View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Technische Universität Dresden: Qucosa

Schedules for Dynamic Bidirectional Simulations on Parallel Computers

DISSERTATION

zur Erlangung des akademischen Grades

Doctor rerum naturalium
(Dr. rer. nat.)

vorgelegt

der Fakultät Mathematik und Naturwissenschaften
der Technischen Universität Dresden

von

Dipl.-Math. Uwe Lehmann

geboren am 25. Mai 1972 in Bad Saarow-Pieskow

Gutachter: Prof. Mark S. Gockenbach
Prof. Andreas Griewank
Prof. Wolfgang E. Nagel
Prof. Tor Sørevik

Eingereicht am: 14. Februar 2003
Tag der Verteidigung: 19. Mai 2003

To Tilmann and Catrin

und ich habe keine zeit mehr, ich stell mich nicht mehr an
in den langen wartenschlagen, wo man sich verkaufen kann
und ich habe keine zeit mehr, ich nehm den handschuh auf
ich laufe um mein leben, und gegen den lebenslauf

Gerhard Gundermann in "keine zeit mehr" (1995)

First of all, I would like to express my gratitude to Prof. Andreas Griewank for his guidance and supervision while working on this thesis. Furthermore, I would like to thank Prof. Wolfgang E. Nagel for giving me the opportunity to work on this subject and for all the support he has given me over the last five years. Special thanks are due to Dr. Andrea Walther for all of the helpful discussions we had and hints and support she gave me.

Moreover, I am indebted to Rachel Lichten, John Shaw and Ellen Smith who helped me to bring this thesis into shape. Also, I thank all of my colleagues at the Centre for High Performance Computing for the nice and pleasant working atmosphere.

On the nonmathematical side, I want to thank my parents very much for all of the lifelong support and help they have given me. Last but not least, my special thanks go to Catrin Lorenz for her patience, understanding and continuous support, without which I would not have been able to write this thesis.

Contents

1 Introduction **9**

2 Reversal Schedules **17**
2.1 Notation, Assumptions and Definitions 17
2.2 Offline Constructed Serial Reversal Schedules 18
2.3 Online Constructed Serial Reversal Schedules 21
2.4 Offline Constructed Parallel Reversal Schedules 23

3 Online Constructed Parallel Reversal Schedules **29**
3.1 Instantaneously Reversible Distributions 30
3.2 Forward Computation for Instantaneous Reversal 35
3.3 Reverse Sweep for Instantaneous Reversal 44
3.4 Extended Instantaneously Reversible Distributions 47
3.5 Properties of Instantaneously Reversible Distributions 54
3.6 An Algebraic View onto Instantaneously Reversible Distributions 60

4 Implementation of Parallel Reversal Schedules **65**
4.1 Programming models 65
4.2 Offline Constructed Parallel Reversal Schedules for Distributed Memory Programming Models 66
4.2.1 Checkpoint Oriented Implementation Approach 67
4.2.2 Process Oriented Implementation Approach 67
4.3 Online Constructed Parallel Reversal Schedules for Distributed Memory Programming Models 73
4.4 TOPAS – User interface 74
4.4.1 Predefined function 74
4.4.2 User defined function 76

5 Numerical Results **79**
5.1 Schedules 79
5.2 Example: Formula One Car Model 80
5.2.1 Car model 82
5.2.2 Cost function 83

5.2.3 The Forward Computation 84
5.2.4 The Reverse Computation 84
5.2.5 The Computation of the Jacobians m_j and n_j 86
5.2.6 Optimisation 86
5.2.7 Results 86
5.3 Example: Lipschitz tracking 89

6 Summary and Outlook **95**
6.1 Summary 95
6.2 Outlook 97

A Glossary **99**
A.1 General Notations 99
A.2 Notations for Car Model Problem 100

List of Figures **101**

Bibliography **103**

Chapter 1

Introduction

The mathematical description of application problems is often done using a nonlinear vector function

$$\vec{F} : \mathbb{R}^n \longrightarrow \mathbb{R}^m, \qquad x \longmapsto \vec{F}(x)$$

which is evaluated by a computer or evaluation function F, or for short function F. For many mathematical methods, some kind of reversal or adjoint $\bar{F}$ of such an evaluation function F is needed. From now on, the term "function F" represents the procedure evaluating the vector function $\vec{F}$. For example, the reversal $\bar{F}$ may be required for optimal control problems solved using adjoints or parameter adaption problems for given computer models. Computing the adjoints of the evaluation function F is related to the reverse mode of algorithmic differentiation (AD), also called automatic or computational differentiation. Using the reverse mode of AD, one needs to provide the reversal of the evaluation function F. Another example for techniques needing the reversal of a computer function is debugging. The program runs until an error appears and is stopped by the debugger. To find out why the program execution fails, the debugger has to check what happened previously.

Difficulties appear in all cases if the reversal of the evaluation function F cannot be easily created. The runtime and the spatial complexity of reversing such an evaluation function F, which is difficult to reverse, can be large. This can be illustrated by the following examples.

Example 1.1 *(Explicit Euler method).* Let an optimal control problem be described by a cost function J and a system of ordinary differential equations f, which are given by

$$J\big(x(1)\big) \longrightarrow \min \qquad \text{such that} \qquad \dot{x} = f(x, u, t) \qquad \text{with} \qquad x(0) = x_0 \ .$$

The time parameter t lies in the time interval $t \in [0, 1]$. The value $x = x(t)$ describes the state of the system at the time t while the value $u = u(t) \in \mathbb{R}$ defines the control done at the time t. This control is bounded, i.e. $u \in [a, b]$ with $a, b \in \mathbb{R}$. The initial value for the state values is given by $x(0) = x_0$. The ODE system is integrated using an equidistant grid of step size h and a given control. For simplification, the explicit Euler scheme is used, which is defined by

$$x_i = x_{i-1} + hf(x_{i-1}, u_{i-1}, t_{i-1})$$

for $i = 0, \ldots, N \subset \mathbb{N}$ and $N = \lceil h^{-1} \rceil$. The time t_i is defined by $t_i = ih$ and the approximated state value is defined by $x_i = x(t_i)$. The control u given at time t_i is denoted by u_i. For many optimisation algorithms one needs to know how the control influences the result, i.e. the sensitivities of the cost function J with respect to u. Hence, one wants to compute the discrete adjoints J_u. These can be obtained by using a backwards method for the explicit Euler scheme (compare [GW01]) given by

$$\lambda_i = \left[I + h f_x(x_i, u_i, t_i) \right]^{\mathrm{T}} \lambda_{i+1}$$

for $i = N, \ldots, 0$. The computation of the local Jacobian f_x needs the results x_i of the explicit Euler scheme, but in the reverse order. There are different ways to get the needed values in reverse order. One obvious way is to store all values x_i during the forward computation, i.e. during the execution of the explicit Euler scheme. This approach leads to a large spatial complexity of the computation. Another obvious way is to recompute all values, which leads to an increase in runtime.

Example 1.2 *(MPEG Decoding).* In order to illustrate the problem once more, a "nonnumerical example" is chosen, namely the decoding of MPEG video streams. MPEG stands for Motion Picture Expert Group. This is a working group of ISO/IEC (International Organisation for Standardisation / International Electrotechnical Commission) in charge of the development of standards for coded representation of digital audio and video [MPG02]. This group established different standards for video compression, depending on the quality and other properties. This example mostly deals with the MPEG-1 standard. The basic idea behind MPEG video compression is to remove spatial redundancy within a video frame and temporal redundancy between temporally neighboured video frames. A frame is a picture displayed on the screen during the motion picture. Within a frame, the spatial redundancy is reduced by using techniques known from the still image compression such as JPEG. To exploit temporal redundancy, one notes that the images in a video stream usually do not change much within small time intervals. Hence, the idea of reducing the temporal redundancy is to encode a video frame based on other video frames temporally close to it. This technique is also called motion-compensation.

When using the MPEG video compression, the motion picture is encoded using three types of frames, namely the I-, the P- and the B-frame. The I-frame is encoded as a single image, with no references to any past or future frame. The P-frame is encoded with a reference to a frame in the past. The reference frame can either be the previous I- or the previous P-frame. The information stored for a P-frame is roughly ten times smaller then the information stored for an I-frame. The third frame type is the B-frame. The reference one needs for the decoding of this frame is the closest I- or P-frame in the past and the closest I- or P-frame in the future. This frame type roughly needs to store just a tenth of the size of the information of the P-frame. The frame dependencies can be summarised by

$$\begin{array}{lcl} dec(I) & \longrightarrow & \text{picture} \\ dec(P, ref(I^-/P^-)) & \longrightarrow & \text{picture} \\ dec(B, ref(I^+/P^+), ref(I^-/P^-)) & \longrightarrow & \text{picture} \ . \end{array}$$

The way a typical frame sequence is stored is shown in Figure 1.1 on the top. The frame order needs not follow a static pattern. The output of the decoded frames can be seen in Figure 1.1

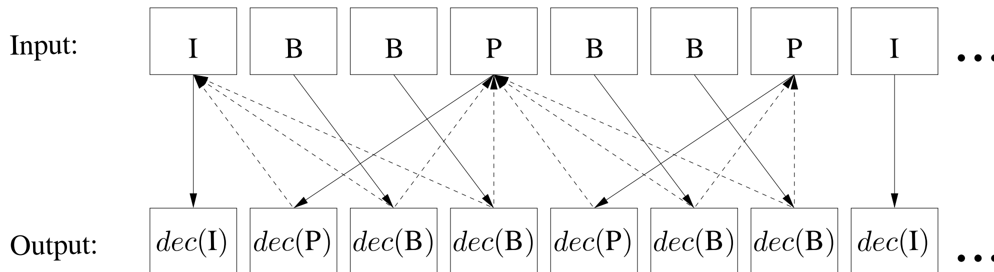


Figure 1.1: Dependencies and temporal order of the frames of a MPEG-decoded video stream.

at the bottom. The solid arrows show how the order changes. The dashed arrows illustrate the dependencies to the reference frames. The maximum number of frames needed to be buffered is three.

So far, the goal is to play the motion picture forward. But what happens, if the video needs to be played backwards? Due to the temporal dependencies, this becomes complicated if the distance between two I-frames increases. If one wants to keep the initial frame rate per second, one must store all P-frame information between two I-frames in order to decode the B-frames. This can enormously increase the memory need of the MPEG-decoder. The approach is later referred to the full-logging approach. Another possibility is to redecode all P-frame information. This will significantly reduce the frame rate and the reverse motion picture will run in slow motion. In order to avoid the slow down one possible solution is redecoding of the P-frames in parallel. In the worst case the procedure leads to a need in computing power proportional to the number of P-frames between two I-frames times the computing power of one MPEG-decoder. One way out is to just remember a few P-frames and redecode the missing one. Such alternative approaches will be discussed in this work.

The two examples above both have in common that the considered vector function $\vec{F} : \mathbb{R}^n \to \mathbb{R}^m$, with $x \mapsto y = \vec{F}(x)$ is an iteration or evolutionary system. That means that the function can be divided into vector subfunctions $\vec{F}_i$ as shown in Figure 1.2. The evaluation

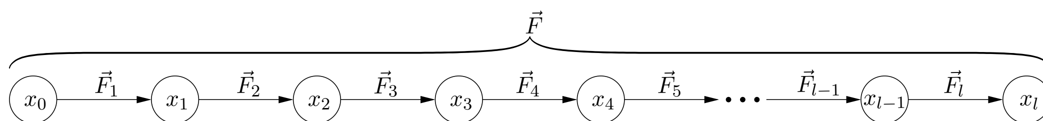


Figure 1.2: Evaluation of vector function $\vec{F}$.

functions F_i of these vector subfunctions $\vec{F}_i$ are called physical steps or advancing steps.

Examples of applications requiring the reversal of such a vector function $\vec{F}$ are programs calculating the adjoints of evaluation function F, programs adapting parameters for a given mathematical model (e.g. [HL$^+$00]) and program debuggers. More examples can be found in

weather or ocean modelling (see e.g. [HHG02] and [RLG98]), or production optimisation (see e.g. [KGW00]).

In [HHG02], the adjoint of the ocean simulation is computed to minimise the misfit between the computer model and the observation. To solve a parameter adaption problem like this, a huge amount of memory is usually required. In order to handle the memory requirement of the program in [HHG02], a serial checkpointing strategy is used.

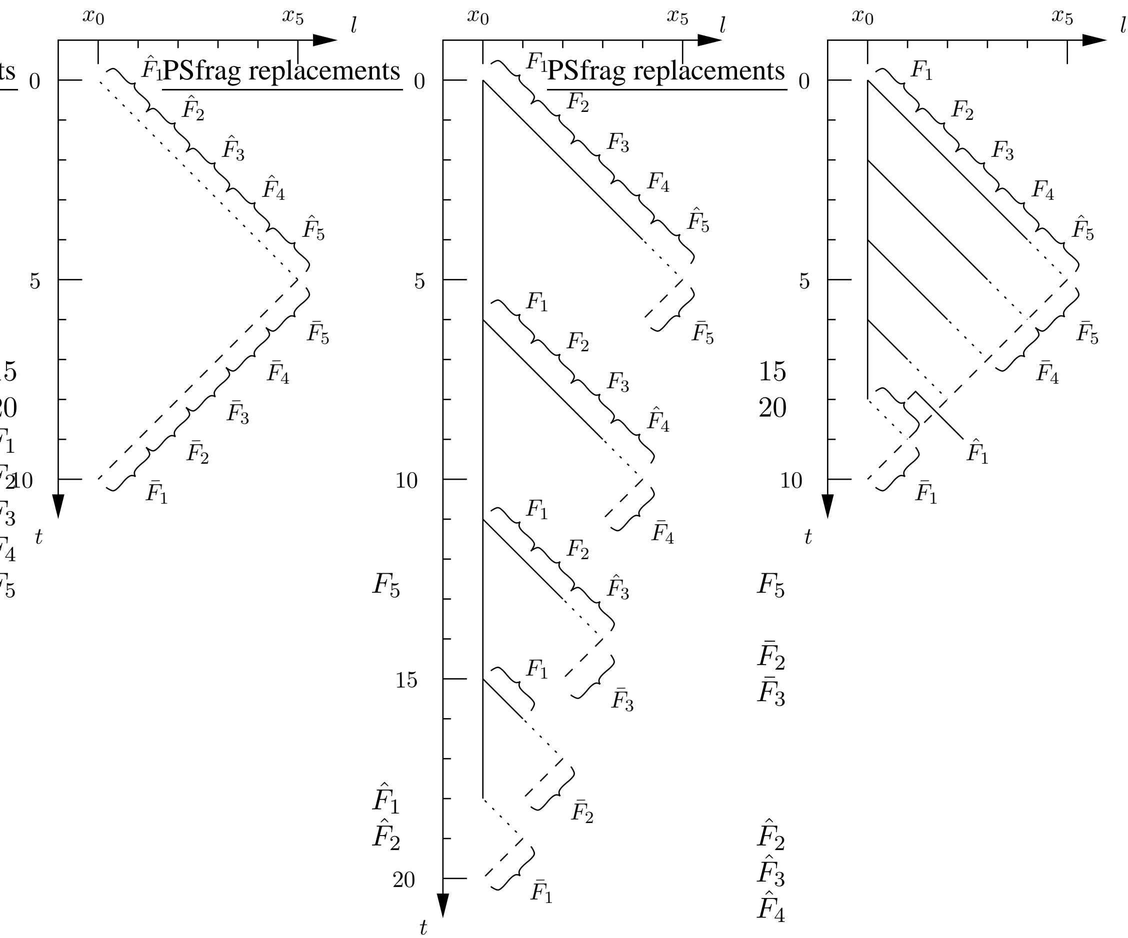
The parameters to be adapted in [HL$^+$00] are the size parameters of a frame used in an injection moulding machine. The resulting nonlinear optimisation problem is solved by sequential quadratic programming. The gradient information needed for this algorithm is obtained by using the reverse mode of AD.

An example of a debugging problem can be found in [RdB99]. This article describes how to search for the reason for nondeterministic behaviour in a parallel program. During the execution of the parallel program, all potential nondeterministic actions, such as system calls or unsynchronised memory access are recorded. During the analysis, this data is read in order to get the same program behaviour.

Discussions dealing with the reversal of computer programs in general can be found in [Sne93]. For example, in Chapter 11, van Snepscheut discusses problems occurring due to data dependencies or problems occurring if one wants to reverse loops. In [Ben73] a first effort estimate was described. Bennett has indicated that the spatial complexity to reverse a function F may grow logarithmically.

As described in the examples, there are two obvious ways to reverse the execution of the function F. The first scheme, illustrated in Figure 1.3(a), writes a complete execution log (dotted line) to an appropriated data structure called trace. The function writing the execution log is called $\hat{F}$ and the associated subfunctions are called $\hat{F}_i$. During the reversal (dashed line in Figure 1.3(a)), this data structure is read backwards. The reverse function is called $\bar{F}$, consisting of the subfunctions $\bar{F}_i$, which is the reversal associated with the subfunction F_i. The second scheme is the recomputation (solid line) of all data required for the function reversal (Figure 1.3(b)). The first case leads to an enormous spatial complexity while the second approach increases the runtime in terms of wall clock time. This increase can be avoided, if the recomputation is carried out by additional processors in parallel (Figure 1.3(c)). The first two cases are examples of a serial reversal of the function F, while the latter case is an example of the parallel reversal of the evaluation function F.

A more general estimate of [Ben73] for the runtime and the spatial complexity, especially for the reverse mode of AD, is given in [Gri92]. The first approach to establish these results for AD in practice was done in [Ben96], [Ben95], their continuation in [Bra98], as well as in [RDG93]. The spatial complexity of the problems discussed in these references is reduced by storing the state values in temporally equidistant steps (called checkpoints) and recomputing the missing state values between two stored checkpoints in parallel. Hence, the increase in runtime is converted into an increase in computing power, i.e the additional runtime is converted into an increase in the number of processes used at the same time. The first optimal results and bounds for the complexity of a program reversal were given in [Wal99] and [WG01]. In [Wal99] two major cases were distinguished. For the first case, it is assumed that only one processor is available, which leads to serial reversal schedules. In the second case, it is assumed that the



(a) Full-logging approach leads to an increase in spatial complexity

(b) Complete recomputation in serial leads to an increase in runtime

(c) Complete recomputation in parallel requires an increase in number of processes

Figure 1.3: The reversal of a function F consisting of 5 steps.

reversal is carried out on a multi-processor machine, which yields parallel reversal schedules. In [Wal99] it is assumed that the evaluation function F, the function that one wants to reverse, can be divided into N subfunction F_i as shown in Example 1.1. Every execution of such a subfunction yields a new state of the system. A reversal schedule specifies which states have to be stored and when, as well as which states have to be recomputed and when the recomputation starts at the stored states. In the used termonology parallel reversal schedule does not specify which process or which processor on a parallel computer has to perform the tasks.

