice of the input from which receive a message, for generic parallel modules (e.g. implementing a farm o a data parallel program). Inside any parallel module *communications are deterministic* for data parallel programs, or follow a well-defined pattern (e.g. they implement a pipeline semantics, as in the farm case). Consequently, we can exploit these properties in supporting fault tolerance. This assumption cannot be done in structure-unaware programming models: any communication point is assumed to be nondeterministic.

In the previous sections of this chapter we have studied only solutions which avoid the domino-effect. Thus, we cannot compare the optimistic logging techniques with ours *if we apply it to the whole computation*. The comparison can be done if we limit the extent of optimistic logging to a subset of the application modules. Then we have to guarantee that all causally precedent nodes in the program graph are domino-effect free and that they can recover lost messages. In this way the whole fault tolerance of the application is domino-effect free. This solution, which study is demanded to future work, exploits the knowledge of the interconnections, by means of stream, between parallel and sequential modules of our programming model (see Chapter 3).

In Section 5.5 we have implemented a pessimistic logging protocol: whenever a VPM receives a message, it is logged onto stable storage. VPMs perform an unco-

ordinated checkpointing protocol and during recovery messages are recovered from stable storage. The consistency definition on which is based this solution is the same of pessimistic logging protocols (see Section 5.5). We discuss the possibility of analyzing the performance in the case in which we do not have structural information. Suppose we are supporting data parallel programs with fault tolerance at the level of processes communicating through message-passing. We do not know that the process programs implement a data parallel computation. We exploit uncoordinated checkpointing on each process: differently from our solution, we do not have "hooks" in which we can abstractly characterize the state of each process, which correspond to the beginning of the data parallel loop (see Figure 5.16) in our protocol. We have indeed to save the whole state of the process (data and code sections, information related to the operating system). The checkpointing operations on each process can be triggered, for instance, by some time-based mechanisms. Thus, we *statically* know the frequency of checkpointing on each process, but the information is stochastic. We extend checkpointing with pessimistic message logging: at compile-time we do not know how much messages will be passed between processes, because we do not have the information on the dependencies between processes. As a consequence, we cannot obtain a formula expressing the total costs incurred because of checkpointing *and* message logging: this information will be only available at run-time.

In more general terms, if we cannot derive an expression of the overheads incurred because of this fault tolerance technique we cannot decide if pessimistic message logging is well-suited depending on application- and platform-specific information. This can be done by exploiting our framework, by simply computing the costs of each VPM during failure-free executions.

We also consider causal logging protocols. These are based on recording on the volatile memories of multiple processes the dependency graphs of the computation (actually antecedence graphs), by replicating it as piggybacked on application messages. Piggybacking can enjoy incremental techniques, for which the size of the piggybacked information does not increase with the number of dependencies during the computation. In our methodology of supporting fault tolerance, the antecedence graph of processes (VPMs) is known at compile time, as it is specified as part of the program itself. The determinants of nondeterministic events are: in part represented in the stencil graph and are part of the distributed programs of processes; in part collected on stable storage (or possibly multiple volatile memories) during the execution. For instance, the protocol described in Section 5.4 saves on stable storage the reached computation step at each program iteration. This information, both with the stencil knowledge, is sufficient to re-build correctly the computation graph of the lost computation steps. As a consequence, the solution presented in Section 5.4 can be seen as a causal message logging protocol, in which we optimize the information that must be saved on stable storage during the execution. The main difference with this approach and the one of structure-unaware programming models is that we know statically the graph of dependencies between processes. Thus, we can exploit this information to introduce proper actions *during compile-*

time and, as a consequence, we can study the impact on the failure-free performance before the applications are actually executed. Also in this case, static performance analysis is a key point in the comparison with structure-unaware solutions.

Message logging techniques for structure-unaware programming models are well-suited to meet the needs of large applications, both in sense of the number of used processes and their distribution on execution platforms. In particular causal logging seems to optimize the trade-offs of communication and computation required by these kinds of applications [4]. To evaluate this issue we should define an experimental session which goals are to compare our modular approach to parallel programming with structure-unaware ones. The experimental configuration should be defined in the following way:

- define an application (or more applications) composed of several parallel and sequential modules composed by means of streams. Apply our message logging protocols, possibly exploiting optimistic logging for some nodes of the graph. The choice of the techniques should be guided by the cost models of the protocols to target optimizations for specific application and execution platform parameters. Finally, test the implementation to prove the correctness of the results of the exploitation of the cost models;

- define the same application(s) of the previous point according to a structure-unaware programming model. Support the whole application(s) with pessimistic and casual logging protocols and test the implementation.

We demand the implementation of this testing experience and the comparison of the numerical results to future work.

Chapter 6

Implementation of the Fault-Tolerant Stream

In this chapter we show an implementation of the Fault Tolerant-Stream (or FT-Stream), which we preliminarily described and exploited in Chapter 3. We first introduce some preliminary mechanisms related to interprocess communication on a same machine and on remote ones. The FT-Stream implementation is presented as a composition and extension of these mechanisms. This implementation is not done on actual computing platforms, but it is based on the preliminary concepts which we introduce in the next section. This choice is done because our aim is to perform analytical studies. We address in future work actual implementations and experiments (see Chapter 8).

6.1 Preliminaries

We introduce some preliminary notions that are used in the description of the FT-stream. This section discusses the following points:

- How synchronous interactions between processes in a same computational node are implemented. The aim is to minimize the number of information copied during this operation.
- The communication protocol without fault tolerance support in a distributed environment, i.e. for processes running on different nodes.

The following subsections analyze separately each point.

6.1.1 Efficient Interprocess Communications on a Same Node

We consider the case of message exchanging between two processes on a same computational node. The communication is synchronous and avoids any copies of the content of the message (0-copy property).

In the most general case, suppose two processes are respectively executed on the main processor (say process A) and on an auxiliary one (say process B). A passes information to B through the interruption mechanism. Our goal is to minimize the quantity of information copied in this process. We assume the processes can access a same shared memory support. To implement the communication we make processes pass references to the variables they want to communicate. We illustrate a possible

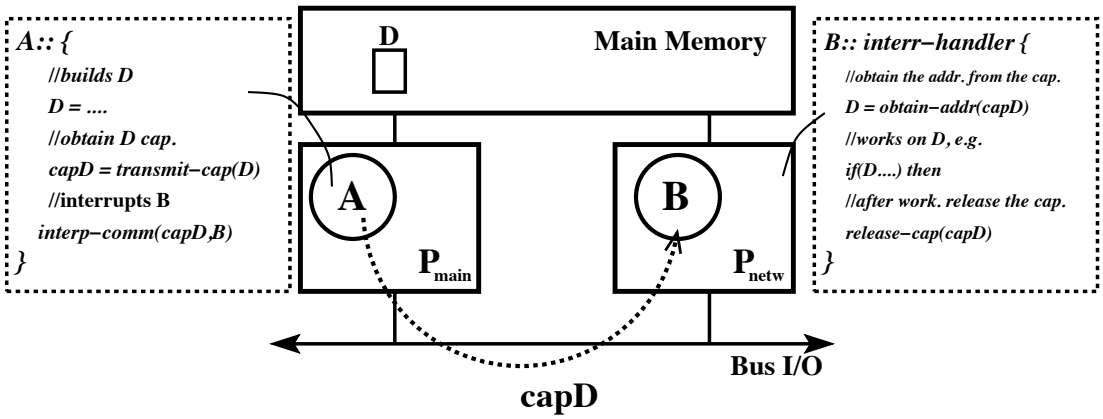


Figure 6.1: Example of synchronous communication between two processes on a same node.

instance of this kind of communication in an example. In Figure 6.1 we show the communication of a data structure D from A to B. A is executed on the main processor P_{main}. B on another one P_{netw} and it is specialized in the implementations of communications between different nodes. B is responsible of managing input and output communications and it is connected directly to the network interface card (NIC in short). The communication protocol is the following: A builds the data structure D by filling up its fields. Next it obtains a reference to D (that we call *capability* of D) and it copies it in a variable, say *capD*. The function that performs this task is called *transmit-cap*. Finally A passes B the reference *capD* through the interruption mechanism. We denote this operation with the function *interp-comm*. When B receives the interruption from A it performs the corresponding handler and (possibly in the main loop, or in the handler) it uses the reference *capD* to D to address it. The function that makes this task is called *obtain-addr*. After working on D B releases its capability (*release-cap* function). This last step implements the minimal privilege mechanism for which any variables cannot be accessed unless explicitly authorized for the minimum possible time required. That is, A authorizes B to access D and when the access terminates B has no more privilege to access it. Notice that the protocol avoids to copy the variable D during the communication between A and B.

6.1.2 Communication Protocol without Fault Tolerance

Suppose two processes S (for sender) and R (for receiver) are executed on different nodes (N_S and N_R respectively). We assume that there is no an external shared memory support. We show the implementation of the communication support (as send and receive primitives) in the case without fault tolerance support.

The channel *CH*, for the communication from S to R, is mapped in the receiver node N_R. On N_S a smaller data structure CH_{rem} is exploited for the interactions between S and its local communication process. We will name *KP* the communication process on each node, i.e. the process implementing communications. Thus on N_S we exploit a KP_S and on N_R a KP_R. Messages from S to R are typed and are copied only in the channel *CH*. CH_{rem} does not contain the messages to be sent. We notice that this choice is a point in the differences with the model on which message logging protocols are based (see [79]). In that model there is a message buffer on both the sender and the receiver nodes.

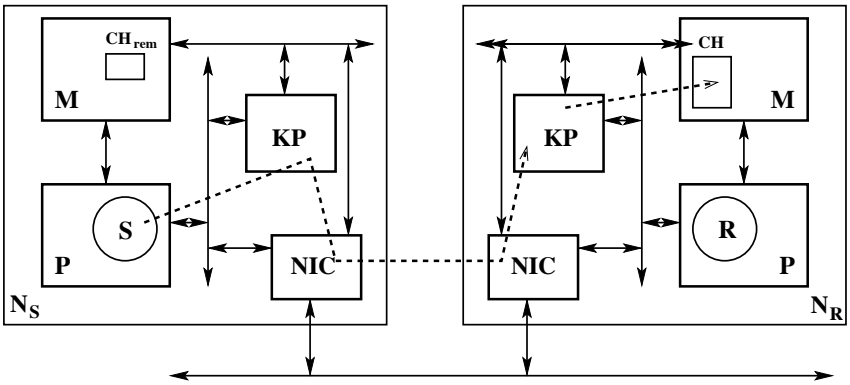


Figure 6.2: Architecture of N_S and N_R nodes. The *KP* on the nodes are mapped on a special hardware, and implement the communication protocol. The dotted line shows the path of a message sent from S to R.

Figure 6.2 shows the entities implementing the communication and their mapping onto the nodes. The communication protocol is composed of the behaviors of S, KP_S, KP_R and R. We describe each behavior according to their sequence in the protocol. In the example we have mapped the *KP* processes in a special hardware on each node. Clearly, this is not mandatory and the protocol works also in the case *KPs* are executed on the main processor. In the rest of the chapter we will assume to be supported with this special hardware. We also assume that each node is provided of a Network Interface Card (NIC) that implements the lower levels of the networking protocol. The NIC in a node can access the main memory of that node through the Direct Memory Access (DMA) mechanism. In the description of the communication protocol and its data structures we make use of the following terms for the states characterizing a process execution:

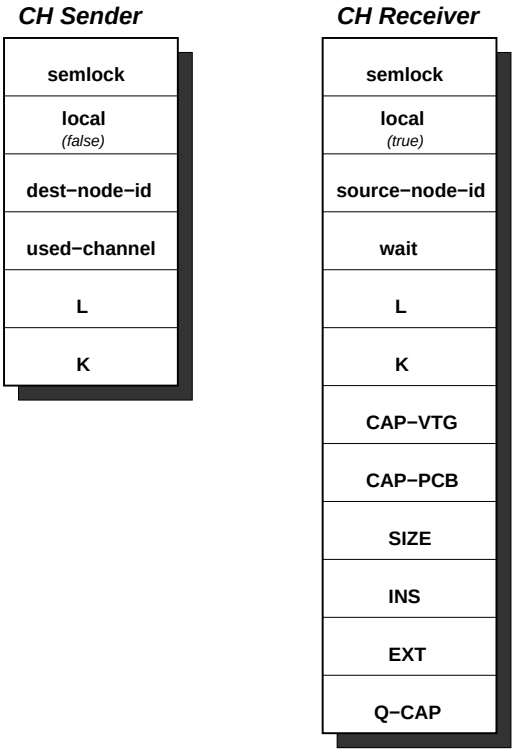


Figure 6.3: Data structures implementing the communication channel on the sender and receiver, respectively. The message buffer is only on the receiver side.

Running the process is in execution on one of the available processors.

Waiting the process is waiting for an event to happen. For events related to the communication we make use of specific macros, which we describe below.

Ready the process is ready for being executed on one of the processors.

Figure 6.3 shows the fields of *CH* and CH_{rem}, i.e. the data structures implementing the communication channel on the receiver and on the sender, respectively. A description of their fields follows. CH_{rem} contains:

semlock this is the semaphore used to support concurrent accesses on the data structure from KP_S and S.

local this is a boolean variable indicating if the channel is local or remote. It is used in the case we want to implement the message queue on the sender and on the receiver. As we have decided that the message queue is mapped onto the receiver space this variable is always set to false in this data structure.

dest-node-id this is the identifier of the node executing the receiver process. It is used by the lower layer routing support to obtain the N_R network address.

used-channel this is used to synchronize S and KP_S in successive send operations.

L this is the length of messages exchanged on this stream (recall that communication channels are typed).

K this is the asynchrony degree of the channel, which indicates the maximum number of messages that can be placed in the message queue. In the case in which the queue is filled up of $K+1$ messages the sender blocks until the next receive is invoked.

CH contains:

semlock this is the semaphore used to support concurrent accesses on the data structure from KP_R and R.

local with the same meaning as above, here it is set to true.

source-node-id this is the identifier of N_S. It is used by the lower layer routing support to obtain its network address.

wait this is a boolean indicating if one of the two parties is waiting for the other one.

L this is the length of messages (equal for all messages).

K this is the asynchrony degree of the channel (see above).

CAP-VTG this is the capability of the target variable, which is used in the case the receiver waits for a message (see below).

CAP-PCB this field can contain the Process Control Block (PCB) of the sender or the receiver, in the case in which they transit in the waiting state.

SIZE this is the number of messages in the queue.

INS this is the index of the next free position in the queue.

EXT this is the index of the next position in the queue from which extract a message.

Q-CAP this is the message queue. It can contain up to $K+1$ messages: in the case the queue is filled the next send will block the sender. Before transiting into the waiting state the support copies the message in the $K+1$ position. In this way, when the sender is woken up, it is not needed to copy the message.

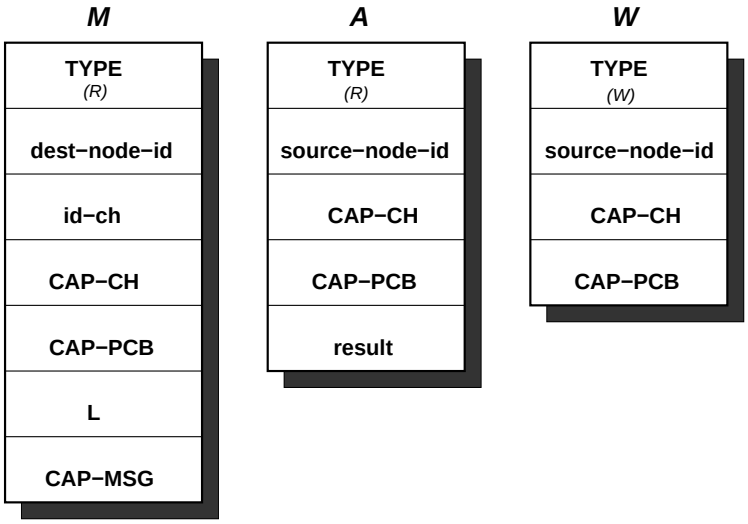


Figure 6.4: Data structures used to implement the communication protocol. The **M** data structure is used to implement the send operation, and it is used between S, KP_S and KP_R. The **A** data structure is used to notify KP_S of the result of the local send on the receiver. The **W** data structure is used from R to wake up S, when the receiver removes a message from a full channel queue.

Figure 6.4 shows the data structures passed between the processes to implement the communication protocol. All data structures contain a **TYPE** field, which contains the type of the message. It is used by *KPs* to select the proper actions when a request is received.

The **M** structure is used to pass a message from the sender to the receiver. It contains:

dest-node-id this is the identifier of the destination node. It is used by the NIC on the sender node to implement the routing protocol, i.e. to address N_R.

id-ch this is the identifier of the channel in which to place the carried message. KP_R uses this value to obtain a reference to the channel[1].

CAP-CH this is the capability of CH_{rem}, which is used for the management of control messages from R and KP_R to KP_S.

CAP-PCB this field contains the capability of the PCB of S and it is used to implement process operations, like waiting and wake up ones.

L this is the length of the message.

CAP-MSG this is the capability of the message.

[1]Recall that *KPs* implement all the communications incoming to an outgoing from a node.

The **A** structure is used to transmit the send result from KP_R to KP_S. It contains:

source-node-id this is the identifier of N_S, which is used for routing purposes.

CAP-CH this is the capability of CH_{rem}.

CAP-PCB this is the capability of *S*.

result this field contains the result of the send operation.

The **W** data structure is used from *R* to wake up the sender in the case it goes into waiting status, i.e. when up to $K + 1$ unreceived messages are placed in the message queue. It contains:

source-node-id this is the identifier of N_S, which is used for routing purposes.

CAP-CH this is the capability of CH_{rem}.

CAP-PCB this is the capability of *S*.

Refer to the next paragraphs for a complete explanation of the meaning of the channel and message fields.

S Sends a Message to R *S* calls the send primitive with the following parameters: *send(ch,m)*, where *ch* is the identifier of the channel on which sending the message and *m* is the message to be sent. The following operations are performed:

- *S* acquires the lock on CH_{rem}.
- It checks if the channel is used, i.e. if the previous send operation is terminated. If the *used-channel* field is true *S* sets its state in its Process Control Block (PCB) to **SPEC-WAIT**[2], releases the lock on the channel and it transits in the special waiting state. It will be moved back to the ready queue by the local KP_S when the result of the previous send will be received (see below). When *S* is woken up it automatically re-acquires the lock on the channel.
- Either if the CH_{rem}.used-channel field is false or after *S* is woken up it continues the communication protocol. It sets the used-channel field to true and it releases the lock on the channel.
- *S* fills up a data structure **M** (see above) with: (1) the identifier of the destination node, which is used for routing purposes; (2) the identifier of *CH* used by KP_R to access the channel data structure; (c) the capability of its PCB; (d) the capability of CH_{rem}, to implement the send result notification and

[2]**SPEC-WAIT** denotes that the process is waiting for the previous communication on *CH* to terminate. We denote that the process is waiting for other events with the **WAIT** macro.

wake up operations; (e) the length in bytes of the message, which is obtained from the $\mathrm{CH}_{rem}.L$ field. (remember that we are implementing *typed* streams); (f) the capability of the message, which is obtained through the *transmit-cap* function. Below we use the name of the type of used variables (e.g. **M**) to denote also their specific instances during a communication. The reference to the PCB of S is used to move back the process from the waiting status to the ready or running ones, in the case it is blocked because the channel was full.

- Next S copies in a variable *capM* the capability of the built **M** variable that is obtained exploiting the *transmit-cap* function.
- Finally, S performs an interprocess communication passing *capM* on KP_S (see the local interprocess communication protocol).

