Tartu University
Faculty of Mathematics and Computer Science
Institute of Computer Science

Fault tolerance method for MPI FORTRAN programs

Master's Thesis

Oleg Batrashev

Supervisor: Meelis Roos

Tartu 2007

Contents

Introduction

Fault tolerance is a feature of a software or hardware system that allows it to avoid failures or restore system state after an error has occurred. There are many different kinds of systems that require such a feature and all require different approaches to the problem. One distinct area that is constantly considered for the fault tolerance is distributed scientific computing.

In scientific computing applications may run for many hours using computational power of thousands of processors. The probability of some node failing is higher than usual and the cost of restarting application from the start in case of a single failure is unacceptable.

Over the past fifty years there has been a constant evolution of hardware and software systems. Over last two decades there has been explosion in the number of systems available on the market with a clear shift toward distributed systems. There is a stack of different layers in the system: network, operating system, system libraries, application, all of which are susceptible to a failure. Every new implementation of a software system requires fault tolerance features to be redesigned and reimplemented.

For a long time scientific computing has been done on supercomputers or private clusters which had specific fault tolerance mechanisms and local programmers were aware of those. Nowadays, GRID computing is spreading, offering everyone the computational power provided by large number of institutions. The multiplicity of the available systems is overwhelming, and each of them is providing different mechanism to fault tolerance.

The main goal of this work was to add fault tolerance feature to DOUG package[1] — parallel iterative linear solver with domain decomposition. DOUG is written in FORTRAN 90 and use MPI (Message Passing Interface) for communication. It is portable and may be run on many provided clusters and it was highly desired to keep that portability after adding fault tolerance.

At the beginning of the search for an appropriate solution there was no expertise available locally to narrow the set of possible solutions. During the search many available publications were studied and the whole area of fault tolerance techniques was explored. Finally, appropriate solution was found and an attempt was made to implement it. Although no real results for DOUG has been achieved the solution showed to be appropriate for the task.

First three chapters of this paper shortly summarize general classification of fault tolerance systems provided by other articles and surveys. Then the selected solution written in FORTRAN 90 is presented using many concepts from the earlier chapters.

The structure of this paper is as follows. The first chapter describes the high-level classification and general terms. The second and third chapters further explore classification specifically focusing attention on the basic concepts of such systems. The fourth chapter presents an application-level checkpointing approach and the last, fifth chapter discusses gained experience and general ideas about checkpointing in distributed computing.

Chapter 1

Overview of fault tolerant systems

This chapter introduces high-level classification of fault tolerant systems. The first section presents two strategies for fault tolerance systems, namely forward and backward fault tolerance. The second section gives brief overview of classification from other perspectives.

1.1 High-level classification

From a high perspective there exist two different approaches to fault tolerance: forward and backward. The former anticipates errors and tries to tolerate them without any time-consuming actions while the latter invokes a recovery mechanism after a fault has occurred.

1.1.1 Forward fault tolerance

Forward fault tolerance uses additional resources to tolerate a failure in advance and does not allow it to defect execution. This kind of solution is needed in critical (nuclear plant, airplane software, Mars rover) or real-time (important public servers) systems.

This is mainly achieved by duplicating hardware and software components. It is assumed that in case of a one component instance failure its clones are still functional and respond correctly to requests. For example, some electronic part of a Mars rover may have three identical chips and an integration circuit which gives answer provided by at least two components so failure of a one of three chips does not affect the system. There exist software systems which run multiple instances of the same application or databases where writes are performed simultaneously to all of them and reads are compared for identity.

This approach requires several times more resources than needed by a single instance of the application and is considered too expensive for scientific applications.

1.1.2 Backward fault tolerance

Backward fault tolerance, in contrast, does not use duplication but invokes some recovery mechanism to tolerate a failure when it occurs. This may take substantial time. Therefore, it is only acceptable for systems that are not time or life critical.

Systems using this approach store some additional information during execution of the process. When an error occurs, they rollback the process to some earlier saved state and sometimes recover it further to the state just before the error. This approach is called *rollback-recovery* and the saved snapshot of a process state is referred to as *checkpoint*.

This is the typical approach to fault tolerance in scientific applications because it just slightly affects performance of the computation.

1.2 Other classifications

The *checkpointing mechanism* defines how a *local* snapshot of the application is taken. It can be accomplished on many levels: by specific hardware, operating system, linked library or by application itself. The functionality available and presentation of an application data that goes into snapshot vary for different levels.

The *checkpointing protocol* defines how to make global snapshot of a distributed application. It solves the problem of combining checkpoints of separate processes and communication channels into one *meaningful* global checkpoint. The solution for this problem is not trivial because there is no shared clock to take snapshots simultaneously.

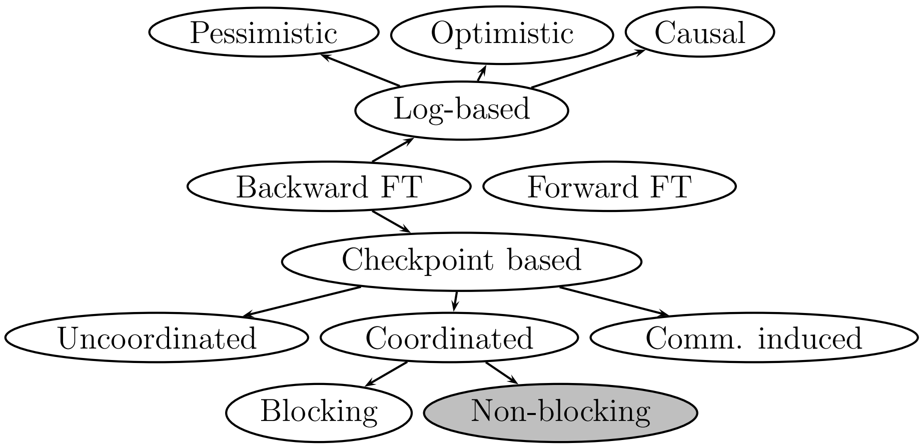

Figure 1.1: Classification of fault tolerance techniques

Diagram 1.1 shows basic classifications of fault tolerance techniques with filled oval of technique used by the algorithm which is presented in chapter 4.

Chapter 2

Classification by checkpointing mechanism

This section discusses approaches to saving the state of a single process. It is not obvious which data should be considered as process state and it is usually dictated by the requirements to the system under consideration. For example, should the state of the libraries be considered as part of the state of the process or whether the ID number of the process must be preserved. If we proceed to a distributed environment then opened sockets and the number of sent and received messages can also be viewed as a part of the process state.

The most common data sections that are considered for checkpointing are process stack, global variables and process heap. Even with these basic sections there is no standard way to inspect and handle them.

Some libraries allow a user to create objects inside the library and an application to hold only object handles that are used to manipulate the objects. The application does not have access to the internals of the library or at least does not have enough knowledge to decide how its state should be saved. It is the library that should be aware of checkpointing functionality and provide means to accomplish that.

Two main approaches exist to above problems: system- and application-level checkpointing. System-level checkpointing (SLC) saves the states of the whole sections of application without knowledge of internal data structures or algorithms. Application-level checkpointing (ALC), on the contrary, can use information about the algorithm and can usually change the code of application to add checkpointing functionality.

2.1 System-level checkpointing

System-level checkpointing operates on the level of system library or operating system itself. It uses OS specific mechanisms and knowledge to take a snapshot of the application state. These approaches can be divided into two groups: *user mode* and *kernel mode* mechanisms. The first one operates on the level of all other libraries and applications and usually does not need any changes in the operating system, whereas the second one lives in operating system kernel and operates on the level of hardware and software drivers.

There is a survey [2] on this topic which discusses requirements and existing SLC implementations. Most of implementations use user mode because of the complexity of a conventional operating system code and design, but they satisfy less requirements because they are unable to properly handle OS specific data like process credentials, signal handlers and sockets.

Many of the developed solutions were aimed for process migration in the first place which is similar to checkpointing. Snapshots of registers, stack and heap are taken, copied to another node and process state is reconstructed there. As pointed out in [2] there exists a large difference between checkpointing and process migration — the latter may assume that there are no failures during the run of an application. This assumption may result in much simpler algorithms. For example, there is no need to remember writes to the file or files themselves because there will be no need to rollback them, although this feature is considered one of the toughest to implement.

2.1.1 Issues with SLC

Here we describe most common problems to the checkpointing mechanism and their solutions in user and kernel mode.

The most obvious requirement is to save the program state including the current location and the program data. Kernel mode solutions use operating system internal structures like *task_struct* in Linux to save program registers and *vm_area_struct* to find and save the process address space. User mode solutions have to use special system library functions (*setjmp/longjmp*) to save registers and must have platform specific knowledge about process stack location. In Linux solutions the special *proc* file system is often used to locate process memory regions.

Some implementations, those running in kernel mode, save process credentials (PID, GID), especially those which are considered for process migration. The *BProc* system[3] uses *process ID masquerading* when copying a process to a slave node — its PID is mapped to the PID on the slave node and the old PID is still returned to the migrated application. The process group identifier (PGID) is also considered for storing, because scripts and pipes are common in Unix environment. Pending, running and blocked signals are handled by most kernel mode implementations. There are lots of other process information that should be stored and recovered: file descriptors, resource usage limits, process priority.