The minimum time needed for the function reversal using a serial reversal schedule is the time for a complete logging of the evaluation function F, plus the time of the reverse computation. The use of a serial reversal schedule increases the runtime due to recomputation done in a sequential manner. In [Wal99] this increase is minimised. More specifically, two cases are investigated. The first case is given if each subfunction F_i requires the same execution time. The second case assumes that the execution of the subfunctions needs different amounts of time. This first property is called uniform step cost while the second property is called nonuniform step cost. Serial reversal schedules for nonuniform step costs were developed further in [Ste02]. For both kinds of step costs (uniform and nonuniform), cheap algorithms to create serial reversal schedules are introduced in [Ste02].

In this thesis parallel reversal schedules will be discussed and developed. The construction of parallel reversal schedules in [Wal99] will be extended for online application. That means that the step number l does not need to be known priori and can in fact be modified repeatedly. The step number l is the number of subfunctions F_i evaluated during one evaluation of F and defines a measurement for the length of the evolutionary system. For the construction of a parallel reversal schedule developed in [Wal99], the length of the evolutionary system must be known priori. This construction of parallel reversal schedules is called offline, while the construction of parallel reversal schedules developed in this thesis is called online. The goal is to use minimal resources for the complete program reversal. Thereby, the overall complexity will stay within the bounds established in [Wal99].

A parallel reversal schedule starts with the evaluation of the function F. During the evaluation certain intermediated states are stored into checkpoints. The distribution of checkpoints, after a subfunction is evaluated, will be explored. The the intermediate state information which was currently worked on and the stored checkpoints together, are called resources. One can now give a description of resource distributions, which are instantaneously reversible. First, it will be defined as to what it means for a resource distribution to be instantaneously reversible. The definition of an instantaneously reversible resource distribution depends on the current step number l, i.e. the state x_l with the largest state number. Furthermore an algorithm is discussed, which changes an instantaneously reversible resource distribution for the step number l to an instantaneously reversible resource distribution for the step number $l+1$. The needed time for this change is the time that one subfunction F_i takes to be executed. If the algorithm is used starting from the step number $l = 0$ up to an arbitrary step number, it defines the forward computation for parallel reversal schedules constructed online. Later on in the thesis, the description of instantaneously reversible resource distributions will be extended, such that the resource distributions of reverse computation, i.e. the actual function reversal, can also be covered. This extension is done by generalising the position of a resource, i.e. the number of the

intermediated state of F, to an interval, containing possible resource positions. This gives the opportunity to define an algorithm, which converts a given instantaneously reversible resource distribution (defined by using the intervals), either to an instantaneously reversible resource distribution for the step number $l+1$, or to an instantaneously reversible resource distribution for the step number $l-1$ on a parallel computer. Thus, one can use this algorithm to switch between the forward computation and the reverse computation at any time. The occurring resource distribution are all instantaneously reversible. Once again the required time for the change of the instantaneously reversible resource distribution in any direction is the time that one subfunction F_i needs in order to be executed. Therefore, the resulting schedules can be used to run a simulation in both directions (forward and reverse) at any time, i.e. one obtains schedules for dynamic bidirectional simulations.

Within the next chapter, reversal schedules will introduced more precisely and known theoretical results will be reviewed. The different types of schedules (for uniform and nonuniform step cost, offline and online construction, serial and parallel) will therefore be discussed. In Chapter 3 the theory of the offline construction of parallel reversal schedules will be extended to online construction and generalised, such that schedules for both the function evaluation and the function reversal can be described. Thereby one looks at resource distributions which are instantaneously reversible. This leads to a description of how a restart of the function evaluation can be done without the loss of instantaneous reversibility. Chapter 4 explains implementation issues for parallel reversal schedules. It will be discussed, under which circumstances the theoretical optimality can be reached, if the implementation effort is small. Chapter 5 will represent the results obtained by using an implementation for the online construction and execution of reversal schedules on parallel computers. First, the runtime behaviour of a reversal schedule with restart is shown. Then, the implemented reversal schedules are applied to the parallel computation of the adjoints of an optimal control problem, namely the steering of a Formula One car. The last chapter, Chapter 6, summarises the thesis and gives an outlook on further problems.

Sfrag replacements

Chapter 2

Reversal Schedules

2.1 Notation, Assumptions and Definitions

It is supposed that the evaluation of the vector function $\vec{F} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ can be divided into l parts $\vec{F}_{i+1} : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_{i+1}}$ with $x_i \mapsto x_{i+1} = \vec{F}_{i+1}(x_i)$. The evaluation functions F_i of the vector subfunctions $\vec{F}_i$, are called advancing steps, or just steps for short. From now on the term "function F" denotes the evaluation function, in the sense of a computer or C function of the mathematical function or mapping $\vec{F}$. The arguments x_i and the results x_{i+1} of the functions F_{i+1} are called the state or state vector. The index i is called the state counter. It is assumed throughout this thesis that all state vectors have the same size, i.e. $n_i = n_{i+1}$ for all i, or at least that there exists an upper bound n such that $n_i \leq n$ and $n_i \approx n$ for all i. If an intermediate state is retained in memory it is called a checkpoint. The goal is to reverse the execution of the function F. The reverse function is denoted by $\bar{F}$. Difficulties occur if the original function F is not invertible, in the sense that the original subfunctions F_i are ill conditioned or difficult to reverse. Another problem is that the results of the corresponding forward computation are required for the reverse computation as in Example 1.1. The information flow which are as-

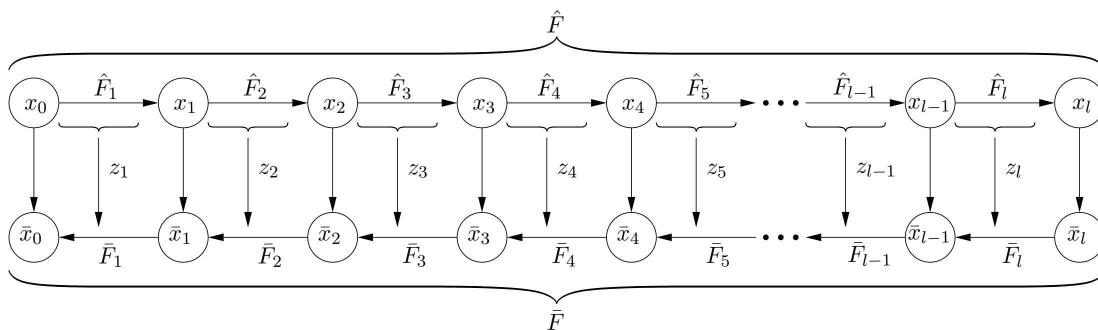


Figure 2.1: The evaluation and reversal of a function F.

sumed throughout this thesis are illustrated in Figure 2.1. The function $\bar{F}_i$ is the corresponding reverse function to the advancing step F_i. These reverse functions are also called reversing

steps. The arguments of such a function $\bar{F}_i$ are the result $\bar{x}_i$ of the previous reversing step $\bar{F}_{i+1}$, the intermediate state x_{i-1} and possibly some more information or other intermediate values, which are obtained from the execution of the corresponding advancing step F_i. An advancing step that does the additional collecting of required data is called preparing step $\hat{F}_i$. The data obtained while executing the preparing step $\hat{F}_i$ is denoted by z_i. Thus, one can formalise the definition of a reverse step as

$$\bar{x}_i = \bar{F}_{i+1}\big(x_i, z_{i+1}, \bar{x}_{i+1}\big) \ .$$

The variables $\bar{x}_i$ are assumed to have the same dimension, or the dimension that is uniformly bounded for all i. The same is assumed for the dimension of the variables z_i. Of course the dimensions of $\bar{x}_i$ and z_i can be different, which is usually the case.

Since the information z_i needed for the reverse computation is required in a reverse order to that in which it is obtained during the forward computation, the data structure holding this information can be realised as a stack. This data structure is usually called a trace.

In order to obtain information about the runtime behaviour of the introduced functions, times τ_i, $\bar{\tau}_i$ and $\hat{\tau}_i$ are used. These values denote the computation time of functions F_i, $\hat{F}_i$ and $\bar{F}_i$ respectively. Two scenarios have to be considered.

The first case assumes that all advancing steps F_i require the same time τ. An equivalent assumption is made for the preparing and the reversing steps. Such a function is called an evolutionary function with a uniform step cost. To simplify, τ can be scaled to one w.l.o.g.. The second case is called an evolutionary function with a nonuniform step cost. Here, the execution times for the advancing steps F_i vary and are given explicitly by a sequence $\langle\tau\rangle = \langle\tau_1, \tau_2, \ldots, \tau_{l-1}, \tau_l, 0, 0, \ldots\rangle$. Again, an equivalent assumption is made for the preparing steps $\hat{F}_i$ and the reversing steps $\bar{F}_i$.

2.2 Offline Constructed Serial Reversal Schedules

For this section it is assumed that only one processor is available for the reversal. The simple way to realise the reversal of function F is to store all information during the forward computation. The reverse computation is carried out by simply reading the trace information backwards and carrying out the reverse computation. This kind of approach is called the full-logging approach and is illustrated in Figure 2.2. The dashed line going from the middle down to the bottom lefthand side represents the computed reversing steps, while the dotted line from the top left corner to the middle represents the preparing steps. All computation done until the final state x_l is reached, is called forward computation (light grey area in Figure 2.2). All computation performed during the reversing step $\bar{F}_i$ is called reverse computation (dark grey area in Figure 2.2).

The l-axis shows the current intermediate state reached, i.e. the state counter. The vertical t-axis represents the normalised time. As already stated, the drawback of this algorithm is the memory requirement. This approach yields a runtime minimal program reversal on a single processor machine. For every step which has to be reversed, the memory of size x_i and z_i is needed. Hence, the memory requirement for the full-logging approach, as illustrated in

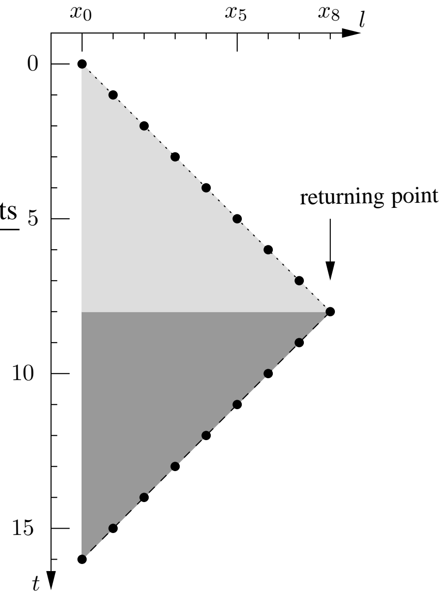


Figure 2.2: Full-logging approach to calculate the reversal of function F.

Figure 2.2, is l times the size of z_i and $l + 1$ times the size of x_i at the returning point. Thus, the memory requirement is proportional to the length of the computation.

The use of serial reversal schedules allows a reduction in the memory requirement to a given limit. The drawback is an increase in computing time.

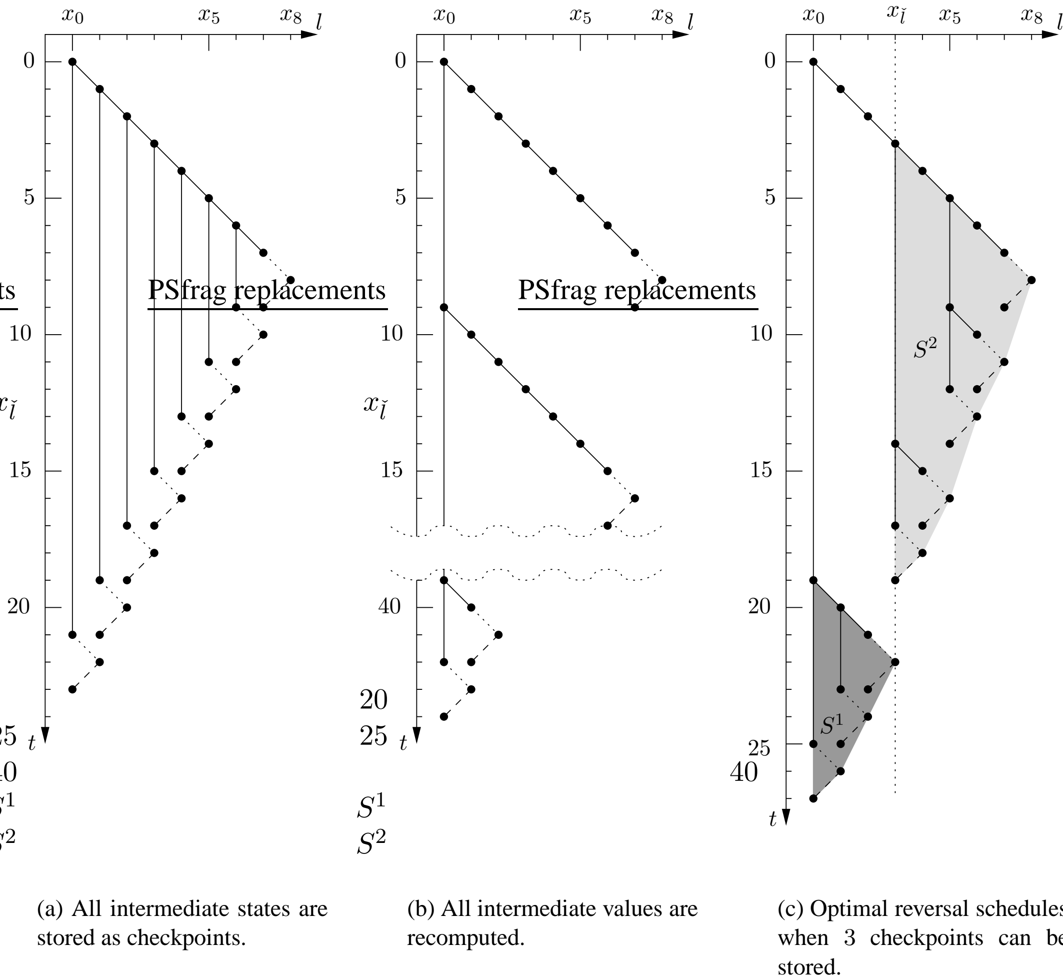
The first improvement can be seen in the approach illustrated in Figure 2.3(a). During the forward computation (illustrated by the solid line from the top lefthand side going down to the right), only the intermediate states are stored. The time (one of these checkpoints is kept in memory) is shown by the vertical lines within the schedules.

The trace information z_i is obtained by recomputing the step F_i and logging the data, i.e. the preparing step $\hat{F}_i$ is carried out. The computation time increases by $l - 1$ times the time required for one advancing step F_i. The memory requirement still depends on the number of time steps, however, because after each advancing step, the intermediate state x_i has to be stored.

Another solution is shown in Figure 2.3(b). Here, no intermediate state is stored except x_0, and all the information needed is recomputed. The constant and minimal memory requirement means that a lot of time is required. Hence, one spends about $l/4$ times the amount of time one advancing step needs more than the full-logging approach needs. Runtime optimal solutions for a given number of intermediate states that can be stored (i.e. checkpoints) are discussed in [Wal99, GW00, Gri00, Ste02].

To obtain a serial reversal schedule, more notation is required. A reversal schedule S consists of a chain of instructions such as:

- carry out n advancing steps F_i for $i = j, j + 1, \ldots, j + n - 1$,



(a) All intermediate states are stored as checkpoints.

(b) All intermediate values are recomputed.

(c) Optimal reversal schedules when 3 checkpoints can be stored.

Figure 2.3: Serial reversal schedules for step number $l = 8$.

- set a checkpoint, i.e. copy the actual state x_i into a checkpoint,
- carry out the returning step U_i, or
- copy a checkpoint and restart the forward computation using the copy of the checkpoint.

The preparing step $\hat{F}_i$ and the reversing step $\hat{F}_i$ are merged into the returning step U_i. The number l denotes the currently maximal state counter during the execution of the schedules S.

From now on, it is assumed that all advancing steps need the same time to be executed. In order to judge the quality of a schedule S, the total cost of an execution of the schedule S is introduced. The cost function denotes the sum of the amount of time the executed advancing steps F_i need. During the execution of a reversal schedule, certain individual steps are executed more than once. In order to obtain an optimal serial reversal schedule, the total cost of a

schedule must be minimised. There can be more than one schedule having the property of minimal total execution cost. In order to select a schedule out of the number of schedules with minimal total cost a secondary objective function may be used. One possibility for such a secondary objective is the number of intermediate states which are copied and used for a restart during the execution of a schedule S. Still, there can be more than one schedule S satisfying both optimality criteria.

One main characteristic of an optimal reversal schedule is checkpoint persistence. Checkpoint persistence means that if a intermediated state x_i is copied into j^{th} checkpoint, then this checkpoint will only be deleted after carrying out the returning step U_i, if $l \leq i$. Furthermore, no action takes place concerning the intermediate state x_k for $k \in [0,..,i]$, between setting the checkpoint and starting the returning step U_i. The proof of the checkpoint persistence of an reversal schedule can be found in [Wal99], [Gri00] or [Ste02]. The major conclusion of checkpoint persistence is that any optimal serial reversal schedule can be decomposed into two optimal subschedules: One named S^1, concerning all computation having state counters between 0 and $\check{l}$, and the other subschedule named S^2, concerning all computation having state counters between $\check{l}$ and l

$$S = S^2 + S^1 \quad . \tag{2.1}$$

The decomposition of a serial reversal schedule is illustrated in Figure 2.3(c).