KP_S is interrupted from S: by looking at the *dest-node-id* field it understands that the communication is related to a remote channel. It forwards to *NIC* the capability *capM* of *M*. We call *delegate* this forwarding operation performed by *KPs*. *NIC* acquires the capability of *M*, obtains the address of N_R from the logical identifier of the receiver ($M.dest-node-id$ field) and it copies *M* and *m* in the network. Recall that *NIC* can access *m* by means of its capability copied in *M.CAP-MSG* field.

When *NIC* on the receiver side receives *M*, it copies it and *m* in the main memory and it passes its capability to KP_R through an interrupt. KP_R performs the following actions which we call "local send":

- It acquires the capability of *M* and *m*.
- It obtains the local channel *CH* exploiting its identifier *M.id-ch*.
- It acquires the lock on the channel.
- If R is waiting for a message, it obtains the capability of the target variable in which to copy the received message. It copies m in this variable, releases its capability and wakes up R[3].
- Otherwise, if R is not waiting for a message, *m* is copied in *CH*. KP_R obtains the capability of the first free position in the message queue *CH.Q-CAP*. It copies the capability of the message (i.e. *m*) in that position and it releases the acquired capability.
- KP_R updates the field indicating the number of messages in the channel, and the next free position in *CH.Q-CAP*.
- KP_R checks if the channel queue is full. In the case it is full it copies to *CH* both the capability of the remote channel (on the sending side) and the capability of the PCB of *S*. Otherwise no action is performed.

[3]Notice that this is an optimization inspired from [73]

- KP_R builds up an **A** variable that includes the result of the send: **stop** in the case the channel is full, **go** in the case the channel is not full.

KP_R delegates the local *NIC* to send the *A* variable to N_S.

When KP_S receives the **A** variable it checks the send result: (a) if the result is **go** *CH.used-channel* is set to false and if *S* was in the **SPEC-WAIT** state it is moved back to the **Ready** or **Running** ones (depending on its priority); (b) if the result is **stop** and the sender is not waiting *CH.used-channel* is left to true. Otherwise, if *S* is waiting (not only in the **SPEC-WAIT** state), *CH.used-channel* is set to false.

Finally we describe the receive protocol performed by *R*, which possibly sends a wake up message to KP_S if the channel was full. *R* performs the following actions that correspond to the call of *receive(ch,tgv)*, where *ch* is the identifier of the communication channel and *tgv* is the target variable on which to copy the message:

- *R* acquires the lock on *CH*.
- *R* checks if the channel is empty. In that case it sets *CH.wait* to true and it copies the capability of its PCB and of the target variable in the fields *CH.PCB* and *CH.TGV*, respectively. Next *R* releases the lock on *CH.R*. It will be woken up by KP_R at the next message reception (see above the KP_R behavior). Notice that, in this case, the message will be directly copied in the target variable and the receive terminates without further actions.
- Otherwise, if there is at least one message in the channel, *R* obtains it by accessing the *CH.EXT* field and the queue *CH.Q-CAP*.
- *R* checks if the sender was waiting because the message queue was full by checking the wait field. If *S* is waiting *R* builds a **W** variable with the capability of the remote channel CH_{rem} and the capability of the PCB of the sender. This information was copied in *CH* by KP_R in the case the channel was full. *R* sets *CH.wait* to false.
- Finally *R* releases the lock on the channel.

In the case *S* was waiting KP_S receives the *W* message and it will move *S* back to states **Ready** or **Running** only if it is in the **SPEC-WAIT** state. Otherwise no action is performed.

6.2 Introduction to the FT-Stream

We introduce the FT-Stream that extends the stream abstraction with message logging. Message logging have been widely employed in the context of fault tolerance to support uncoordinated checkpointing (see the Chapter 2). In this context, to minimize the overhead incurred in logging messages, we target an asynchronous logging technique (i.e. an optimistic message logging technique). To minimize the

number of lost messages in the case of failure we exploit an implementation feature (K-asynchrony), which is essentially based on avoiding the producer on a stream to send "too many" messages.

From the fault tolerance modeling viewpoint the FT-Stream abstraction allows the communicating parties to uncoordinately perform checkpointing operations. That is, communicating parties decide independently when to take checkpoints. This comes at the cost of the message logging overhead but it enables a simple and optimized rollback recovery scheme. Uncoordination of checkpointing is a suitable feature in the case checkpointing operations cannot be introduced in a system-wide logic. It is the case in which we are building our parallel application up of separated modules, possibly internally parallel (see Chapter 3, for a description of the programming model we target). Moreover, the FT-Stream enables localized rollback recovery and it allows us to model the performance behavior of parallel programs independently of the behavior of other modules which interact with it.

In this chapter we give a detailed description of the implementation of FT-Streams and we show an example of its exploitation in a simple test-bed. We also define the performance model of communication and we show some interesting properties. The test-bed represents an important component to implement structured parallel programs. We have seen that this scheme is exploited in farm computations to implement input and output streams (see Chapters 4 and 5, where we make extensive use of FT-Streams).

In Section 6.3 we describe the FT-Stream abstraction, a possible implementation of it, together with its cost model. In Section 6.4 we show how we can exploit it in a simple parallel program, which is used as building block of more complex programs. The example includes a complete rollback recovery protocol and its evaluation.

This document does not discuss the following issues:

- Failure model and failure detection: we assume the fail-stop model, in which processes can restart. Failures are detected in unbounded but finite time [26].
- Process restart mechanisms: we assume processes are restarted by some *ad-hoc* sub-system which is resilient to failures.
- Unknown process dependencies which are built at run-time.

The last one is a key point in our research work: we exploit the knowledge on the interaction patterns of structured parallelism models to insert proper and optimized checkpointing procedures and rollback recovery protocols (see Chapters 3, 4 and 5).

6.2.1 FT-Stream Abstract Model and Implementation Features

To model message logging techniques we exploit I-structures (see Chapter 3) which are defined as sequences of named and typed elements. We map each stream ele-

ment in a I-Structure element. A whole stream is modeled as a single I-Structure connecting the producer to the consumer. Operations on I-Structures are:

- **put(position,element)** This operation stores the given element in the given position. It is semantic of I-Structures that no more than one put can be performed on the same position (write-once property). That is, it is not possible for two elements to have the same name.

- **get(position,target)** This operation stores the content of the given position in the target variable (used as output of the operation). According to the semantic of I-Structures the get operation is blocking: if a position is empty the control-flow of the process performing a get operation blocks until a put is performed on that position (blocking property).

Clearly, the producer will exploit put operations to produce elements on the stream, whereas the consumer get ones.

The properties of put and get operations model message logging on streams as:

- The write-once property is used to uniquely identify stream elements. We exploit the element identities to implement message logging and fault recovery schemes. As stream elements cannot be re-written at the level of the model, the write-once property also expresses the general semantics of rollback recovery: elements passed on streams are not lost but they can be recovered when needed.

- The blocking property implements the receive blocking semantics in the case the message queue is empty (see the communication protocol above).

The controlled asynchrony degree, blocking the sender after $K + 1$ sends, is not modeled by the I-Structure model and it is an implementation feature. The general idea behind this choice, which leaves such a property at the implementation level, is motivated by the fact that the consequences of I-Structure properties are sufficient to implement rollback recovery strategies. The controlled asynchrony degree is an optimization of the number of messages that must be re-sent on a stream, in the case of failure.

The implementation of streams is based on the communication channels described in Section 6.1. We focus on single producer and consumer streams and we leave the implementation for multiple producers and consumers to future work.

The main feature of the implementation of the FT-Stream we describe is that it is based on a controlled asynchrony: the number of messages enqueued cannot be larger than a given value, which is known at compile-time. As a consequence of this property, the number of messages re-sent during fault recovery is upper bounded. This property is independent of the actual frequencies of checkpointing operations. Moreover, the message logging support is based on the same implementation of unlogged channels. The versatility of the channel implementation is well-suited to transparently support different logging strategies. For instance, we can support

logging of messages onto the local disk of the receiver or onto the volatile memories of remote nodes.

6.3 FT-Stream Implementation

In this section we describe the FT-Stream implementation. We introduce a message logging procedure in the implementation of streams, and we provide hooks to support rollback recovery protocols. Whenever one of the two parties communicating through an FT-stream fails, the other one is informed at the next communication operation. The communication exceptions are handled by executing some rollback recovery protocol. The semantics of rollback recovery depend on the application semantics and its parallel structure (as shown in Chapter 3).

We can consider different implementations of the FT-Stream, each depending on the way in which the stable storage is implemented:

- Stable storage implemented on the local disk of the node executing the receiver.
- Stable storage implemented by a subsystem executed remotely w.r.t. the sender and receiver.

In the first case message logging is implemented as a local communication, in the receiver node, between the process managing communications (the **KP**) and the local disk manager. In the second case the implementation can be made in two ways:

- The *KP* of the sending node locally performs the send to the stable storage address, along with the remote send to the receiver node on the application channel.
- The *KP* of the receiving node locally performs the send to the stable storage address, along with the local support on the application channel.

In this chapter we show the implementation for the first case as represented in Figure 6.5. In the figure the sender process (S) is executed on the main processor (P) on the sending node (N_S). The receiver process (R) is executed on the main processor of the receiving node N_R. We exploit two processes implementing the communication protocol which we denote KP_S and KP_R. These are executed on a special hardware: also in this case we avoid to distinguish between the process name *KP* and the hardware executing it. A network interface card (or *NIC*) is an I/O device that implements the low-level network accesses. We denote with NIC_S and NIC_R these devices on the sender and receiver node, respectively. To implement message logging on the receiver hard disk we exploit an hard disk manager, which we name *message logger* (or *ML* in short). We assume that *ML* is executed on the I/O device managing the hard disk on the receiver. The interactions between any

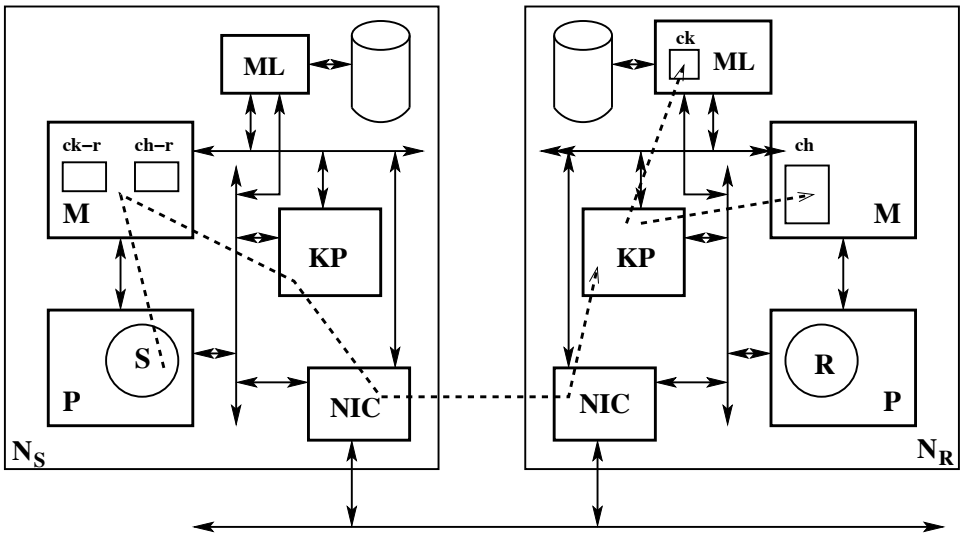


Figure 6.5: Architecture of the nodes executing the sender and the receiver processes for the implementation of FT-Streams.

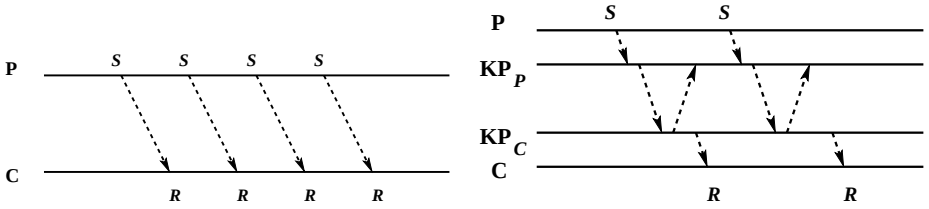
processes executed on N_R with *ML* are implemented through local communication channels. For this purpose KP_R can address the local memory of the I/O device executing *ML*, according to the Memory Mapped I/O mechanism.

The implementation of FT-Streams exploits two channels: one for the communication between *S* and *R*; the other one for the communication between *S* and *ML*. This is transparent to the programmer which only exploits an FT-Stream abstraction at the level of programming language. The local memory on the sending node contains two data structures: one representing the channel between *S* and the receiver (*R*), denoted with (*ch-r*); the other one representing the channel between *S* and the checkpointing process, denoted with (*ck-r*)[4]. These data structures are used only to synchronize *S* with KP_S, i.e. we avoid to copy messages in such data structures. On N_R we exploit two data structures, which correspond to the ones on N_S. *ch* is mapped on the main memory, and it implements the communication between *S* and *R*. *ck* is mapped on the memory of the I/O device executing *ML* and it implements the communication between *S* and *ML*. These data structures include both a message queue.

The presented solution is valid *also* if:

- The sender and the receiver are mapped on the same resource. In this case we can optimize the support by implementing communications in a shared memory environment.

[4]The letter "r" of the channels on the sender stands for *remote*, in the sense they are remote to the actual variable implementing the channel that is implemented on the receiver node.



(a) Example of graph of a communication on a stream between a producer P and a consumer C. The horizontal lines denote the execution of the process. The ones connecting the horizontal lines denote communications. The points on the lines in which the horizontal ones are connected denote the instants of times in which the send and receive operations are performed.

(b) Example of graph of the implementation of send operations. Unlike the abstract example we have also included two horizontal lines denoting the communication processes (KP). The example shows the protocol performed during a communication, which we detail in the text.

Figure 6.6: Graphs of communication of two processes on a stream, from an abstract (left) and implementation viewpoint (right).

- There is not a specialized hardware supporting communications and *KP* is executed on the main processor.

6.3.1 Notation

In this chapter we will make use of time/event graphs to describe message logging and rollback recovery. Figure 6.6(a) shows a simple example of a producer-consumer interaction pattern: two processes communicate through a stream. The upper horizontal line denotes the execution of the producer process, the lower one of the consumer. Each line that links the horizontal ones denotes a communication between the producer and the consumer. The connection points denote the execution of a send operation on the producer and of a receive operation on the consumer.

In Figure 6.6(b) it is depicted the same behavior from an implementation viewpoint. The communications are partially implemented by local communication processors (*KP*, see above). In this case the horizontal lines denote the execution of single processes (P, KP_P, KP_C, C). The other lines, connecting points on the horizontal ones, denote communications (both local and remote). Notice that, at this level, there are communications also from the consumer node to the producer one: we will see below that these ones carry control messages which implement the message logging protocol.

In some cases we will add vertical lines to denote computational steps, whenever it makes some sense to relate the discretization of the execution of the producer and the consumer of a stream.

6.3.2 Communication Support for Stable Storage on the Receiver Disk

For brevity in this thesis we present only the description in the case in which the stable storage is mapped onto the local hard disk of the node executing the receiver process. The other cases (e.g. stable storage implemented on a remote node) enjoy a very similar semantics to this case and the implementations can be made by slightly modifying the one presented here. We demand to future work a full description of the implementations for all stable storage mappings.

We first describe the abstract behavior of the protocol. Next we define the data structures and we show the implementation of the communication protocol.

The implementation that we describe extends the one for streams presented in the previous sections with a simple modification. In the implementation of streams we copy (a) the capability of the data structure on the sending node (*CAP-CH* field) and (b) the capability of the PCB of the sender in the message sent from the sender to $textKP_R$. This reference was used by KP_R to notify KP_S of the send result and to possibly implement the wake up procedure of S. In the case of the FT-Stream we are assuming that:

- Processes can fail and restart.
- A failed process can be restarted on a different computational node. In the implementation that we describe we limit this property to the sender process because we implement the stable storage on the receiver secondary storage.

In the implementation of streams a capability is a logical reference in the physical memory of a computational node. In the case of failure and restart the physical memory associated to a logical variable changes. This problem is especially highlighted in the case in which processes restart on different nodes: references to physical addresses cannot be equal on different machines. In the implementation of FT-Streams we modify the implementation of capabilities: we exploit logical identifiers for the communication of information between remote nodes that we assume to be defined statically in the compilation of processes. In this way we avoid to pass capabilities between processes, to implement the communication protocol. We exploit specific functions to map identifiers onto capabilities and vice-versa. In the case of failure and restart we assume that such functions are re-initialized by the process restart sub-system. Unlike this approach the capabilities that are *not* passed between nodes (e.g. the list of capabilities of queue positions in the channel data structure on the receiver nodes) have the same implementation of the stream case without message logging. We assume that, after a failure and restart, they are properly re-initialized by the restart sub-system.

High Level Protocol Description

Suppose that a process S wants to send a message *msg* marked with sequence number *seq-num*[5]to the receiver onto the application channel *CH*. The implementation of the send is partially performed remotely onto the sender node and locally onto the receiver node. As usual the terms "remote" and "local" are used w.r.t. the mapping of the communication data structures that include the message queue (in our case onto the receiver node). The sent messages are copied both onto the application channel and the checkpointing one. S checks if the previous send has terminated. If it has terminated it collects the needed information in a special data structure (called **M**). Next it delegates its local *KP* to perform the remote send for it. It also sets a boolean field in the channel to *true* (**used-channel** field, see below). If the previous send has not yet terminated S goes into **SPEC-WAIT-ONE** or **SPEC-WAIT-BOTH** status: the choice depends on which send results have been received (as stated above, we perform two copies, one for the application channel, the other one for the checkpointing one). S will transit back in the ready or running state only after all needed results will be received from KP_S. The variable **M**, which S passes to KP_S, includes information to retrieve the communication channel, the destination node, a reference to the message to be sent, its PCB, and information for fault tolerance purposes. *KP* delegates itself the transfer of the message to the NIC_S if the channel is not marked as *failed*. In this latter case the receiver signaled a failure and restart event: S can perform proper actions to recover to a correct state (see Section 6.4.1).

The request is passed from NIC_S to NIC_R that simply forwards the request to the local KP_R. KP_R, which receives a message **M** from a remote node, performs a sequence of actions for the related application channel and a similar sequence for the checkpointing one. The operations for the application channel are: (1) it retrieves the channel on which sending the message, (2) it copies the messages in the next free position, (3) it increments the number of messages in the queue, (4-stop) in the case the channel is full, the result of the local send is **stop**, (4-go) otherwise it is **go**. The sequence of operations for the checkpointing channel are equal to those for the application one. We also obtain an equal result: **stop** if the channel is full, **go** otherwise. The results of the two send operations are collected in a single message of type **A** with information for fault tolerance purposes and for KP_S to retrieve the correct channel and the sender process.

When KP_S receives a result notification for a send it behaves according to the values of the result fields:

app = stop, checkp = stop both channels (application and checkpointing) are full. If S is not waiting KP_S leaves the *used-channel* variable set to true in

[5]The sequence numbers are not directly used in the implementation of fault tolerance for streams. In the next section, we will show how they can be exploited by rollback recovery protocols defined at higher-levels.

both of them (see the data structures below). The sender will stop at the next attempt to send a message on the application channel. Otherwise, if the sender is waiting, both *used-channel* fields are set to false. In both cases *KP* waits for *two* wake up messages from the destination node before re-activating the process (see below).

app = stop, checkp = go the application channel is full. If the sender is not waiting the *used-channel* field in the application and checkpointing channels are, respectively, left to true and changed to false. The next attempt to send a message will block the calling process. Otherwise, if the sender is waiting, both *used-channel* fields are set to false. In both cases *KP* waits for a wake up message from the destination node.

app = go, checkp = stop this case similar to the previous one but the *used-channel* variables are set with reversed values w.r.t. the previous case. *KP* waits for a single wake up message.

app = go, checkp = go both *used-channel* variables are set to false. In the case that the sending process is waiting for the send result it is moved to the ready queue. Otherwise, if the process is not waiting or it is waiting for another event to occur, its state does not change.