Storing and recovering files is almost not covered topic in existing implementations. The reason for this is, probably, that file data checkpointing or file operation logging is considered too expensive and superfluous. Systems aimed for process migration usually use remote file IO calls or any kind of distributed file system. They also limit checkpoint information with file open attributes and seek pointer value which works only with fail-free execution. Even the latest implementation of BLCR [4] does not support this feature and it is not planned.

System-level kernel mode checkpointing does not have any knowledge about application specific behavior and has to save all data in the application. Despite this, there exist a number of optimizations that prove to be very useful. Not all data need to be saved when second checkpoint is taken, only data changed between two sequential checkpoints needs to be identified and stored. Platform specific tricks (e.g. dirty pages) can be used to identify changed sections of the data to be checkpointed. Another optimization also uses platform specific feature to avoid delay that arises from the need to stop application execution for the time the checkpoint is taken. The need to stop execution is necessary, otherwise the snapshot will not be consistent. Many systems use *fork* and *copy-on-write* mechanisms to allow the application to continue its execution while the checkpoint is taken.

2.1.2 Kernel mode solutions

CRAK [5] and VMADump (part of BProc [3]) are extensions to the Linux kernel via kernel module mechanism to support process migration in a cluster environment. Both seem to be abandoned or at least not active for the last several years.

BLCR [4] is based VMADump and is the most promising project which is still active. A new version 0.5.2 was released recently. BLCR is a Linux kernel module and supports x86 and x86_64 architectures for the old 2.4 and most the recent 2.6 kernel versions. BLCR is aimed more toward High Performance Computing (HPC) than other solutions. Even socket migration is not planned because this problem is assumed to be solved by higher levels like MPI.

Another attempt is an extension [6] to the Mach operating system. This solution is interesting because Mach has micro-kernel architecture and shows whether adding checkpointing functionality to this kind of system is simpler than to monolithic kernels. It was necessary to add some extensions to several Mach interfaces and in general extending modularized kernel is not simpler. This is because checkpointing functionality is orthogonal to the whole system, i.e. it requires small changes to many modules.

2.1.3 User mode solutions

One of the first user mode solutions to the checkpointing was libckpt [7] which used *setjmp/longjmp*, *fork*, *copy-on-write* and other common techniques. It is referenced in many later articles, but the implementation itself was not developed further. It is an overall tendency that system-level checkpointers running in user mode are not in research in recent years. The problem is that they are still as highly platform dependent as kernel mode solutions, but much more limited in functionality.

Another solution which gets attention is libckp[8] because it adds file content checkpoint functionality. Unfortunately, no source code of this implementation was found and further research of file data checkpointing is not very active.

Condor[9] is a load balancing system for the high-throughput computing. It supports checkpointing and process migration [10] by using very the same techniques as other user mode checkpointers. Comparing to other attempts this project is a fully functioning system with recent new stable and development releases.

Table 2.1 gives shortened and fixed summary acquired from [2]. None of the programs supports the full feature list, but Condor and BLCR are two correspondingly user and kernel mode implementations that are still evolving and extending their functionality.

Name	**Mode**	**File Data**	**Creden-tials**	**Signals**	**File Descrip-tors**	**Address Space**	**Registers**
libckp	user	○	-	-	○	○	●
libckpt	user	-	-	-	○	○	●
Condor	user	-	-	●	○	○	●
CRAK	kern	-	○	●	○	○	●
BPRoc	kern	-	△	●	-	○	●
BLCR	kern	-	●	●	○	○	●

- — none, △ — weak, ○ — good, ● — complete

Table 2.1: Summary of SLC implementations

2.2 Application-level checkpointing

Application-level checkpointing does not use operating system mechanisms to save process state. Special code within the application itself is responsible for this functionality. The code is written by a programmer, generated by some tool or inserted by a compiler. Considering the last variants, we distinguish three methods of ALC: *manual*, *code preprocessing* and *compiler-assisted*.

ALC has more control over application state, but does not have access to the internals of operating system or used libraries and is thus very limited. For example, it cannot manage sockets in the MPI subsystem, so there must exist some other way to deal with it.

2.2.1 Issues with ALC

Saving application state with ALC is not simple, although it may seem that having more control over application code must lead to the opposite. Most programming languages do not give control over global variables and getting a snapshot of the stack is something not present in any widely used language. The reason for such limitations is that conventional algorithms do not require mentioned features and programming languages, following common patterns, provide means to separate and hide state of different components within application. To clear this point — there is no need to get access to the whole stack state until you start implementing checkpointing mechanism.

There are two known techniques to save the stack by programming language means. First, as shown in [11], the application saves pointers and lengths of local variables every time a function is entered and forgets them upon exiting the function. If the program wants to make a checkpoint it iterates over all saved pointers and saves corresponding memory areas. This approach must ensure that virtual address of the stack is the same upon restart of application. Alternative approach is shown in [12] where the application starts the checkpointing procedure by sequentially returning from function calls and saving values of local variables until it reaches the main function. Then it proceeds in opposite direction by restoring stack state until it reaches start location. During an usual run of the application there are much less steps that must be done and therefore this way is expected to be more efficient.

Catching the state of heap variables is no more easier than the ones on the stack. Pointers in the C language do not contain information about the type or size of the memory area it points to. Also, there is no heap interface that provides any knowledge or means by which allocated memory areas can be located and extracted. One possible solution is to redefine memory allocation and deallocation routines and to track this information using additional data structures.

Arguably the most difficult problem is to save the states of libraries and other used subsystems like MPI. Here there are also two common approaches, either the subsystem provides an interface to save its state or the programmer may build a layer on top of the subsystem and save/replay corresponding steps whenever needed.

2.2.2 Manual ALC

This method of application checkpointing is straightforward and easy to comprehend. The programmer himself is responsible for managing application state and it uses ordinary programming techniques. The programmer decides what data must be saved and the best location in the program to activate the checkpointing mechanism. That way, he can choose the location where the data amount is the smallest and there are no *in-flight* messages in case of a parallel application. The problem here is that it requires new analysis and implementation for every new application and it may be substantial effort for complex algorithms. Because this approach is so application dependent, there are no remarkable works that describe this in detail.

2.2.3 Code preprocessing method

Considering the effort it takes to write and maintain code for application fault tolerance there exists another approach — analyze and instrument the code with a special *source-to-source compiler*. This is called automated application-level checkpointing (AALC) and is implemented in the C3 [13] precompiler. This approach provides fault tolerance with much less effort than changing the source code manually, but still encounters all issues related to ALC.

2.2.4 Compiler-assisted method

A little different approach than using source-to-source compiler is to analyze and instrument the code within the compiler itself. This was used in CATCH [14] which extends GNU C compiler (GCC) version 1.34 to generate additional code during compilation.

2.3 Mixed-level checkpointing

There is a new approach under research mentioned in [15] which uses combination of ALC and SLC or even different SLC techniques for different parts of the same system.

For example, one may wish to use ALC for his application data, but SLC to store the internal state of the libraries. Another example is handling the result of the *gettime()* function. Depending on the invocation semantics, it may desirable to return the pre-failure result or, on the contrary, reinvoke the call and return the new result.

Chapter 3

Classification by checkpointing protocol

Classification of fault tolerance methods by checkpointing mechanism presented in 2 is valid for local as well as distributed applications. Orthogonal to the checkpointing mechanism there exists a classification by checkpointing protocol which studies methods and algorithms to synchronize local states of processes in a distributed application. This topic is actually researched more frequently because it allows theoretical approach as opposed to technical issues that arise with checkpointing mechanism.

There exists a comprehensive survey of rollback-recovery protocols[16] which classifies and examines different techniques. All techniques are divided into *checkpoint based* which rely solely on checkpoints and *log-based* which in addition use message logging to track global state of application.

The survey and many other papers use the idea of *consistent global states* in distributed application and often refer to the *domino effect*. First subsection introduces these and other concepts, then overview of the two mentioned rollback-recovery techniques is given.

3.1 Introduction to concepts

Paper [17] by Chandy and Lamport introduces concept of consistent global states and algorithm to capture one of these states. Next we present brief description of the model and results from this paper, algorithm description is partly covered later. We give a little extended description of the model to help later explaining the difference between checkpoint based and log-based rollback-recovery.

3.1.1 Model of distributed system

The presented model of distributed system consists of the number of processes and two-ended channels which are used to pass messages between processes. Every process is independent from other processes in the way that it does not share clocks or memory. Process can be in one of the known states and sequence of process states defines progression of calculation within the process. Channels are assumed to be unidirectional, sequential, error-free and have infinite buffers. Every process adds messages to its outgoing channels and takes messages from its incoming channels. Delivery of a message takes arbitrary but finite time. State of a channel is a sequence of all messages that are sent to this channel but not yet received.