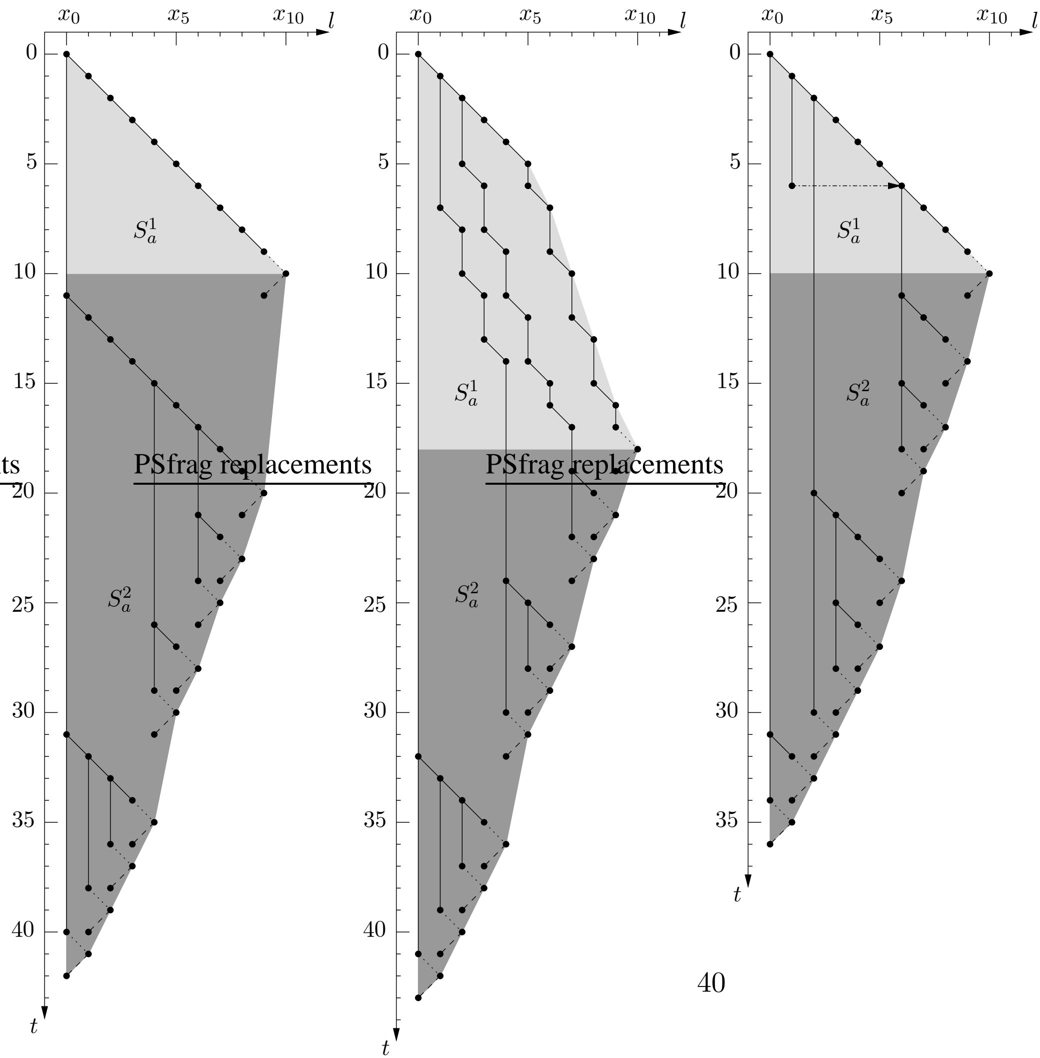
Serial reversal schedules for nonuniform step cost, i.e. the subfunctions F_i of the function F, do not need the same computation time. This is discussed in [Wal99, Ste02].

2.3 Online Constructed Serial Reversal Schedules

Until now it was assumed for all schedules that the number of steps l one wants to reverse was known beforehand. In reality, however, there are cases in which this is not true. An example is an iteration using a reversal criterion or stopping criterion. Before the online construction of a serial reversal schedule S_a can start, the number of checkpoints c one can store has to be known. The computation is carried out in two phases. The first phase is the forward computation. This phase lasts as long as the reversal criterion is not satisfied. This phase is also called the adaption phase and S_a^1 denotes the corresponding schedule. The second phase is the computation after the number of steps is known. It is called the reversal phase, and the schedule is named S_a^2. Since the number of steps is known during the second phase, offline constructed schedules must be used.

There are different methods to construct a reversal schedule when l is unknown. The easiest way is to carry out a normal function evaluation until the reverse computation is initiated, stopping at the intermediate state x_l, and then run a reversal schedule for l steps. This approach of "test computing" is shown in Figure 2.4(a).

Figure 2.4(b) shows another approach. All c checkpoints are set within the first c computation step. When the step number becomes larger then the checkpoint number c, and the reversal criterion is not yet satisfied, then at the end of every advancing step F_i it must be checked, which checkpoint has to be moved to the next intermediate state. This keeps the checkpoint



(a) Computing until reversal criterion is satisfied, then using a serial reversal schedule.

(b) Changing of the checkpoint position until the reverse computation is initiated.

(c) Deleting and reusing of the checkpoint position.

Figure 2.4: Online constructed serial reversal schedules (reversal criterion is satisfied at $l = 10$) for three checkpoints.

positions in such a way as it would be in an offline constructed optimal reversal schedules. In this case, additional computing power is required to keep the checkpoints distributed like they are distributed in an offline constructed optimal reversal schedule.

As it can be seen in Figure 2.4(c), there is a better solution which needs less time than the two previous approaches (Figure 2.4(a) and 2.4(b)). The procedure starts with the computation of c steps, and every intermediate state computed is stored in a checkpoint at the end of each advancing step. For step numbers larger than c, it is checked after each step, whether or not one checkpoint can be deleted for storing the current state, in order to obtain a smaller total execution cost than for the execution of the schedule using the current checkpoint distribution. This is illustrated by the dotted arrow in Figure 2.4(c). Once a checkpoint is stored it will not be moved if using this approach.

In [Ste02], numerical tests are carried out to compare the total execution cost of the online and offline constructed serial reversal schedules for a uniform, as well as for a nonuniform step cost. The results show a tolerance of a maximum of twenty percent between the total computation cost of an online constructed reversal schedule, and the total computation cost of an offline constructed reversal schedule for both kind of step costs.

The results, described above, are used in the implementation of the software package REVOLVE and REVOLVE++. Both include procedures acting as a controller when evaluating a program reversal. By the return value of the function `revolve(..)` the user program is told to perform one of the following actions:

- `advance`: carry out the computation of the advancing step F_i,
- `takeshot`: copy the actual state into a checkpoint,
- `restore`: read a checkpoint and
- `firstturn`: carry out the first return, i.e. carry out a preparing step $\hat{F}_i$, initialise the reverse value $\bar{x}$, and make the first reversing step $\bar{F}_i$.
- `youturn`: carry out a return step U_i, i.e. carry out a preparing step $\hat{F}_i$ and a reversing step $\bar{F}_i$.

The user has to implement corresponding functions and routines adapted to the specific problem. Two further return values of the revolve routines are `terminate` and `error`.

2.4 Offline Constructed Parallel Reversal Schedules

For all schedules discussed so far, the computation runtime has always been larger than the smallest possible runtime, even if an offline constructed optimal serial reversal schedule was used. The aim of parallel reversal schedules is to achieve a minimal runtime by using additional processes for the recomputation of the intermediate states. Since the time one preparing step $\hat{F}_i$ requires to be executed is usually larger than the time for one normal advancing step F_i, the minimal time $T(S(l))$ a parallel reversal schedule S needs, is smaller than for the full-logging approach. It can also be given by the time needed for $l-1$ advancing steps F_i, plus the time

needed for one preparing step $\hat{F}_i$, plus the time needed for l corresponding reversing steps $\bar{F}_i$ (see Figure 2.5), i.e.

$$T(S(l)) = (l-1) + \hat{\tau} + l\bar{\tau} = (\bar{\tau}+1)l + \hat{\tau} - 1$$

PSfrag replacements
for $\bar{\tau} = 1$. For comparison the runtime for the full-logging approach is given by $l\bar{\tau} + l\hat{\tau}$. Provided a parallel reversal schedule needs the minimum time to execute, the goal is to minimise the number of checkpoints and processes needed for the reversal. The number of checkpoints and the number of processes are added to the number of resources.

One example of what a parallel reversal schedule can look like is given in Figure 2.5. Here, the bisection strategy was applied for the reversal of $l = 8$ steps. Furthermore, $\hat{\tau} = \bar{\tau} = 1$ was

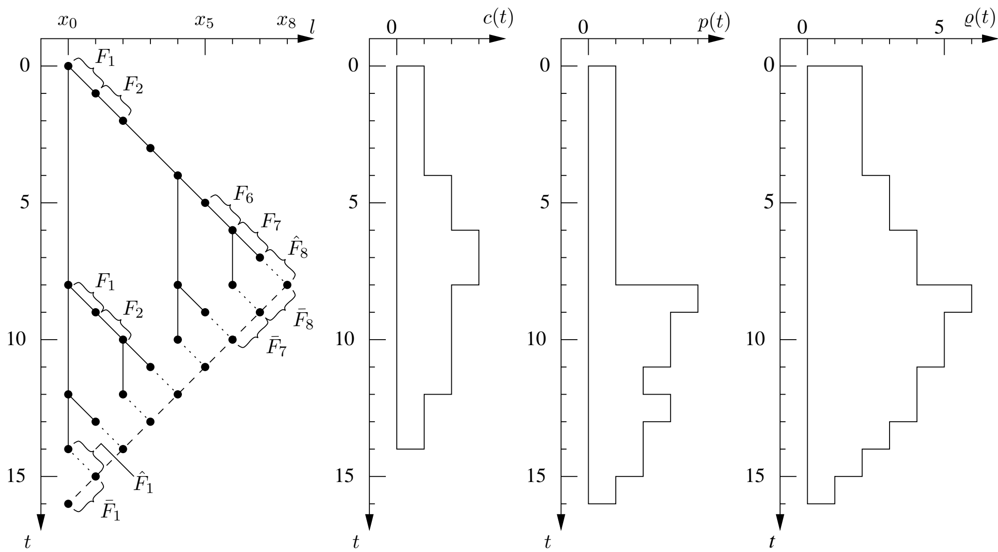


Figure 2.5: A schedule for $l = 8$ steps with $\hat{\tau} = \bar{\tau} = 1$ using the bisection strategy.

assumed. As can be seen in the diagram the furthest left in Figure 2.5, one process copies the initial state into a checkpoint, carries out the forward computation up to the 4th state and copies it into a checkpoint. After that, the process continues up to x_7, thereby copying the state x_6 into a checkpoint. Then it carries out the preparing step up to the 8th intermediate state and the first reversal afterwards. The checkpoints are always set at the midpoint of the subrange that remains for computation. At the time $t = 8$ (when the first reversing step starts), three other processes must start: One process carrying out the forward computation starts from state x_0, one process starts the forward computation from state x_4 and one starts a preparing step from state x_6. At the time $t = 10$, the preparing step starting at state x_4 must be carried out. Furthermore, one process must delete the checkpoint holding the state x_4 and store the intermediate state x_2. The checkpoint at state x_2 is deleted at the time $t = 12$.

During the time between $t = 8$ and $t = 16$, one process carries out the reversing steps. The computation stops when the state count 0 is reached by a reversing process. More about

implementation strategies and how to assign the task to available processors can be found in Chapter 4.

The three remaining diagrams in Figure 2.5 are called resource profiles. These diagrams show how many resources are used at which time. The first resource profile shows the use of checkpoints $c(t)$, the second profile the need of processes $p(t)$ and the third resource profile shows the sum of checkpoints and processes $\varrho(t) = c(t) + p(t)$, i.e. the resource use over the time.

A slightly more general approach was developed in [Ben96]. Here the checkpoints are uniformly distributed. This means that the checkpoints are recursively distributed, using a fixed ratio between the last checkpoint set and the final state x_l. The bisection strategy described above uses the ratio 0.5. Implementations of this approach were discussed in [Ben95] and [Bra98].

For the serial case, one knows from [Gri92] that the resource need (runtime and checkpoints) grows logarithmically with the length of the computational graph of the function F, i.e. the resource need grows logarithmically with the number of steps l. For the parallel case the increase of the resource need also depends logarithmically on the length of the computational graph of the function F as stated in [Ben96]. There, the logarithmic dependency is achieved using the uniformly distributed checkpoint strategy.

The question which must be answered, however, is what is the maximum number of steps which can be reversed in minimal time for a given number of resources. The answer is given and proved in [Wal99] or [WG01]. There, the recursion formula for $\hat{\tau} = 1$ depending on $\bar{\tau}$ is given by

$$l_\varrho = \begin{cases} \varrho & \text{if} \quad \varrho < 2 + \dfrac{1}{\bar{\tau}} \\ l_{\varrho-1} + \bar{\tau}\, l_{\varrho-2} & \text{else} \quad . \end{cases} \tag{2.2}$$

For $\bar{\tau} = 1$, the construction of an optimal parallel reversal schedule which satisfies (2.2) is similar to the construction of an optimal serial reversal schedule. Equivalent to the construction of an optimal serial reversal schedule, the schedule S^0 consists of two subschedules S_1^1 and S_2^1 (compare the two diagrams on the right in Figure 2.6). One subschedule on the far right corresponds to the step number $l_{\varrho-1}$, while the other subschedule corresponds to the step number $l_{\varrho-2}$. The resulting schedules are further divided into three subschedules S_2^2, S_3^2 and S_4^2 connected to the step numbers $l_{\varrho-2}$, $l_{\varrho-3}$ and $l_{\varrho-4}$, respectively. This is carried out recursively, until the trivial subschedule length one is reached. The resource profiles are constructed equivalently. As shown in Figure 2.7, the resource profile is composed of the resource profiles of the two subschedules plus the resources needed for the startup phase (dark grey areas). Again, this is carried out recursively, until only subschedules of a step size $l = 1$ remain.

Using $\bar{\tau} = 1$ in formula (2.2) yields the definition of the ϱth Fibonacci number. Therefore, using the results in [Knu97] and [HP98], an explicit estimate for the maximum number of steps which can be reversed in minimal time using ϱ resources can be given by

$$l_\varrho \sim \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{\varrho+1}$$

PSfrag replacements

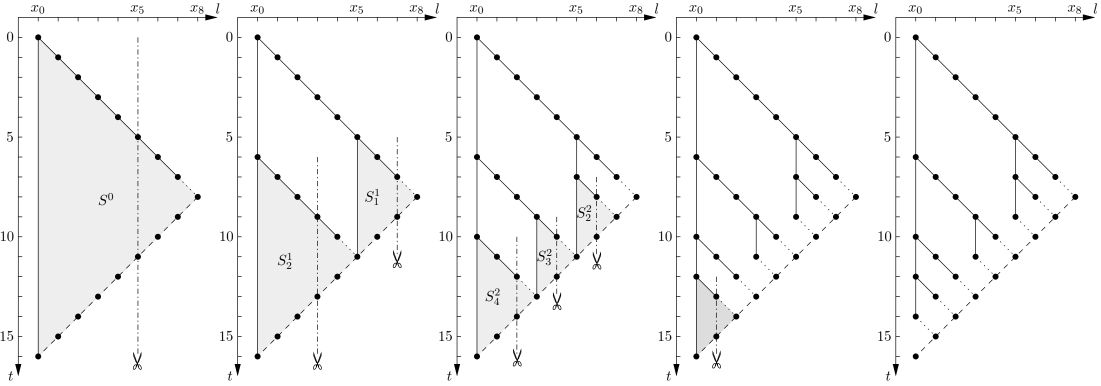


Figure 2.6: Recursive construction of an optimal schedule for $l = 8$ steps and $\hat{\tau} = \bar{\tau} = 1$.

or for general $\bar{\tau}$ by

$$l_\varrho \sim \frac{1}{2}\left(1 + \frac{3}{\sqrt{1+4\bar{\tau}}}\right)\left(\frac{1+\sqrt{1+4\bar{\tau}}}{2}\right)^{\varrho-1} .$$

This formula clarifies the exponential growth of the number of steps l, which can be reversed in minimal time using a given number of resources. Using parallel computers, one question is, how many processes are needed to solve this problem. If possible, it should be known before the program is started. The number of processes p needed is defined by the maximum value in the process resource profile, i.e. $p = \max_t\{p(t) \,|\, 0 \le t \le T(S(l))\}$, and can be bounded by

$$p \le \left\lceil \frac{\varrho+1}{2} \right\rceil \tag{2.3}$$

as shown in [Wal99]. Hence, roughly half of the resources need to be processes, if an offline constructed parallel reversal schedule is executed.

Sfrag replacements

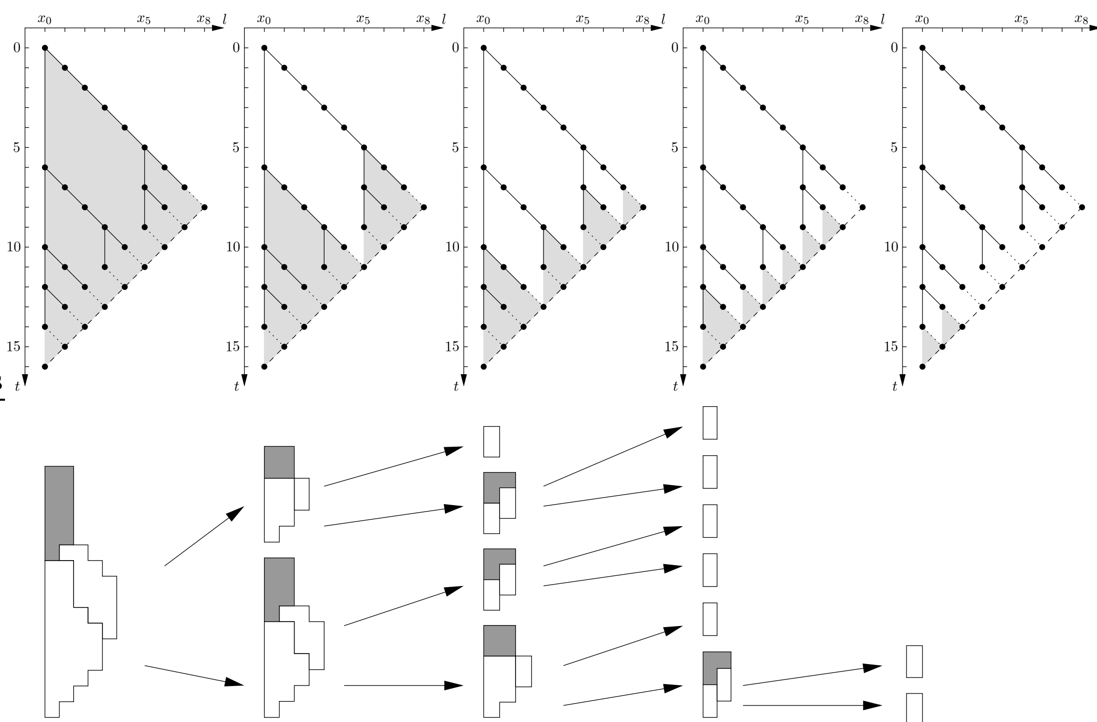


Figure 2.7: Recursive construction of the resource profi le for an optimal schedule for $l = 8$ steps and $\hat{\tau} = \bar{\tau} = 1$.