The behavior of the process *R* or *ML* performing a receive is: (1-empty) if the queue is empty the process blocks waiting for a message; (1-not empty) otherwise the message is copied in the target variable; (2-wait) if the sender was waiting to complete a send operation (channel full) it delegates KP_R to send a wake up message. It passes KP_R a data structure of type **W** which includes all the information needed for waking up the sender, i.e.: the capability of the PCB of *S*, the capability of the channel and the sequence number of the received message. Otherwise (2-not waiting) no more actions are needed.

When a wake up message is sent from the destination node to the sending one KP_S behaves according to the state of the *S* process and of the values of the *used-channel* variables:

- Suppose the sender is waiting for both channels. When *KP* receives the first wake up message it sets the *used-channel* variable to false in the related channel. It avoids to wake up the process by looking at the *used-channel* field in the other channel (that is set to true). When *KP* receives the second wake up message it sets the used-channel of the related channel to false. By analyzing the used-channel of the counter part channel it knows that the process state can be set to *ready*.

- Suppose the sender is waiting for only one of the two channels. When *KP* receives the wake up message, by checking the *used-channel* variable of the counter part channel, it knows that the process can be woken up. It sets

the *used-channel* variable of the related channel to false and it wakes up the process.

Clearly it is not possible that S is not waiting for the wake up message and a wake up message is received.

Similarly to the case of the stream implementation, we optimize the copy of messages in the case the receiver is waiting. KP_R directly copies the received message in the target variable, which capability has been provided form R before passing in the waiting status.

In the case of failure of S or R the restart sub-system, after having re-initialized their data structures, restarts them by calling proper rollback recovery procedures. Such procedures are callbacks provided by some upper-level. To enable the failure detection between the communicating parties, when a process restarts it passes a message of type **V** to the other party. The purpose of the message is:

- To communicate the new address of the process if it is different from the previous one;
- To set a special variable in the channel indicating that the other party has failed. In this way the process performing communication on the channel is informed of the failure.

We do not employ any kind of timeouts: these are hidden by the process restart mechanism that we do not discuss in this thesis and can be supported by existing techniques.

Implementation of the Communication Support: Data Structures

The fields of the local and remote channels, respectively mapped in the destination and source nodes, are shown in Figure 6.7.

We extend the channel data structure on the sending side (w.r.t. the stream implementation, see Section 6.1), with the following fields:

seq-num this contains the last sequence number that was passed to a send.

hasCounterPart this indicates if the channel has an application or checkpointing counter part. We recall that we associate a checkpointing channel to each application channel only for FT-Streams. This field is used by the *KP* support to choose if to implement an unlogged communication or a logged one.

CAP-CP this is the capability of the counter part channel.

failed this field is a boolean value used by KP_S to signal to S that the receiver failed and restarted.

The fields of the channel on the receiving side are extended with the following fields:

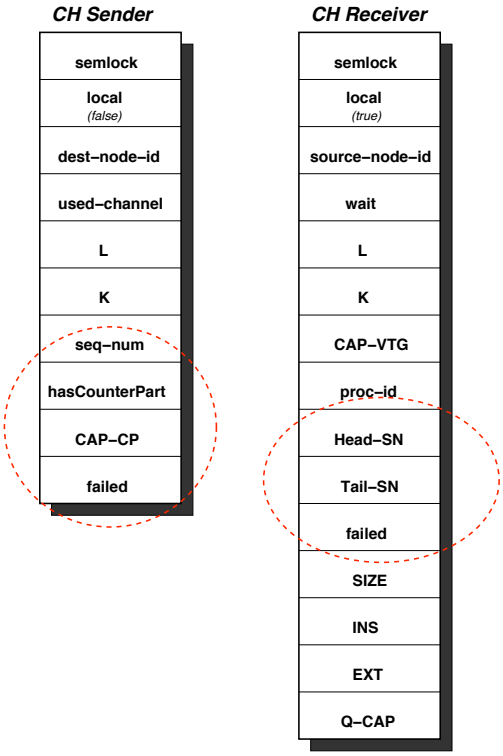


Figure 6.7: Implementation of the communication channels, on the sending (left) and receiving (right) sides. Highlighted fields are related to the FT strategies (both for stream and application levels).

Head-SN this is the sequence number of the first un-received message, i.e. the message at the head of the queue.

Tail-SN is the sequence number of the last message put in the channel, i.e. the message at the tail of the queue. Because we avoid unordered sequences of messages in the queue **Tail-SN** $\geq$ **Head-SN**.

failed this is set by *KP* in the case in which the sender node is restarted. It is used to support rollback recovery and process migration in the case of failure.

The data structure exchanged to implement the communications are shown in Figure 6.8. As usual each data structure contains a first **TYPE** field that contains the operational code of the message. The fields of the **M** structure which are passed from the sending process to KP_S are:

dest-node-id this is the identifier of the destination node. *NIC* uses this value to obtain the node address.

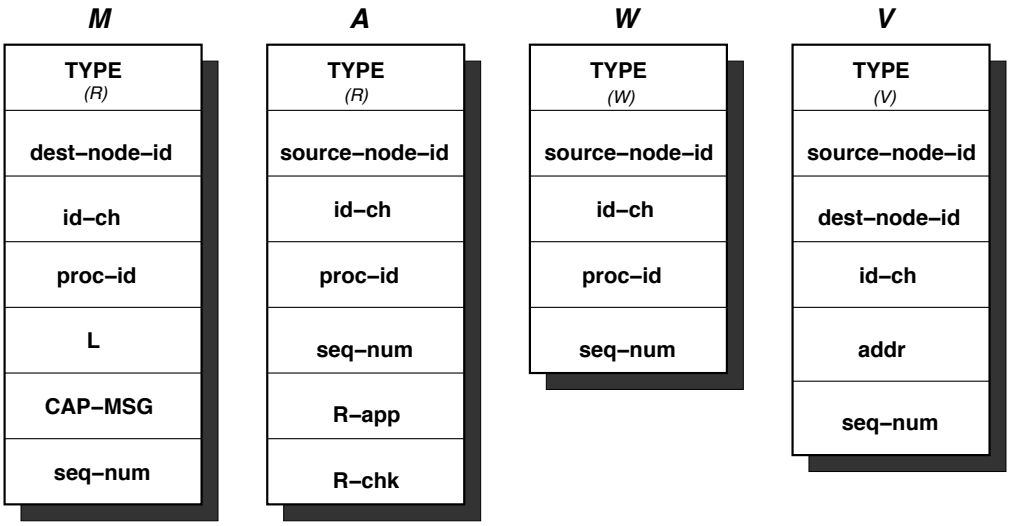


Figure 6.8: Message types implementing the communication protocol with message logging.

ch-id this is the identifier of the channel. It is used by KP_R to obtain the capability of the local channel.

proc-id this is the identifier of the sending process.

L this is the length of the message.

CAP-MSG this is the capability of the message.

seq-num this is the sequence number of the message.

The fields of the **A** structure, which are passed from the receiving process to the local KP_R to notify S of the send results, are:

source-node-id this is the identifier of the source node. *NIC* will use this value to obtain the node address.

ch-id this is the identifier of the channel. It is used by KP_S to obtain the capabilities of the data structures *ch-r* and *ck-r*.

proc-id this is the identifier of the sending process.

seq-num this is the sequence number of the message whose receive induced the sending of this message.

R-app this is the result of the local send onto the application channel.

R-chk this is the result of the local send onto the checkpointing channel.

The fields of the **W** structure which is passed from the receiving node to the sending one to wake up S are:

source-node-id this is the identifier of the source node. *NIC* will use this value to obtain the node address.

ch-id this is the identifier of the channel. It is used by KP_S to obtain the capabilities of the data structures *ch-r* and *ck-r*.

proc-id this is the identifier of the sending process.

seq-num this is the sequence number of the message which receive induced the sending of this message.

The **V** structure is used to implement rollback recovery protocols in the case of process re-start. It is exchanged between *KPs* (when they are re-started) to notify the failure and restart and the rollback sequence number. Its fields are:

source-node-id this is the identifier of the source node, i.e. the node that has rolled back. It is used to notify the other *KP* of the identity of the failed node.

dest-node-id this is the identifier of the destination node, i.e. the one notified of the failure.

ch-id this is the identifier of the channel on which notifying the failure. We assume that channel identifiers are statically assigned to (pair of) processes.

addr this is valid only in the case S is the failed and restarted process. It includes the new address of the node in the case the sender has been re-started on a different computational node. We recall that in this implementation we are assuming the receiver restarts on the same node.

seq-num this is the sequence number of the checkpoint to which the notifying node has rolled back. It is used by upper-level rollback recovery protocols.

Implementation of the Communication Support

We now give the implementation of the communication protocol in pseudo-codes. Figure 6.9 shows the implementation of the send operation. S sends a message *msg* on the channel identifier *ch-id*, with sequence number *seq-num*. After obtaining the capability of the channel and its address from the capability (copied in the local variable *ch-r*), it locks the associated semaphore. The *chidToCap* function, given a channel identifier, returns its capability. Next it checks if the channel is supported by message logging (*hasCounterPart* field). We are interested in the case message logging is supported. S locks the semaphore associated to the checkpointing channel and it checks if the previous communication, which are composed of two copies of

the message on the receiver node, is terminated. It do so by checking the *used-channel* fields on the application and checkpointing channels. The behavior is the one described above: if one of the two channels is used by a previous send it transits into the waiting status. We choose the **SPEC-WAIT-BOTH** status in the case both channels are used, **SPEC-WAIT-ONE** otherwise. Next S unlocks the related semaphores. When S is moved back to the running state, or if the channels were not used, it automatically re-acquires the lock on the semaphores. It sets the *used-channel* fields to true and it unlock the semaphores. Next it allocates and fills up a **M** data structure with proper values. It obtains the capability of the **M** variable and it passes it to KP_S with an interprocessor communication.

The implementation of a *KP* process is described in Figure 6.10. It acts as a server by waiting for requests incoming from processes executed on the same node or from remote processes, through the network. When *KP* is signaled of the arrival of a new request it checks the type of the received message. We describe the behavior for each message type separately. The behavior in the case a **M** message is received is shown in Figure 6.11. If the request is received from a remote node, *KP* implements the local send operations by performing two *local-send* procedures on the application and checkpointing channels. The procedures return the send results, which are encapsulated in a **A** message and passed back to KP_S by means of a request to the local NIC. Otherwise, if the request **M** is for a remote node, *KP* simply delegates the forwarding of the message to the local NIC.

The *local-send* procedure takes as input the message m and a channel variable *ch* on which to perform the send. After locking the related semaphore it checks if the receiver is waiting for a message. In the case it is waiting it directly copies the message in the target variable, which the receiver previously copied in the *CAP-VTG* field of the channel. Next, it moves the receiver to the *ready* status if it has not the privilege to go directly into the *running* status or if preemption is not admitted. Otherwise it is moved to the *running* status. This task is performed by calling a *wake* procedure (which implementation is not shown here) on the the PCB of the receiver, which is obtained by means of the *procIdToPCB* and *obtain-addr* functions. Otherwise, if the receiver is not waiting, the message is copied in the queue. If this message fills up the queue, the field *wait* is set to true, and the process identifier of the sender is copied in the *proc-id* field. In this case the result of the procedure is *stop*. Otherwise, if the message queue is not full, the result is *go*. Notice that we assume that process identifiers are names that scope to the whole application, independently of their actual mapping on computational nodes.

Now we return back to Figure 6.10. In the case the *KP* receives an **A** message it calls the *send_results* procedure, which is shown in Figure 6.13. The procedure implements the behavior described above. In the case one of the two results is *stop* and if S is not waiting the *used-channel* field is left to *true*. This blocks the sender to perform a further send operation (see the send implementation in Figure 6.9).

In the case *KP* receives a *W* message (see Figure 6.10) we perform the *wake_sender* procedure, which is shown in Figure 6.14. Each **W** message is related to one chan-

nel. KP_S behaves according to the following rule:

- If S is in the **SPEC-WAIT-BOTH** status it moves it in the *SPEC-WAIT-ONE* status.
- If S is in the **SPEC-WAIT-ONE** status it moves it in the *ready* or *running* states.

The management of **V** message by *KP* is shown in Figure 6.10.

Finally, Figure 6.15 shows the implementation of the receive operation. R transits into the *waiting* status if the message queue is empty. Otherwise it extracts the message on the top of the queue and it checks if the sender is in the waiting status. If the sender is waiting it allocates and fills up a **W** variable by copying the information from the channel and the actual parameter of the procedure. The built variable is forwarded to the local *KP*.

6.3.3 Synchronous FT-Streams

We show how the above implementation can be modified to support the following properties:

- The sender is synchronized with the message logger.
- We ensure that, when a receive operation terminates, the related message is copied on stable storage.

The modifications are the following:

- The send onto the checkpointing channel which is performed by the KP_R is synchronous. This means that the control-flow is returned to the KP_R only after the message logger has copied the message in the secondary storage.
- In the *local_send* procedure we first perform the send on the checkpointing channel. Next we perform the one on the application channel. The second one can be also asynchronous.

As a result the sender is synchronized with the message logger and the receiver can obtain only logged messages. We notice that, during a local send, the KP_R is blocked until the message logger has stabilized the element. This can be a drawback because all other communications are delayed.

6.3.4 Analysis of the Communication Latency

The communication latency depends on the hypotheses on the execution environment we target. In this thesis we just give a qualitative analysis of the impact of the checkpointing channel on the communication latency.

If we consider a communication without message logging we can abstractly represent the latencies in the following way:

- The time needed to perform a send operation depends on the state of the channel. We consider the case in which the channel is not full and the sender does not block. The execution latency is denoted by T_{send}. This time includes: (a) the time needed to check the state of the channel, (b) the time needed to fill a R variable and (c) the interprocessor/process communication latency. This last choice depends on the resource that is executing KP.

- The KP on the sending side just delegates the operation of copying the message in the network to the local network control unit. We denote the latency needed to perform this operation as $T_{KP-Send}$.

- The KP on the receiver side: (1) performs a local send to the communication channel; (2) performs a local send to the checkpointing channel; (3) fills an A variable and sends it back to the sender KP. If the time needed to perform a local send is $T_{loc-send}$, we can compute the time needed to perform the two sends as $T_{loc-send-check} = 2 \cdot T_{loc-send}$. Notice that we do not assume any specific optimizations in the execution of the send and, consequently, the evaluation we give is necessarily an upper bound of the actual time. Each local send takes the same latency of a send to a channel mapped onto a shared memory (the main memory of the receiver that can be addressed from KP and the receiver).

- The latency needed to perform a receive, in the case the channel is not empty, is denoted with T_{recv} and is equal to the time needed to perform a receive on a channel mapped on a shared memory support.

- We can denote with T_{ack} and T_{wake} the receiver KP latency needed to send back the send results to the sender and the time needed to send a wake-up message to the sender, respectively. We also denote with $T_{fail-notify}$ the time needed on a KP to notify the other party of the failure of the local process (valid for both sender and receiver). All these latencies includes (a) the time needed to build the variable to be transferred (b) the time needed to delegate the communication to the network control unit.

- The time needed to transfer a message of length m between two network interfaces is $T_{transf}(m)$.

We can instantiate each quantity above to the actual cases to obtain a general evaluation of the communication latency.

6.4 Exploiting the FT-Stream

We now describe an example of exploitation of FT-Streams, which we use to highlight general properties of rollback recovery protocols. The example is described as a test-bed and it is exploited in several structured parallel programs. For instance,

we exploited it in Chapter 3 to implement the communication of emitter and collector processes with the external entities, which provide input elements and consume output results respectively.

6.4.1 The Test-Bed

Suppose a sender process executes the program shown in Figure 6.16. The `count` variable is used to generate sequence numbers and to perform periodic checkpoints. The **DELTA_SEND** macro is an integer constant locally defined. In this program, similarly to the cases of farm and data parallel ones, we have a mapping between the computational steps, represented by a loop iteration, and the sequence numbers of stream elements. That is, between computational steps and I-Structure positions.

Similarly suppose that the receiver process performs the computation which pseudo-code is shown in Figure 6.17.

We exploit message logging as an optimization of the recovery overhead and to maximize the decoupling of the fault tolerance strategies of processes.

6.4.2 A Simple Rollback Recovery Protocol

We describe the recovery protocols performed by the sender and the receiver. The protocol is based on uncoordinated checkpointing and asynchronous message logging. The uncoordination of the checkpointing strategy allows the two parties to take checkpoints without any synchronizations in their implementation. This is especially important in the case we are composing processes belonging to different applications/modules: according to this approach the only requirement for the implementation is to exploit the FT-Stream abstraction. Otherwise the approach based on coordinated checkpointing forces a design constraint in the process implementation. We could implement coordination of checkpoints by choosing the same checkpointing interval in the two processes. Asynchronous message logging should be preferred to the synchronous blocking of the local state if the stable storage access features an high overhead.

The rollback recovery protocols include the behavior of S, KP_S, R and KP_R. Thus the protocol is an extension of the FT-Stream communication protocol presented above. The fault recovery behavior is implemented as an extension of the procedures performed by KP in the case of failure and restart.

In the protocols below we have defined with C the name of the message logger process (ML above).

Sender Protocol

Suppose the sender fails at step f and rolls back to step r. We known that the receiver can be in any steps between t and $f+1$, where $t \leq f-k$ (this last because of the controlled asynchrony degree). Notice that, if the rollback depth is greater or

equal than the asynchrony degree (i.e. $f - r \leq k$), then it could be that the receiver is executing some steps before the rollback one (i.e. $t \leq r$). Otherwise, if $f - r > k$, then it is mandatory that the receiver is executing at a step after the rollback step (i.e. $t \geq r$).

In principle during the recovery from the rollback step to the failure one we can: (a) avoid the re-sending operations by making the receiver send its sequence number to the sender process or (b) re-send the messages and force KP_R to discard the duplicated messages. In both cases we keep the invariant property for which *Head-SN* $\leq$ *Tail-SN* (see Figure 6.7).

To minimize the recovery overhead we choose the first strategy. When the sender is restarted KP_S and S perform some preliminary operations:

- KP_S re-generates the data structures for the application and checkpointing channels.
- S recovers its last checkpoint both with its sequence number and it sets it in the local channels (CH.seq-num field).
- S passes KP_S a **V** data structure including its new address and its sequence number.
- KP_S delegates the NIC_S the sending of the message to KP_R.

First suppose the receiving node does not fail during the rollback protocol (i.e. from the sender failure to the beginning of the recovery protocol). In this case all the send operations performed from r to f must be avoided. To know the actual value of f the sender asks it to the receiver. The value of f is equal to the sequence number of the last sent message, which can be found in the *Tail-SN* variable in the channel data structures on the receiver. The KP_R knows that it is not executing a rollback protocol: it builds a new **V** structure including the bottom sequence number (*bsn* below) of the application channel and it sends it to S. During recovery S knows that it can avoid re-sending operations until the *bsn* step. In the case of recursive failures of S it will rollback to r again and it will re-perform the same protocol: *bsn* will not change in the meantime. Suppose now that the receiver concurrently fails during the S rollback protocol. When KP_R receives the **V** data structure from S it knows that it has restarted (the failed field in the channels are set to true). Denote with r_R and r_C the rollback steps of the receiver and the checkpointing process, respectively. Looking at Figure 6.18 if the rollback step of S is less than both r_R and r_C KP_R selects the minimum between them. Next it sends it to KP_S in a **V** variable. Otherwise, if both r_R and r_C are less than r, then S has to rollback again. Notice that the rollback course of the sender is limited to two steps at most also in the case of recursive failures. The sender rolls back to the nearest checkpoint before the minimum between r_R and r_C. KP_S sets the minimum between the rollback steps in the channel to allow the sender to avoid re-sending operations until the

notified rollback step. Notice that, if r_R and r_C are different, KP_R must discard some messages for the channel with the greatest sequence number. It can do so by simply checking the message sequence number and the channel ones. Finally suppose that either r_R or r_C is less than r. In this case S has to rollback as in the previous case and the minimum sequence number is selected by KP_R to be included in the **V** data structure.