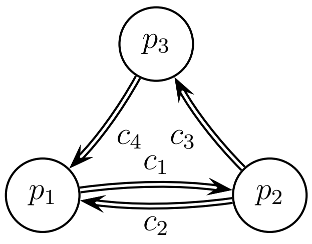

Figure 3.1: Processes and channels

Event in this model designates change of the state of exactly one process and at most one channel incident on the process. Events may be logically divided between processes and partially ordered by local physical time, so every process p has sequence of events $E^p = (e_i^p | i >= 1)$ that occur within it whereas sequences are pairwise different $E^{p_i} \bigcap E^{p_j} = \varnothing$ and union of all sequences $\bigcup_p E^p = E$ is set of all events in the system.

As an example, figure 3.2 shows the execution lines of 3 processes with the time axis along vertical direction. Event $e_1^{p_2}$ changes state of process p_2 from state $s_1^{p_2}$ to state $s_2^{p_2}$ and receives message m_1 from process p_1. Messages m_1 and m_2 are passed through some unidirectional channels which is not shown on the picture.

Event can be written as the tuple $e = (p, s, s', c, m)$ where p is a process where state change occurs, s and s' are correspondingly states before and

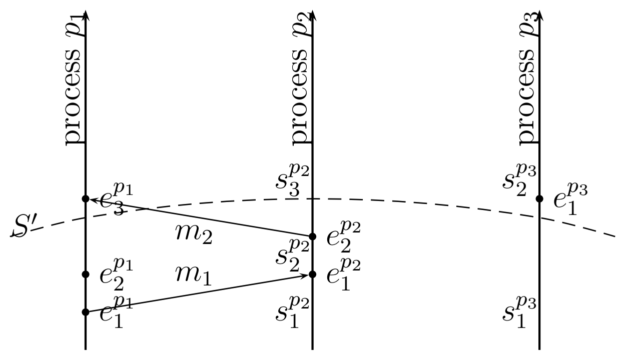

Figure 3.2: Events, process states and messages

after the event, m is a message sent to or received from channel c. Message and channel may have a special value *null* meaning no message is sent or received in the event. Event is *legal* if it is defined by the algorithm of the running process.

Chandy in his article [17] does not discuss whether more than one event is legal from the same state as his algorithm is used in checkpoint based rollback-recovery and his proofs are correct in both cases. Transitions to different states mean that data or code position of the process differ after the event which may result from *gettime()* system call, incoming channel polling or any other *non-deterministic event*. We assume that there may exist several legal transitions from the same process state, but for the simplicity we can ignore this issue until the discussion of early messages (3.1.4).

3.1.2 Global time and state

Processes do not share global clock and there is no unique way to order events of all processes. There are two rules that must be obeyed:

1. events within single process p have fixed ordering E^p
2. if event e_s is message m' send and event e_r is this message m' receive at another process then e_s must occur earlier in the ordering than e_r.

In this sense (see figure 3.2) events $e_1^{p_2}$ and $e_2^{p_2}$ happen after event $e_1^{p_1}$ but before $e_3^{p_1}$, the order of $e_2^{p_1}$, $e_1^{p_2}$ is undefined. These two rules create partial ordering of all events in the system.

Further ordering of the events is artificial and all reorderings of the events that follow above rules are equivalent. All such reorderings can be explained by variations in processor speeds and delays in message delivery. Imaginary physical global clock or logical clock from [18] can be used to define order of all events in the system $E_g = (e_i^{\pi(i)})$. The latter may be useful in the algorithms like distributed shared mutex handling, but we use the former one further as more natural from human perspective.

Global state S_i is a set of the states of all processes and all channels in the system at some fixed time point i defined by a global clock. Global state S' on figure 3.2 consists of process states $s_3^{p_1}, s_3^{p_2}, s_1^{p_3}$ and one channel state $(m2)$ (other channels are empty).

Using E_g, the definition of sequence of the global states $(S_i | i >= 1)$ is straightforward. One such sequence (S_i) is shown on figure 3.3 which was defined by the event sequence $(e_1^{p_1}, e_1^{p_2}, e_1^{p_3}, e_2^{p_1}, e_2^{p_2}, e_3^{p_1})$.

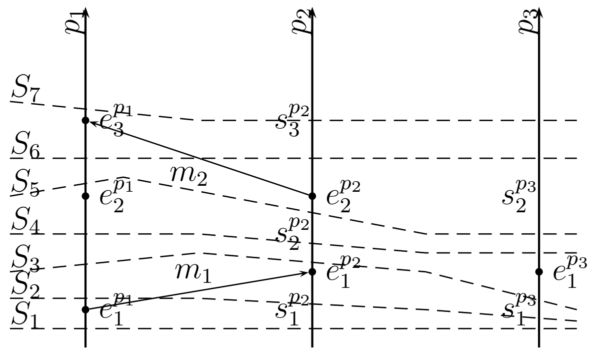

Figure 3.3: Global states

3.1.3 Consistent global states

Note that global state S' from figure 3.2 does not appear within mentioned sequence, although the global computation is the same except event $e_1^{p_3}$ appears at different global time. If process p_3 starts saving its local state later than other two processes then S' may be saved as a global checkpoint.

Similarly, if sender's local state is saved later than receiver's (see figure 3.4) then all six messages sent between those actions must be saved as the channel state, although there were at maximum two messages on the channel at a time. We can justify that state by assuming that message delivery takes longer than in the original computation, so message receipts will be shifted up the time axis.

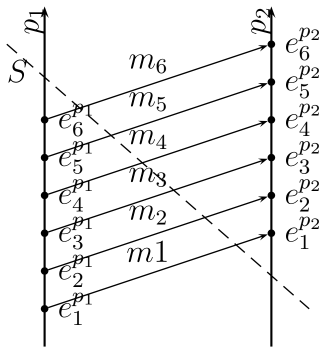

Figure 3.4: Channel state

Any global state that is potentially possible in the computation is meaningful and such states are called *consistent*.

3.1.4 Late and early messages

The messages that in global state appear as sent but not received (see figure 3.5a) are called *late*, and sometimes referred to as *in-flight* messages. Such situation is legal and messages are saved as channel state.

The messages that marked as not sent but received (see figure 3.5b) are called *early*. Such global state is not possible in normal execution and early messages are sometimes called *inconsistent*.

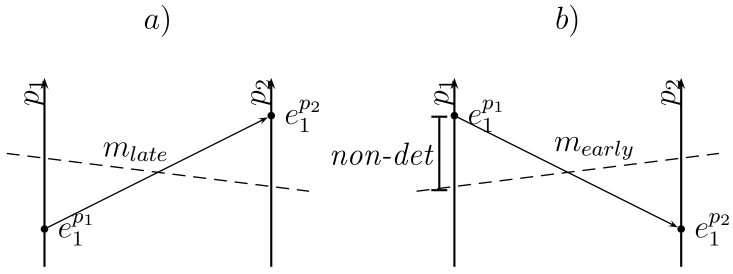

Figure 3.5: Late and early messages

The question arises if early messages can be saved as already received and then suppressed on the sender side after state recovery. This way it would be possible to store and replay inconsistent global states. If an early message is not resent after recovery because non-deterministic events in the sender (see figure 3.5b, *non-det* section) behave differently than before taking the checkpoint then the whole application state becomes illegal. Therefore, to use global checkpoint with early messages it must be assured that all early messages saved with the checkpoint are resent after recovery.

3.1.5 Piecewise determinism

Piecewise determinism (PWD) divides a process into deterministic execution sections with non-deterministic events between them. By logging and replaying these events, deterministic execution section can be extended. This allows to exactly predict execution of the algorithm from the start to the end of the section.

The very important feature of such section is that its output to the external world (console, other processes, network) is fully determined by the starting state. In the previous section we showed that this feature is needed to introduce early messages into consistent global state.

3.1.6 Domino effect

On figure 3.6 two processes are shown each having taken 3 local checkpoints with $x_0^{p_i}$ being initial state. If processes do not rely on the piecewise determinism then as shown before early messages must not exist in the system. Now, if process p_1 fails and must be rolled back to the checkpoint $x_2^{p_1}$, message m_5 is unsent and p_2 must be rolled back to the checkpoint $x_2^{p_2}$. This in turn causes message m_4 to be unsent and now process p_1 must be rolled back further to the checkpoint $x_1^{p_1}$. This is called *rollback propagation* and continues until there are no early messages in the system (here both processes reach their initial state). The phenomenon is most often referred to as *domino effect* in literature.

Set of all process and channel states where rollback propagation stops is called *recovery line*.

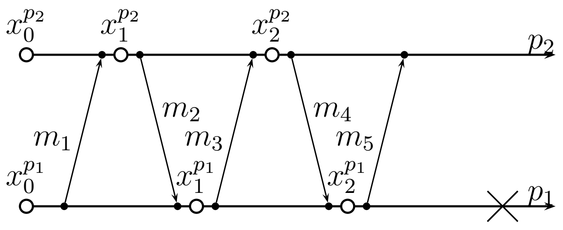

Figure 3.6: Domino effect

3.1.7 Piggybacking

Many algorithms add information to outgoing messages which is read and interpreted by the receiver of the message. This technique is called *piggy-backing.* The information may contain logical clock value, information about sender state or even information updating some distributed data structure.