Chapter 3

Online Constructed Parallel Reversal Schedules

ich fange immer wieder von vorne an
alle sieben jahre kommt ein neuer mann
alle sieben jahre oder irgendwann
drum fang ich immer wieder von vorne an
Pension Volkmann in "von vorn" (1988)

For the function F considered so far, one knows the number of steps l, before the evaluation starts. Hence, the length of the computation is known beforehand. But for many applications, this information is not given. An example of such an application is a function evaluation based on an iteration, where a stopping criterion is checked after each iteration step. Since the construction of the parallel reversal schedules described in Chapter 2 needs the information about the number of steps l, parallel reversal schedules which can be constructed online have to be developed. For the development of such a schedule one has to construct instantaneously reversible resource distributions first. A resource distribution $\mathfrak{r}(l)$ is the set of all intermediated states available at the end of the lth iteration during the execution of a reversal schedule. Thereby, the intermediate state information with the largest index defines the step number of the resource distribution. The step number is named l. The construction of such an instantaneously reversible resource distribution is possible for any arbitrary given number l. Furthermore, a way should be found to change a given resource distribution for the step number l to a resource distribution for the step number $l + 1$, without loss of the instantaneous reversibility.

Assuming the instantaneous reversibility of a resource distribution, an appropriate reverse computation has to be found. The offline construction of a schedule for the reverse computation can be used to extend the definition of instantaneously reversible distributions. The online construction of a schedule for the forward computation, together with the offline construction of a schedule for the reverse computation, yield a parallel reversal schedule constructed online.

3.1 Instantaneously Reversible Distributions

From now on, only the case $\hat{\tau} = \bar{\tau} = 1$ is considered. From (2.2), it follows that n resources are needed for the reversal of φ_n steps. From now on it is assumed that φ_i denotes that i^{th} Fibonacci number with the index shifted by one, i.e. the sequence starts with $\varphi_1 = 1$ and $\varphi_2 = 2$. It is assumed that the intermediate state x_l is reached during a forward computation, i.e. the number of steps is given by l. Further, it is assumed that $l \in [\varphi_n, \varphi_{n+1}) \subset \mathbb{N}$. The interval $[n_1, n_2)$ denotes the set of natural numbers defined by

$$[n_1, n_2) := \{n \in \mathbb{N} \mid n_1 \leq n \ \wedge \ n < n_2 \ \wedge \ n_1, n_2 \in \mathbb{N}\} \ .$$

In order to distribute the other intermediate states x_{r_i} for $0 \leq i < n$, or simply to distribute the resource positions r_i, a sequence of intervals $[a_i, b_i)$ for $i \in [0, n-1)$ is constructed by $[a_0, b_0) = [\varphi_n, \varphi_{n+1})$ and

$$(3.1) \qquad [a_i, b_i) = \begin{cases} [a_{i-1}, b_{i-1}) & \text{if } a_{i-1} = \tilde{b}_{i-1} \ , \\ [a_{i-1}, \tilde{b}_{i-1}) & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \ \wedge \ l < \tilde{b}_{i-1} \ , \\ [\tilde{b}_{i-1}, b_{i-1}) & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \ \wedge \ l \geq \tilde{b}_{i-1} \ . \end{cases}$$

Here and throughout, the value $\tilde{b}_{i-1}$ is defined by $b_{i-1} - \varphi_{n-i-1}$. For all intervals $[a_i, b_i)$, one obtains that $l \in [a_i, b_i)$. Now the resource positions r_i of the resource distribution $\mathfrak{r}(l) := \{r_i \mid i \in [0, n)\}$ are chosen as

$$(3.2) \qquad \begin{aligned} r_0 &= 0 \ , \\ r_i &= \begin{cases} r_{i-1} + \varphi_{n-i} & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \ \wedge \ l \geq \tilde{b}_{i-1} \ , \\ r_{i-1} + \varphi_{n-i-1} & \text{else} \ , \end{cases} \\ r_{n-1} &= l \ . \end{aligned}$$

Definition 3.1 (formula for the resource distribution). The interval definition (3.1) and the position definition (3.2) together are called formula for the resource distribution.

Example 3.1. It is assumed that the intermediate state x_{31} is reached, i.e. $l = 31$. Because of $31 \in [21, 34) = [\varphi_7, \varphi_8)$, the number of resources needed equals 7. Hence, the resource distribution can be computed by

$$\begin{array}{llll} i = 0: & & [a_0, b_0) = [21, 34), & r_0 = 0 \\ i = 1: & \tilde{b}_0 = 34 - \varphi_5 = 26, & [a_1, b_1) = [26, 34), & r_1 = 13 \\ i = 2: & \tilde{b}_1 = 34 - \varphi_4 = 29, & [a_2, b_2) = [29, 34), & r_2 = 21 \\ i = 3: & \tilde{b}_2 = 34 - \varphi_3 = 31, & [a_3, b_3) = [31, 34), & r_3 = 26 \\ i = 4: & \tilde{b}_3 = 34 - \varphi_2 = 32, & [a_4, b_4) = [31, 32), & r_4 = 28 \\ i = 5: & \tilde{b}_4 = 32 - \varphi_1 = 31, & [a_5, b_5) = [31, 31), & r_5 = 29 \\ i = 6: & & & r_6 = 31 \ . \end{array}$$

Figure 3.1 shows the resource distributions up to $l = 34$.

PSfrag replacements

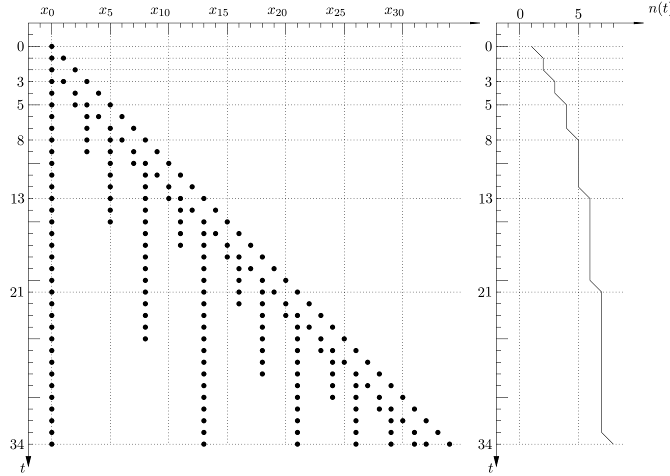


Figure 3.1: Resource distributions for $l \in [0, 34]$.

Lemma 3.1
Every distribution defined by the formula for the resource distribution (Definition 3.1) is instantaneously reversible if an arbitrary number of processors is available. □

As illustrated in Figure 3.2, it suffices to show that the distance between two neighbouring resources r_{i-1} and r_i is small enough; that is, that the time needed to evaluate the steps from r_{i-1} up to r_i is less than or equal to the time needed to calculate the reversal starting at r_{n-1} down to r_i, i.e.

$$\begin{aligned} &(3.3) & l - r_i &\geq r_i - r_{i-1} \\ &\iff & l - r_i + (r_i - r_{i-1}) &\geq r_i - r_{i-1} + (r_i - r_{i-1}) \\ &(3.4) \quad \iff & l - r_{i-1} &\geq 2(r_i - r_{i-1}) \ . \end{aligned}$$

The left hand side in (3.3) describes the distance ③, while the right hand side depicts the distance labelled with ② in Figure 3.2. After changing (3.3) to (3.4), the distance ④ in Figure 3.2 illustrates the lefthand side in (3.4), while the right hand side describes ①. Equivalent to (3.4), one may prove

$$l \geq 2r_i - r_{i-1} \tag{3.5}$$

for all $i \in [1, n)$. In Section 3.2 and in Section 3.3 it will be shown, that the number of processors required for the reverse computation starting with a instantaneously reversible resource distribution is bounded by the number of processors required for offline constructed parallel reversal schedules (2.3).

The proof of (3.5) consists of two steps. First, a corollary will be proven which shows that the resource distribution $\mathfrak{r}(l)$ is recursively constructed using a resource distribution $\mathfrak{r}(\lambda)$ for a smaller λ, which corresponds to l shifted by φ_{n-1} and the resources between 0 and

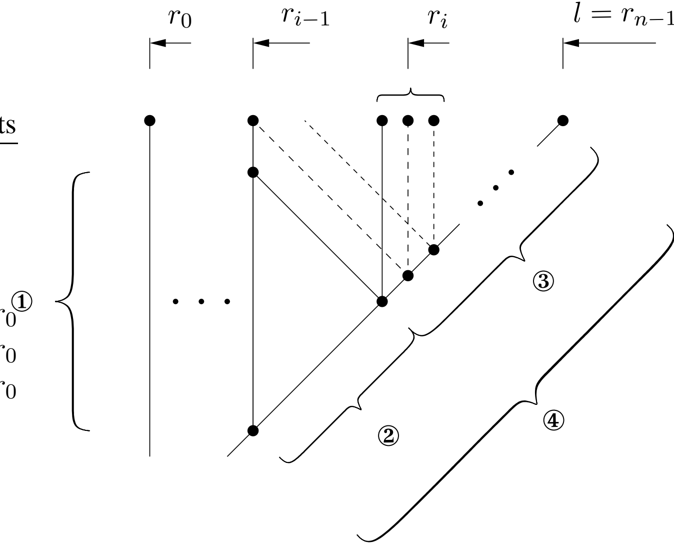


Figure 3.2: Suffi cient criterion of reversibility.

φ_{n-1}. This corollary can be seen as an equivalent to the formula for the resource distribution (Definition 3.1), but it defines a recursive algorithm. The second step is the actual proof of Lemma 3.1, carried out by recursively splitting the resource distributions down to obviously instantaneously reversible resource distributions.

Corollary 3.1
For $l \in [\varphi_n, \varphi_{n+1})$ with $n \geq 3$ let the resource distribution $\mathfrak{r}(l) = \{r_i \,|\, i \in [0, n)\}$ be defined by the formula for the resource distribution. Set $\lambda = l - \varphi_{n-1}$, and $d = 1$ if $\lambda \in [\varphi_{n-1}, \varphi_n)$, and $d = 2$ if $\lambda \in [\varphi_{n-2}, \varphi_{n-1})$. Consider for λ the resource distribution $\mathfrak{r}(\lambda) = \{\varrho_j \,|\, j \in [0, n-d)\}$ again defined by the formula for the resource distribution. Then, for every resource position $r_i \in \mathfrak{r}(l)$, there exists a unique resource position $\varrho_j \in \mathfrak{r}(\lambda)$, such that

$$\varrho_j + \varphi_{n-1} = r_i = r_{j+d} \quad \textit{with} \quad \begin{cases} d = 1 & \textit{if } \lambda \in [\varphi_{n-1}, \varphi_n) \textit{ and} \\ d = 2 & \textit{if } \lambda \in [\varphi_{n-2}, \varphi_{n-1}) \end{cases} .$$

for $i \in [1, n)$ or $i \in [2, n)$ respectively A similar statement can be specified for the sequences of intervals $[a_i, b_i)$ and $[\alpha_i, \beta_i)$. □

Proof of Corollary 3.1:
Within this proof all Greek written variables belong to the resource distribution $\mathfrak{r}(\lambda)$ for the smaller λ. The proof is carried out by induction. Since $\lambda = l - \varphi_{n-1}$, it follows that $\lambda \in [\varphi_{n-2}, \varphi_n)$. Two cases have to be distinguished (Figure 3.3): first $\lambda \in [\varphi_{n-2}, \varphi_{n-1})$, which implies that $l \in [\varphi_n, \varphi_n + \varphi_{n-3})$, or second $\lambda \in [\varphi_{n-1}, \varphi_n)$, which implies that $l \in [\varphi_n + \varphi_{n-3}, \varphi_{n+1})$.

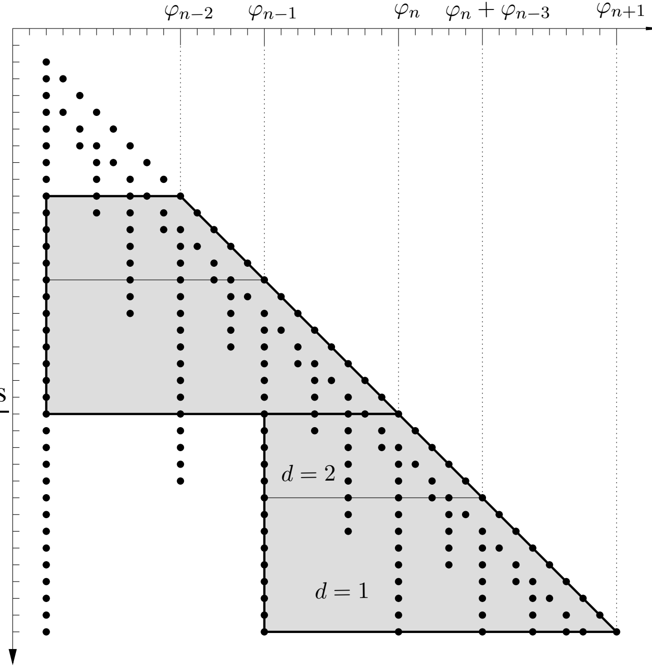


Figure 3.3: Proof sketch for Corollary 3.1.

Induction base case ($j = 0$): for $j = 0$ and $\lambda \in [\varphi_{n-2}, \varphi_{n-1})$, it has to be shown that $r_2 = \varrho_0 + \varphi_{n-1}$ as well as, $[a_2, b_2) = [\alpha_0 + \varphi_{n-1}, \beta_0 + \varphi_{n-1})$. Since $d = 2$, it follows that $l \in [\varphi_n, \varphi_n + \varphi_{n-3})$. By definition, one knows that for $i = 0$

$$[a_0, b_0) = [\varphi_n, \varphi_{n+1}) \quad \text{and} \quad r_0 = 0 \ . \tag{3.6}$$

Since $b_0 - a_0 = \varphi_{n-1} \neq \varphi_{n-2}$ for $n > 2$, and $l < \tilde{b}_0$ because of

$$\begin{aligned} \varphi_n + \varphi_{n-3} &= a_0 + \varphi_{n-3} \\ &= \varphi_{n+1} - \varphi_{n-2} \\ &= b_0 - \varphi_{n-2} \\ &= \tilde{b}_0 \ , \end{aligned}$$

it follows from (3.1) for $i = 1$ that $[a_1, b_1) = [\varphi_n, \varphi_n + \varphi_{n-3})$, and from (3.2) that $r_1 = \varphi_{n-2}$. For $i = 2$ follows $b_1 - a_1 = \varphi_{n-3}$ and $\tilde{b}_1 = b_1 - \varphi_{n-2}$. Therefore, one has

$$\begin{aligned} a_2 &= a_1 &&= \varphi_n &&= \varphi_{n-2} + \varphi_{n-1} &&= \alpha_0 + \varphi_{n-1} \ , \\ b_2 &= b_1 &&= \varphi_n + \varphi_{n-3} &&= \varphi_{n-1} + \varphi_{n-1} &&= \beta_0 + \varphi_{n-1} \ , \\ r_2 &= r_1 + \varphi_{n-3} &&= \varphi_{n-2} + \varphi_{n-3} &&= \varphi_{n-1} &&= \varrho_0 + \varphi_{n-1} \ . \end{aligned}$$

For the case $\lambda \in [\varphi_{n-1}, \varphi_n)$, it has to be shown that $[a_1, b_1) = [\alpha_0 + \varphi_{n-1}, \beta_0 + \varphi_{n-1})$, as well as that $r_1 = \varrho_0 + \varphi_{n-1}$. Since $d = 1$ it follows that $l \in [\varphi_n + \varphi_{n-3}, \varphi_{n+1})$. Again

for $i = 0$, the interval and resource position is defined by (3.6). For $i = 1$, one knows that $b_0 - a_0 = \varphi_{n-1} \neq \varphi_{n-2}$, and $l \in [\varphi_n + \varphi_{n-3}, \varphi_{n+1})$. Hence, $b_0 - \varphi_{n-2} = \varphi_n + \varphi_{n-3} \leq l$. Therefore, one obtains

$$\begin{array}{lllllllll} a_1 & = & b_0 - \varphi_{n-2} & = & \varphi_{n+1} - \varphi_{n-2} & = & 2\varphi_{n-1} & = & \alpha_0 + \varphi_{n-1} \ , \\ b_1 & = & b_0 & = & \varphi_{n+1} & = & \varphi_n + \varphi_{n-1} & = & \beta_0 + \varphi_{n-1} \ , \\ r_1 & = & r_0 + \varphi_{n-1} & = & \varphi_{n-1} & & & = & \varrho_0 + \varphi_{n-1} \ . \end{array}$$

This proves the induction base case for $j = 0$.