The description of the protocol as sequence of actions follows:

- KP_R receives a **V** notification message from S. It checks if *CH.failed* is set to true (*CH* is the name of the application channel data structure):
- case *false*: KP_R builds a new **V** message including *CH.Tail-SN* and sends it to S.
- case *true*: KP_R computes the minimum between *CH.Head-SN* and *CK.Head-SN* (*CK* is the name of the checkpointing channel). It copies it to a new **V** message and it sends it to S.

KP_S receives back a **V** message and it checks if the received sequence number is greater or lower than its rollback step: (case $V.sn \geq CH.sn$) R has not failed and S avoids re-sending operations until $V.sn$. KP_S sets this value in both data structures (the one for the application and checkpointing channels). S avoids to send all messages with sequence number lower than $CH.sn$. (case $V.sn < CH.sn$) R has failed and S has to rollback. KP_S sets the new sequence number in the channel. S, by checking this value before re-starting the recovery protocol, decide to recover to the checkpoint with sequence number lower and nearest to $CH.sn$. Then it avoids re-sending operations for all messages until $CH.sn$. Notice that this can be done in a transparent way to the programmer in the send operation: S checks if the message sequence number is lower than $CH.sn$.

Receiver Protocol

Suppose the receiver fails at step f and is restarted at step r. During the rollback it has to re-obtain the messages from r to f (lost messages, see Chapter 3). To do so it recovers the ones checkpointed on the local disk. For the remaining ones, which are not copied on the local disk but that are copied in the checkpointing channel, it forces the sender to rollback and re-send them. The rollback protocol performed by R, C and KP_R when they are restarted follows:

- R obtains its last checkpoint and the corresponding sequence number from the local disk.
- C obtains the sequence number of the last message copied in the local disk.

- KP_R re-builds the data structures for the communication channels and it waits R and C to set their sequence numbers in the respective channels. The failed fields in the channels are set to true. In the meantime KP_R discards all messages received from S until the first phase of the recovery is terminated.

- R and C set their sequence numbers in the new channels.

- KP_R obtains the two sequence numbers, it computes the minimum of them and it includes it in a **V** message. The message is passed to KP_S.

- When KP_S receives the **V** message it sets the failed field of the channel to true and *V.seq-num* in the channel. At the next send S gets aware of the R failure (by checking the failed field) and it rolls back to the checkpoint which sequence number is lower and nearest to the sequence number set in the local channel (i.e. the sequence number of the receiving node computed by R). We denote with *ssn* such a sequence number.

- S knows that it has to avoid some message sending operations. These are the ones from *ssn* to CH.seq-num $-$ 1. We select the minimum between the C and R sequence number. Thus, some messages that KP_R receives are only for one of the two channels, and must be discarded for the other one. KP_R re-starts performing the normal operations (i.e. copying the message in the channel queue) (a) for *CH* when a message with sequence number equal to *CH.seq-num* is received, (b) for *CK* when a message with sequence number equal to *CK.seq-num* is received.

The recovery protocol consists in replaying a part of the computation. The first part (possibly empty) consists in the recovery of S from *ssn* to *CH.seq-num* (on the sender node). In this phase S avoids all sending operations. Next S re-start to send messages and KP_R properly manages them, by checking the message sequence number and the ones in the channels. That is, it can happen that some messages are copied in just one channel, depending on the minimum sequence number contained in them.

In the case of recursive failures (i.e. the receiving node fails again) the same rollback step is re-selected and the above protocol is re-executed identically. In the case of concurrent failure of S during the rollback protocol we perform the protocol of above. In fact, when KP_S receives a **V** message from R it checks if *CH.failed* is set to true. In that case it sends back to R a **V** message according to the protocol performed in the case of S failure. This means that, in the case of failure concurrency, we perform the rollback recovery protocol for the S failure.

Protocol Performance

We study now the performance of rollback recovery, in terms of the number of message replayed and the rollback depth, i.e. the number of discarded checkpoints.

Number of Re-Sent Messages It should be clear from the description above that orphan messages, i.e. messages which sending operation has been unrolled but not the related receive one (see Chapter 3), are not replayed. We avoid the formal proof for this property.

The performance of the protocol depends on how lost messages are managed. Lost messages are the ones for which the sending operation is not unrolled but the receive one is (see Chapter 3). In the case of rollback we can characterize them in two main set:

- The ones stored in the local disk of the receiver. The cardinality of this set depends of the rollback step of R. That is, if the last checkpointed message has a sequence number lower than the R sequence number the stable storage does not contain useful messages.

- The rest of them, i.e. the ones from the last checkpointed one to the last sent by S. S replays all their sending operations.

Looking at the protocol we can see that the cardinality of the second set is limited by the asynchrony degree of the checkpointing channel. That is, greater asynchrony degrees in the checkpointing channel make lower the probability of S to be blocked (better failure-free performance). From the other side, in the case of failure, greater degrees can induce a larger number of messages to be re-sent (worst recovery performance).

Number of Discarded Checkpoints The number of discarded checkpoints is different for the receiver and the sender:

- Receiver: it rolls back only in the case it fails and restarts. It always recovers to its last checkpoint: no checkpoints are lost and older checkpoints can be directly garbage collected. The checkpointing process has not a state to be checkpointed.

- Sender: in the case it fails and rolls back it selects its last checkpoint. The rollback is deeper only in the case the receiver fails (both alone or concurrently). In this case the checkpoint preceding the receiver one is selected. Consequently the rollback depth depends on the receiver checkpointing frequency as in the previous case.

Clearly, the number of messages to be kept on stable storage depends of the checkpointing frequency, and the number of messages exchanged between two checkpoints. We observe that this information can be deduced from the parallel structure of programs (see Chapter 3).

6.5 Related Work

We relate the FT-Stream message logging mechanism with existing ones. For this purpose we briefly subsume the message logging mechanisms which were previously described in Chapter 2.

There exist three main flavors of message logging protocols: *pessimistic*, *optimistic*, and *causal* [39]. Message logging protocols typically support checkpointing protocols, representing a way to speed-up the rollback recovery execution [39]. Typically they are not used in coordinated checkpointing techniques [39] in which checkpoints are made consistent by coordinating process actions. They are required in uncoordinated checkpointing protocols [39], to ensure recoverability to consistent states after a rollback.

In pessimistic logging whenever a process receives a message its determinant is logged synchronously to stable storage. The determinant of a message receipt event include the content of the message together with other information related to the causal execution of the process. In the case of rollback the recovery is straightforward because the lost receive events are replayed by obtaining them from stable storage.

The MPICH-V1 [19] implementation of the MPI interface exploits pessimistic logging of messages. A special channel memory abstraction is exploited: each process has its own channel memory. When a process wants to send a message to another process it sends it to its channel memory. The channel memory both saves the message on stable storage and it delivers the message to the sender. In [19] the channel memory is implemented as a process and each channel memory can support multiple communications. In [21] it is reported that this approach does not provide good performance which depends on the number of exchanged messages and the number of channel memories and processes in the computation (in fact the communication bandwidth is divided by a factor of 2 [21]).

In optimistic message logging message receipt events are logged to stable storage asynchronously. For instance, some buffering of events is exploited: when the buffer of events becomes full it is flushed to stable storage. In the case of rollback *some* of the message receipt events can be lost. The sender of such messages (called orphan messages) must rollback to re-execute the corresponding send operations. In some cases the re-sending of unrolled messages can be done in parallel with the actual process execution, without forcing the process to rollback. Ordering of messages is needed to support this kind of parallel recovery.

In our approach message logging is performed on a subset of communication channels in an asynchronous fashion. Differently from optimistic logging protocols, we impose a limit on the degree of asynchrony. This limit is equivalent to the level of asynchrony of checkpointing channels, and can be properly tuned statically or dynamically. This impacts the performance of recovery because the maximum number of lost messages, which sending must be replayed, is actually limited from the asynchrony degree of the checkpointing channel. This is different from existing works where, in the case of failure, an uncontrolled number of messages is lost. In

fact the number of lost message in such works only depends on the frequency of checkpointing operations.

In casual logging protocol, messages are logged asynchronously, as in optimistic logging protocols. In some cases [21] the messages are logged on the volatile memory of the senders. Some information related to process inter-dependencies is piggybacked to messages to replicate it on all processes. Also events can be logged synchronously or asynchronously on stable storage. In the literature the minimization of the impact of piggybacked information on application message has been deeply studied.

Causal logging protocols have been introduced to support the second version of MPICH-V, namely MPICH-V2. In [22] three protocols are described, each reducing in a different way the size of piggybacked information on application messages. The implementation stores the content of messages on senders (sender-based message-logging) and it asynchronously logs dependency information on a stable storage abstraction which is called Event Logger (EL).

In contrast with the motivations behind the introduction causal logging protocols, we address structured parallel computations in which some compile-time knowledge is assumed on the process dependencies. We exploit such information to avoid to log dependencies between processes implementing our parallel computations because they are known at compile-time.

```
1  send(ch-id, msg, seq-num) {
2      CAP-CHR = chidToCAP(ch-id);
3      ch-r = obtain-addr(CAP-CHR);
4      lock(ch-r.semlock);
5      //checks for the app. or checkp. ch. if it must stop
6      if(ch-r.hasCounterPart == true) {
7          ck-r = obtain-addr(ch-r.CAP-CP);
8          lock(cp.semlock);
9          switch(ch-r.used-channel, ck-r.used-channel) {
10         case true, true:
11             PCB.state = SPEC-WAIT-BOTH;
12             unlock(ch-r.semlock);
13             unlock(ck-r.semlock);
14             wait();
15             break;
16         case false, true:
17         case true, false:
18             PCB.state = SPEC-WAIT-ONE;
19             unlock(ch-r.semlock);
20             unlock(ck-r.semlock);
21             wait();
22             break;
23         default: //false false
24             unlock(ch-r.semlock);
25             unlock(ck-r.semlock);
26         }
27       } else { //check just the app. one
28         if (ch-r.used-channel == true)
29             PCB.state = SPEC-WAIT-ONE;
30         unlock(ch-r.semlock);
31     }
32     //sends the message
33     ch-r.used-channel = true;
34
35     //if there is counter part, it is properly managed
36     if(ch-r.hasCounterPart == true) {
37         ck-r.channel-used = true;
38         unlock(ck-r.semlock);
39     }
40     unlock(ch-r.semlock);
41     M.dest-node-id = ch-r.dest-node-id;
42     M.CAP-CH = trasmit-cap(ch-r);
43     M.CAP-PCB = transmit-cap(PCB);
44     M.L = ch-r.L;
45     release-cap(CAP-CHR);
46     M.CAP-MSG = transmit-cap(msg);
47     M.seq-num = seq-num;
48     CAP-M = transmit-cap(M);
49     interprocessor-comm(CAP-M, KP);
50 }
```

Figure 6.9: Pseudo-code describing the implementation of the send operation performed by the sender process.

```
1  KP {
2    while(true) {
3      wait();
4      //when KP returns to running state, it has received an
           interrupt
5      //and it can access a msg variable
6      switch(msg.TYPE) {
7      case M:
8        manage-M-msg(msg);
9        break;
10     case A:
11       msg-a = (A) msg;
12       send_results(msg-a);
13       break;
14     case W:
15       msg-w = (W) msg;
16       wake-sender(msg-w);
17       @?bf break^;
18     case V: //sender or receiver signals its restart
19       msg-v = (V) msg;
20       if(msg-v.dest-node-id != rt.local-id()) {
21         //remote update of routing tables
22         msg-v.addr = rt.local-addr();
23         delegate(NIC, msg-v);
24       } else { //else local update routing tables
25         //modifies source id -> address for NIC
26         rt.updateRouting(msg-v.source-node-id,
27         msg-v.addr);
28         //set the failed field
29         CAP-CHR = chidToCAP(msg-v.ch-id);
30         ch-r = obtain-addr(CAP-CHR);
31         ch-r.failed = true;
32         if(ch-r.hasCounterPart == true) {
33           ck-r = obtain-adrr(ch-r.CAP-CP);
34           ck-r.failed = true;
35           release-cap(ch-r.CAP-CP);
36         }
37         release-cap(CAP-CHR);
38
39       }
40       break;
41
42     default: //unknown message type
43     }
44   }
45 }
```

Figure 6.10: Pseudo-code describing the implementation of the *KP* process.

```
1  manage_M_msg(msg) {
2    msg_m = (M) msg;
3
4    //M received by KP on the receiving side
5    if(msg_m.dest_node_id == my_id) {
6      //send on the application channel
7      CAP_CH = channel_list(msg_m.ch_id);
8      ch = obtain_addr(CAP_CH);
9      ch.seq_num = msg_m.seq_num;
10     r1 = local_send(msg_m);
11
12     //send on the checkpointing channel:
13     if(ch.hasCounterPart == true) {
14       ck_ch = obtain_addr(ch.CAP_CP);
15       ck_ch.seq_num = msg_m.seq_num;
16       r2 = local_send(msg_m, ck_ch);
17       release_cap(ch.CAP_CP);
18     }
19
20     //builds the answer for the sender
21     A msg_a;
22     msg_a.source_node_id = my_id;
23     msg_a.ch_id = msg_r.ch_id;
24     msg_a.proc_id = msg_r.proc_id;
25     msg_a.seq_num = msg_r.seq_num;
26     msg_a.R_app = r1;
27     if(ch.hasCounterPart == true)
28       msg_a.R_chk = r2;
29
30     release_cap(CAP_CH);
31
32     delegate(NIC, msg_a);
33   } else //The sending KP receive an M msg from S
34     //delegates the send to the NIC
35     delegate(NIC, msg);
36 }
```

Figure 6.11: Pseudo-code describing the implementation of the *KP* process when receiving a message of type **M**.

```
1  local_send(m, ch) {
2    lock(ch.semlock);
3    if(ch.wait == true) {
4      //receiver waiting: direct copy
5      ch.wait = false;
6      unlock(ch.semlock);
7      vtg-addr = obtain-addr(ch.CAP-VTG);
8      msg-addr = obtain-addr(m.CAP-MSG);
9      copy(msg-addr, vtg-addr);
10     release-cap(ch.CAP-VTG);
11     release-cap(m.CAP-MSG);
12     CAP-PCB = procIdToPCB(ch.proc-id);
13     pcb = obtain-addr(CAP-PCB);
14     wake(pcb);
15     release-cap(CAP-PCB);
16   } else {
17     //copy in the message queue
18     vtg-addr = obtain-cap(ch.Q-CAP[CH.INS]);
19     msg-addr = obtain-cap(m.CAP-MSG);
20     copy(msg-addr, vtg-addr);
21     release-cap(ch.Q-CAP[CH.INS]);
22     release-cap(m.CAP-MSG);
23     ch.INS = (ch.INS + 1) mod (ch.K + 1);
24     ch.SIZE = ch.SIZE + 1;
25     //the queue is full
26     if (ch.SIZE = ch.K + 1) {
27       ch.wait = true;
28       ch.proc-id = m.proc-id;
29       unlock(ch.semlock);
30       return 'stop';
31     }
32   }
33   return 'go';
34 }
```

Figure 6.12: Pseudo-code describing the implementation of the send operation performed on the receiving node.

```
1  send_results(a) {
2    CAP_PCB = procIdToPCB(a.proc_id)
3    pcb = obtain_addr(CAP_PCB);
4    if(a.R_chk != NULL) {
5      //logged channel
6      CAP_CH = chIdToCAP(a.ch_id);
7      ch_r = obtain_addr(CAP_CH);
8      ck_r = obtain_addr(ch_r.CAP_CP);
9      switch(a.R_app, a.R_chk) {
10     case 'stop', 'stop':
11       if(pcb.state == WAITING or
12                pcb.state == SPEC_WAIT_BOTH) {
13           pcb.state = WAITING;
14           ch_r.used_channel = false;
15           ck_r.used_channel = false;
16       }
17       break;
18
19     case 'stop', 'go':
20       ck_r.used_channel = false;
21       if(pcb.state == WAITING or
22                pcb.state == SPEC_WAIT_BOTH) {
23              ch_r.used_channel = false;
24              pcb.state = WAITING;
25       }
26       break;
27
28     case 'go', 'stop':
29       ch_r.used_channel = false;
30       if(pcb.state == WAITING or
31                pcb.state == SPEC_WAIT_BOTH) {
32              ck_r.used_channel = false;
33              pcb.state = WAITING;
34       }
35       break;
36
37     case 'go', 'go':
38       ch_r.used_channel = false;
39       ck_r.used_channel = false;
40       if(pcb.state == WAITING or
41                pcb.state == SPEC_WAIT_ONE or
42                pcb.state == SPEC_WAIT_BOTH) {
43              pcb.state = RUNNING;
44              wake(pcb);
45       }
46       break;
47
48     default: //unknown result types
49   }
50   release_cap(ch_r.CAP_CP);
51   release_cap(CAP_CH);
52   } else {
53       ch_r = obtain_addr(a.CAP_CH);
54       if(a.R_app == 'stop') {
55         if(pcb.state == SPEC_WAIT_ONE)
56             ch.used_channel = false;
57         else
58             ch.used_channel = true;
59       } else {
60           ch.used_channel = false;
61           if(pcb.state == SPEC_WAIT_ONE) {
62             pcb.state = RUNNING;
63             wake(pcb);
64           }
65         }
66         release_cap(a.CAP_CH);
67       }
68   release_cap(CAP_PCB);
69 }
```

Figure 6.13: Pseudo-code describing the implementation of the send result notification, performed by the receiving KP.

```
1  wake_sender(w) :: {
2      CAP-CH = chIdToCAP(w.ch-id);
3      ch = obtain-addr(CAP-CH);
4      ch.used-channel = false;
5      if(ch.hasCounterPart == true) {
6          pcb-s = obtain-addr(a.CAP-PCB);
7          if(pcb-s.state != SPEC-WAIT-BOTH) {
8            //I can wake up the process
9            pcb-s.state = RUNNING;
10           wake(pcb-s);
11         } else { //process now waits for 1 wake up message
12             pcb-s.state = SPEC-WAIT-ONE;
13         }
14         release-cap(a.CAP-PCB);
15     }
16     release-cap(CAP-CH);
17 }
```

Figure 6.14: Pseudo-code describing the implementation of the receiving behavior of the wake-up messages on the sending KP.

```
1  receive(ch-id, tgv, seq-num) {
2    CAP-CH = chidToCAP(ch-id);
3    ch = obtain-addr(CAP-CH);
4    lock(ch.semlock);
5    if(ch.SIZE == 0) {
6      ch.wait = true;
7      ch.proc-id = getMyID();
8      unlock(ch.semlock);
9      wait();
10   }
11
12   //a message is ready
13   msg-addr = obtain-addr(Q-CAP[EXT]);
14   copy(msg-addr, tgv);
15   release-cap(Q-CAP[EXT]);
16   ch.SIZE = ch.SIZE - 1;
17   ch.EXT = (ch.EXT + 1) mod (ch.K + 1);
18
19   if(ch.wait == true) { //sender waiting:
20     //KP delegated to send a wake up message
21     ch.wait = false;
22     unlock(ch.semlock);
23     W msg-w;
24     msg-w.source-node-id = getNodeID();
25     msg-w.ch-id = ch-id;
26     msg-w.proc-id = ch.proc-id;
27     msg-w.seq-num = ch.seq-num;
28     delegate(NIC, msg-w);
29   } else unlock(ch.semlock);
30   release-cap(CAP-CH);
31 }
```

Figure 6.15: Pseudo-code describing the implementation of the receive operation.

```
1 S {
2   int count = 0;
3   while(cond) {
4     element = F(state);
5     send(CH, element, count);
6     count++;
7     if(count % DELTA_SEND == 0)
8       checkpoint(state,count);
9   }
```

Figure 6.16: Pseudo-code of the sender in the test-bed of FT-Streams.

```
1 R {
2   int count = 0;
3   while(cond) {
4     receive(CH, tgv, count);
5     G(state, tgv);
6     count++;
7     if(count % DELTA_RCV == 0) then
8       checkpoint(state,count);
9   }
```

Figure 6.17: Pseudo-code of the receiver in the test-bed of FT-Streams.