If communication channels are reliable and preserve order of messages, piggybacking may be replaced by the second message which is sent just before or after the original message.

3.2 Checkpoint based

Checkpoint based rollback-recovery relies only on checkpoints of processes and channels and does *not* use assumptions about PWD.

Different techniques of checkpoint based rollback-recovery differ in the way they solve rollback propagation problem. *Uncoordinated* checkpointing does not make any steps to avoid it, *coordinated* checkpointing ensures that every global snapshot is consistent and *communication-induced* checkpointing ensures that recovery line progresses with local checkpoints.

3.2.1 Uncoordinated checkpointing

With uncoordinated checkpointing every process decides when to take local checkpoint independently from the others. This allows to avoid any synchronization steps and also to minimize the size of a local checkpoint by taking it when program data set is minimal. The disadvantage of such approach is the possibility of the domino effect. Another problem is determining recovery line which must be done during rollback-recovery. In addition several checkpoints must be maintained for each process because it is not known which of them belongs to recovery line. Periodic algorithm should be used to update recovery line and reclaim unneeded checkpoints.

This approach is not very useful, especially for the algorithms with intensive inter-process communication, because probability of early messages is high. It becomes more valuable with addition of message logging in log-based checkpointing (see section 3.3).

3.2.2 Coordinated checkpointing

With coordinated checkpointing all processes synchronize their activity to take one consistent global checkpoint.

It can be further divided into *blocking* and *non-blocking* coordinated checkpointing. The blocking approach stops all communications and ensures that all channels are empty then saves local checkpoints of every process. By doing that it ensures that there are no early or late messages in the system and the set of local checkpoints is complete. The non-blocking approach takes global checkpoint while at the same allowing application to run.

Blocking algorithm is often implemented in MPI libraries and supercomputers because it is the simplest approach to coordinate local checkpoints.

Chandy-Lamport algorithm One of the most referred non-blocking algorithm is presented next. Chandy-Lamport algorithm uses model presented in 3.1.1 and is a classical checkpoint based rollback-recovery algorithm. It uses one additional message type — *marker* message — that can be distinguished from application messages. Special *control* channels between processes may be used for the instruction to start checkpointing.

State saving activity starts by initiator process or by external instruction to one or more processes. It is necessary to deliver initiating instruction through control channels to any process which by some reason avoids normal communication.

After a process takes local checkpoint, it:

- sends a marker message to all outgoing channels before sending any other messages;

- starts recording messages of all incoming channels as corresponding channel state.

When a process receives a marker message from any channel:

- if process state is not saved then takes local checkpoint (with mentioned additional actions) and finishes saving state from the channel where marker was observed (saves this channel state as empty);
- if process state is already saved then stops recording this channel's messages (saves all messages recorded from this channel as channel state).

After all process and channel states are saved they compose one consistent global checkpoint. Consistency is achieved by ensuring that local process state is saved before receiving any further messages after the marker message.

During recovery, saved channel messages are resent or gathered at the receiver side and ready to be processed. This changes the delivery time of the saved messages and due to non-deterministic events may cause different result of the application after recovery. Nevertheless, recovered state is still legal because message delivery times and processor speeds are not fixed in the model and can be adjusted to justify new state.

3.2.3 Communication-induced checkpointing

Communication-induced approach is a trade-off between coordinated and uncoordinated checkpointing. Additional checkpoint dependency information is exchanged between processes by piggybacking it on the messages and some dependency graph may be constructed, tracked and analyzed. As the result taking of local checkpoints may be forced to ensure that recovery line progresses with the new checkpoints.

3.3 Log-based

Log-based rollback-recovery relies on PWD in addition to checkpoints to handle early messages. During normal run all non-deterministic events are logged and then replayed during recovery of the process. Three approaches exist that differ in how strictly they ensure that current computation state is achievable during recovery. *Pessimistic* logging commits every non-deterministic event log before allowing it to affect the computation, *optimistic* logging commits log asynchronously and rolls back state if latest events are not recoverable, *causal* logging uses quite sophisticated logging algorithm to ensure recoverability of the current state.

The main advantage of the log-based rollback-recovery is possibility to recover failed processes without touching other processes. The drawback is the overhead related to non-deterministic event logging.

3.3.1 Pessimistic logging

Pessimistic logging assumes that failure may occur after every non-deterministic event (pessimistic assumption) and therefore considers event and log write as an atomic action. Such synchronization of every event with the log may result in poor performance, but the system is fully recoverable at any time.

3.3.2 Optimistic logging

Optimistic logging stores recent logs in the volatile memory and periodically flushes them to stable storage. This results in better performance but may require more processes to rollback because of lost logs.

3.3.3 Causal logging

This is trade-off between reliability of the pessimistic logging and synchronization free execution of optimistic logging. For more detailed description of this and other algorithms the survey [16] of the rollback-recovery protocols can be explored.

Chapter 4

Solution for FORTRAN 90 and MPI

This chapter introduces one approach to implement fault tolerance feature for parallel programs written in FORTRAN 90 programming language and using MPI system for communication. First section describes motivation and requirements, second section proceeds with the description of the algorithm and third section brings out implementation problems and details.

4.1 Choice of the techniques

One of the motivations for this work was need to add fault tolerance feature to the DOUG package [1]. DOUG is a parallel iterative linear solver with two-level preconditioners based on geometric and aggregation methods. First version of this project written in FORTRAN 77 was started at University of Bath [19] and a new version is developed in cowork with University of Tartu. The new version of DOUG is written in FORTRAN 90 and uses MPI library for communication. It is tested with several FORTRAN compilers and MPI implementations.

4.1.1 Requirements

DOUG is not intended for one particular cluster or supercomputer but should run on any available installation in GRID environment. Because of that most requirements to the fault tolerant DOUG are there to keep it's portability. Here we give list of the requirements to the system:

- has impact on application performance
- compiles with any FORTRAN 90 compiler (anycomp)
- works with many MPI implementations (anympi)
- is OS/system independent (anysys)
- does not require special extensions to the standard installations (kernel modules, extended systems) (noext)
- is transparent to the user (transp)
- does not need the source code (nosource).

The results of previous attempts to the fault tolerance for scientific computing show that any carefully designed backward fault tolerant implementation gives no more than ten percent drop in application speed, although SLC gives some better performance during run-time than ALC. So we ignore the performance impact and use other requirements to select the technique.

4.1.2 Checkpointing mechanism

Checkpointing mechanism defines how checkpoints are taken and as the variety of available systems is tremendous with each having different versions and extensions, good choice of checkpointing mechanism is of high importance.

Here we consider following candidates:

- SLC in kernel mode (SLCk)
- SLC in user mode (SLCu)
- ALC with manually inserted code (ALCm)
- automated ALC with compiler inserted code (AALCc)
- automated ALC with precompiler generated code (AALCg).

Any SLC technique is compiler independent because it manipulates system library calls or kernel internals. Within ALC techniques only compiler driven fault tolerance is not acceptable, others should work with any compiler.

MPI lies between application and system holding all the information about messages and their delivery status. MPI layer should track all late and early messages and decide when local checkpoints must be taken. Making fault tolerance MPI implementation independent without extending MPI interface is arguably the most difficult problem.

MPI implementations usually support one particular SLC solution to make checkpoint of the library and application at once. They also often provide callback for the application to handle ALC based checkpoints. Because none of the SLC implementations support socket checkpointing there is no truly independent solution. But there exists an ALC solution [13] which builds a layer on top of MPI and tracks any sent or received messages. As most widely used MPI implementations gradually extend list of supported SLCs and implementing a coordination layer is quite complicated we do not favor any of these solutions.

SLCk is always built into particular operating system, SLCu solutions largely use OS dependent mechanisms, all ALC solutions are mostly OS independent.

SLCk requires special modules or even patch to the operating system, SLCu may be statically linked into application but often needs special extensions for MPI to work. ALC does not rely on any extensions during run-time, although this makes ALC solutions more complicated.

All solutions except ALCm are transparent to the user, although all of them limit what user can use in the program. SLCk requires no original code, SLCu may require object files to link it's library, all ALC solutions require source code of the application. The last requirement is one of the reasons why there is more effort toward SLC than ALC.

Table 4.1 lists all requirements and possible techniques with evaluation and importance for the DOUG project. Despite that evaluation and importance values are quite artificial, it nevertheless shows that AALCg may be the best solution for DOUG.

	SLCk	SLCu	ALC	AALCc	AALCg	importance (0-3)
anycomp	+	+	+	-	+	3
anympi	.	.	.	.	.	3
anysys	-	-	+	+	+	2
noext	-	-	.	.	.	2
transp	+	+	-	+	+	3
nosource	+	.	-	-	-	0
	2	2	2	2	8	

Table 4.1: Evaluation of checkpointing mechanisms

4.1.3 Checkpointing protocol

When choosing checkpointing protocols we primarily consider DOUG requirements. DOUG as a parallel iterative solver has quite intensive communication between processes during the whole computation. Considering this any log-based protocol may cause too large overhead when saving application messages. As it also supposed to be run on tenths or hundreds of nodes any blocking feature is undesirable.