Induction step ($j \to j+1$): let $m = |\mathfrak{r}(\lambda)|$ be the number of resources in $\mathfrak{r}(\lambda)$ with $m = n-d$. It is required to obtain the definition of ϱ_j right. Assume for $\lambda \in [\varphi_{n-d}, \varphi_{n-d+1}) = [\varphi_m, \varphi_{m+1})$ that $[a_{i-1}, b_{i-1}) = [\alpha_{j-1} + \varphi_{n-1}, \beta_{j-1} + \varphi_{n-1})$, as well as $r_{i-1} = \varrho_{j-1} + \varphi_{n-1}$ with $i = j + d$. Obviously, since $\tilde{\beta}_j$ is defined by $\beta_j + \varphi_{m-j}$, one has

$$\begin{array}{lll} \tilde{b}_{i-1} & = & b_{i-1} - \varphi_{n-i+1} \\ & = & \beta_{j-1} + \varphi_{n-1} - \varphi_{m-j+1} \\ & = & \tilde{\beta}_{j-1} + \varphi_{n-1} \ . \end{array}$$

As a result, if $l < \tilde{b}_{i-1}$ it follows that

$$\begin{array}{llll} & l & < & \tilde{b}_{i-1} \\ \Longleftrightarrow & l - \varphi_{n-1} & < & \tilde{b}_{i-1} - \varphi_{n-1} \\ \Longleftrightarrow & \lambda & < & \tilde{\beta}_{j-1} \ . \end{array}$$

Then the interval $[a_i, b_i)$ is given by

$$\begin{array}{lll} [a_i, b_i) & = & \begin{cases} [a_{i-1}, b_{i-1}) & \text{if } a_{i-1} = \tilde{b}_{i-1} \ , \\ [a_{i-1}, \tilde{b}_{i-1}) & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \text{ and } l < \tilde{b}_{i-1} \ , \\ [\tilde{b}_{i-1}, b_{i-1}) & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \text{ and } l \geq \tilde{b}_{i-1} \end{cases} \\ & = & \begin{cases} [\alpha_{j-1} + \varphi_{n-1}, \beta_{j-1} + \varphi_{n-1}) & \text{if } \alpha_{j-1} = \tilde{\beta}_{j-1} \ , \\ [\alpha_{j-1} + \varphi_{n-1}, \tilde{\beta}_{j-1} + \varphi_{n-1}) & \text{if } \alpha_{j-1} \neq \tilde{\beta}_{j-1} \text{ and } \lambda < \tilde{\beta}_{j-1} \ , \\ [\tilde{\beta}_{j-1} + \varphi_{n-1}, \beta_{j-1} + \varphi_{n-1}) & \text{if } \alpha_{j-1} \neq \tilde{\beta}_{j-1} \text{ and } \lambda \geq \tilde{\beta}_{j-1} \end{cases} \\ & = & [\alpha_j + \varphi_{n-1}, \beta_j + \varphi_{n-1}) \ . \end{array}$$

Hence the resource position r_i is given by

$$\begin{array}{lll} r_i & = & \begin{cases} r_{i-1} + \varphi_{n-i} & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \text{ and } l \geq \tilde{b}_{i-1} \ , \\ r_{i-1} + \varphi_{n-i-1} & \text{else } . \end{cases} \\ & = & \begin{cases} \varrho_{j-1} + \varphi_{n-1} + \varphi_{m-j} & \text{if } \alpha_{j-1} \neq \tilde{\beta}_{j-1} \text{ and } \lambda \geq \tilde{\beta}_{j-1} \ , \\ \varrho_{j-1} + \varphi_{n-1} + \varphi_{m-j-1} & \text{else } , \end{cases} \\ & = & \varrho_j + \varphi_{n-1} \ , \end{array}$$

which completes the proof of Corollary 3.1. ∎

Proof of Lemma 3.1:
Using the terminology of Corollary 3.1, it is assumed that the resource distribution $\mathfrak{r}(\lambda)$ for $\lambda = l - \varphi_{n-1}$ steps is instantaneously reversible, i.e. $\lambda \geq 2\varrho_j - \varrho_{j-1}$ for all $j \in [1, m)$. Hence, all resource positions r_i for $i \in [d, n)$ also fulfil (3.5), i.e. $l \geq 2r_i - r_{i-1}$. Now, for $\lambda \in [\varphi_{n-2}, \varphi_{n-1})$, the first resources $i \in [0, d)$ are given by $r_0 = 0$, $r_1 = \varphi_{n-2}$ and $r_2 = \varphi_{n-1}$. Thus, (3.5) follows from

$$\begin{array}{rclclclcl} 2r_1 - r_0 & = & 2\varphi_{n-2} & = & \varphi_{n-1} + \varphi_{n-4} & < & \varphi_n & \leq & l \\ 2r_2 - r_1 & = & 2\varphi_{n-1} - \varphi_{n-2} & = & \varphi_{n-1} + \varphi_{n-3} & < & \varphi_n & \leq & l\ . \end{array}$$

Furthermore, if $\lambda \in [\varphi_{n-1}, \varphi_n)$, the inequality

$$2r_1 - r_0 = 2\varphi_{n-1} = \varphi_n + \varphi_{n-3} \leq l$$

holds for the first two resources.

Finally, to complete the proof, it must be shown that all small resource distributions for one, two, three and four steps fulfil the proposition in Lemma 3.1. This is obvious and can be shown by simply writing down the resource positions

$$\begin{array}{llll} l = 1: & r_0 = 0, & r_1 = 1, & \\ l = 2: & r_0 = 0, & r_1 = 2, & \\ l = 3: & r_0 = 0, & r_1 = 1, & r_2 = 3 \\ l = 4: & r_0 = 0, & r_1 = 2, & r_2 = 4\ , \end{array} \tag{3.7}$$

inserting them in (3.5) and recalculating the inequality (3.5). This completes the proof of Lemma 3.1. ■

3.2 Forward Computation for Instantaneous Reversal

Having obtained the resource distributions defined by the formula for the resource distribution (Definition 3.1), the question is whether or not it is possible to change the resource distribution $\mathfrak{r}(l-1) = \{\varrho_i \,|\, i \in [0, m)\}$ for a given l to the resource distribution $\mathfrak{r}(l) = \{r_i \,|\, i \in [0, n)\}$. Furthermore, if this transition is possible, one must study the computational effort.

Only time minimal transitions are acceptable. Such a time minimal transition takes the time $\tau = 1$ that one advancing step $F_i(x_i)$ lasts, because the intermediate state x_{l-1} has to change to the intermediate state x_l by carrying out one advancing step. Also one can say that the resource position $\varrho_{m-1} \in \mathfrak{r}(l-1)$ must be propagated to the resource position $\varrho_{m-1} + 1 = r_{n-1} \in \mathfrak{r}(l)$. Thus, in order to obtain a time minimal transition, all resource positions can only change from ϱ_i to $r_j = \varrho_i + 1$ or $r_j = \varrho_i$. The next lemma shows how such a time minimal transition works and how large the computational effort is.

Lemma 3.2
The effort to change a given resource distribution $\mathfrak{r}(l-1)$ for given $l \geq 3$ to the resource

distribution $\mathfrak{r}(l)$ is equivalent to two advancing steps $F_i(x_i)$ carried out in parallel, i.e. only two resource positions $\varrho_j \in \mathfrak{r}(l-1)$ have to be changed to the resource positions $r_i = \varrho_j + 1$. For all other resource positions r_i, there exists a resource position $\varrho_j \in \mathfrak{r}(l-1)$ such that $r_i = \varrho_j$. Furthermore, the resource positions that have to be moved are the resource positions ϱ_{m-2} and ϱ_{m-1}, which are moved to the resource positions $\varrho_{n-2} + 1 = r_{n-2}$ and $\varrho_{m-1} + 1 = r_{n-1}$, respectively. □

Proof:
Two cases must be considered. In the first case, the resource distributions for $l-1$ and l have different numbers of resource positions, and, in the second case, the distributions for $l-1$ and l have the same number of resource positions. By definition, the last case can only happen if the number of steps l equals a Fibonacci number. From here on, all variables written in Greek letters belong to the resource distribution with the smaller number of $l-1$ steps. The variables corresponding to l steps are written in Latin letters.

First, it is assumed that $l = \varphi_n$. The number of resources equals n for $\mathfrak{r}(l)$, and $m = n-1$ for $\mathfrak{r}(l-1)$. To prove Lemma 3.2, it has to be shown that the resource positions of the resource distributions $\mathfrak{r}(l-1)$ and $\mathfrak{r}(l)$ are equal for all resources with an index $i < n-2 = m-1$.

All intervals $[a_i, b_i)$ are defined such that l always lies inside the interval. Additionally, $a_0 = l$ by definition (3.1). Thus the values $a_i = l$ will never be changed for all i. Therefore, only the first two cases of the interval definition (3.1) will be used to calculate the intervals. Hence, only the second case of the resource positions' definition (3.2) is applied. That is why all resources are computed by $r_i = r_{i-1} + \varphi_{n-i-1}$ for $i \in [0, n-1)$, if the number of steps l equals φ_n.

For $l-1$ steps, the upper interval limit $\beta_i = \varphi_n$ will never be changed for all $i \in [0, m-1)$. This can be shown inductively. For case $i = 0$, the definition (3.1) of the intervals yields $\beta_0 = \varphi_n$. Now, for $i-1$ it is assumed that $\beta_{i-1} = \varphi_n$, $l-1 \geq \tilde{\beta}_{i-2}$, and $\alpha_{i-1} \neq \tilde{\beta}_{i-2}$. For the calculation of the intervals for the case i, one knows that $\tilde{\beta}_{i-1} = \beta_{i-1} - \varphi_{m-i-1} = \varphi_n - \varphi_{m-i-1}$. This expression defines a sequence of increasing numbers for the increasing value of i. The maximum is reached at $i = m-2$, i.e. $\tilde{\beta}_{i-1} = \varphi_n - \varphi_1 = \varphi_n - 1$. This value, however, is always less than or equal to $l-1$, such that only the third case of the definition (3.1) of the intervals will be used. Therefore, only the first case of the definition (3.2) of the resource positions will be applied. Hence, the definitions of all resource positions ϱ_i and r_i for all $i \in [0, m-1)$ are identical, i.e. it was shown that all resources $\varrho_j \in \mathfrak{r}(l-1)$ are given by $\varrho_j = \varrho_{j-1} + \varphi_{m-j}$, $j \in [0, m-1)$, and all resources r_i of $\mathfrak{r}(l)$ are given by $r_i = r_{i-1} + \varphi_{n-i-1} = r_{i-1} + \varphi_{m-i}$, $i \in [0, n-2) = [0, m-1)$.

The two movements of resource positions are $\varrho_{m-1} = l-1$ moved to $r_{n-1} = l$ and ϱ_{m-2} moved to r_{n-2}. The resource positions ϱ_{m-1} and r_{n-1} define the step number $l-1$ and l respectively. Hence, the forward step $F_l(x_{l-1}) = x_l$ has to be carried out. The next smaller resource position starts at $\varrho_{m-2} = r_{n-3}$ and is transformed to $r_{n-2} = r_{n-3} + \varphi_{n-(n-2)-1} = r_{n-3} + \varphi_1 = r_{n-3} + 1$. The situation for the last two resource positions is illustrated in Figure 3.4. The additional resource needed to change the resource distribution $\mathfrak{r}(l-1)$ to the resource distribution $\mathfrak{r}(l)$ is used for the resource position ϱ_{m-2}. On the one hand, this resource

position is used to obtain the resource position r_{n-3}, and on the other hand, this resource is used to compute the resource position r_{n-2}.

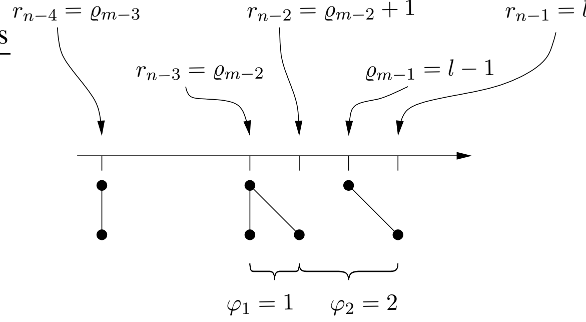


Figure 3.4: Changing of the resource positions ϱ_{m-2} and ϱ_{m-1} to the resource positions r_{n-3}, r_{n-2} and r_{n-1}

In order to prove the second part of the assertion, it is assumed that the resource distribution $\mathfrak{r}(l-1)$ is comprised of the same number of elements as the resource distribution $\mathfrak{r}(l)$. This is equivalent to the assumption that $l-1, l \in [\varphi_n, \varphi_{n+1})$. The proof for this case is carried out recursively using the subdistributions defined by Corollary 3.1.

Obviously all resource distributions with a number of steps between one and four satisfy Lemma 3.2.

Now, using the approach of Corollary 3.1, it is assumed that for $\lambda = l - \varphi_{n-1}$, Lemma 3.2 is true for the resource position within the resource distribution $\mathfrak{r}(\lambda-1)$ for $\lambda-1$, and resource distribution $\mathfrak{r}(\lambda)$ for λ.

Now the effort needed for the resource positions smaller than φ_{n-1} has to be computed. This must be zero for all $l \in (\varphi_n, \varphi_{n+1})$. The resource positions are explicit given by $r_0 = 0$, $r_1 = \varphi_{n-2}$ and $r_2 = \varphi_{n-1}$, if $l-1, l \in [\varphi_n, \varphi_n + \varphi_{n+3})$, or by $r_0 = 0$ and $r_1 = \varphi_{n-1}$, if $l-1, l \in [\varphi_n + \varphi_{n+3} + \varphi_{n+1})$. In both cases, for every resource position r_i a corresponding resource position ϱ_j can be found such that $r_i = \varrho_j$. Even if $l-1 \in [\varphi_n, \varphi_n + \varphi_{n+3})$ and $l \in [\varphi_n + \varphi_{n+3}, \varphi_{n+1})$, the resource r_1 has its predecessor ϱ_2. This concludes the proof. ■

Lemma 3.2 provides an indication of how one should proceed if no final number of steps l is known. As shown in Figure 3.5, one has to start two asynchronous running advancing processes. The second process starts, when the first process reaches the intermediate state x_2. All other resources act as checkpoints and are set by the second process calculation. These checkpoints are written and deleted at certain times. The specification of these times is carried out using the state count of the last resource r_{n-1}, which is equivalent to the current number of steps. The state count, when the checkpoint r_i is written, is named $l_O(r_i)$, while the deletion state count is named $l_I(r_i)$. The indices O and I stands for *outer* and *inner*, whereby the meaning will be explained in more detail in the following Section 3.3.

During the forward computation of the second process, every state the process reaches has to be checked, whether a checkpoint has to be written or not. A checkpoint is written if the step

PSfrag replacements

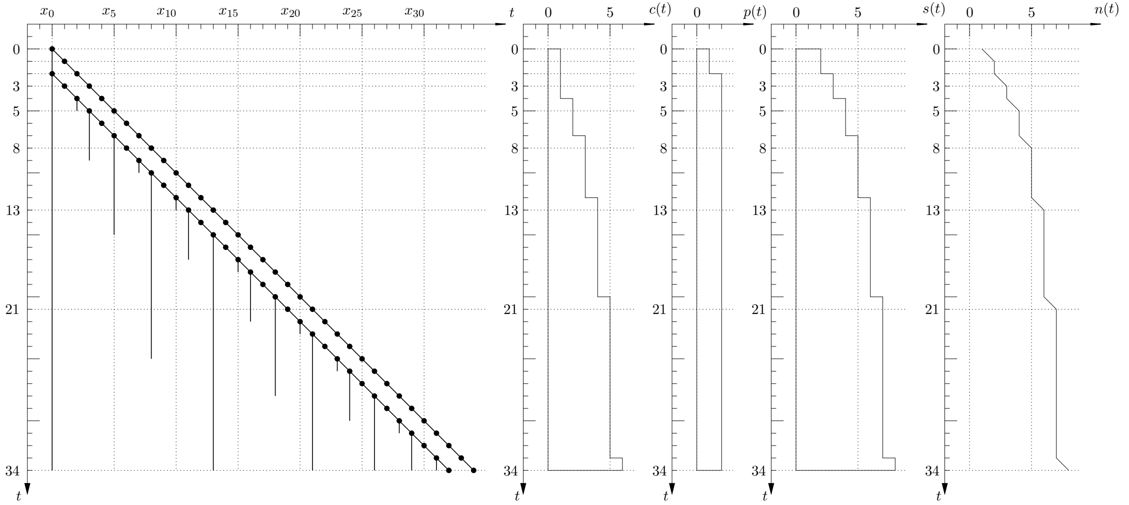


Figure 3.5: Forward computation for an unknown number of steps.

number l equals $l_O(r_i)$. This step number is always given by

$$l_O(r_i) = r_i + 2 \quad . \tag{3.8}$$

Generally, every intermediate state has to be stored as a checkpoint. Some of them, however, have a life time of $0 = l_I(r_i) - l_O(r_i)$, such as the intermediate state x_1, and x_4.

The deletion of a checkpoint is carried out if the step number $l = l_I(r_i)$. For the definition of $l_I(r_i)$, one must first look at the second resource position r_1. By definition, this resource position is always a Fibonacci number, w.l.o.g. $r_1 = \varphi_n$. The last step number l, when this resource position is used, is one step before the condition $l < b_0 - \varphi_n = \varphi_{n+3} - \varphi_n = 2(\varphi_n + \varphi_{n-1})$ which can be found in the formula for the resource distribution (Definition 3.1) fails. If $l = b_0 - \varphi_n$, the resource position r_1 equals φ_{n+1}. Hence, the step number $l_I(r_1)$, when a resource position $r_1 = \varphi_n$ has to be deleted, is defined by

$$l_I(\varphi_n) = 2(\varphi_n + \varphi_{n-1}) - 1 = 2\varphi_{n+1} - 1 \quad .$$

To determine the deletion step number of an arbitrary resource position, the following corollary is required. It shows that if a checkpoint has the resource position $r_i \in (\varphi_n, \varphi_{n+1})$, and it is set at step $l_O(r_i)$, and it is deleted at step $l_I(r_i)$, then a checkpoint exists with the resource position $\varrho_j = r_i - \varphi_n \in (0, \varphi_{n-1})$, which is set at step $l_O(\varrho_j) = l_O(r_i) - \varphi_n$ and deleted at step $l_I(\varrho_j) = l_I(r_i) - \varphi_n$. In abstract, the corollary shows that the inner structure formed by all resource positions between two Fibonacci numbers φ_n and φ_{n+1} (right grey area in Figure 3.6(a)) is equivalent to the inner structure formed by all resource positions lying between 0 and φ_{n-1} (left grey area in Figure 3.6(a)).

Using this knowledge, the deletion step of an arbitrary resource position can be recursively defined by $l_I(r_i) = l_I(r_i - \varphi_n) + \varphi_n$, if $r_i \in (\varphi_n, \varphi_{n+1})$.