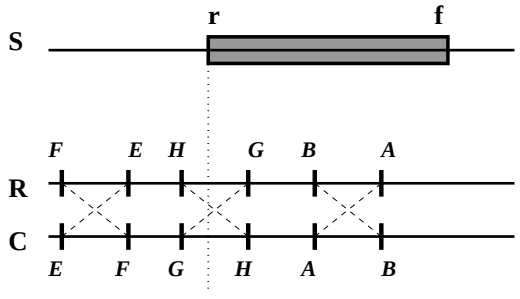


Figure 6.18: Relative positions of S and R in the case of concurrent and recursive failures. The italic letters indicates the relative positions of R and C w.r.t. the one of S.

Chapter 7

Performance of Parallel Computations in Presence of Failures

In the previous chapters we have defined cost models describing the performance of the (chosen) structured parallel constructs. The models quantify the overhead induced by the fault tolerance support to the performance of the computations and they can be used for optimization purposes. The idea is that, by exploiting such cost models, we can decide between several available fault tolerance mechanisms and optimize them w.r.t. the actual configuration of the execution platforms and the application. The models are applied to the parallel structure and to quantitative aspects of the implemented applications (e.g. the time needed to perform a task), and the (possibly dynamic) configuration of the exploited execution platform. We have assumed that such quantities can be derived by simulation and/or analysis of the applications and environments.

In this chapter we start from the observation that the presented cost models can be used to obtain an evaluation of the performance of applications *in the absence of failures*. In this thesis we are also interested in studying the behavior of the performance in the case of failures during the computation. This is especially important if our aim is to provide some guarantee of the final performance of applications, given by the overhead of both the fault tolerance support and the failures. We target *task parallel* computations and we base our study on the farm parallel construct which we described in Chapter 3. We present a study of the completion time of failure execution of farm computations [15, 16]. The research is based on the *master-slave* implementation strategy which is presented in Chapter 4: as we focus on the modeling of the impact on the performance of task execution failures, we are not interested in decoupling the scheduling and collection tasks. We present an analytical study, based on Markov chains, which abstracts the actual implementation model to obtain an upper bound of the completion time. The abstraction of the implementation model is made to introduce in a simple way a Markov model and

to allow us to obtain upper bounds of the completion times. That is, the abstract Markov model is not used to implement farm computations, but *only* to model thei performance in presence of failures. We briefly describe the implementation of the master-slave strategy for farm computations and we show the assumptions under which we work in this study.

7.1 Implementation of the Master-Slave Farm Strategy and its Fault Tolerance Support

Recall that in the master-slave implementation strategy for farm computations (see Chapter 4, Figure 4.1) the master is responsible of scheduling tasks to slaves and of collecting results. The slaves are the unit of parallelism, which behavior is to iteratively obtain a task from the master, perform it and return the result. In short, they implement the computation stage of the farm. In this chapter we illustrate our approach by implementing farm computations as distributed processes communicating through message passing:

- The master is mapped on a process.
- Each slave is mapped in a (different) process.
- The I-Structures interconnecting the master to the slaves and vice-versa are implemented as asynchronous communication channels supported by failure detection without message logging.

We assume that each process of the computation is executed on a different node. We do not exploit stable storage mechanisms, as we assume (see below) that the master cannot fail. Actually, the master represents the centralization point of the fault tolerance implementation. We claim that the failure-free property of the master can be obtained by mapping it on a robust node or by exploiting some replication techniques as we described in Chapter 2 (see for example [14]).

The fault tolerance support is implemented by the master process and it is based on the *re-scheduling* of failed tasks: whenever a slave fails the master detects the failure and it re-schedules the lost task to an available slave. We also assume that failed slaves are eventually restarted by some sub-system and that they re-join the computation. In Section 7.8(b) we detail an implementation of this strategy for the master-slave scheme of the `muskel` programming environment.

7.2 Working Assumptions

We make further assumptions w.r.t. the previous chapters, which we claim to be fundamentally viable in the context of high-performance computing and platforms. A complete list of the assumptions follows:

- Failure semantics is according to the fail-stop model and it is limited to slave failures. Failure detection is assumed to be performed in known average time. Failed slaves are re-started after known average time (see below).
- Failure probability is uniform on the set of slave processes. Each slave fails independently w.r.t. the other ones. The failure probability for the execution of a task is assumed to be known and we denote it with the variable q. Consequently, it is also known which is the probability of succeeding in performing a task, which we denote with the variable p (where $p = 1 - q$). The value of q (and p) is an input of our model.
- The average task execution time is known, it is an input of our model, and it is denoted with the variable δ. It can be noticed that an high variance of actual values lowers the accuracy of the abstract model.
- The fault correction time is known and it is sum of two known variables, which are an input of our model: (a) the time needed to detect a failure, denoted with Δ_F and (b) the time needed to restart a slave process, denoted with Δ_R. Their sum, plus half the time to perform a task ($\delta/2$), is denoted with Δ.
- The number of tasks to be performed in the whole computation is known and we denote it with n.
- The number of exploited resources is known and fixed (i.e. the parallelism degree is fixed). We denote this value with m.

The stronger assumptions we make are related to the failure distribution (uniform) and to the knowledge of its probability (q). We claim that such assumptions can be made for massively parallel systems by performing proper static analysis of the architectures. For instance, see [75] for an example of estimation of failure probability analysis by means of system logs.

7.3 Upper Bounding Completion Time by Means of a Bulk-Synchronous Model

In this analysis we are interested in obtaining an upper bound of the actual completion time of farm computations. For this purpose we build a proper abstract computation model and we show the differences with the actual implementation model.

In the actual computation behavior consider a steady-state of the master execution in which all slaves are scheduled with a task. Suppose that k tasks remain to be scheduled: whenever one of the slaves returns the result of a task the master directly re-schedules it with a new task. Thus $k-1$ tasks remain to be scheduled. In another case suppose that one of the slave fails. The master detects the failure and,

if a slave is available, it directly re-schedules the lost task. We notice that there is not an explicit coordination between the scheduling actions of the master towards different slaves.

We build an abstract computation model according to a Bulk-Synchronous Parallel computation model [80]. The abstract model we introduce here is used *only* for modeling purposes, as we do not exploit it in the implementation. Actually, it is the quantitative difference between the BSP and implementation models which introduces the upper-bounds over the completion times. In the abstract model the master executes according to iterated supersteps, each one including the following actions:

- It schedules a task to each slave.
- It waits for *all* slaves to either return a result or fail.
- In the case of failure(s), it waits for all failed slaves to be re-started.

According to the computation variables described above and assuming that we include in δ and Δ the scheduling time of the master, each superstep completion time is the maximum between δ and Δ. This is true because the master waits for all slaves to terminate or fail before re-scheduling the next set of m tasks. We denote the superstep completion time with $\mu = \max(\delta, \Delta)$.

It is straightforward to notice that the implementation behavior provides lower completion time w.r.t. the BSP-like behavior because it opportunistically re-schedules slaves whenever they are available independently of all other slaves. In this thesis we do not give a formal proof of the quantitative differences between the two models and we study it in future work (see Chapter 8).

7.4 A Markov Model for Fault-Tolerant Farm Computations

A Markov chain [54] describes a stochastic process as a set of states describing the process states, and transitions linking states. Transitions between states feature the probability of being chosen, i.e. the probability of being in the source state and transit towards the destination state. They can also feature costs, i.e. whenever a transition is taken we pay the related cost. Markov chains have been exploited to model the timing behavior of computations (both sequential and parallel), in the following way [85]:

- Proper state values of the computation are mapped into chain states.
- Transition probabilities between the state of the computation (due for example to conditional jumps) are mapped into transition probabilities between Markov states.

- Transition costs, which represent the time needed to perform a sequence of operations, are mapped into costs of transitions between markovian states.

We briefly review two notable research studies previously presented in the literature from which we inherit the analytical formulas. Next we introduce a Markov model of the BSP-like computation described in the previous section.

7.4.1 Existing Approaches to Reliability Quantification by Exploiting Markov Chains

In [87] a markovian model is exploited to analyze the performance impact, due to failures, in the case fault tolerance is achieved by means of a two-level recovery scheme. The computation model consists in a set of processes executing a distributed program and which interact through message-passing. The processes are assumed to periodically take consistent checkpoints. The recovery scheme is two-level in the sense that checkpointing is performed on both volatile and stable storage supports at different frequencies. The recovery procedure changes according to the kind of failure and the availability of collected information to the restarted process. The markov model that is developed allows the author to: (a) prove that the two-level scheme enable higher resiliency degrees w.r.t. one-level schemes; (b) show which is the time needed to perform a portion of the computation in the case of failures.

In [94] the computation model consists of performing a set of independent task, supporting fault tolerance by checkpointing, i.e. partial execution of tasks is periodically checkpointed. The failure model considered includes software errors, i.e. the execution of a task can deliver an incorrect result. A Markov Reward Model is exploited to analyze the performance impact of checkpointing schemes based on task duplication. Parameters of the Markovian model are the average time intercurring between checkpoints, and the total average task execution time. The paper analyzes four different schemes for fault tolerance, mixing checkpointing, software replication techniques and forward recovery.

7.4.2 Simple Model Examples

We start the description of the Markovian model for farm computations by showing two simple examples. These will be used as building blocks of model for the general case.

One task performed by one slave. Figure 7.1 shows a Markov chain $\mathcal{M}_1$ describing the execution of a single task on a single slave. In the case of failure, we re-schedule the task on the same slave after it is restarted, according to the fault tolerance strategy. In the figure, circles are states which labels represent the number of tasks that remain to be computed. Each transition is labeled with a triplet. The first component is a symbol in the set $\{F, S\}$, denoting respectively the failure or success of the task execution. The second component is the transition probability. The

third component is the cost of the transition, that in our model is the time needed to perform it. This component is a symbol in the set $\{delta, \Delta, \mu\}$. The computation starts from the state 1, denoting that one task remains to be performed. In the case of failure F, we remain in state 1 with probability q. In the case of success S we move to state 0 with probability p. State 0 corresponds to the end of the execution. For modeling purposes, we added the dummy arc from the state 0 to itself labeled with probability 1. In the model, this arc denotes that when we transit to 0 we cannot transit out of it. By definition of Markov chains, this property defines the chain as an *Absorbing Markov Chain*[85]. In an absorbing Markov chain, any computations eventually terminates in one of the absorbing states, with probability 1. Indeed, if we compute the probability of ending in this state, we obtain:

$$p + pq + pq^2 + pq^3 + \cdots = p(1 + q + q^2 + q^3 + \cdots) = \frac{p}{1-q} = 1$$

In our study, we exploit the properties of absorbing Markov chains to evaluate the completion time for farm computations. We will see that all the graphs that we derive, especially the one for the general model, are absorbing Markov chains.

In Figure 7.1 notice that the case of failure takes $\Delta = \Delta_F + \Delta_R + (\delta/2)$ time, according to its definition. The meaning behind this choice is that, in the case of failure of one of the slaves, we have to wait for the detection of the fault and for the restart of the slave (or transparently for the recruiting of a new computational node). As the failure happens some time after the execution has started, we count an average task execution time before the fault, equal to half of the task execution time.

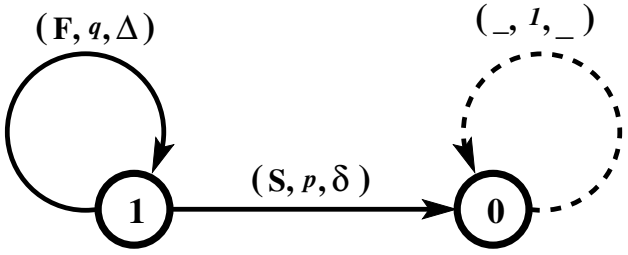


Figure 7.1: Markov chain $\mathcal{M}_1$ modeling the execution of 1 task on 1 slave. Arcs are labeled with triplets (a, b, c), where a is the result of the try (success or failure), b is the probability of the transition, and c is the time required by the transition.

We compute the average number of re-tries τ_1 in $\mathcal{M}_1$. $\mathcal{M}_1$ can be described with the recursive equation $\tau_1 = q(\Delta + \tau_1) + p\delta$. Defining $t_1 = p\delta + q\Delta$ we get $\tau_1 = q\tau_1 + t_1$ and finally

$$\tau_1 = \frac{1}{1-q} t_1 = \frac{1}{p} t_1 = \delta + \frac{q}{p}\Delta$$

It is tempting to compute τ_1 directly as an expectation, then:

$$\tau_1 = \Delta p + (\Delta + \delta)qp + (2\Delta + \delta)q^2p + (3\Delta + \delta)q^3p + \cdots$$

The first term of the expression represents the case without failures. We multiply the time needed to perform the task with the probability of having no failures. The following terms represent increasing numbers of failures, and are multiplied by the corresponding overheads (i.e. Δ time for each failure, and δ time for the last successful execution). We explicit the computations: using $1 + q + q^2 + q^3 + \cdots = 1/(1-q) = 1/p$ we rewrite as:

$$\begin{aligned}
&\delta p + \delta qp + \delta q^2p + \delta q^3p + \cdots + \Delta qp + 2\Delta q^2p + 3\Delta q^3p + \cdots \\
&\qquad = \delta p\left(1 + q + q^2 + q^3 + \cdots\right) + \Delta pq\left(1 + 2q + 3q^2 + \cdots\right) \\
&\qquad = \delta + \Delta pq\left(1 + 2q + 3q^2 + \cdots\right)
\end{aligned}$$

A closed form for $1 + 2q + 3q^2 + \cdots$ is computed just deriving,

$$\frac{d}{dq}\Big(\frac{1}{1-q}\Big) = \frac{d}{dq}(1+q+q^2+q^3+q^4+\cdots) = 1+2q+3q^2+4q^3+\cdots = \frac{1}{(1-q)^2} = \frac{1}{p^2}$$

Thus, the second expression is $q\Delta/p$ and we re-obtain, as expected $\tau_1 = \delta + \Delta\frac{q}{p}$

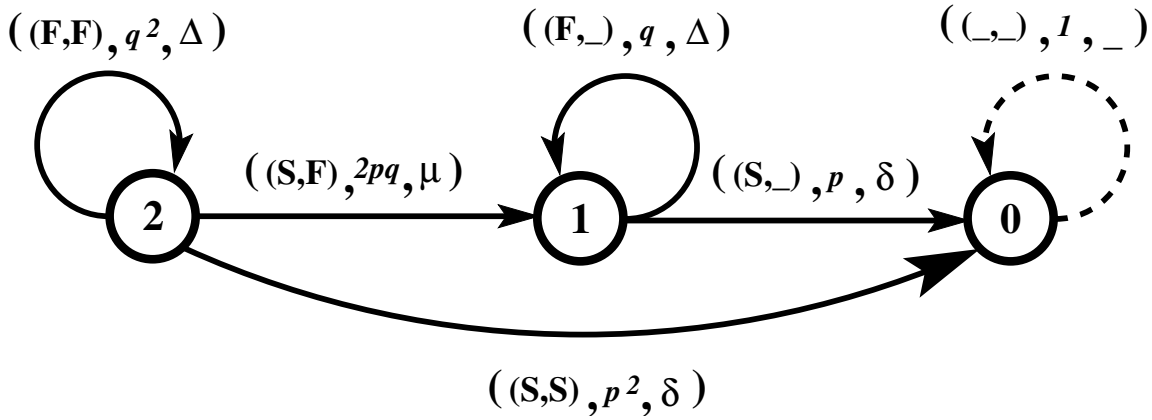


Figure 7.2: Markov chain $\mathcal{M}_2$ modeling the execution of 2 tasks on 2 interpreters.

Two tasks performed by two slaves. The next example we consider is the execution of 2 tasks on 2 slaves modeled by the Markov chain $\mathcal{M}_2$ of Fig.7.2. The initial state 2 corresponds to the assignment of the 2 tasks to the 2 interpreters. If both fail (edge labeled with F, F), we remain in state 2. This event has probability q^2 and takes time Δ. If one fails, and the other succeeds, we transit in the state 1 (transition labeled F, S) with probability $2pq$. It takes $\mu = \max\{\delta, \Delta\}$ to execute it. Recall

that we use the max over the two times because, in the BSP-like model, we have to wait for a results from both tasks to decide for the next state. If both succeed, we transit to state 0 (edge S, S) with probability of p^2. It takes δ time (i.e. the parallel execution time). From state 1 we re-assign the failed task to an interpreter (possibly the first available). Notice that similarly to the previous example, we have defined an absorbing Markov chain, as the state 0 is an absorbing state.

We compute the average expected time τ_2. The Markov chain $\mathcal{M}_2$ give us the recursive equation $\tau_2 = q^2(\Delta+\tau_2)+2pq(\mu+\tau_1)+p^2\delta$, defining $t_2 = q^2\Delta+2pq\mu+p^2\delta$ we rewrite $\tau_2 = q^2\tau_2 + 2pq\tau_1 + t_2$ and substituting τ_1 we obtain

$$\tau_2 = \frac{2qt_1}{1-q^2} + \frac{t_2}{1-q^2} = \frac{2pq}{1-q^2}(\delta+\mu) + 3\Delta\cdot\frac{q^2}{1-q^2} + \delta\cdot\frac{p^2}{1-q^2}$$

The preceding formula can also be obtained computing directly the expectation. To do that, we compute the average completion time of the paths to the absorbing states independently. Using $i \rightsquigarrow j$ to denote paths going from state i to state j we have

$$\tau_2 = \sum_{w_1=2\rightsquigarrow 1} t(w_1)\cdot p(w_1) + \frac{2pq}{1-q^2}\cdot\sum_{w_2=1\rightsquigarrow 0} t(w_2)\cdot p(w_2) + \sum_{w_3=2\rightsquigarrow 0} t(w_3)\cdot p(w_3)$$

We have

$$\sum_{w_1=2\rightsquigarrow 1} t(w_1)\cdot p(w_1) = 2\mu\frac{pq}{1-q^2} + 2pq\Delta\frac{q^2}{(1-q^2)^2}$$

$$\frac{2pq}{1-q^2}\cdot\sum_{w_2=1\rightsquigarrow 0} t(w_2)\cdot p(w_2) = \frac{2pq}{1-q^2}\left(\delta+\Delta\frac{q}{p}\right)$$

$$\sum_{w_3=2\rightsquigarrow 0} t(w_3)\cdot p(w_3) = \delta p^2\frac{1}{1-q^2} + p^2\Delta\frac{q^2}{(1-q^2)^2}.$$

Summing up the three parts and after some simplifications we get the result. For brevity we avoid to show all the computations.