Any version of non-blocking checkpoint based protocol is a good solution for the fault tolerance feature of DOUG. But as was mentioned in the previous section ALC based solutions that do not rely on special MPI implementations may have major problems with coordinating local checkpoints, so we let existing solutions of checkpointing mechanism to dictate on the choice of checkpointing protocol.

4.1.4 Summary

Having evaluated all possible techniques AALC with source code generation and a variant of non-blocking coordinated checkpointing protocol were chosen. This should give enough portability to run on almost any available cluster.

4.2 Algorithm description

The algorithm is presented using FORTRAN 90 language terms and syntax, but the same ideas have been applied for other languages [13] in the past.

Fault tolerance is achieved by inserting special code sections around a number of subroutine calls and modules. Insertions are done by the source-to-source compiler (precompiler) which reads original source code, finds locations where the code must be inserted and generates new source files with fault tolerance functionality woven into them. These new source files are then compiled and run as usual, except checkpointing library must be additionally linked to them.

Programmer of the application decides where local checkpoints should be taken and places annotations (special comments) into those locations. The command to start checkpointing may come from several sources: by signal, received from control channel or triggered by a piggybacked information of the message from another processor. Wherever that command is received checkpoint of the local state may only be taken at the annotated locations, so the process continues execution until reaching potential checkpoint location.

As this is a coordinated checkpoint based algorithm all processes cooperate to take one global consistent checkpoint. The whole execution is divided into *epochs* with new epoch starting after local checkpoint is taken. Epochs are sequentially numbered starting at 1 and epoch numbers of two processes may not differ more than 1 (see figure 4.1). Dashed ellipse marks the procedure of saving global state which is described in more detail in 4.2.2. The transitions of the processes to the epoch 2 happen by saving their local checkpoints ($x_1^{p_1}, x_1^{p_2}, x_1^{p_3}$) and transitions to a new epoch may not start until global state is saved.

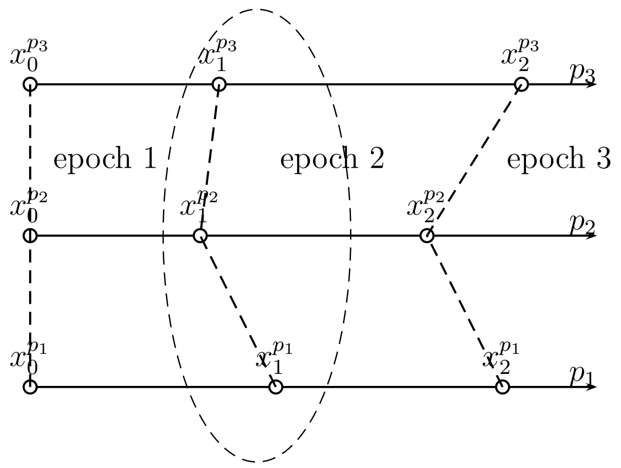

Figure 4.1: Epochs

The current epoch number `cpr_epoch` and other information about the progress of the checkpointing is stored in additional FORTRAN module `cpr_m`.

The next two sections describe procedures of saving local state and coordination algorithm to save consistent global checkpoint.

4.2.1 Saving process state

Saving of process state is started at one of annotated locations marked by FORTRAN comment `!!CPR potentialcheckpoint`. This location is found by source-to-source compiler and instrumented with the conditional checking for the need of starting new checkpoint.

As there is no simple way to save stack in FORTRAN, algorithm uses one of two known techniques — it first collapses the stack by gradually exiting from subroutines and has a chance to save all local variables then it restores the original stack back by entering exactly the same subroutines. These two phases are called correspondingly *save* and *restore*, the third phase called *recover* being almost identical to the restore phase is used when process is rolled back because of a failure and restored from the latest saved checkpoint.

The following FORTRAN code is inserted after every potential checkpoint location and the last *if* block is inserted after every subroutine call that may be involved in stack collapse/extend procedure.

```
  !!CPR potentialcheckpoint
1 if(cpr_state==CPRS_RESTORE .OR. cpr_state==CPRS_RECOVER) then
   call cpr_setState(CPRS_NORMAL)
end if
if(cpr_state==CPRS_NORMAL .AND. cpr_docheckpoint) then
    cpr_docheckpoint = .FALSE.
    call cpr_setState(CPRS_SAVE)
end if
! Save state and exit from the call
if(cpr_state==CPRS_SAVE) then
   call cpr_saveLabel(1)
   call cpr_save(r)
   call cpr_save(rho)
   ...
   call cpr_save(alpha)
   return
end if
```

The first *if* block finishes and the second *if* block starts saving the local state. The third *if* block saves local variables and exits function call (i.e. stack collapse).

Variable `cpr_state` tracks the phase of collapsing (*CPRS_SAVE*) or extending (*CPRS_RESTORE* or *CPRS_RECOVER*) stack and has the value *CPRS_NORMAL* during normal execution. When an external command to take new checkpoint comes the `cpr_docheckpoint` logical variable is set to *true*.

Application position

When exiting subroutine call the exact location must be remembered, so it can be restored during restore/recovery phase. This is done by placing unique label before the call and saves this label after returning from the call if the stack is collapsing.

```
3  par_x = Kaasgradientide_meetod(par_A,par_y,eps,it)
! is stack collapsing
if(cpr_state==CPRS_SAVE) then
   call cpr_saveLabel(3)
   ...
   return
endif
```

When entering subroutine containing above code and stack is extending then the saved label must be inspected and execution transferred directly to the saved location. The following code fragment is inserted just in the beginning of a subroutine.

```
! is stack extending
if(cpr_state==CPRS_RESTORE.or.cpr_state==CPRS_RECOVER) then
  ...
  call cpr_restoreLabel(lcpr_label)
  select case (lcpr_label)
     case(1); goto 1
     case(2); goto 2
```

```
    case(3); goto 3
  end select
end if
```

This continues until the original `!!CPR potentialcheckpoint` is reached.

Loop constructs Special care must be taken when jumping into loop constructs (*while, for*), because due to compiler optimizations it may cause incorrect execution. This problem may be avoided by jumping to the loop construct, entering the loop and jumping further to the desired location.

Application data

Local variables are saved and restored exactly at the same locations as the labels: saved (i) after an annotation and (ii) after a subroutine call, restored (iii) at the beginning of a subroutine. Variables are not saved by their name but in fixed order, so they must be restored in reverse order.

Subroutine parameters in FORTRAN have *in*, *out* or *inout* attributes attached to them meaning correspondingly that the value of the parameter is passed to the subroutine, returned from the subroutine or both. By the semantics it is necessary to save out and inout parameters, but because many FORTRAN compilers pass parameters by reference it is often possible to ignore parameters at all.

Global and module variables are saved when stack is fully collapsed and *main* function is reached. They are restored only when the process state is recovered because of the failure, otherwise they are not touched.

Pointers Saving and restoring pointers may be tricky in any language. In addition FORTRAN also has *allocatable* variables which is some analogue of a pointer but dynamic memory of an allocatable variable must be freed as soon as execution exits the subroutine where the variable is declared. As a simplification our code only tracks pointers that hold references to the allocated dynamic memory regions, in other words it does not handle case when pointer is assigned to an already existing memory location.

When pointer to an array is saved, *lower* and *upper* bounds of the array must be saved. If the pointer is not associated with dynamic memory 1 and -1 are saved as array bounds.

```
if(associated(so_reg)) then
  call cpr_save(so_reg)
  call cpr_save(lbound(so_reg))
  call cpr_save(ubound(so_reg))
else
  call cpr_save(1)
  call cpr_save(-1)
endif
```

To restore a dynamically allocated array referenced by pointer its upper and lower bounds are read and the size of the array says if the pointer was associated during checkpointing.

```
call cpr_restore(lcpr_ubound)
call cpr_restore(lcpr_lbound)
lcpr_size = lcpr_ubound+1-lcpr_lbound
if(lcpr_size>=0) then
  if(cpr_state==CPRS_RECOVER) allocate(so_reg(lcpr_lbound:lcpr_ubound))
  call cpr_restore(so_reg)
else
  nullify(so_reg)
endif
```

Derived types For every user defined type a special procedure is generated that goes through all its elements and saves them as usual. Here is the example of saving *par_sparse_m* derived type.

```
subroutine cpr_save_par_sparse_mD0(t)
  use cpr_m
  integer :: lcpr_label, lcpr_size, lcpr_lbound, lcpr_ubound

  type(par_sparse_m),intent(in) :: t
  call cpr_save(t%siseyhendusi)
  call cpr_save(t%variyhendusi)
  call cpr_save(t%sise_mat)
  call cpr_save(t%vari_mat)
end subroutine cpr_save_par_sparse_mD0
```

4.2.2 Saving MPI state

This section describes how MPI state is tracked and checkpointed, presented algorithm closely follows the ideas from [20]. A special *coordination* layer is inserted between application and MPI library (see figure 4.2) which intercepts all MPI calls. This is done by the same source-to-source compiler which replaces original MPI calls with the delegating method calls. The coordination layer recognizes different MPI calls and takes specific actions to track logical state of MPI library.