Corollary 3.2
Let $\mathfrak{S}(\varphi_n, \varphi_{n+1})$ with $n \geq 3$ be the set of all possible resource positions $r \in (\varphi_n, \varphi_{n+1})$, i.e.

$$\mathfrak{S}(\varphi_n, \varphi_{n+1}) = \{r \mid r \in (\varphi_n, \varphi_{n+1}) \quad \textit{and} \quad r \in \mathfrak{r}(l) \textit{ for } l \in \mathbb{N} \ .\}$$

Then, for every resource position $r_i \in \mathfrak{S}(\varphi_n, \varphi_{n+1}) \cap \mathfrak{r}(l)$, there exists a resource position $\varrho_j \in \mathfrak{S}(0, \varphi_{n-1}) \cap \mathfrak{r}(\lambda)$ for a step number λ such that

$$r_i = \varrho_j + \varphi_n \qquad \textit{and} \qquad l = \lambda + \varphi_n \ .$$

□

Proof:
The proof will be carried out in three parts. First, it will be shown that if a resource position r_i lies in $(\varphi_n, \varphi_{n+1})$, the range of possible number of steps l with $r_i \in \mathfrak{r}(l)$ can be bounded. Secondly, the statement of Corollary 3.2 is shown for a subset of the possible number of steps l. Finally, the assertion will be proven by induction for all steps numbers $l \in \mathbb{N}$.

To prove the first part, it has to be shown that if $r_i \in (\varphi_n, \varphi_{n+1})$, then $l \in (\varphi_n, \varphi_{n+2}) =: (l_l, l_u)$. If $r_i > \varphi_n$, then $l > \varphi_n = l_l$ by definition. The upper bound l_u for l is reached at the number of steps, where no resource position lies between the resource positions $r_k = \varphi_n$ and $r_{k+1} = \varphi_{n+1}$. By Lemma 3.2 the position r_k and r_{k+1} will not be changed once they are set, and k is small enough. The resource position can only be deleted. Also one obtains from Lemma 3.2 that every resource position r_i, $i > 0$ needs a predecessor. Therefore, a resource position between φ_n and φ_{n+1} cannot exist for all step numbers greater than the upper bound l_u. That means that only one situation, i.e. one k, has to be found where $r_k = \varphi_n$ and $r_{k+1} = \varphi_{n+1}$. Since $n > 0$ and $r_0 = 0$, it follows that $k > 0$. One possible number of steps where no resource position exists between φ_n and φ_{n+1} is given by $l = \varphi_{n+2} - 1$. Then k equals 1. This can be verified by the formula for the resource distribution. As a result of $l \in [\varphi_{n+1}, \varphi_{n+2})$, the intervals and the resource positions are given by

$$\begin{array}{rrrr} & a_0 = \varphi_{n+1} \ , & b_0 = \varphi_{n+2} \ , & r_0 = 0 \ , \\ \tilde{b}_0 = \varphi_{n+2} - \varphi_{n-1} \ , & a_1 = \tilde{b}_0 \ , & b_1 = \varphi_{n+2} \ , & r_1 = \varphi_n \ , \\ \tilde{b}_1 = \varphi_{n+2} - \varphi_{n-2} \ , & a_2 = \tilde{b}_1 \ , & b_2 = \varphi_{n+2} \ , & r_2 = \varphi_{n+1} \ . \end{array}$$

Lemma 3.2 provides that for all l larger than $\varphi_{n+2} - 1$, there is no resource position between $r_1 = \varphi_n$ and $r_2 = \varphi_{n+1}$.

In order to prove the second part, all resource positions $r_i \in (\varphi_n, \varphi_{n+1})$ for all $l \in [\varphi_{n+1}, \varphi_{n+2})$ are considered. The area containing these resource positions is denoted by $Rec(\varphi_n, \varphi_{n+1}, \varphi_{n+2})$ (see Figure 3.6(a)). By Corollary 3.1, resource positions $\varrho_j \in (0, \varphi_{n-1})$ for $\lambda \in (\varphi_{n-1}, \varphi_n)$ exist such that $r_i = \varrho_j + \varphi_n$ for $r_i \in \mathfrak{r}(l)$ and $l = \lambda + \varphi_n$. Hence, for each resource position r_i within the considered rectangle area $Rec(\varphi_n, \varphi_{n+1}, \varphi_{n+2})$, an equivalent resource ϱ_j can be found, which lies within the rectangle area defined by $\varrho_j \in (0, \varphi_{n-1})$ for $\lambda \in (\varphi_{n-1}, \varphi_n)$. This rectangle is denoted $Rec(0, \varphi_{n-1}, \varphi_n)$ in Figure 3.6(a). Thus one can

PSfrag replacements PSfrag replacements

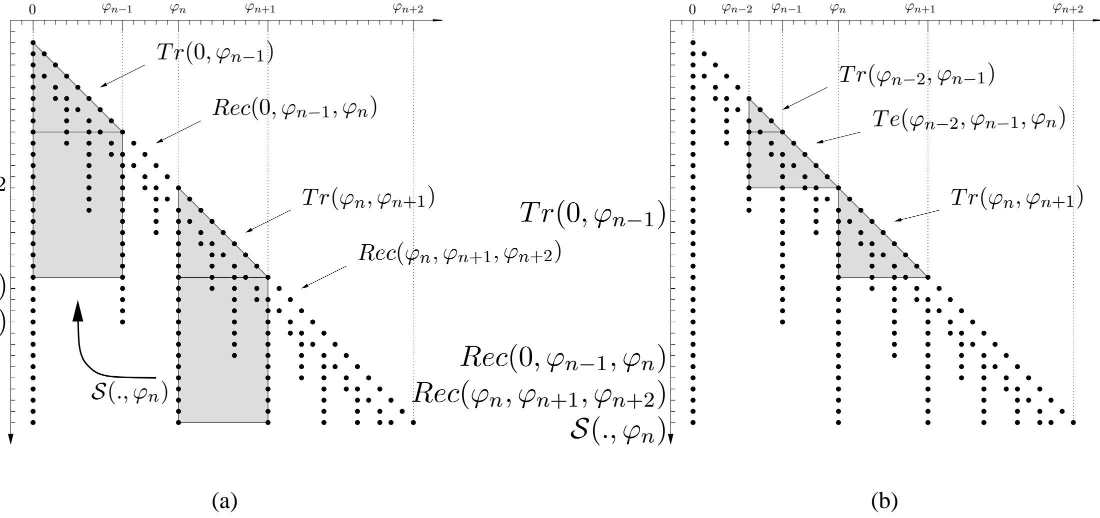


Figure 3.6: Proof sketch for Corollary 3.2.

say that the rectangle $Rec(\varphi_n, \varphi_{n+1}, \varphi_{n+2})$ is shifted by φ_n in both the step number direction and the time direction, which yields the rectangle $Rec(0, \varphi_{n-1}, \varphi_n)$.

Part three: For the remaining resource positions within the triangle area defined by $r_i \in (\varphi_n, \varphi_{n+1}) \cap \mathfrak{r}(l)$ with $l \in [\varphi_n, \varphi_{n+1})$, it now has to be proved that they are equivalent to resource positions within the triangle defined by $\varrho_i \in (0, \varphi_{n-1}) \cap \mathfrak{r}(\lambda)$ for $\lambda \in [0, \varphi_{n-1})$ shifted by φ_n. The first area will be termed $Tr(\varphi_n, \varphi_{n+1})$ and the second $Tr(0, \varphi_{n-1})$, as can been seen in Figure 3.6(a). Hence, it has to be shown that

$$Tr(\varphi_n, \varphi_{n+1}) = \mathcal{S}(Tr(0, \varphi_{n-1}), \varphi_n) \ , \tag{3.9}$$

where $\mathcal{S}(.,.)$ denotes the shift in the down-rightward direction referring to the Figures 3.6(a), 3.6(b), 3.7(a) or 3.7(b). The triangle $Tr(\varphi_n, \varphi_{n+1})$ is a subset of all resource positions $\mathfrak{r}(l)$ for $l \in [\varphi_n, \varphi_{n+1})$. Further, all resource positions within $Tr(\varphi_n, \varphi_{n+1})$ are larger then φ_{n-1} by definition. Therefore, from Corollary 3.1, one knows, that for each resource position $r \in \mathfrak{r}(l)$ there exists a unique resource position $\varrho \in \mathfrak{r}(\lambda)$ such that $r = \varrho + \varphi_{n-1}$ and $l = \lambda + \varphi_{n-1}$. Hence, it is known that

$$Tr(\varphi_n, \varphi_{n+1}) = \mathcal{S}(Tr(\varphi_{n-2}, \varphi_n), \varphi_{n-1}) \ . \tag{3.10}$$

As shown in Figure 3.6(b), $Tr(\varphi_{n-2}, \varphi_n)$ can be divided into the triangle $Tr(\varphi_{n-2}, \varphi_{n-1})$ and a tetragon $Te(\varphi_{n-2}, \varphi_{n-1}, \varphi_n)$, which contain all resource positions $r_i \in (\varphi_{n-2}, \varphi_n)$ for $l \in [\varphi_{n-1}, \varphi_n)$. Using Corollary 3.1 once more, one obtains that

$$Te(\varphi_{n-2}, \varphi_{n-1}, \varphi_n) = \mathcal{S}(Te(0, \varphi_{n-3}, \varphi_{n-1}), \varphi_{n-2})$$

PSfrag replacements
PSfrag replacements

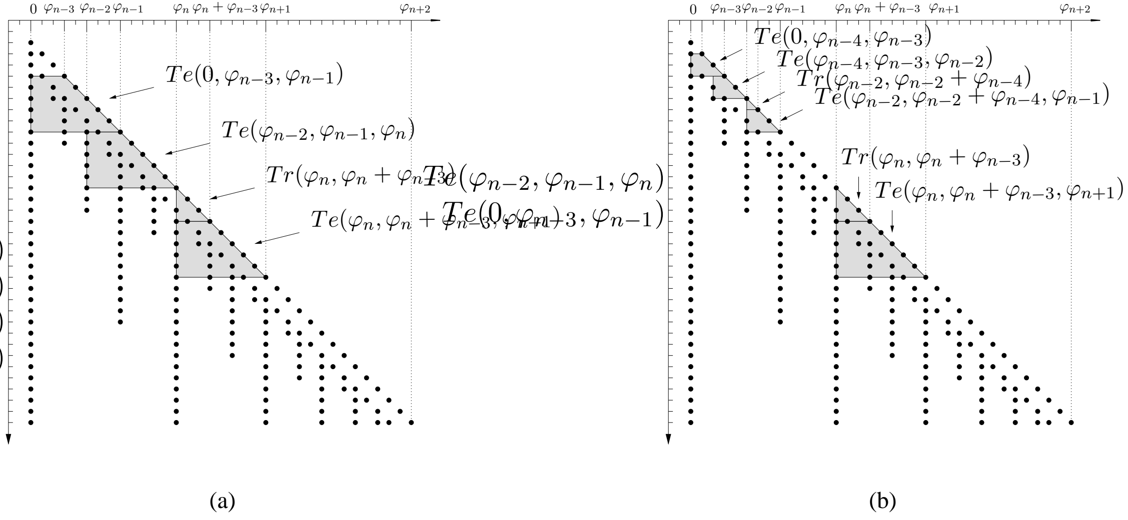


Figure 3.7: Proof sketch for Corollary 3.2.

and hence

$$
\begin{aligned}
Te(\varphi_n, \varphi_{n+1} - \varphi_{n-2}, \varphi_{n+1}) &= \mathcal{S}(Te(\varphi_{n-2}, \varphi_{n-1}, \varphi_n), \varphi_{n-1}) \\
&= \mathcal{S}(\mathcal{S}(Te(0, \varphi_{n-3}, \varphi_{n-1}), \varphi_{n-2}), \varphi_{n-1}) \\
&= \mathcal{S}(Te(0, \varphi_{n-3}, \varphi_{n-1}), \varphi_{n-2} + \varphi_{n-1}) \\
&= \mathcal{S}(Te(0, \varphi_{n-3}, \varphi_{n-1}), \varphi_n) \ , \qquad (3.11)
\end{aligned}
$$

as can be seen in Figure 3.7(a). For the remaining triangle defined by $Tr(\varphi_n, \varphi_{n+1} - \varphi_{n-2}) = Tr(\varphi_n, \varphi_n + \varphi_{n-3})$, it must be shown that

$$Tr(\varphi_n, \varphi_{n+1} - \varphi_{n-2}) = \mathcal{S}(Tr(0, \varphi_{n-3}), \varphi_n) \ .$$

As a result of (3.10), the claim can be rewritten as $Tr(\varphi_{n-2}, \varphi_{n-1}) = \mathcal{S}(Tr(0, \varphi_{n-3}), \varphi_{n-2})$, which is an equivalent claim to (3.9).

Part three of the proof is achieved by proving

$$
\begin{aligned}
Tr(\varphi_n, \varphi_{n+1} - \Phi_j) &= Tr(\varphi_n, \varphi_n + \varphi_{n-2j-1}) \\
&= \mathcal{S}(Tr(\varphi_{n-2j-2}, \varphi_{n-2j}), \bar{\Phi}_j) \qquad (3.12)
\end{aligned}
$$

with

$$\Phi_j = \sum_{k=1}^{j} \varphi_{n-2k} \qquad \text{and} \qquad \bar{\Phi}_j = \sum_{k=0}^{j} \varphi_{n-2k-1} \qquad (3.13)$$

first. An induction will be used.

Induction base case ($j = 0$): by setting $j = 0$ in (3.12) one obtains

$$\begin{aligned} Tr(\varphi_n, \varphi_{n+1} - \Phi_0) &= \mathcal{S}(Tr(\varphi_{n-2}, \varphi_n), \bar{\Phi}_0) \\ Tr(\varphi_n, \varphi_{n+1}) &= \mathcal{S}(Tr(\varphi_{n-2}, \varphi_n), \varphi_{n-1}) \ , \end{aligned}$$

the induction base case, which is the formula (3.10).

Induction hypothesis: It is assumed that (3.12) is true for j.

Induction step ($j \to j+1$): the given area can be subdivided once more into a triangle and a tetragon such that

$$\begin{aligned} Tr(\varphi_n, \varphi_{n+1} - \Phi_j) &= \quad Tr(\varphi_n, \varphi_{n+1} - \Phi_j - \varphi_{n-2j-2}) \\ &\cup \ Te(\varphi_n, \varphi_{n+1} - \Phi_j - \varphi_{n-2j-2}, \varphi_{n+1} - \Phi_j) \\ &= \quad Tr(\varphi_n, \varphi_n + \varphi_{n-2j-1} - \varphi_{n-2j-2}) \\ &\cup \ Te(\varphi_n, \varphi_n + \varphi_{n-2j-1} - \varphi_{n-2j-2}, \varphi_n + \varphi_{n-2j-1}) \\ &= \quad Tr(\varphi_n, \varphi_n + \varphi_{n-2j-3}) \\ &\cup \ Te(\varphi_n, \varphi_n + \varphi_{n-2j-3}, \varphi_n + \varphi_{n-2j-1}) \ . \end{aligned}$$

Using the formula (3.12), one can rewrite this equation as

$$\begin{aligned} Tr(\varphi_n, \varphi_{n+1} - \Phi_j - \varphi_{n-2j-2}) &= Tr(\varphi_n, \varphi_{n+1} - \Phi_{j+1}) \\ &= \mathcal{S}(Tr(\varphi_{n-2j-2}, \varphi_{n-2j} - \varphi_{n-2j-2}), \bar{\Phi}_j) \\ &= \mathcal{S}(Tr(\varphi_{n-2j-2}, \varphi_{n-2j-1}), \bar{\Phi}_j) \end{aligned} \tag{3.14}$$

and

$$\begin{aligned} & Te(\varphi_n, \varphi_{n+1} - \Phi_j - \varphi_{n-2j-2}, \varphi_{n+1} - \Phi_j) \\ & \quad = Te(\varphi_n, \varphi_{n+1} - \Phi_{j+1}, \varphi_{n+1} - \Phi_j) \\ & \quad = \mathcal{S}(Te(\varphi_{n-2j-2}, \varphi_{n-2j} - \varphi_{n-2j-2}, \varphi_{n-2j}), \bar{\Phi}_j) \\ & \quad = \mathcal{S}(Te(\varphi_{n-2j-2}, \varphi_{n-2j-1}, \varphi_{n-2j}), \bar{\Phi}_j) \ . \end{aligned} \tag{3.15}$$

The resulting triangle in (3.14), $Tr(\varphi_{n-2j-2}, \varphi_{n-2j-1})$, can be shifted again by utilising Corollary 3.1 such that

$$Tr(\varphi_{n-2j-2}, \varphi_{n-2j-1}) = \mathcal{S}(Tr(\varphi_{n-2j-4}, \varphi_{n-2j-2}), \varphi_{n-2j-3}) \ .$$

By combining this equation with (3.14), one obtains the induction hypothesis for $j = j+1$, i.e.

$$\begin{aligned} Tr(\varphi_n, \varphi_{n+1} - \Phi_{j+1}) &= Tr(\varphi_n, \varphi_{n+1} - \Phi_j - \varphi_{n-2j-2}) \\ &= \mathcal{S}(Tr(\varphi_{n-2j-2}, \varphi_{n-2j-1}), \bar{\Phi}_j) \\ &= \mathcal{S}(\mathcal{S}(Tr(\varphi_{n-2j-4}, \varphi_{n-2j-2}), \varphi_{n-2j-3}), \bar{\Phi}_j) \\ &= \mathcal{S}(Tr(\varphi_{n-2j-4}, \varphi_{n-2j-2}), \bar{\Phi}_j + \varphi_{n-2j-3}) \\ &= \mathcal{S}(Tr(\varphi_{n-2(j+1)-2}, \varphi_{n-2(j+1)}), \bar{\Phi}_{j+1}) \ , \end{aligned}$$

which finishes the induction.

Secondly, it must be proven that the tetragon part $Te(\varphi_n, \varphi_n + \varphi_{n-2j-3}, \varphi_n + \varphi_{n-2j-1})$ of every triangle $Tr(\varphi_n, \varphi_n + \varphi_{n-2j-1})$ can always be shifted by φ_n to the tetragon given

by $Te(0, \varphi_{n-2j-3}, \varphi_{n-2j-1})$. The tetragon lays between 0 and φ_{n-2j-1}, as can be seen in Figure 3.7(a).