7.4.3 A General Model

We generalize the two example to the case of n tasks performed on m slaves, where $n > m$. The Markov chain-based model is shown in Fig. 7.3. At the beginning of the computation, m tasks are scheduled to the slaves. The next state depends on the number of failures/successes:

- If we get m failures, we remain in the initial state. The probability that this happens is equal to q^m. The time spent in this operation is equal to Δ, and the time needed to terminate the computation is equal to the execution of n tasks on m slaves, as at the beginning.

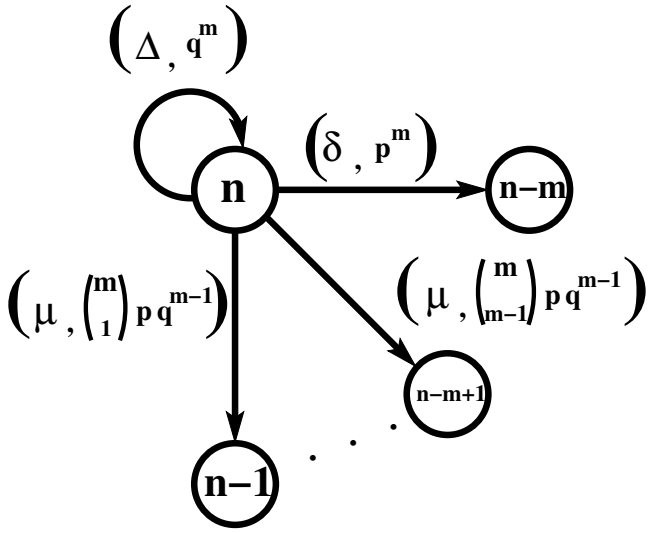


Figure 7.3: Markov chain modeling the execution of n task on m slaves, in the case of failure/restart of the processes.

- The cases from 1 to m-1 failures make the state transit in states from $n-m+1$ to 1, respectively. The probability of each transition is computed exploiting a binomial expression, properly multiplied for the probabilities. The time needed to perform the operation is the maximum between δ and Δ (i.e. μ). The time needed to complete the computation depends on the amount of remaining tasks.

- If we get all successes, we transit in the state labeled with $n - m$. The probability of such a transition is p^m, with time equal to the average task execution time. The time needed to complete the computation, in this case, is equal to the computation of $n - m$ tasks.

We can exploit these observations to represent the general model with a single recurrence over the completion times:

$$\tau_{n,m} = q^m(\Delta + \tau_{n,m}) + \sum_{k=1}^{m-1} \binom{m}{k} p^k q^{n-k}(\mu + \tau_{n-k}) + p^m(\delta + \tau_{n-m,m})$$

The first part of the expression is related to m failures. This takes Δ time and the additional time needed to perform the whole computation, as we have still to perform all tasks. The second part of the expression is related to 1 to m-1 failures. It takes μ time, depending on the actual δ and Δ values, plus the time needed to perform the rest of the computation. The last part is related to M successes, that takes δ time, plus the time needed to perform the rest of the computation.

It is interesting to show how the general model can be instantiated (Fig. 7.4), to get some insights on the complexity of the computations required to evaluate the completion time. For the case of n tasks performed on 2 slaves, we show the linear recurrence expression, and its solution. The mathematical steps we show are only the main ones, according to a standard methodology to solve linear recurrences. The recurrence expression for n tasks performed by 2 slaves is:

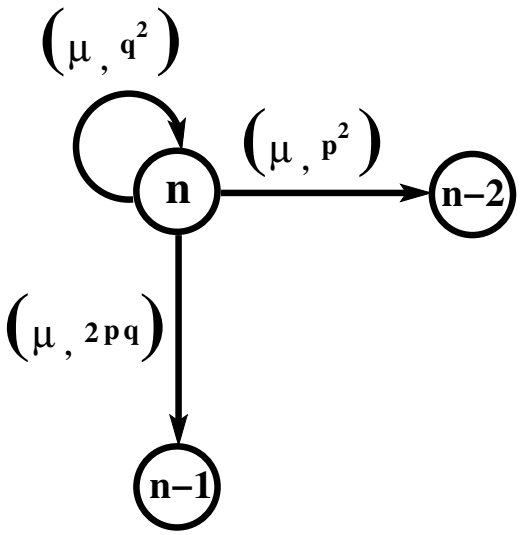


Figure 7.4: Markov chain of the model for n tasks performed on 2 slaves.

$$\tau_n = q^2 \cdot (\Delta + \tau_n) + 2pq \cdot (\mu + \tau_{n-1}) + p^2 \cdot (\delta + \tau_{n-2})$$

This formula can be expressed in the classical form of linear recurrences as:

$$\tau_n = \underbrace{\frac{q^2}{1-q^2}\tau_{n-1} + \frac{2pq}{1-q^2}\tau_{n-2}}_{homogeneous\ part} + \underbrace{\frac{q^2}{1-q^2}\Delta + \frac{2pq}{1-q^2}\mu + \frac{p^2}{1-q^2}\delta}_{inhomogeneous\ part}$$

Direct Solution for 2 Slaves

To solve the recurrence, three main steps can be followed:

1. Solve the homogeneous part of the recurrence.

2. Find particular solutions.

3. Substitute boundary conditions in the general solution to develop it.

The homogeneous part of the linear recurrence is:

$$(1-q^2)\tau_n - 2pq\tau_{n-1} - p^2\tau_{n-2}$$

and we impose it to be equal to zero:

$$(1-q^2)\alpha^2 - 2pq\alpha - p^2 = 0$$

We solve it, and we obtain two solutions:

$$\alpha = \begin{cases} 1 \\ \frac{q-1}{q+1} \end{cases}$$

Thus, the homogeneous solution has the following form:

$$\gamma_n = A\left(\frac{q-1}{q+1}\right)^n + B(1)^n$$

Next, we find the particular solutions: we guess one of the same or higher degree of the inhomogeneous, and we verify it is correct. We have found that

$$\tau_n = cn$$

is a particular solution, where:

$$c = \frac{q^2\Delta + 2pq\mu + q^2\delta}{2p}$$

Finally, we obtain the general solution. It has the form:

$$\tau_n = \textit{homogeneous solution} + \textit{particular solution}$$

We substitute with the expressions found above and we obtain.

$$\tau_n = A\left(\frac{q-1}{q+1}\right)^n + B + \frac{q^2\Delta + 2pq\mu + q^2\delta}{2p}$$

Classically, we substitute the following boundary conditions:

$$t_1 = p\delta + q\Delta$$

and

$$t_2 = p^2\delta + 2pq\mu + q^2\Delta$$

We can rework the solution in a way that it depends only on the A variable:

$$\tau_n = A\left(\frac{q-1}{q+1}\right)^n + \frac{t_2}{2p}n - A\left(\frac{q-1}{q+1}\right)^2 - \frac{q}{1-q^2}(t_2 - 2t_1)$$

By substituting the boundary conditions t_1 and t_2 we obtain:

$$A = \frac{\frac{1}{1-q^2}\cdot\left(\frac{3+q}{2}\cdot t_2 - 2qt_1\right) - \left(1+\frac{2pq}{1-q^2}\right)\cdot\frac{t_2}{1-q^2} - \frac{1}{1-q^2}\left(\frac{2pq}{1-q^2}\cdot 2pq + p^2\right)\cdot\frac{t1}{p}}{\left(\frac{q-1}{q+1}\right)^2\cdot\frac{2}{(q+1)}}$$

Below we re-organize the general model to allow an evaluation of the completion time by means of approximations, without finding a direct solution to the Markov chain.

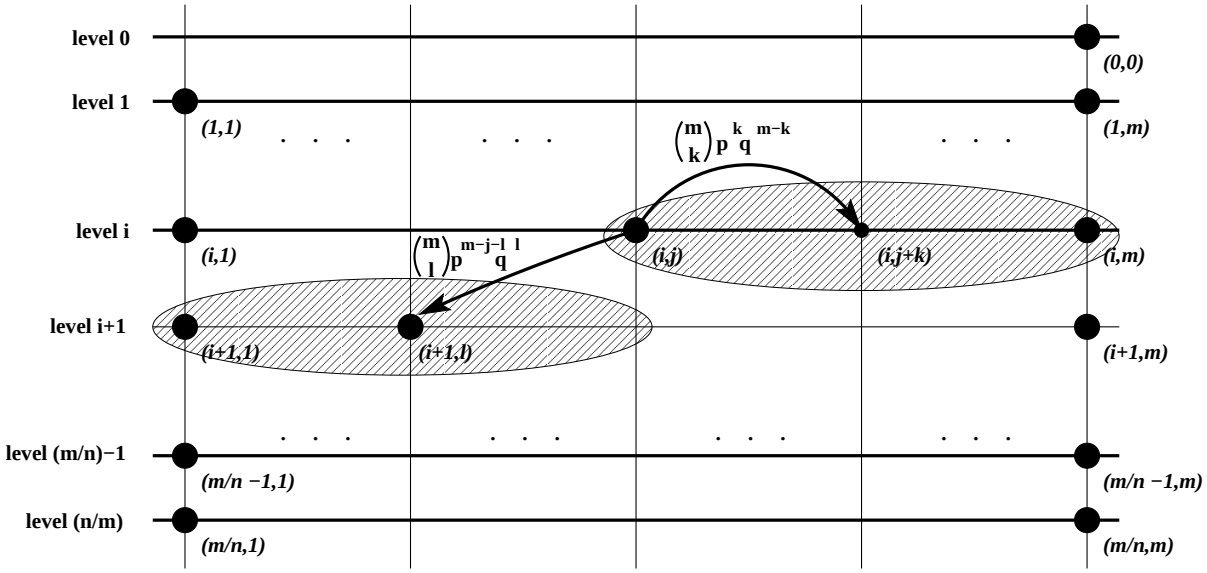


Figure 7.5: Representation of the Markov model of computation of n tasks on m slaves (with $n >> m$), where each slave can fail to perform a task with probability q.

7.4.4 Re-Organizing the Model

We re-organize the Markov model of Figure 7.3 to highlight interesting properties for its evaluation. Figure 7.5 is a graphical representation of the re-organized model. The states of the Markov chain are represented as black circles labeled with two indexes. The first index denotes the *level* (see below) at which we place the state, while the second one denotes the relative position inside the level. A state of the Markov chain represents a specific state in the computation we have reached. It can be formalized as the number of tasks that remain to be performed:

$$t(i,j) = \begin{cases} n & i = 0 \text{ and } j = 0 \\ n - ((i-1)m + j) & \text{oth.} \end{cases}$$

With the initial state $(0,0)$ we represent the fact that n tasks have to be performed. Each level of the model includes exactly m states. For simplicity, we assume that n is a multiple of m. We define an initial level 0 that includes just the initial state $(0,0)$. Level 1 include states from $(1,1)$ to $(1,m)$, where: in $(1,1)$ we have performed a single tasks, and $n-((1-1)m+1) = n-1$ are still to be performed. In $(1,m)$ we have performed m tasks, and $n-((1-1)m+m) = n-m$ are still to be performed. Below, for labeling the states, we sometimes exploit their $t(i,j)$ value, instead of the pair notation (i,j).

Now consider the state $(0,0)$. Recall that we exploit exactly m resources to perform tasks. We assign a task to each of the m resources, and we can obtain 0 to

m failures. In the case we obtain m failures, we remain in state $(0,0)$. In the case we obtain $m-1$ failures, we have performed 1 task, and we transit in state $(1,1)$, in the next level. If we obtain 0 failures, we go directly down to state $(1,m)$. As the maximum degree of parallelism is m, we cannot obtain more than m successes. Thus, we cannot go from level 0 to level 2 *with just one transition*. We have first to pass from at least one of the states belonging to level 1. This is the general semantics behind the characterization in levels of the Markov model.

In a generic computation, we will transit in some states inside the same level. Next, we jump into a state in the next level. The transition probabilities and their weights are given by the general formula for $n \geq m$, that we recall:

$$T_{n,m} = q^m(\Delta + T_{n,m}) + \sum_{k=1}^{m-1} \binom{m}{k} p^k q^{n-k}(\mu + T_{n-k,m}) + p^m(\delta + T_{n-m,m})$$

Consider the highlighted state (i,j) in Fig. 7.5. We schedule m tasks, and we can:

- Remain in (i,j), if we obtain m failures. This event has probability q^m and it costs Δ.
- Transit in a state at the right of (i,j), if the number of successes is not sufficient to change level. For instance, we can transit in state $(i,j+k)$, if we obtain k successes, and $m-k$ failures. This transition has probability $\binom{m}{k}p^k q^{m-k}$, and it has a weight of μ seconds.
- Transit down-left in a state $(i+1,j-l)$ in the next level. In this case, we obtain a number of successes sufficient to make us change level. In this specific case, the transition has probability $\binom{m}{l}p^{m-j-l}q^l$, and it costs μ seconds.
- Transit directly down, if we obtain m successes. We take this transition with probability p^m, and it costs δ seconds, i.e. the average tasks execution time (as we perform all tasks in parallel).

It remains to consider the case $0 \leq n < m$. Of course, when $n = 0$ it holds $T_{0,m} = 0$. For $0 < n < m$, we have

$$T_{n,m} = q^n(\Delta + T_{n,m}) + \sum_{k=1}^{n-1} \binom{m}{k} p^k q^{n-k}(\mu + T_{n-k,m}) + p^n\delta$$

Note that, two types of transitions are not admitted in our model. Transitions to the left are forbidden. According to the fault tolerance model, we cannot lose results that we have performed in previous steps (see Sect. 7.1). Transitions to the down-right are also forbidden. This happens we can obtain up to m successes for a transition, and going to the down-right means that we obtain more than m successes.

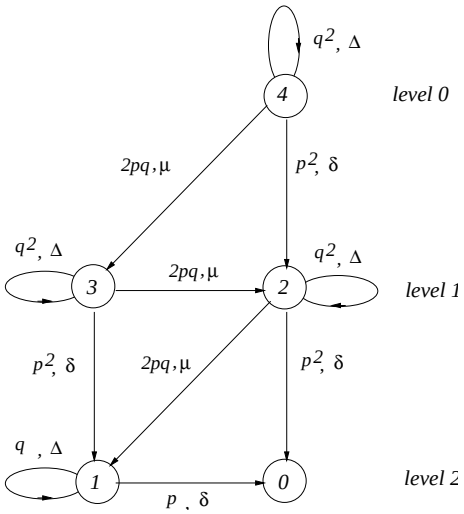


Figure 7.6: Markov model of computation of $n = 4$ tasks on $m = 2$ slaves, each slave can fail with probability q. State numbering denotes the number of pending tasks.

In general, each computation features a different path in the computation states, but we can characterize all paths as a behavior. Consider the level k: we transit in one of the states of level k, from a state of the level $k-1$; we transit in one or more states (possibly all) of the level k, until we jump to one of the states at the next level $k+1$. From a quantitative viewpoint, we can characterize each level with two quantities. $T_{stay}(k)$ is the *average time passed in level k* and $T_{jump}(k)$, the *average time needed to jump from level k to level* $k+1$. According to [87], $T_{stay}(k) = \sum_{a,b \in k} V_{a,b} Q_{a,b} N_a$, the sum is over all transitions $a \rightarrow b$ in level k, $V_{a,b}$ is the time needed in $a \rightarrow b$, $Q_{a,b}$ is the state transition probability of $a \rightarrow b$ and N_a and that is the average number of entries into state a. Similarly, we can compute the average time needed to jump from a level k, to its successor $k+1$ by considering all the transitions from a state a at level k to the state b at level $k+1$: $T_{jump}(k) = \sum_{a \in k, b \in k+1} V_{a,b} Q_{a,b} N_a$. The expected time can be computed as

$$T_{n,m} = \sum_{0 \leq k \leq m/n} T_{stay}(k) + \sum_{0 \leq k < m/n} T_{jump}(k)$$

Example with 4 tasks and 2 slaves

We develop the preceding approach in a small example with $m = 2$ and $n = 4$. Denoting $T_{k,2} = \tau_k$ for $0 \leq k \leq 4$ we have the equations

$$\begin{aligned} \tau_k &= q^2(\tau_n + \Delta) + 2pq(\mu + \tau_{k-1}) + p^2(\delta + \tau_{k-2}) \ , \ 2 \leq k \leq 4 \\ \tau_1 &= q(\tau_1 + \Delta) + p\mu \ , \ \tau_0 = 0 \end{aligned}$$

For $2 \leq k \leq 4$ the recurrence for τ_k follows the general formula. For $k = 1$ only one slave is needed because only one task need to be executed.Finally for $k = 0$, $\tau_0 = 0$ because there are no pending tasks. The Figure 7.6 is a rewriting of the recursive equations in terms of the absorbing Markov chain (the Figure 7.6 is a concrete example of Figure 7.5). States in Figure 7.6 are numbered by the number of pending tasks, 4 is the initial state and 0 is the final absorbing state.

In the following we use recurrence equations to find $T_{stay}(k)$ and $T_{jump}(k)$ for $0 \leq k \leq 2$. Starting from 4, the expected time before absorption verifies $\tau_4 = q^2(\tau_4+\Delta)+2pq(\mu+\tau_3)+p^2(\delta+\tau_2)$ and therefore $\tau_4 = \frac{q^2}{1-q^2}\Delta+\frac{2pq}{1-q^2}(\mu+\tau_3)+\frac{p^2}{1-q^2}(\delta+\tau_2)$. This equation give us directly $T_{stay}(0) = \frac{q^2}{1-q^2}\Delta$ and $T_{jump}(0) = \frac{2pq}{1-q^2}\mu+\frac{p^2}{1-q^2}\delta$. Using the notations given in Section 7.4.4

$$\begin{aligned} T_{stay}(0) &= V_{4,4}Q_{4,4}N_4 = \Delta q^2\frac{1}{1-q^2} \\ T_{jump}(0) &= V_{4,3}Q_{4,3}N_4 + V_{4,2}Q_{4,2}N_4 = \mu 2pq\frac{1}{1-q^2} + \delta p^2\frac{1}{1-q^2} \end{aligned}$$

where we can identify $N_4 = \frac{1}{1-q^2}$. From this point, we have to analyze the system from level 1 represented by the expression $\frac{2pq}{1-q^2}\tau_3 + \frac{p^2}{1-q^2}\tau_2$. Unfolding this equation an enough number of times we obtain

$$T_{stay}(1) = \frac{q^2}{1-q^2}\Delta + \Big(\frac{2pq}{1-q^2}\Big)^2\Big(\mu + \frac{q^2}{1-q^2}\Delta\Big)$$

The term $\frac{q^2}{1-q^2}\Delta$ is the expected time looping around states 3 and 4. The term $\Big(\frac{2pq}{1-q^2}\Big)^2\Big(\mu + \frac{q^2}{1-q^2}\Delta\Big)$ give us the probability to take transition $(3, 2)$ coming from 4 multiplied by the expected time going from 3 to 2 and looping in 2 before leaving. It can be rewritten as

$$\begin{aligned} T_{stay}(1) &= V_{3,3}Q_{3,3}N_3 + V_{3,2}Q_{3,2}N_3 + V_{2,2}Q_{2,2}N_2 \\ &= \Delta q^2\frac{2pq}{1-q^2}\frac{1}{1-q^2} + \mu 2pq\frac{2pq}{1-q^2}\frac{1}{1-q^2} + \\ &+ \Delta q^2\Big(\Big(\frac{2pq}{1-q^2}\Big)^2 + \frac{p^2}{1-q^2}\Big)\frac{1}{1-q^2} \end{aligned}$$

We have $N_3 = \frac{2pq}{1-q^2}\frac{1}{1-q^2}$ and $N_2 = \Big(\Big(\frac{2pq}{1-q^2}\Big)^2 + \frac{p^2}{1-q^2}\Big)\frac{1}{1-q^2}$. Following this analysis we find

$$\begin{aligned} T_{jump}(1) &= \frac{2pq}{1-q^2}\mu\Big(\Big(\frac{2pq}{1-q^2}\Big)^2 + \frac{p^2}{1-q^2}\Big) + \frac{p^2}{1-q^2}\delta\Big(\Big(\frac{2pq}{1-q^2}\Big)^2 + 1\Big) \\ T_{stay}(2) &= \Big(\Big(\frac{2pq}{1-q^2}\Big)^3 + 2\frac{pq}{1-q^2}\frac{p^2}{1-q^2}\Big)\Big(\delta + \frac{q}{1-q}\Delta\Big) \end{aligned}$$

7.4.5 Computation of the Average Number of State Entries

Recall that we denote with N_s the average number of entries in the state s (where s is the number of tasks that remain to be performed in that state). We want to estimate the value of N_s for each non absorbing state s. When s is different from the initial state, we have [87]:

$$N_s = \sum_{\{t|(t,s)\text{is an arc}\}} P_{t,s} N_t$$

where $P_{t,s}$ is the probability of taking the transition $t \rightarrow s$. When s is the initial state we have $N_s = 1 + \sum_{\{t|(t,s)\text{is an arc}\}} P_{t,s} N_t$.