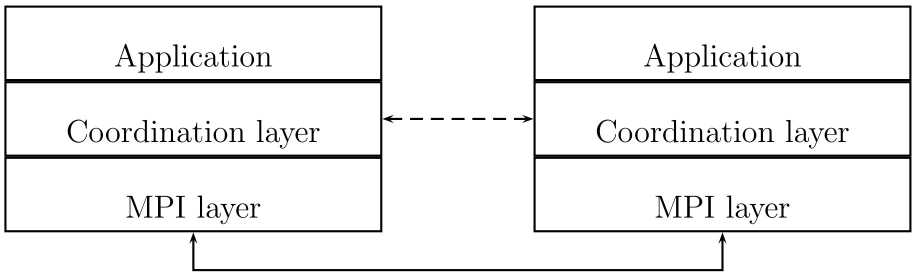

Figure 4.2: Layer stack

High level description of the algorithm

This algorithm is a variation of the Chandy-Lamport algorithm described in 3.2.2. There are 3 modes for each process during global state checkpointing (see figure 4.3): normal, non-deterministic logging (*non-det*) and late-message capture (*late-capture*).

Tuple $(tag, comm, source, target)$ consisting of MPI tag, MPI communicator, sender and receiver identities defines a separate channel between two processes. By the Chandy-Lamport, to capture state of the channel we need to send special marker message to every channel and reception on the other side marks where the channel state ends. Unfortunately, it is not a very good solution for MPI: (i) tag is an integer value which makes number of possible channels very large, (ii) recognizing special message on the channel can be an unresolvable problem in MPI, at least there is no obvious solution. It is possible to use piggybacking on every message to overcome this problem — we add epoch number from which message was sent. For example, on figure 4.3 the message m_{late} has number n piggybacked on it which tells receiver that it is a late message. Late messages must be locally saved into the *late-message-registry* within checkpoint as the corresponding channel state.

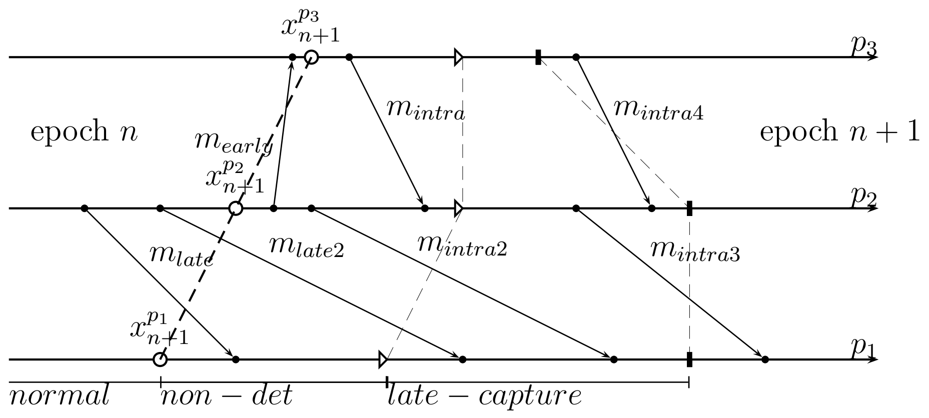

Figure 4.3: Allowed messages

Handling early messages One substantial difference between SLC and ALC is that we cannot force checkpoint in ALC, it can only be taken at predefined code locations. This means that early message (m_{early}) must be received without first taking local checkpoint as expected by Chandy-Lamport algorithm. This makes early messages possible and we must rely on PWD and save non-deterministic events. This mode is showed as *non-det* on the figure 4.3 and must continue until all processes save their local state and this information reaches a process. This transition is shown by a triangle on the figure.

In addition to any application internal non-deterministic event source there are 2 more sources related to MPI:

- *MPI_Recv* subroutine call with `ANY_SOURCE` or `ANY_TAG` parameters
- *MPI_Test* subroutine call (message polling).

In the first case process saves actual source and tag of the received message into *nondet-event-registry* and then substitutes these values during recovery before looking into *late-message-registry* (for m_{late}) or calling actual *MPI_Recv* (for m_{intra}). In the second case the process must save how many times *MPI_Test* call has been invoked and what values were returned in each call, and then replay it during recovery.

The receiver of an early message must save its identity in *early-message-registry* and during recovery the sender should use this information to suppress the message. The identity may be the message channel and a sequence number of the message sent to that channel in that epoch. It is, actually, sufficient to hold the number of messages that must be suppressed for every channel.

This mode completes when all processes save their local state because it makes new early messages impossible. Every process informs the master process when it saves local state. When the master knows that every process has finished saving local state it broadcasts that information through the control channel.

There is another subtle problem here, note that the only disallowed message is the one sent from *late-capture* mode and received in *non-det* mode. Such message would be sent from non-deterministic region and received in deterministic one. It may occur that after a recovery the message is not resent and deterministic region will not receive the message its further *determined* execution relies on. The mode may be piggybacked on the message and if the sender has stopped logging non-deterministic events the receiver must do the same. This is intuitive as it means that sender knows every process has taken local checkpoint and it is now made known to the receiver.

Capturing late messages To finish checkpoint a process must capture all late messages (we assume that every sent message will be received), then it can save the checkpoint and leave *late-capture* mode. The process keeps count of messages sent to and received from every process in current epoch and the number of early messages received from every process. Number of early messages must be added to the next epoch receive count.

When a process takes local checkpoint it informs other processes about the number of sent messages during the previous epoch. The difference of sent and receive counts is the number of late messages to receive.

Recovery mode When process is recovered from a checkpoint it uses *early-message-registry* to suppress sending early messages, *nondet-event-registry* to replay non-deterministic events, including *MPI_Recv* calls with `ANY_TAG` or `ANY_SOURCE` values, and *late-message-registry* to get data of the late messages.

When all three registries are empty a process may proceed to *normal* mode.

Point to point communication

Here is pseudo code of how MPI wrappers for point to point communication should look like. It follows the rules given in previous sections. First, *checkControlChannel()* checks if any control messages have arrived that inform about the need to start taking checkpoint or the end of *non-det* mode.

The wrapper subroutine (see 4.1) *CPR_MPI_Send* only has to track early messages in *recovery* mode that must be suppressed. If another sent message makes all 3 registries empty then *recovery* mode finishes.

Code 4.1 *MPI_Send* wrapper

```
CPR_MPI_Send(data,comm,tag,target)
   ! check control channel for state transitions
   checkControlChannel()

   source = me
   channel = (tag,comm,source,target)

   ! supress early messages
   if(mode==recover)
      if(early-message-registry[channel]>0)
         early-message-registry[channel]--
         if(registries are empty)
            mode=normal
         return MPI_SUCCESS

   MPI_Send() with piggybacked (epoch,mode)
   sent-count[target]++
```

The *CPR_MPI_Recv* wrapper (see 4.2) in *recovery* mode must replay non-deterministic receive events and grab messages from the saved input channel until it is empty. If all three registries are empty then *recovery* mode is over. When a new message is received and it appears to be late or early message then corresponding registry must be updated. The last condition ensures that if the message says that sender has stopped logging non-deterministic events then receiver must also stop this.

It is also possible to extend above algorithm to use non-blocking and collective MPI communications as described in [20].

Collective communication

Collective MPI operations may cross recovery line as shown on the figure 4.4. Consider situation given for *MPI_Allgather* collective operation: process p_1 takes local checkpoint before *MPI_Allgather* call while processes p_2 and p_3 do the opposite. This is possible not only because collective operation does not have to be blocking but taking local checkpoint is also non-blocking operation.

The problem is that unlike others, process p_1 invokes collective operation after recovery which leads to incorrect state. The process p_1 should save gathered data into checkpoint and return it without calling actual collective operation during recovery. For better understanding, collective operation may be viewed as *sendreceive* operation on a separate channel (`comm`, `COLLECTIVE_TAG`). This requires process p_1 to save/substitute late data from other processes and remember/suppress early data to other processes.