The tetragon $Te(\varphi_{n-2j-2}, \varphi_{n-2j-1}, \varphi_{n-2j})$, which is the result in (3.15), can be shifted using Corollary 3.1 such that

$$Te(\varphi_{n-2j-2}, \varphi_{n-2j-1}, \varphi_{n-2j}) = \mathcal{S}(Te(0, \varphi_{n-2j-3}, \varphi_{n-2j-1}), \varphi_{n-2j-2}) \ .$$

Combining this result with (3.15) one obtains

$$\begin{aligned} & Te(\varphi_n, \varphi_n + \varphi_{n-2j-3}, \varphi_n + \varphi_{n-2j-1}) \\ &= Te(\varphi_n, \varphi_{n+1} - \Phi_j - \varphi_{n-2j-2}, \varphi_{n+1} - \Phi_j) \\ &= Te(\varphi_n, \varphi_{n+1} - \Phi_{j+1}, \varphi_{n+1} - \Phi_j) \\ &= \mathcal{S}(Te(\varphi_{n-2j-2}, \varphi_{n-2j-1}, \varphi_{n-2j}), \bar{\Phi}_j) \\ &= \mathcal{S}(\mathcal{S}(Te(0, \varphi_{n-2j-3}, \varphi_{n-2j-1}), \varphi_{n-2j-2}), \bar{\Phi}_j) \\ &= \mathcal{S}(Te(0, \varphi_{n-2j-3}, \varphi_{n-2j-1}), \bar{\Phi}_j + \varphi_{n-2j-2}) \\ &= \mathcal{S}(Te(0, \varphi_{n-2j-3}, \varphi_{n-2j-1}), \varphi_n) \ . \end{aligned} \tag{3.16}$$

Now, one can build a tetragon $Te(\varphi_n, \varphi_n + \varphi_{n-2j-3}, \varphi_{n+1})$ by the union of all tetragon up to index j, i.e.

$$Te(\varphi_n, \varphi_n + \varphi_{n-2j-3}, \varphi_{n+1}) = \bigcup_{k=0}^{j} Te(\varphi_n, \varphi_n + \varphi_{n-2k-3}, \varphi_n + \varphi_{n-2k-1}) \ .$$

Since by (3.16) each element of this union can be shifted by φ_n, the whole union can be shifted as well, i.e.

$$Te(\varphi_n, \varphi_n + \varphi_{n-2j-3}, \varphi_{n+1}) = \mathcal{S}(Te(\varphi_n, \varphi_n + \varphi_{n-2j-3}, \varphi_{n+1}), \varphi_n)$$

For $j = \lceil (n-3)/2 \rceil$ the tetragon $Te(\varphi_n, \varphi_n + \varphi_{n-2j-3}, \varphi_{n+1})$ either covers the whole triangle $Tr(\varphi_n, \varphi_{n+1})$, or for the remaining triangle $Tr(\varphi_n, \varphi_n + 1)$ it is trivial to show that it can be shifted by φ_n, i.e. $Tr(\varphi_n, \varphi_n + 1) = \mathcal{S}(Tr(0, 1), \varphi_n)$. Hence, one obtains (3.9), i.e.

$$Tr(\varphi_n, \varphi_{n+1}) = \mathcal{S}(Tr(0, \varphi_{n-1}), \varphi_n) \ .$$

This completes the proof. ■

Using Corollary 3.2, one can concretise the upper bound of the step number, where a resource r_i is deleted, i.e. an upper bound on $l_I(r_i)$, given in the first part of the proof of Corollary 3.2, to the exact definition

$$l_I(r_i) = \begin{cases} 2\varphi_{n+1} - 1 & \text{if } r_i = \varphi_n \ , \\ l_I(r_i - \varphi_n) + \varphi_n & \text{if } r_i \in (\varphi_n, \varphi_{n+1}) \ . \end{cases} \tag{3.17}$$

3.3 Reverse Sweep for Instantaneous Reversal

Assume a computation with checkpointing is carried out as described in Section 3.2. This computation is referred to as the forward computation. This forward computation stopped with a step number l. Now, the corresponding reversal of this forward computation will be derived. This reversal is also called the reverse computation. From inequality (3.5) and Figure 3.2, one knows that for the Fibonacci number $2\varphi_j = r_i - r_{i-1}$, an equilateral triangle of the height φ_j and the base line length $2\varphi_j$ fits always between the resource positions r_{i-1} and r_i. According to the formula for the resource distribution (Definition 3.1), the distance between two resource positions is always a Fibonacci number. Hence, one may insert a time optimal reversal schedule for a fixed number of steps φ_j as explained in Chapter 2.4.

PSfrag replacements

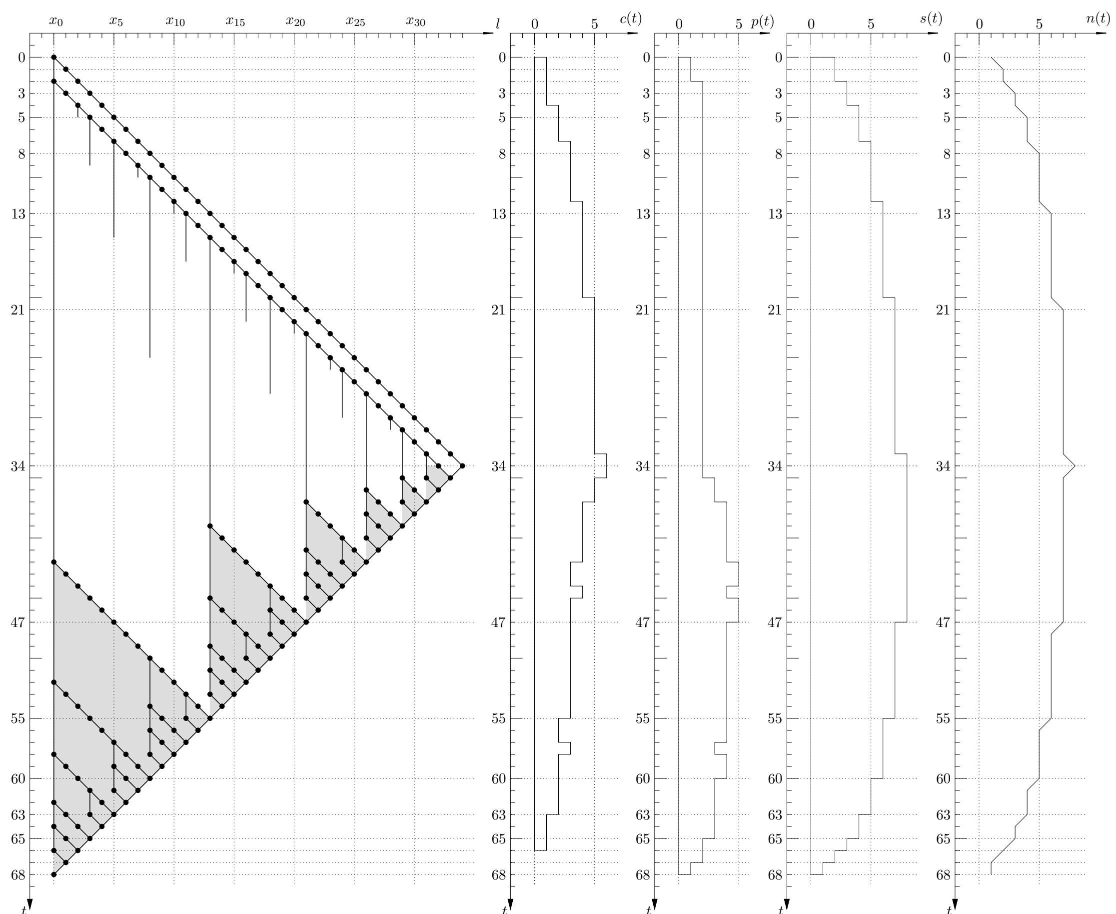


Figure 3.8: One forward and backward computation for an instantaneous reversal.

A resulting schedule can be seen in Figure 3.8. Once again, the diagrams $c(t)$ and $p(t)$ show the checkpoints and the processes needed during the advancing steps, respectively. The diagram for the value $s(t)$ (second from the right) illustrates the sum of all resources needed during the advancing steps. The sum of all resources at the end of an advancing step can be

seen on the diagram on the far right (value $n(t)$).

From the symmetry observed in this diagram, one can conclude that the schedules for the forward computation and the reverse computation are symmetric in some sense. To exploit this ”symmetry”, the resource distributions of a reverse computation (illustrated by $\times$ in Figure 3.9) are mirrored on the apex (the axis determined by the number of steps where the reverse calculations starts) onto the resource distributions of the forward computation (illustrated by $\Box$ in Figure 3.9). This yields a new view of the ”behaviour” of the resources during the forward or

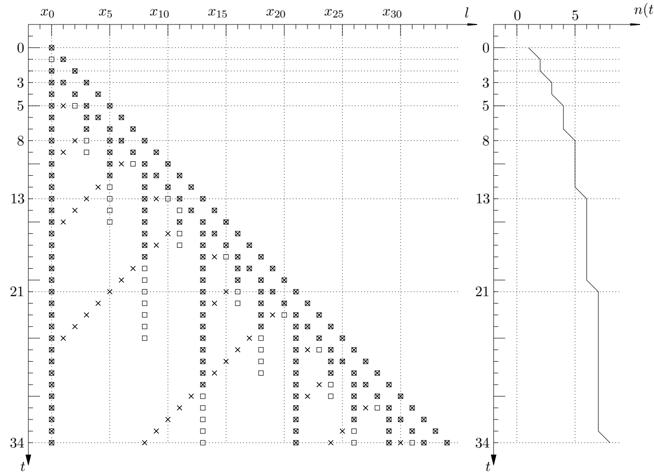
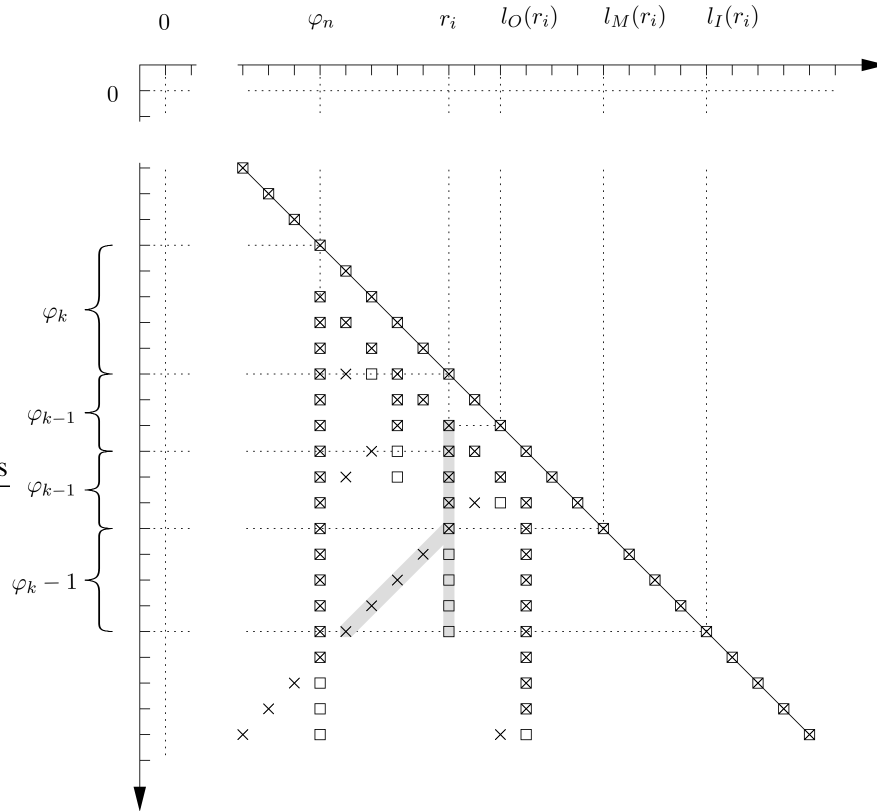


Figure 3.9: Resource distribution of the reverse calculation ($\times$) mirrored onto resource distribution of the forward computation ($\Box$).

reverse computation using a parallel schedule. During the forward computation, a resource is set at the resource position r_i, when the leading process reaches the step number $l_O(r_i)$. The leading process is always the process using or computing the intermediate state with the largest state number. During the reverse computation, the step number $l_O(r_i)$ is the same step number of the leading reversing process, when the resource r_i is used to start the computation of the preparing step. It is the step number where the resource at the resource position r_i is deleted during the reversal (compare Figure 3.10).

At the other end of the life cycle of a resource assigned to the resource position r_i, the step number $l_I(r_i)$ can be defined. If during the forward computation the leading process reaches the step number $l_I(r_i)$, then the resource assigned to the resource position r_i must be deleted. If a reverse computation is carried out and if the leading process reaches $l_I(r_i)$, then the resource assigned to the resource positions r_i begins to ”exist”. The actual resource position or the intermediate state of this resource is not yet the resource position r_i or the intermediate state x_{r_i} (therefore the resource is just assigned to the resource position r_i). This resource position has to be computed by carrying out advancing steps. Once the resource reaches the resource position r_i, the forward computation stops and the resource becomes a checkpoint. This happens when the leading process position reaches the step number $l_M(r_i)$ (see Figure 3.10).

In Figure 3.10, one can also see how the step numbers $l_O(r_i)$, $l_I(r_i)$ are computed and, as explained in the next section, how the step number $l_M(r_i)$ is computed. Here, the life cycle of



PSfrag replacements

Figure 3.10: Life cycle of the resource position r_i.

a resource position $r_i = \varphi_n + \varphi_k$ for some $k < n$ is illustrated. As defined in formula (3.8), the resource position starts to exist during the forward computation or is deleted during the reverse computation, if the leading process reaches the step number $l_O(r_i) = r_i + 2$. If the leading process reaches the step number $l_I(r_i)$, then the resource assigned to the resource position r_i is deleted during the forward computation and is created during the reverse computation. In this case, it is computed by formula (3.17), i.e. $l_I(r_i) = l_I(r_i - \varphi_n) + \varphi_n = 2\varphi_{k+1} - 1 + \varphi_n$.

Due to the different meaning of the step numbers $l_O(r_i)$ and $l_I(r_i)$ during the forward and the reverse computation, these are labelled with O for the outer step number, i.e. close to the leading process, and I for the inner step number, away from the leading process. During the reverse computation, the resource changes from a process resource to a checkpoint resource at the step number $l_M(r_i)$ of the leading process. This step lies somewhere in the middle of the life cycle interval $[l_O(r_i), l_I(r_i)]$. Another meaning for this step number $l_M(r_i)$ is given by the point where the resource position of the forward and the backward computation merge. Both are indicated by the index M. A precise description of the step number $l_M(r_i)$ and how it can be used to extend the formula for the resource distribution (Definition 3.1) is given in the next section.

3.4 Extended Instantaneously Reversible Distributions

In order to get a more general characterisation for instantaneously reversible resource distributions the results above are extended. Since the resource distributions of the reverse computation are instantaneously reversible by definition, the extension will lead to a definition including the resource distributions of the forward computation as well as the resource distributions of the reverse computation.

Using the results in Section 3.1 and Section 3.3, one can extend the formula for the resource distribution (Definition 3.1). Again, a step number $l \in [\varphi_n, \varphi_{n+1})$ is given. Instead of specific resource positions, intervals of possible resource positions are used. These intervals are defined by

$$\mathbf{r}_i(l) = \Big[\min\big(r_i, r_i - l + l_M(r_i)\big), r_i\Big] \quad . \tag{3.18}$$

Thereby, it is assumed w.l.o.g. that $r_i \in [\varphi_m, \varphi_{m+1})$ for a $m \in \mathbb{N}$ and the step number $l_M(r_i)$ are defined by

$$l_M(r_i) = \begin{cases} \varphi_{m-1} + \varphi_{m+1} & \text{if } r_i = \varphi_m \ , \\ l_M(r_i - \varphi_m) + \varphi_m & \text{if } r_i \in (\varphi_m, \varphi_{m+1}) \ . \end{cases} \tag{3.19}$$

The values r_i are computed by the formula for the resource distribution (Definition 3.1) for the given step number l. The step number $l_M(r_i)$ defines the step number of the leading process when the resource connected to the resource position r_i changes from a computing resource to a checkpoint resource, as described in Section 3.3. Figure 3.11 shows the intervals of the resource positions (grey shaded areas) up to $l = 34$. One important property of these intervals

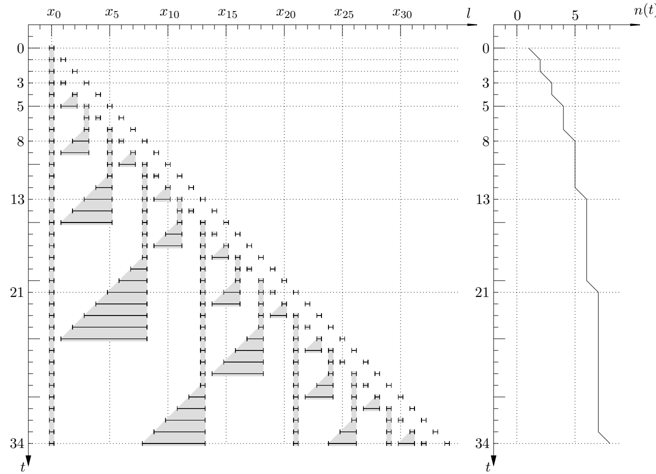


Figure 3.11: Intervals of instantaneous reversible resource distributions.

is that they do not intersect, which is shown by the following corollary.

Corollary 3.3
For an arbitrary step number $l \in [\varphi_n, \varphi_{n+1})$ and $n \in \mathbb{N}$, let the resource intervals be defined by (3.18) and (3.19). Then for all $i \in [1, n)$ one has

$$\mathbf{r}_{i-1}(l) \cap \mathbf{r}_i(l) = \emptyset \ .$$

□

Proof:
It must be shown that the upper bound of the interval $\mathbf{r}_{i-1}(l)$ is always smaller than the lower bound of the interval $\mathbf{r}_i(l)$. By definition (3.2), one knows $r_i(l) > r_{i-1}(l)$. Therefore, only the step number l is of interest, where the minimum in (3.18) is formed by $r_i - l + l_M(r_i)$, i.e. where $l_M(r_i) \leq l$. Hence, l is restricted to $l \in [l_M(r_i), l_I(r_i)]$. The statement of Corollary 3.3 can be rewritten as

$$r_i - l + l_M(r_i) > r_{i-1} \qquad (3.20)$$

for $l \in [l_M(r_i), l_I(r_i)]$ and $i \in [1, n)$.