Example Let us recompute N_k for $4 \leq k < 0$ for Figure 7.6. Using the equations

$$N_4 = 1 + q^2 N_4 \ , \ N_3 = q^2 N_3 + 2pq N_4 \ , \ N_2 = q^2 N_2 + 2pq N_3 + p^2 N_4$$

we re-obtain N_4,N_3 and N_2. Finally $N_1 = qN_1 + 2pqN_2 + p^2 N_3$ and

$$\begin{aligned} N_1 &= \frac{1}{1-q} 2pqN_2 + \frac{1}{1-q^2} p^2 N_3 \\ &= \frac{1}{1-q} \Big(\frac{2pq}{1-q^2} \Big)^3 + 2 \frac{1}{1-q} \frac{2pq}{1-q^2} \frac{p^2}{1-q^2} \end{aligned}$$

We can check the correctness of the whole approach recomputing

$$\begin{aligned} T_{jump}(1) &= \delta p^2 N_3 + \mu 2pq N_2 + \delta p^2 N_2 \\ T_{stay}(2) &= \Delta q N_1 + \delta p N_1 \end{aligned}$$

General Case for N Let us consider the general case. According to the Figure 7.5, $(0,0)$ corresponds to the initial state with n tasks to be executed. As in the example we identify $(0,0)$ and n. As $N_n = 1 + q^m N_n$ we obtain $N_n = \frac{1}{1-q^m}$. Consider the state $(1,1)$ with $n-1$ tasks has to be executed, as $N_{n-1} = mpq^{m-1} N_n + q^m N_{n-1}$,

$$N_{n-1} = m \frac{pq^{m-1}}{(1-q^m)^2}$$

Example Before writing the general case we consider the case $m = 5$ and $n = 15$. We want to compute the expected number of entries into state $s = 7$. Note that state 7 appears in the second level of the Markov chain. In Figure 7.7 we show the representation of the portion of graph needed to compute N_7. We sum up all the contributions to enter 7 and we get

$$N_7 = \frac{1}{1-p^5} \Big(5pq^4 N_{s+1} + 10p^2 q^3 N_{s+2} + 10p^3 q^2 N_{s+3} + 5p^4 q N_{s+4} + p^5 N_{s+5} \Big)$$

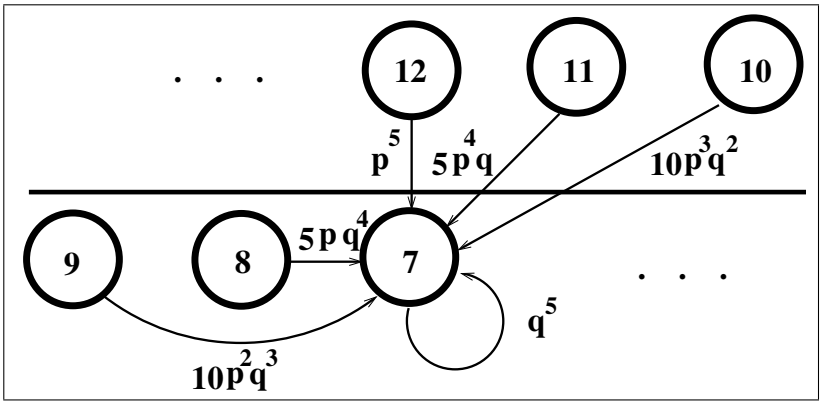


Figure 7.7: Snapshot of the graphical representation of Markov models in the case of $m = 5$ and $n = 15$, to compute the average number of entries into state 5.

General Case We can now easily generalize this formula to the case of m resources:

$$\begin{aligned} N_s &= p^m N_{s+m} + q^m N_s + \sum_{0<k<m} \binom{m}{k} p^k q^{m-k} N_{s+k} \\ &= \frac{1}{1-q^m}\Big(p^m N_{s+m} + \sum_{0<k<m} \binom{m}{k} p^k q^{m-k} N_{s+k}\Big) \end{aligned}$$

7.5 A Framework to Study the Completion Time

As we have seen the values of N_s are different for different nodes s. We study experimentally the possibility to replace all the different values by a unique average. We can apply the formula allowing to compute N_s to concrete examples, in order to get an idea of the behavior of this value. We have implemented a simple program that takes as input the quantities describe in Section 7.2, and produces the N value for all states of the Markov chains. The results are shown in Tables 7.1, 7.2, 7.3, and 7.4.

From these numerical results, it seems a reasonable hypothesis to take the same average value $\overline{N}$. Therefore we assume the following:

> Working Hypothesis. In the computation of T_{stay} and T_{jump} we replace all the values N_a by an average value $\overline{N}$.

First consider the average time passed in the level k:

$$\begin{aligned} T_{stay} &= \sum_{a,b\in k} V_{a,b} Q_{a,b} N_a \approx \overline{N} \cdot \sum_{a,b\in k} V_{a,b} Q_{a,b} \\ &= \overline{N} \cdot \left[m\Delta q^m + \sum_{0<k<m} (m-k)\mu \binom{m}{k} p^k q^{m-k} \right] \end{aligned}$$

n	m	δ	Δ	p	$\mathbf{N_{avg}}$	$\mathbf{N_{var}}$
500	5	10	5	0.9	0.223	0.0022
500	5	10	5	0.8	0.251	0.0015
500	5	10	5	0.5	0.401	0.001
1000	5	10	5	0.9	0.223	0.0011
1000	5	10	5	0.8	0.250	0.0007
1000	5	10	5	0.5	0.400	0.0004
2000	5	10	5	0.9	0.222	0.0006
2000	5	10	5	0.8	0.250	0.0004
2000	5	10	5	0.5	0.400	0.0002

Table 7.1: Evaluation of N for $n = 5$, and $n = 10$.

n	m	δ	Δ	p	$\mathbf{N_{avg}}$	$\mathbf{N_{var}}$
500	10	10	5	0.9	0.112	0.0026
500	10	10	5	0.8	0.126	0.0020
500	10	10	5	0.5	0.201	0.0014
1000	10	10	5	0.9	0.112	0.0013
1000	10	10	5	0.8	0.125	0.0010
1000	10	10	5	0.5	0.200	0.0007
2000	10	10	5	0.9	0.111	0.0007
2000	10	10	5	0.8	0.125	0.0005
2000	10	10	5	0.5	0.200	0.0004

Table 7.2: Evaluation of N for $n = 5$, and $n = 10$.

We can similarly consider all the transitions from any states at a level k to any states at level $k + 1$, and compute T_{jump} as:

$$\begin{aligned} T_{jump} &= \sum_{a\in k, b\in k+1} V_{a,b} Q_{a,b} N_a \approx \overline{N} \cdot \sum_{a\in k, b\in k+1} V_{a,b} Q_{a,b} = \\ &= \overline{N} \cdot \left[m\Delta p^m + \sum_{0<k<m} (m-k)\mu \binom{m}{k} p^{m-k} q^k \right] \end{aligned}$$

7.6 Experimental Results

In this section we show the experimental results that we performed to test the effectiveness of the models we introduced. We show results for the simple case of 2 tasks performed by 2 slaves, and n tasks performed by m slaves. The first set of tests is described to show the differences between the completion time evaluated with the direct solution to the Markov model, and actual values. The second set of

n	m	δ	Δ	p	$\mathbf{N_{avg}}$	$\mathbf{N_{var}}$
500	20	10	5	0.9	0.056	0.0030
500	20	10	5	0.8	0.063	0.0023
500	20	10	5	0.5	0.101	0.0018
1000	20	10	5	0.9	0.056	0.0015
1000	20	10	5	0.8	0.063	0.0011
1000	20	10	5	0.5	0.100	0.0009
2000	20	10	5	0.9	0.056	0.0007
2000	20	10	5	0.8	0.063	0.0006
2000	20	10	5	0.5	0.100	0.0004

Table 7.3: Evaluation of N for $n = 5$, and $n = 10$.

n	m	δ	Δ	p	$\mathbf{N_{avg}}$	$\mathbf{N_{var}}$
510	30	10	5	0.9	0.038	0.0030
510	30	10	5	0.8	0.043	0.0023
510	30	10	5	0.5	0.068	0.0019
1020	30	10	5	0.9	0.037	0.0015
1020	30	10	5	0.8	0.042	0.0012
1020	30	10	5	0.5	0.067	0.0009
2010	30	10	5	0.9	0.037	0.0008
2010	30	10	5	0.8	0.042	0.0006
2010	30	10	5	0.5	0.067	0.0005

Table 7.4: Evaluation of N for $n = 5$, and $n = 10$.

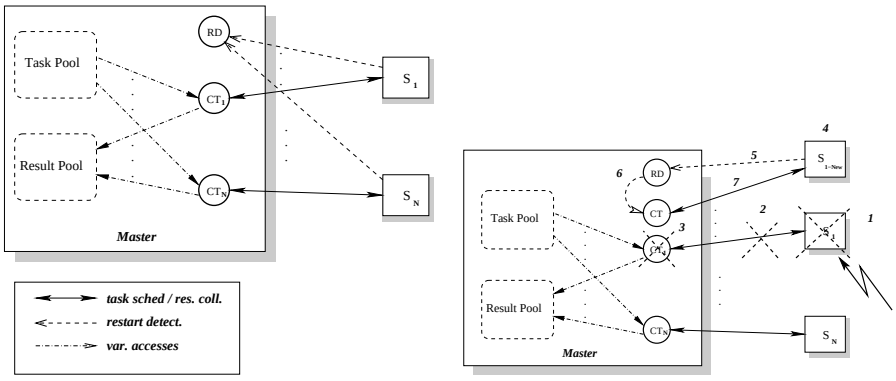
tests, for the general case, shows the differences between the approximated solution, derived from the framework of above, and actual values. The aim is to show that the experimental numbers are always lower than the ones derived evaluated with the model, i.e. the model gives effective upper bounds of the completion time. We also study the discrepancies between the two sets of values.

We describe the environment chosen to perform tests, and how the experimental configuration is set up to simulate failures.

7.6.1 Testing Environment

We exploit the `muskel` programming environment to test the effectiveness of the theoretical study. We shortly describe the `muskel` programming model, and its implementation to motivate its choice.

A `muskel` program is a macro data-flow graph composed of sequential and parallel modules, connected through data streams. Parallel modules are expressed



(a) Graphical representation of the implementation of `muskel` exploiting a master-slave strategy.

(b) Graphical representation of the master behavior in the case of slave failure and restart.

Figure 7.8: `muskel` implementation as a master-slave, and its FT strategy.

as skeletons (e.g. farm, pipeline), and can be nested in arbitrary hierarchies. In `muskel` , farm computations express a task parallel computation, where a set of *workers* performs the same program on different input data (the tasks) incoming from an input stream, and producing an output stream of results. In a pipeline computation, each input data received from an input stream is transformed from a sequence of nested functions (e.g. $out = F_1(F_2(\ldots F_n(in)\ldots))$). Each function F_i is performed in a different *stage*, and the evaluation of two functions on different input data (can) happen in parallel.

The implementation of `muskel` is based on a *master-slave* strategy:

- A whole data-flow graph is mapped in each slave, that represents the unit of parallelism. The data-flow graph can be modeled as a function $\mathbf{F}$, which application to an input data represents the *task* of the computation. A slave computation consists in iteratively: (1) receiving an input value from the master; (2) applying it as actual parameter of $\mathbf{F}$, and producing a result; (3) returning the result to the master. This scheme is applied to a stream of input values (produced by the master), and it produces an output stream of results (consumed by the master).

- The master owns a local queue of tasks (i.e. the input data), and is responsible of coordinating slaves: it schedules to workers the input data, and it receives results back from slaves.

This implementation reflects the implementation model of farm computations described in the previous sections. In this section, we conversely use the terms *slave* and *remote interpreter* mixing up the terminology of the theoretical study described in the previous sections, and the mapping in the `muskel` terminology. The fault tolerance strategy is implemented in the master process, which structure is represented

in Fig 7.8(a). We denote processes with squares, threads with circles, and state variables with rounded rectangles. The master includes two pools, one for tasks, and the other for their results. Each slave is managed by an independent thread (slave S_i is managed by CT_i, where CT stands for Control Thread). Control threads and slaves are linked with bold double-ended arrows, denoting the task scheduling and result collection operations. Control threads are also linked to tasks and results to obtain and collect them (arrows half dotted, half dashed). We also exploit a restart detection thread (denoted with RD), that is responsible of receiving restart messages from slaves (dotted arrows). The behavior in the case of slave failure is represented in Fig. 7.8(b): slave S_1 fails (1) and the corresponding bold arrow with its control thread is closed (2). The control thread detects the closing of the connection, and it terminates (3). In the implementation, a list of active slaves is also exploited. In the event of failure, the control thread removes the slave from the list of actives. The list is used to control the degree of parallelism of the computation. In the same figure we show the case of slave restart. Suppose slave S_1 is restarted by some external mechanism (4). It communicates the RD thread on the master that it is available to perform tasks (5). The RD thread spawns a new control thread for its management (6), that connects to the slave to assign it tasks (7).

The implementation of `muskel` and its fault tolerance strategy reflect the computation model and fault tolerance strategy we exploited to study farm computations. This motivates our choice of this programming environment. Also notice that `muskel` implements the behavior describe in Section 7.1, in which lost tasks are immediately re-scheduled to available slaves. This states the difference between `muskel` and the BSP-like model, that we use for the theoretical study, and will be reflected in the differences between the experimental results, and the theoretical values (see below).

We extended the `muskel` implementation to set up failure testing. We show the abstract failure models, and their implementation in `muskel` , the represent the mapping of the abstract assumption described in Section 7.2.

7.6.2 Mapping the Failure Model in `muskel`

We introduce two *failure models* reflecting in the `muskel` environment the abstract one defined in the theoretical study. From a general viewpoint, the first model is more similar to the theoretical model (see Section 7.2) than the second one, which better simulates the behavior of an actual execution environment subject to failures. We characterize failure models w.r.t. *qualitative* and *quantitative* aspects. Qualitative aspects describe the behavior of a resource when it fails, and the features of the failure detection sub-systems. Quantitative aspects describe the frequency of faults, their patterns, and duration. The instances of the abstract failure model are derived by instancing the quantitative aspects in different ways. The qualitative ones are the same both models. The main features of both models are:

- *Qualitative.* Computational resources fail by stopping their execution, according to the fail-stop model [76], i.e. after a failure a resource stops to perform its task. After a failure, a computational resource is eventually restarted. The detectability of failures and restarts are demanded to specific sub-system, which implementation is not described here.

- *Quantitative.* The failure of a resource is subject to probability q. Conversely, we denote with $p = 1 - q$ the probability of success of a resource. The failure is expressed in terms of slave invocations: every t invocations to a slave there is a q probability of failure. The time needed to detect a failure is represented by the variable T_F. After a failure, it takes a time T_R for a resource to be restarted. No constraints are given for the value that T_R can assume, the fault frequency and their patterns.

In both models we can set the value p. The way in which the failure and restart latencies are controlled characterize the models.

- *First Failure Model.* Failures can happen every time a slave is invoked, i.e. $t = 1$. The time needed to detect a failure can be upper bounded, but it cannot be directly controlled: the actual value can vary at each failure and, in general, it will depend on some lower virtualization level features. We denote with Δ_F the upper bound on failure detectability. The time needed to restart an interpreter is a random variable in the range $0 \leq T_R \leq \Delta_R$, and we can set the upper bound. Also, we can decide the probability distribution the restart time.

- *Second Failure Model.* In the second model, failures can happen every $t \geq 1$ time a slave is invoked. We can control both the time needed to detect a failure, and the time needed to restart it, by specifying their upper bounds and the failure distributions they follow. The constraints on the values of T_F and T_R are: $0 \leq T_F \leq \Delta_F$, and $0 \leq T_R \leq \Delta_R$. Probability distributions of T_F and T_R can be possibly different. In the experiments we can study different configurations of these parameters to address actual execution environments.

7.6.3 Implementation of Fault Injection

We extend the implementation of `muskel` to simulate failures and restarts of the processes. In the implementation, instead of halting and restarting machines, we kill and restart processes implementing `muskel` slaves to simulate failures. The simulation avoids to restart resources because:

- The fail-stop model of the theoretical study does not strictly requires computational resources to fail, but it just states that the computation module (the process, in our case) fails by stopping to execute.

- From a empirical viewpoint, managing failures at software-level is simpler than dealing with the stop and restart of machines.

We describe the implementation of the two abstract failure models.

Implementation of the First Failure Model. Failures of calls to a slave are implemented as part of their behavior:

```
RemoteInterpreter::
...
compute(Task t) {
   //compute for random time...
   //...
   double coin = rand();
   if(coin <= q) exit(FAILURE);
   //compute the remaining of the task
   Result r = t.execute();
   return r;
}
...
```

When a request for task execution is received, the slave performs a part of the task for a random time. Next, a random number is generated and, depending if it is lower than a fixed value q, the slave either executes the tasks or terminates with failure. The q value can be configured on each slave as an initialization parameter. Eventually, we will allow modification of this value at run-time to simulate environments in which the failure probability of resources can change during the computation. The feature $t = 1$ is obtained by tossing a coin at each invocation, in the remote interpreter code.

The restart of a slave is performed by an external process that monitors for slave failures and, when it is the case, it restarts them. We will call this module the *Restarter*. A parameter for the monitoring is the time between two successive fault detections (this value should be chosen to target the trade-off between the overhead it causes to the computation, and the restart latency). This value represents the upper bound over the fault detection latency, i.e. Δ_F.

In Fig. 7.9 we show the implementation of the first instance of the abstract failure model. Each Interpreter (denoted with I_i) is augmented with a fault injector (denoted with *finj*). The master process, running on a robust node, is responsible of scheduling tasks to interpreters and, in the case of failure, it re-contacts the new restarted interpreter (bold arrow). The restart subsystem is implemented as a single process and is executed on a robust node. Whenever a slave fails (1) it detects the failure in T_F time (2), and it re-spawns it (3) in T_R time (dotted arrows). In the figure, we show the failure of I_n, and the re-spawning of a new instance of it from the restarter. After the restart, the manager re-connects to the new instance of I_n.