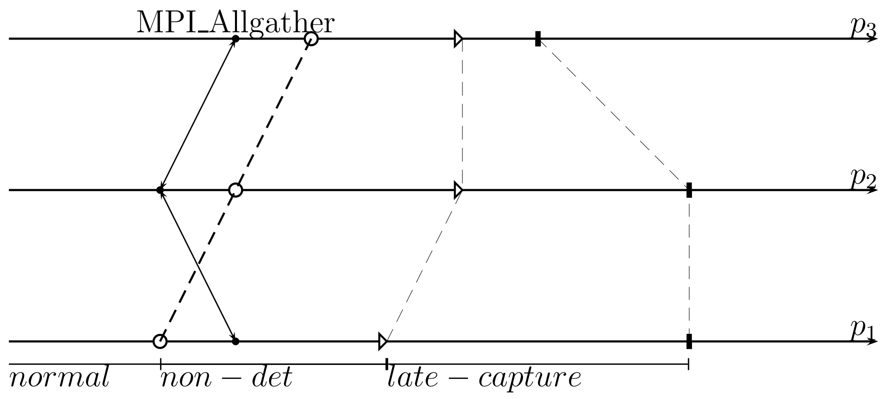

Figure 4.4: Collective communication

Notice that transition line from *non-det* mode to *late-capture* mode must not be crossed by collective communication exactly for the same reason as early messages. So they must inform others if some of them stopped logging non-deterministic events. It may be more tricky to piggyback that data

Code 4.2 *MPI_Recv* wrapper

```
CPR_MPI_Recv(data,comm,source)
   ! check control for state transitions
   checkControlChannel()

   ! determine channel (i.e. replay this non-det event)
   if(mode==recover)
      if(tag or comm are undefined)
         tag,comm = next(nondet-event-registry[
                          (INDETERMINATE-CHANNEL, me)])
         if(registries are empty)
            mode=normal

   target = me
   channel = (tag,comm,source,target)

   ! substitute late message
   if(mode==recover)
      if(late-message-registry[channel] not empty)
         data = next(late-message-registry[channel])
         if(registries are empty)
            mode=normal
         return data

   MPI_Recv() read piggybacked (r_epoch,r_mode)
   isEarly = r_epoch>epoch
   isLate = r_epoch<epoch

   if(isEarly)
      early-received-count[source]++
      early-message-registry[channel]++
   elseif(isLate)
      late-received-count[source]++
      add message to late-message-registry[channel]
      if(late-received-count==expected-late-count)
         mode=normal
   else
      received-count[source]++
      if(mode==nondet and r_mode!=nondet)
         mode=normal
```

on collective operation, so before every collective operation all concerning processes exchange their epochs and modes by executing *MPI_Allgather* with these values.

Non-blocking communication

Non-blocking send and receive operations are shown on figure 4.5. The question is if operation *ISend* or *Wait* should be considered as message send event. The process must not access the buffer passed to *ISend* until *Wait* returns nor knows it anything about the message during this period, so the message is considered *in-flight* after *MPI_ISend* returns. We may look at it from the other point — any data that gets passed into the message must be defined before the *MPI_ISend* call and following non-deterministic events or just calculations do not affect the message.

The behavior of *MPI_IRecv* is the opposite — message gets only known to the process when *MPI_Wait* returns, so data in the message may not influence application execution before the *MPI_Wait* call.

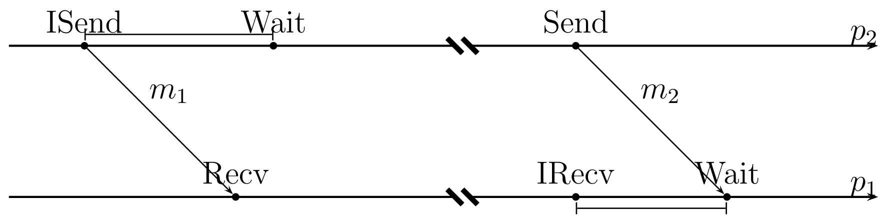

Figure 4.5: Non-blocking communication

If non-blocking operation initiation appears in one epoch and *MPI_Wait* in the next epoch then special care must be taken of requests' information, because MPI state is not checkpointed and these requests will be unknown to the MPI layer after recovery. For every created request, all its information necessary to restore it after a recovery is duplicated in coordination layer and saved with the checkpoint.

There exists a problem with non-blocking communication and stack collapsing. Some memory may be temporary deallocated during this process which may occur to be a communication buffer.

MPI objects

MPI datatypes, communication channels and groups may be handled by remembering all the operations on them during original execution and then replaying it on recovery. This way coordination layer holds copy of all created datatypes, channels and groups within itself, saves them into checkpoint and then recreates them after reading the checkpoint.

4.3 Implementation details

The project is written mostly in Java programming language with some C and FORTRAN code.

4.3.1 FORTRAN parser

For FORTRAN 90 parser and code generator ANTLR[21] tool was used together with FORTRAN 90 grammar taken from Eclipse PTP[22] subproject. At the time of writing ANTLR project was in beta version and PTP FORTRAN parser only handled recognition of language tokens.

Application extends given versions of frameworks by building partial syntax tree and some analysis of FORTRAN source code. Then original source files are modified with the StringTemplate engine[23] by inserting checkpointing code into calculated locations.

Because ANTLR was not ready and PTP FORTRAN parser was in a very early stage it has occurred that application supports quite limited set of FORTRAN syntax. It does not handle FORTRAN labels and some specific constructs.

4.3.2 Coordination layer

Implementing a fully featured coordination layer appeared to be a hard task. To save time and achieve the results, only MPI features needed for the "proof of concept" and performance tests were implemented in the coordination layer. This does not include most of the collective routines, communicators, groups and derived types.

4.4 Performance tests

For the tests, a FORTRAN 90 implementation of CG (conjugate gradient) algorithm was used. The CG method is a simple iterative parallel solver for linear equations and it has the same behavior and communication pattern as DOUG.

4.4.1 Test algorithm

The main part of the algorithm consists of the iterations with varying point to point and collective communication calls. During point to point communication every process sends messages (several kilobytes each) to about three other processes. The collective communication sums one floating point value of every process and delivers the result to all processes (*MPI_Allreduce*). The starting snippet of the CG iteration code was instrumented with `!!CPR potentialcheckpoint` comment and new set of source files were generated.

The Laplacian matrix with 90000 unknowns was generated for the algorithm containing about 450000 non-zero values. Total number of iterations was over 1000 to solve the problem with 10^{-11} precision.

4.4.2 Test setup

The application was developed on a single machine with *gfortran 4.1.2* compiler and *OpenMPI 1.1* implementation of MPI. Later, the performance tests were run on a cluster consisting of eight 1000MHz Dual Core AMD Opteron processors. The cluster runs GNU/Linux operating system, has *Intel Fortran 9.1* compiler and *LAM/MPI 7.1.2* implementation of MPI installed. No source changes to the application were needed to run with different compiler and MPI library. All checkpoints were saved to and restored from the mounted NFS tree.

The CG algorithm was run in six modes

1. original CG (Original)
2. generated CG with only application data checkpointing code, MPI calls were left unchanged (+Data)

3. generated CG with only application data checkpointing code and instrumented MPI point to point communication calls (*MPI_Send*, *MPI_Recv*, *MPI_ISend*, *MPI_IRecv*, *MPI_Wait*), MPI collective communication calls were left unchanged (+Point to point)

4. generated CG with all functionality but without taking checkpoints (+Collectives)

5. generated CG with all functionality and taking single checkpoint (+Checkpoint)

6. restoring from the saved checkpoint (Restore).

All modes were run with 1, 4, 9 and 16 processors 5 times and run times of the algorithm were recorded. The highest and lowest times were ignored and mean of 3 other values was taken. The recovery mode does not have timings recorded because they are meaningless — *gettime()* function call was not instrumented and the resulting run time was generally the time between two executions of the algorithm. All modes worked without errors and gave correct answers.

4.4.3 Test results

Table 4.2 presents mean values of run times given in seconds and ratio coefficients to the original code execution time. The time intervals were measured by *MPI_Wtime* calls and synchronized with *MPI_Barrier* calls, so they do not include start up time of CG application and MPI environment. The total size of the saved global checkpoint for each case is given in the last row.

The results show that there is no substantial difference until collective communication is wrapped by the the coordination layer. The problem is that it requires additional *MPI_Allgather* call for every collective MPI operation to collect remote epoch numbers. This approach is very inefficient when number of processors gets high.

The overhead from saving the program state is noticeable although not large. Two major factors may affect the result:

- process is stalled when taking local checkpoint (*copy-on-write* helps with SLC solutions)

mode/processors	1	4	9	16
Original	13.42	5.62	4.01	4.45
	1	1	1	1
+Data	13.2	5.33	4.41	4.76
	0.98	0.95	1.1	1.07
+Point to point	12.31	5.16	4.51	4.76
	0.91	0.92	1.12	1.07
+Collectives	12.26	5.54	4.97	10.25
	0.93	0.99	1.24	2.3
+Checkpoint	12.99	5.36	6.55	10.91
	0.97	0.95	1.63	2.45
Restore	OK	OK	OK	OK
Size	21M	1x12.4M+ 3x3.2M	1x10.7M+ 8x1.6M	1x10M+ 15x1.08M

Table 4.2: Performance test results

- simultaneous checkpoint transfers over the network may overload it, so both state capture speed and basic algorithm communications are affected.

Tests with larger input data, more processors and longer run time should clear if the overhead resulting from checkpoint data transfer is noticeable.

Chapter 5

Conclusion and future plans

Using the overview given in the first three chapters of this work we found that automated application-level checkpointing could be the best fault tolerance solution for scientific applications with available source code. As a result we have implemented and tested a proof of concept solution for FORTRAN 90 programs that use MPI.