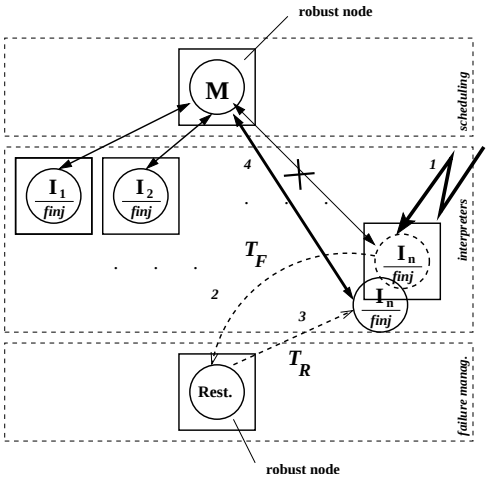


Figure 7.9: Representation of the first instances of the fault injection technique. Whenever a slave fails (1) it detects the failure in T_F time (2), and it re-spawns it (3) in T_R time (dotted arrows).

Implementation of the Second Failure Model. We exploit a single process to terminate slaves (i.e. inject faults), and to restart them. Every T_C seconds the program chooses if it has to terminate a slave (with probability q). If it is the case, it: (1) waits for T_F seconds to simulate the failure detection; (2) it terminates the slave; (3) it waits T_R seconds before restarting the slave.

T_C represents the grain at which the failure can happen, and is a random variable, according to some probability distribution, upper bounded by the value Δ_C that is specified for each machine. We claim that this implementation is correct w.r.t. its specification, as, given that the average completion time of each task is T_T, the t value can be obtained as: $t = \lceil \frac{T_C}{T_T} \rceil$. It is important to notice that, differently from the previous model, the decision to fail is taken independently of the frequency at which an interpreter is called, i.e. it is application independent.

As in the model, T_F and T_R can follow specific probability distributions. Differently from the previous model T_F is not upper bounded by the monitoring frequency, but it is directly simulated and can be controlled to study different situations.

7.7 Numerical Results

In this section we show the numerical results we obtained, and the comparison with corresponding theoretical ones.

p	δ	Δ	$T_{exp}(avg.)$	T_{theo}	Discr.
0.9	10	5	11.37679	12	*0.62321*
0.8	10	5	12.83438	13.95833	*1.12395*
0.5	10	5	20.14205	21.66667	*1.52465*
Av. Disc. for $delta > \Delta$					**1.091**
0.9	10	15	13.07644	13.18182	*0.10538*
0.8	10	15	15.21683	16.875	*1.65817*
0.5	10	15	26.57598	35	*9.42402*
Av. Disc. for $\delta < \Delta$					**3.729**
Total Av. Disc.					**2.429**

Table 7.5: Results of the experiments for 2 tasks performed by 2 slaves. We show the discrepancies between monitored and theoretical completion times. The first set of lines is related to $delta > \Delta$, while the second to its inverse. For each set, we show the average discrepancy (Av. Disc.). At the bottom we show the total average discrepancy for all values (Total Av. Disc.).

7.7.1 Tests for A Simple Case

We have tested the case of 2 tasks performed by 2 interpreter exploiting the environment shown above, with the first failure model. The results are characterized w.r.t. the value assume by μ, δ or Δ.

For each configuration, we compute the average value, and the variance, on the 100 runs we experimented. The average and variance values are computed as following:

$$E = \frac{1}{n}\sum_{x=1}^{n} T^x_{comp}$$

$$Var = \frac{1}{n}\sum_{x=1}^{n}(T^x_{comp} - E)^2$$

Where $n = 100$, T_{comp} is the set of the obtained completion times, and T^i_{comp} is the i-th result.

Table 7.5 subsumes the results we obtained, and their comparison with the theoretical values. The results show that the differences between the theoretical results and the experimental ones depend on the failure rate. This reflects the intuitive difference between the BSP-like model, and the implementation one: in the case of failure, the BSP-like model does not immediately re-schedule the lost task, as the implementation model does. Higher failure rates induce higher probability that this behavior is repeated, giving higher discrepancies with the implementation model.

n	p	T_c exp.	T_c th.	dev. (%)
500	0.9	1014.695	1115.106	9%
500	0.8	1025.893	1253.799	18%
500	0.5	1648.587	1972.688	16.43%
1000	0.9	2037.750	2226.211	8.47%
1000	0.8	2057.511	2503.599	17.82%
1000	0.5	2763.726	3941.438	29.88%
2000	0.9	4047.495	4448.422	9.01%
2000	0.8	4392.028	5003.199	12.22%
2000	0.5	6559.243	7878.937	16.75%

Table 7.6: Experimental results for $m = 5$.

n	p	T_c exper.	T_c theo.	dev. (%)
500	0.9	513.856	560.056	8.25%
500	0.8	595.157	629.500	5.46%
500	0.5	851.208	1004.010	15.22%
1000	0.9	1028.474	1115.611	7.81%
1000	0.8	1036.899	1254.500	17.35%
1000	0.5	1706.031	2003.521	14.85.77%
2000	0.9	2062.921	2226.722	9.01%
2000	0.8	2096.943	2504.500	12.22%
2000	0.5	2165.239	4002.545	16.43%

Table 7.7: Experimental results for $m = 10$.

7.7.2 Tests for the General Model

We have performed experiments exploiting the `muskel` support, as extension of the ones performed for simple cases described above. We have chosen to test the cases for $m = 5$, and $m = 10$, with numbers of tasks equal to 500, 1000, and 2000, with $\delta = 10$, $\Delta = 5$, and $\mu = 10$. Tables 7.6 and 7.7 show the completion times we obtained from the experiments, the corresponding one computed according to the T_{stay} and T_{jump} quantities, and their deviation w.r.t. the theoretical value. Also in this case the deviation seems to increase with the probability of failure, that actually gives more uncertainty to the actual result.

Chapter 8

Conclusions

In this thesis we have studied the issue of failures in the context of High-Performance computing. In particular we highlighted which are the relationships between the parallel programming models and the mechanisms and techniques to support fault tolerance. We have presented a review of the works presented in the literature about parallel programming and fault tolerance computing which characterizes these relationships. From this study we conclude that there is a a need of the introduction of a methodology which allows application programmers to select the best-suited fault tolerance technique and to configure it. We made a step towards this methodology by starting from structured parallel programming models and by deriving cost models to guide the selection and configuration of the supporting techniques.

In particular we have defined a programming model for parallel computations featuring generality from the expressivity viewpoint but also with strong structural properties. The structural properties of the parallel constructs of the model characterize the interactions between the parallel activities which compose the computation. In this way the constructs can be described from the performance viewpoint by cost models which can be instantiated to the actual features of execution platforms and applications. We have introduced a formal tool to study the relationships between the constructs of this programming model (more precisely, its structural properties) and the mechanisms/techniques to support fault tolerance. This tool allows us to describe parallel computations by highlighting the relation between the control- and data-flow. By exploiting this relation we can describe the sequence of events which characterize a parallel computation. We have applied the formal model to two main constructs of our parallelism model and we have derived the relationships between the structural properties of the constructs and the fault tolerance mechanisms. To completely define our research methodology we have shown the implementation of fault tolerance support for the two constructs based on checkpointing and rollback recovery techniques. In the implementation we have exploited the high-level structural properties, characterized by the formal tool, to introduce the possibility of configuring the support and, thus, derive optimizations. In this way we can target the high heterogeneity and dynamicity of parallel computing platforms

and of parallel applications (e.g. in the case they solve irregular problems).

We have also shown how the performance models of structured parallel computations can be extended to include a description of the overhead introduced by the fault tolerance support. The models describe the performance behavior of parallel computations in absence of failures. They can be used to statically target the tradeoff between the performance of computations and their resiliency to failures. We have also introduced a formal model, based on Markov chains, to study the impact on the performance of task parallel computations in presence of failures. The study of the model results in a framework that approximates the actual computation time by providing an upper bound. As a result of these studies we have completely characterized the performance behavior of task parallel computations both in the case of failures and in the absence of them.

The work presented in this thesis leaves open several problems for future research directions. In the short term we have to study the relationships between the structural properties of the parallel computations and the possibility of introducing optimized garbage collection strategies for checkpoints. It is also interesting to show how to derive similar relationships to provide an implementation of parts of the support that we only suggest in our study (e.g. the atomicity properties of communications).

The techniques introduced in this thesis have been studied and compared to existing solutions only from an analytical viewpoint, according to the issues described at the end of Chapter 1. The future work must include experimental assessment of the analytical results. In both Chapters 4 and 5 we discuss when this experimentation needs is more necessary and how we should configure them to meet the desired comparison. The results should be compared with the expected ones, derived from analytical modeling.

In this thesis we have focused on the models of structured parallel programming. The fault tolerance computing research field features similar models (e.g. atomic actions and replication schemes) that we exploit to support parallel constructs. In the long term we can envision to study the possibility of moving up the models of fault tolerance computing at the level of parallel programming. As a result we should obtain more general programming constructs with which the programmers can express parallelism and fault tolerance features at the same level of abstraction. For instance, this research scheme has been partially followed in [60, 14]. We also argue that the results of short and long term future works could help in deriving an extension of the I-Structure formal tool to meet higher expressivity of fault tolerance and concurrency aspects.

Bibliography

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 31, Washington, DC, USA, 1999. IEEE Computer Society.

[2] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7-8):449–462, 2006.

[3] L. Alvisi. *Understanding the message logging paradigm for masking process crashes*. PhD thesis, Cornell University, Department of Computer Science, Ithaca, NY, USA, 1996.

[4] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. In *IEEE Transactions on Software Engineering*, volume 24, pages 149–159. IEEE Computer Society, February 1998.

[5] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for now (networks of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[6] Arvind, P. S. Barth, and R. S. Nikhil. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568, London, UK, 1991. Springer-Verlag.

[7] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, 1989.

[8] R. Badrinath, C. Morin, and G. Vallee. Checkpointing an recovery of share memory parallel applications in a cluster. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 471, Washington, DC, USA, 2003. IEEE Computer Society.

[9] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wijckoff. Charlotte: metacomputing on the web. *Future Generation Computer Systems*, 15(5-6):559–570, 1999.

[10] R. Batchu, Y. Dandass, A. Skjellum, and M. Beddhu. Mpi/ft: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, 2004.

[11] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, and M. Apte. Mpi/fttm: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 26, Washington, DC, USA, 2001. IEEE Computer Society.

[12] F. Baude, D. Caromel, C. Delbe, and L. Henrio. A hybrid message logging-cic protocol for constrained checkpointability. In *Proceedings of EuroPar2005*, LNCS, pages 644–653, Lisbon, Portugal, August-September 2005. Springer.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[14] C. Bertolli, M. Coppola, and C. Zoccolo. The co-replication methodology and its application to structured parallel programs. In *CompFrame '07: Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing*, pages 39–48, New York, NY, USA, 2007. ACM.

[15] C. Bertolli and J. Gabarro. On the cost of task re-scheduling in fault-tolerant task parallel computations. *Poster Session CoreGRID Integration Workshop*, 11(4), 2008.

[16] C. Bertolli, M. Meneghin, and J. Gabarro. A markov model for fault-tolerant task parallel computations. In T. Priol and M. Vanneschi, editors, *From Grids to Service and Pervasive Computing*, pages 123–136. Springer US, 2008.

[17] B. Bhargava and S.-R. Lian. Independent checkpointing and concurrent rollback for recovery–an optimistic approach. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 3–12. IEEE Computer Society, October 1988.

[18] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Symp.*, pages 133–147, 1997.

[19] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[20] A. Bouteiller, F. Cappello, T. Herault, K. Krawezik, P. Lemarinier, and F. Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[21] A. Bouteiller, F. Cappello, T. Herault, K. Krawezik, P. Lemarinier, and F. Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[22] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello. Impact of event logger on causal message logging protocols for fault tolerant mpi. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 97, Washington, DC, USA, 2005. IEEE Computer Society.

[23] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello. Impact of event logger on causal message logging protocols for fault tolerant mpi. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 97, Washington, DC, USA, 2005. IEEE Computer Society.

[24] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. Mpich-v: a multiprotocol fault tolerant mpi. *International Journal of High Performance Computing and Applications*, 20(3):319–333, 2006.

[25] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *The primary-backup approach*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[26] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.

[27] K. Mani Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[28] Z. Chen, G. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[29] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[30] M. Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, MA, USA, 1991.

[31] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[32] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 127, New York, NY, USA, 2006. ACM.

[33] D. E. Culler, A. Gupta, and J. Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[34] K. Davis, A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, S. Pakin, and F. Petrini. A performance and scalability analysis of the bluegene/l architecture. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 41, Washington, DC, USA, 2004. IEEE Computer Society.

[35] X. Defago, A. Schiper, and N. Sergent. Semi-passive replication. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[36] X. Defago, A. Schiper, and P Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[37] P. J. Denning. Fault tolerant operating systems. *ACM Comput. Surv.*, 8(4):359–389, 1976.

[38] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[39] E. N. (Mootaz) Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[40] C. Engelmann and G. A. Geist. Super-scalable algorithms for computing on 100,000 processors. In *Lecture Notes in Computer Science: Proceedings of the 5^{th} International Conference on Computational Science (ICCS) 2005, Part I*, volume 3514, pages 313–320, Atlanta, GA, USA, May 2005. Springer.

[41] G. Fagg, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. Dongarra. Scalable fault tolerant mpi: Extending the recovery algorithm. In *Proceedings of the 12th European Parallel Virtual Machine and Message Passing Interface Conference - Euro PVM/MPI*, pages 76–83. Springer, September 2005.

[42] G. Fagg, A. Bukovsky, and J. Dongarra. Harness and fault tolerant mpi. *Parallel Comput.*, 27(11):1479–1495, 2001.

[43] M J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[44] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*. http://www.mpi-forum.org, June 1995.

[45] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[46] E. Frachtenberg, K. Davis, F. Petrini, J. Fernandez, and J. C. Sancho. Designing Parallel Operating Systems via Parallel Programming. In *Euro-Par 2004*, Pisa, Italy, August 2004.

[47] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *HPDC*, pages 43–50, 2000.

[48] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. Dean Risinger, and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. *Int. J. Parallel Program.*, 31(4):285–303, 2003.

[49] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[50] Grid 5000 homepage, https://www.grid5000.fr/.

[51] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[52] J.-M. Helary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal. Communication-based prevention of useless checkpoints in distributed computations. *Distrib. Comput.*, 13(1):29–43, 2000.

[53] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[54] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer Verlag, 1976.

[55] S. Krishnamurthy, W.H. Sanders, and M. Cukier. An adaptive quality of service aware middleware for replicated services. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1112–1125, November 2003.

[56] H. Kuchen. A skeleton library. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.

[57] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[58] B.-H. Lim and A. Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Trans. Comput. Syst.*, 11(3):253–294, 1993.

[59] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

[60] J. Maier. Fault-tolerant parallel programming with atomic actions. In *Proc. of the 1995 Fault-Tolerant Parallel and Distributed Systems*, pages 210–219, 1995.

[61] P. Sarathi Mandal and K. Mukhopadhyaya. Performance analysis of different checkpointing and recovery schemes using stochastic model. *J. Parallel Distrib. Comput.*, 66(1):99–107, 2006.

[62] O. Marin, M. Bertier, and P. Sens. Darx—a framework for the fault-tolerant support of agent software. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 406, Washington, DC, USA, 2003. IEEE Computer Society.

[63] T. G. Mattson, D. Scott, and S. R. Wheat. A teraflop supercomputer in 1996: The asci tflop system. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 84–93, Washington, DC, USA, 1996. IEEE Computer Society.

[64] S. Monnet, C. Morin, and R. Badrinath. Hybrid checkpointing for parallel applications in cluster federations. In *CCGRID '04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 773–782, Washington, DC, USA, 2004. IEEE Computer Society.

[65] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Commun. ACM*, 39(4):54–63, 1996.

[66] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Eternal: a component-based framework for transparent fault-tolerant corba. *Softw. Pract. Exper.*, 32(8):771–788, 2002.

[67] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. Doors: Towards high-performance fault tolerant corba. In *DOA '00: Proceedings of the International*

Symposium on Distributed Objects and Applications, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[68] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):165–169, 1995.

[69] Object Management Group. *Chapter 23 of CORBA/IIOP 3.0.3: Fault-Tolerant CORBA*, January 2008.

[70] F. Petrini, K. Davis, and J. C. Sancho. System-level fault-tolerance in large-scale parallel machines with buffered coscheduling. In *In 9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems (FTPDS04)*, Santa Fe, NM, April 2004.

[71] D. Powell, I. Bey, and J. Leuridan, editors. *Delta Four: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991.

[72] D. A. Reed, C. Lu, and C. L. Mendes. Reliability challenges in large systems. *Future Gener. Comput. Syst.*, 22(3):293–302, 2006.

[73] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local area communication with fast sockets. In *ATEC'97: Proceedings of the Annual Technical Conference on Proceedings of the USENIX 1997 Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 1997. USENIX Association.

[74] E. Roman. A survey of checkpoint/restart implementations. Technical report, Berkeley Lab Technical Report, July 2002.

[75] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 772, Washington, DC, USA, 2004. IEEE Computer Society.

[76] F. B. Schneider. The fail-stop processor approach. In *Concurrency control and reliability in distributed systems*. Van Nostrand Reinhold Co., New York, NY, USA, 1987.

[77] F. B. Schneider. *Replication management using the state-machine approach*, pages 169–197. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[78] R. Sirvent, J. M. Perez, R. M. Badia, and J. Labarta. Automatic grid workflow based on imperative programming languages: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1169–1186, 2006.

[79] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 223–238, New York, NY, USA, 1989. ACM.

[80] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.

[81] J. A. Smith and S. K. Shrivastava. Fault-tolerant parallel applications using queues and actions. In *ICPP '97: Proceedings of the international Conference on Parallel Processing*, pages 145–149, Washington, DC, USA, 1997. IEEE Computer Society.

[82] Y. Tamir and C. H. Sequin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the 13-th International Conference on Parallel Processing*, pages 32–41, 1984.

[83] K. Tani. Earth simulation project in japan - seeking a guide line for the symbiosis between the earth and human beings - visualizing an aspect of the future of the earth by a supercomputer. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 33–42, London, UK, 2000. Springer-Verlag.

[84] T. Tantikul and D. Manivannan. A communication-induced checkpointing and asynchronous recovery protocol for mobile computing systems. In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 70–74. IEEE, 2005.

[85] K. S. Trivedi. *Probability and statistics with reliability, queuing and computer science applications*. John Wiley and Sons Ltd., Chichester, UK, UK, 2002.

[86] P. Urban, N. Hayashibara, A. Schiper, and T. Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 4–17, Washington, DC, USA, 2004. IEEE Computer Society.

[87] N. H. Vaidya. A case for two-level recovery schemes. *IEEE Trans. Comput.*, 47(6):656–666, 1998.

[88] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Satin: Simple and efficient Java-based grid programming. In *AGridM Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, USA, September 2003.

[89] M. Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, 2002.

[90] M. Vanneschi and L. Veraldi. Dynamicity in distributed applications: issues, problems and the assist approach. *Parallel Comput.*, 33(12):822–845, 2007.

[91] G. Wrzesinska, R. van Nieuwpoort, J. Maassen, and H. E. Bal. Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In *Proc. of 19th International Parallel and Distributed Processing Symposium*, Denver, CO, USA, April 2005.

[92] Y. Zhang, D. Wong, and W. Zheng. User-level checkpoint and recovery for lam/mpi. *SIGOPS Oper. Syst. Rev.*, 39(3):72–81, 2005.

[93] G. Zheng, L. Shi, and L. V. Kale. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 93–103, Washington, DC, USA, 2004. IEEE Computer Society.

[94] A. Ziv and J. Bruck. Analysis of checkpointing schemes for multiprocessor systems. In *Symposium on Reliable Distributed Systems*, pages 52–61. IEEE Computer Society, October 1994.