This chapter summarizes fault tolerance techniques and the effort made toward automated application-level checkpointing. It also presents some ideas of why it is difficult to implement fault tolerance and what can be done to improve the situation.

5.1 Conclusions for AALC effort

Implementing AALC with a layer over MPI is a challenging task which requires much effort in designing and tuning the coordination layer and has many hidden obstacles. The performance issues must be taken seriously and possible solutions in the coordination layer must be carefully evaluated.

Nevertheless, the proposed solution may altogether take less effort than SLC. This is because it uses MPI standard and does not require any changes once implemented. With the SLC the situation may be different [24]:

> By far the most challenging aspect of implementing BLCR was to keep it working as the Linux kernel continued to evolve.

We have showed that when such solution is implemented, adding checkpointing functionality to the scientific application becomes easy. The resulting code is fully portable running on potentially any hardware and software platform that has FORTRAN compiler and MPI implementation.

5.2 General ideas learned

While investigating the existing solutions and implementing one of them, more general ideas about the implementation problems of fault tolerance feature and their sources has been revealed to the author.

In forward fault tolerance the requirement of sustaining failures is highly important and is built into the design of the system. With backward fault tolerance this requirement, usually, does not fit into algorithm definition nor the specification of the subsystem where the application runs. The specification of a protocol or a programming language is strictly narrowed to accomplish tasks that are most desired at the moment of writing. This is reasonable attitude because considering all potential uses during design is not feasible. But as a result, developers have to overcome given limitations by digging into and extending the implementation or building unnatural constructs on top of the system.

5.2.1 Efficiency vs. abstraction

Many problems can be solved by abstracting data and operations as done in high level languages. Efficiency is often the obstacle that keeps us from extending the language or the system. This requirement exists with high priority in HPC world. Nevertheless, we give some possible abstractions that can make implementing fault tolerance much easier.

Neither of the proposed ALC techniques to save application stack is natural. If the programming language would consider stack as a first-class object it could make state manipulation easy. With this object, saving the application state would be much easier.

In many systems it is difficult to piggyback data on the messages. By implementing unnatural solutions it causes drastic slowdown to MPI collective communication when the application is run on many nodes. If the message data passed to MPI would be abstracted with an object it could make piggybacking much easier. This problem exists not only with MPI system but also on other layers of communication stack [25].

5.3 Further development plans

No other AALC solutions for the FORTRAN language are known to the author. As the solution promises to give highly portable fault tolerance feature to the scientific applications it is worth developing it further. Next steps can be taken in order to

- solve efficiency problems with MPI collective communication;
- complete FORTRAN parser or better move to the PTP project [22] implementation if possible;
- solve problem with stack collapse causing memory given to the asynchronous communication to be temporary freed;
- complete implementation by wrapping all MPI features.

The last step can be different considering there is original implementation of the coordination layer in development[11]. It is being written for the C language and it may reasonable to adapt FORTRAN 90 checkpointing functionality to their implementation.

Resümee

Tõrketaluvus on tarkvara omadus üle elada tekkinud vead riistvaras või suhtluskanalites. Üks informaatika valdkond, kus selline omadus on väga tähtis, on suure jõudlusega arvutused (High Performance Computing).

Tüüpiline teadusprogramm jookseb mitu tundi, päeva või isegi nädalat. Sellise programmi töö käigus tekkinud viga, mis põhjustab hetkeseisu tulemuste kaotamist ja arvutuste taaskäivitamist, on väga ebameeldiv. Need programmid töötavad tavaliselt mitme tuhande proetsessoriga superarvutil või arvutite klastril. Viimastel aastatel protsessorite arv superarvutites ja klastrites järjest suureneb. See olukord teeb vea tekkimise tõenäosuse väga suureks ja sunnib arendajaid lisada oma programmidele tõrketaluvuse omadust.

Tänapäeval teevad GRID projektid paljude instituutide teadusarvuteid ja arvutiklastreid avalikult kättesaadavaks. Enamus nendest süsteemidest kasutab oma spetsiifilist tõrketaluvuse mehhanismi või üldse ei paku sellist võimalust. Standartse tõrketaluvuse vahendi puudumine paneb arendajat iga klaastri võimalusi uurida ja lisama vastavat koodi oma programmile.

Selle töö eesmärgiks oli uurida ja lisada DOUG[1] paketile tõrketaluvuse omadust. DOUG on paralleelne, iteratiivne lineaarvõrrandisüsteemide lahendaja, mis kasutab määramispiirkonna jagamise meetodit. See pakett on Bathi Ülikooli ja Tartu Ülikooli koostööprojekt, mis on kirjutatud FORTRAN 90 programmeerimiskeeles ja kasutab MPI suhtlusteeki. Kuna DOUG jookseb potentsiaalselt igal arvutiklastril, kuhu on installeeritud FORTRAN 90 kompilaator ja suvaline MPI teegi implementatsioon, siis oli üheks peamiseks nõudeks säilitada porditavust.

Töö käigus sai uuritud tervet tõrkekindla tarkvara meetodite valdkonda ja leiti sobilik lahendus probleemile. Kolmes esimeses peatükis esitatakse erinevatest allikatest kokku pandud tõrkekindluse pakkuvate lahenduste klassifikatsioon ja nende lahenduste omadused ja töö põhiprintsiibid. Järgnevates peatükides põhjustatake ühe lahenduse valik ja kirjeldatakse selle algoritmi, implementatsiooni ja testimist. Kokkuvõttes näidati, et vajadusele vastav lahendus leidub, kuigi selle juures eksisteerib mitu raskesti lahendatavat probleemi, mis mõjuvad halvasti programmi jõudlusele.

Selle lähenemise edasiarendusel on mõtet, kui see lahendab loetletud probleemid. Lisaks pole esitatud lahenduse implementatsioon täiuslik — implementeeritud FORTRAN 90 parser saab ainult osast keelesüntaksist aru ja lahendus ei käsitle osa MPI teegi funktsionaalsusest.

Bibliography

[1] DOUG: Domain decomposition on unstructured grids. project home page. `http://www.dougdevel.org/`. 2007-05-06.

[2] R Roman. A survey of checkpoint/restart implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, 2003.

[3] BProc: Beowulf distributed process space home page. `http://bproc.sourceforge.net/`. 2007-03-31.

[4] Berkeley lab checkpoint/restart (BLCR) home page. `http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml`. 2007-03-31.

[5] CRAK: Linux checkpoint/restart as a kernel module. `http://www.ncl.cs.columbia.edu/research/migrate/crak.html`. 2007-05-11.

[6] Arthur Goldberg, Ajei Gopal, Kong Li, Rob Strom, and David F. Bacon. Transparent Recovery of Mach Applications. In *Usenix Mach Workshop*, pages 169–183, 1990.

[7] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of USENIX Winter1995 Technical Conference*, pages 213–224, New Orleans, Louisiana/U.S.A., January 1995.

[8] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra M. R. Kintala. Checkpointing and its applications. In *Symposium on Fault-Tolerant Computing*, pages 22–31, 1995.

[9] Condor: High throughput computing. project home page. `http://www.cs.wisc.edu/condor/`. 2007-05-01.

[10] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed

processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[11] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs, 2003.

[12] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 58–67, Seattle, WA, USA, June 1997.

[13] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. C3: A System for automating Application-Level checkpointing of MPI programs. In *Languages and Compilers for Parallel Computing*, volume Volume 2958/2004, pages 357–373. Springer Berlin / Heidelberg, 2004.

[14] Chung-Chi Jim Li and W. K. Fuchs. CATCH - compiler-assisted techniques for checkpointing. June 1995.

[15] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent advances in checkpoint/recovery systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[16] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[17] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[19] M.J. Hagger. Automatic domain decomposition on unstructured grids (DOUG). *Advances in Computational Mathematics*, 9(3-4):281–310, November 1998.

[20] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. *sc*, 00:38, 2004.

[21] ANTLR: Another tool for language recognition. home page. `http://antlr.org`. 2007-05-09.

[22] PTP: Parallel tool platform. home page. `http://www.eclipse.org/ptp/`. 2007-05-09.

[23] Stringtemplate engine. home page. `http://www.stringtemplate.org/`. 2007-05-09.

[24] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. Technical Report LBNL-60520, Lawrence Berkeley National Laboratory, 2006.

[25] Christopher Krgel and Thomas Toth. An efficient, ip based solution to the 'logical timestamp wrapping' problem.

Appendix A

CD contents

Contents of the CD are following:

- `FortranCPR` — source-to-source compiler that instruments FORTRAN 90 source code
  - `antlr-3.0b4.patch` — fixes to the ANTLR 3.0b4 parser generator
  - `ptp-fortran.patch` — patch to PTP project FORTRAN syntax written fpr ANTLR (checkout on 15th of January 2007)
- `examples` — original and FortranCPR generated source files
  - `qsort` — qsort algorithm used in FORTRAN checkpointer tests
  - `pcg` — conjugate gradient algorithm implementation used in the coordination layer tests
- `thesis` — source and PDF of this paper