First it is assumed that $r_i = \varphi_m$. The inequality (3.20) changes to

$$\varphi_m - l + \varphi_{m+1} + \varphi_{m-1} = 2\varphi_{m+1} - l > r_{i-1}$$

which yields $l \in [\varphi_{m+1} + \varphi_{m-1}, 2\varphi_{m+1} - 1]$. In order to obtain the correct definition of r_i, one must subdivide the range of l into the parts $l \in [\varphi_{m+1} + \varphi_{m-1}, \varphi_{m+2})$ and $l \in [\varphi_{m+2}, \varphi_{m+2} + \varphi_{m-1} - 1]$. Using the formula for the resource distribution (Definition 3.1) within the two given intervals, one notes that the only case in which $r_i = \varphi_m$ is if $i = 1$. Hence, $r_{i-1} = r_0 = 0$. Then the proposition can be rewritten as

$$2\varphi_{m+1} - l > 0$$

for $l \in [\varphi_{m+1} + \varphi_{m-1}, 2\varphi_{m+1} - 1]$. Using the maximal value for $l = 2\varphi_{m+1} - 1$, the proposition is still true.

Secondly, it is assumed that $r_i \in (\varphi_m, \varphi_{m+1})$. Using Corollary 3.2, one knows that a resource position ϱ_j exists such that $r_i(l) = \varrho_j(\lambda) + \varphi_m$. Additionally, the interval $[l_M(r_i), l_I(r_i)]$ of possible step numbers l is shifted by φ_m to the interval $[l_M(\varrho_j), l_I(\varrho_j)]$ for possible step numbers λ, as a result of $l_M(r_i) = l_M(r_i - \varphi_m) + \varphi_m = l_M(\varrho_j) + \varphi_m$ and $l_I(r_i) = l_I(r_i - \varphi_m) + \varphi_m = l_I(\varrho_j) + \varphi_m$.

Next, the upper bound of the interval $\mathbf{r}_{i-1}(l)$ must be determined. The smallest possible value for this bound is $r_{i-1} = \varphi_m$. To prove this, one has to show that for all resource positions $r_i \in (\varphi_m, \varphi_{m+1})$, the resource position at φ_m is already set when the resource assigned to r_i is created and that the resource position at φ_m is still the resource assigned to r_i when it is deleted, i.e. one has to show that $l_O(\varphi_m) < l_O(r_i)$ and $l_I(\varphi_m) > l_I(r_i)$. The first statement

is true by definition, since $l_O(\varphi_m) = \varphi_m + 2$, $l_O(r_i) = r_i + 2$, and $\varphi_m < r_i$. The second statement is true because

$$\begin{array}{lrcl} & l_I(\varphi_m) & > & l_I(r_i) \\ \Longleftrightarrow & l_I(\varphi_m) & > & \varphi_m + l_I(\varrho_j) \\ \Longleftrightarrow & 2\varphi_{m+1} - 1 & > & \varphi_m + 2\varphi_m - 1 \\ \Longleftrightarrow & \varphi_{m-1} & > & \varphi_{m-2} \ . \end{array}$$

Now, two possibilities to set the resource position r_{i-1} must be considered. First, if $r_{i-1} = \varphi_m$, then (3.20) changes to $r_i - l + l_M(r_i) > \varphi_m$, and by shifting about φ_m, it is equivalent to $\varrho_j - \lambda + l_M(\varrho_j) > 0$. If the resource position $r_{i-1} \in (\varphi_m, \varphi_{m+1})$, then the problem can be converted into an equivalent problem by a φ_m-shift because

$$\begin{array}{lrcl} & r_i - l + l_M(r_i) & > & r_{i-1} \\ \Longleftrightarrow & \varrho_j + \varphi_m - (\lambda + \varphi_m) + l_M(\varrho_j + \varphi_m) + \varphi_m & > & \varrho_{j-1} + \varphi_m \\ \Longleftrightarrow & \varrho_j - \lambda + l_M(\varrho_j) & > & \varrho_{j-1} \ . \end{array}$$

Hence, the statement (3.20) is proven recursively. ■

The actual purpose of defining intervals $\mathbf{r}_i(l)$ is the instantaneous reversibility of the resulting resource distributions. An equivalent lemma to Lemma 3.1 can be stated.

Lemma 3.3
Resource distributions $\mathfrak{r}(l)$ *satisfying*

$$\mathfrak{r}(l) = \left\{ r_i \mid r_i \in \mathbf{r}_i(l) \ \wedge \ i < n \ \wedge \ i, n \in \mathbb{N} \right\}$$

are instantaneously reversible. □

The second condition ($i < n$) of the resource position r_i ensures that there is only one resource within each resource interval. The statement of Lemma 3.1 is a restriction of Lemma 3.3 to the upper bound. The aim is to prove that

$$l \geq 2r_i - r_{i-1} \ ,$$

which is similar to the inequality (3.5). Different to inequality (3.5), the resource positions r_i can be any resource position within the interval $\mathbf{r}_i(l)$, and the resource positions r_{i-1} can be any resource position within the interval $\mathbf{r}_{i-1}(l)$. Nevertheless, the procedure proving Lemma 3.3 is equivalent to the proof of Lemma 3.1.

In an extra corollary simular to Corollary 3.1, substructures were exploited first. Again, it is assumed that the step number l lies within the interval $[\varphi_n, \varphi_{n+1})$. In addition to this, the interval is divided into parts, but in contrast to Corollary 3.1, these intervals are given by

- $l \in [\varphi_n, \varphi_n + \varphi_{n-3})$

- $l \in [\varphi_n + \varphi_{n-3}, \varphi_n + \varphi_{n-2})$ and
- $l \in [\varphi_n + \varphi_{n-2}, \varphi_n + \varphi_{n-1}) = [\varphi_n + \varphi_{n-2}, \varphi_{n+1})$.

The resource positions equal to or larger than φ_{n-1} of the first interval (labelled ① in Figure 3.12(a)) have their equivalent within the resource positions lying in the interval $[\varphi_{n-2}, \varphi_{n-1})$ of the step number (①' in Figure 3.12(a)), shifted by φ_{n-1}. Considering the two other intervals, only the resource positions larger than φ_n will be looked at (labelled ② and ③ in Figure 3.12(a)). These resource positions will be shifted by φ_n, and their equivalents will be found between the step number φ_{n-3} and φ_{n-1} (②' and ③' in Figure 3.12(a)). Using the notation of the proof of Corollary 3.2, the new corollary will state that

$$Te(\varphi_{n-1}, \varphi_n, \varphi_{\varphi_n+\varphi_{n-3}}) = \mathcal{S}(Te(0, \varphi_{n-2}, \varphi_{n-1}), \varphi_{n-1}) \tag{3.21}$$

and

PSfrag replacements

$$Te(\varphi_n, \varphi_n + \varphi_{n-3}, \varphi_{n+1}) = \mathcal{S}(Te(0, \varphi_{n-3}, \varphi_{n-1}), \varphi_n) \ . \tag{3.22}$$

PSfrag replacements

Using the notations of Figure 3.12(a), one can state ① $= \mathcal{S}($①$', \varphi_{n-1})$, as well as ② + ③ $=$ $\mathcal{S}($②$' + $③$', \varphi_n)$. Having proved this corollary, it remains to be shown that the left over resource positions (dark grey area in Figure 3.12(a)) satisfy the inequality of instantaneous reversibility (3.5).

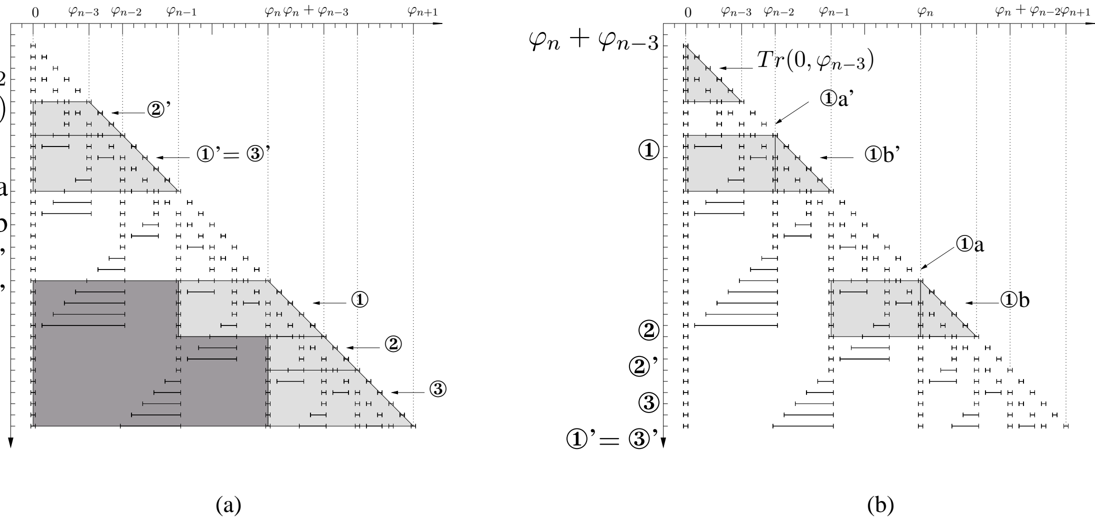


Figure 3.12: Proof sketch for Corollary 3.4.

Corollary 3.4
Let $l \in [\varphi_n, \varphi_{n+1})$ and $n \geq 3$ with the resource distribution intervals $\mathfrak{R}(l) = \{\mathbf{r}_i(l) \,|\, i \in [0, n)\}$ and

$$\lambda = \begin{cases} l - \varphi_{n-1} & \text{if } l \in [\varphi_n, \varphi_n + \varphi_{n-3}) \ , \\ l - \varphi_n & \text{if } l \in [\varphi_n + \varphi_{n-3}, \varphi_{n+1}) \end{cases}$$

with the resource distribution intervals $\mathfrak{R}(\lambda) = \{\mathbf{r}_i(\lambda) \,|\, i \in [0, n-d)\}$. *Then, for every resource interval* $\mathbf{r}_j(\lambda) \in \mathfrak{R}(\lambda)$, *there exists a unique resource interval* $\mathbf{r}_i(l) \in \mathfrak{R}(l)$ *such that*

$$\mathbf{r}_i(l) = \mathbf{r}_{j+d}(l) = \begin{cases} \mathbf{r}_j(\lambda) + \varphi_{n-1} & \textit{with } d = 2 \textit{ if } l \in [\varphi_n, \varphi_n + \varphi_{n-3}) \ ; \\ \mathbf{r}_j(\lambda) + \varphi_n & \textit{with } d = 3 \textit{ if } l \in [\varphi_n + \varphi_{n-3}, \varphi_n + \varphi_{n-2}) \ ; \\ \mathbf{r}_j(\lambda) + \varphi_n & \textit{with } d = 2 \textit{ if } l \in [\varphi_n + \varphi_{n-2}, \varphi_{n+1}) \ . \end{cases}$$

□

Proof:
As with the proof of Corollary 3.1, the Greek written variables belong to the resource interval $\mathfrak{R}(\lambda)$ for λ.

First, it will be shown that the tetragons ①, i.e. $Te(\varphi_{n-1}, \varphi_n, \varphi_{\varphi_n + \varphi_{n-3}})$, and ②+③, i.e. $Te(\varphi_n, \varphi_n + \varphi_{n-3}, \varphi_{n+1})$, include all possible resource positions r_i with $i \geq d$. An equivalent statement is that for all resource intervals $\mathbf{r}_i(l)$ with the $i \geq d$, the left-most bound lies within the tetragons ①, ② and ③. This bound is the resource positions r_d. By using the formula of the resource distribution (Definition 3.1), one can verify that

$$\begin{array}{llll} \text{if} & l \in [\varphi_n, \varphi_n + \varphi_{n-3}) & \Longrightarrow \quad r_2 = \varphi_{n-1} & , \\ \text{if} & l \in [\varphi_n + \varphi_{n-3}, \varphi_n + \varphi_{n-2}) & \Longrightarrow \quad r_3 = \varphi_n & \text{and} \\ \text{if} & l \in [\varphi_n, \varphi_n + \varphi_{n-1}) = [\varphi_n, \varphi_{n+1}) & \Longrightarrow \quad r_2 = \varphi_n & . \end{array}$$

Since the resource intervals do not intersect and all resource intervals $\mathbf{r}_i(l)$ with $i > d$ cover larger resource positions than r_d, it has been shown that the left-most bound r_d lies within the tetragon $Te(\varphi_{n-1}, \varphi_n, \varphi_{\varphi_n + \varphi_{n-3}})$, the tetragon $Te(\varphi_n, \varphi_n + \varphi_{n-3}, \varphi_n + \varphi_{n-2})$, and the tetragon $Te(\varphi_n, \varphi_n + \varphi_{n-2}, \varphi_{n+1})$. This can be achieved by showing that $\mathbf{r}_d(l) = [r_d, r_d]$ for all three step ranges, i.e. $l_M(r_d) \geq l$, such that

$$\begin{array}{llll} l \in [\varphi_n, \varphi_n + \varphi_{n-3}) & \Longrightarrow \quad l_M(r_2) = \varphi_{n-2} + \varphi_n - 1 & \geq & l \\ l \in [\varphi_n + \varphi_{n-3}, \varphi_n + \varphi_{n-2}) & \Longrightarrow \quad l_M(r_3) = \varphi_{n-1} + \varphi_{n+1} - 1 & \geq & l \\ l \in [\varphi_n + \varphi_{n-2}, \varphi_{n+1}) & \Longrightarrow \quad l_M(r_2) = \varphi_{n-1} + \varphi_{n+1} - 1 & \geq & l \ . \end{array}$$

Using the result of the Corollary 3.1 once for the tetragon $Te(\varphi_{n-1}, \varphi_n, \varphi_n + \varphi_{n-3}) \subset Te(\varphi_{n-1}, \varphi_n, \varphi_{n+1})$, or twice for $Te(\varphi_n, \varphi_n + \varphi_{n-3}, \varphi_{n+1})$, the result is proven for the upper bound of each interval $\mathbf{r}_i(l) \in \mathfrak{R}(l)$. To prove the lower bound, one has to show that $l_M(r_i) = l_M(\varrho_j) + \varphi_{n-1}$ for the case ①, or that $l_M(r_i) = l_M(\varrho_j) + \varphi_n$ for the cases ② and ③. Providing this is true, one can rewrite the interval definition (3.18) as

$$\begin{aligned} \mathbf{r}_i(l) &= \Big[\min\big(r_i, r_i - l + l_M(r_i)\big), r_i\Big] \\ &= \Big[\min\big(\varrho_j + \varphi_{n-1}, (\varrho_j + \varphi_{n-1}) - (\lambda + \varphi_{n-1}) + l_M(\varrho_j) + \varphi_{n-1}\big), \varrho_j + \varphi_{n-1}\Big] \\ &= \Big[\min\big(\varrho_j + \varphi_{n-1}, \varrho_j - \lambda + l_M(\varrho_j) + \varphi_{n-1}\big), \varrho_j + \varphi_{n-1}\Big] \\ &= \Big[\min\big(\varrho_j, \varrho_j - \lambda + l_M(\varrho_j)\big) + \varphi_{n-1}, \varrho_j + \varphi_{n-1}\Big] \\ &= \Big[\min\big(\varrho_j, \varrho_j - \lambda + l_M(\varrho_j)\big), \varrho_j\Big] + \varphi_{n-1} \\ &= \mathbf{r}_j(\lambda) + \varphi_{n-1} \end{aligned}$$

for the case that $l \in [\varphi_n, \varphi_n + \varphi_{n-3})$. Equally, if $l \in [\varphi_n + \varphi_{n-3}, \varphi_{n+1})$, then $\mathbf{r}_i(l) = \mathbf{r}_j(\lambda) + \varphi_n$. Using the definition of $l_M(r_i)$, one notes that the lower bounds of all resource intervals within the area $Te(\varphi_n, \varphi_n + \varphi_{n-3}, \varphi_{n+1})$ are equivalent to all lower bounds of all resources intervals of the area $Te(0, \varphi_{n-3}, \varphi_{n-1})$ shifted by φ_n.

By definition, for all l it is known that $l_M(l) = l_M(\lambda) + \varphi_n$. Hence, ② is equivalent to ②' and ③ is equivalent to ③', both shifted by φ_n, i.e. (3.21) is proven.

Since there are resources r with $r = \varphi_n$ in the area ①, some more steps are required to prove that ①, i.e. $Te(\varphi_{n-1}, \varphi_n, \varphi_n + \varphi_{n-3})$, is equivalent to $Te(0, \varphi_{n-2}, \varphi_{n-1})$ shifted by φ_{n-1} (labelled ①' in Figure 3.12(b)). The area $Te(\varphi_{n-1}, \varphi_n, \varphi_n + \varphi_{n-3})$ has to be divided into the area $Rec(\varphi_{n-1}, \varphi_n, \varphi_n + \varphi_{n-3})$ (labelled ①a in Figure 3.12(b)) containing the resource positions $\mathbf{r}_i(l) \subset [\varphi_{n-1}, \varphi_n]$ and $l \in [\varphi_n, \varphi_n + \varphi_{n-3})$ and the area $Tr(\varphi_n, \varphi_n + \varphi_{n-3})$ (labelled ①b in Figure 3.12(b)), containing the resource positions $\mathbf{r}_i(l) \subset [\varphi_n, \varphi_n + \varphi_{n-3}]$. Again, the area ①a, i.e. $Rec(\varphi_{n-1}, \varphi_n, \varphi_n + \varphi_{n-3})$, can be shifted by φ_{n-1}, i.e. ①a $= \mathcal{S}($①a'$, \varphi_{n-1})$, because of the Corollary 3.1 used for the upper bounds of the resource interval, the definition (3.19) of the values $l_M(r_i)$, and the fact that all resources in that area lie within the interval $(\varphi_{n-1}, \varphi_n)$.

The remaining triangle $Tr(\varphi_n, \varphi_n + \varphi_{n-3})$ can be shifted by φ_n using Corollary 3.2 and the definition (3.19) of $l_M(r_i)$ to the triangle $Tr(0, \varphi_{n-3})$, or formal $Tr(\varphi_n, \varphi_n + \varphi_{n-3}) = \mathcal{S}(Tr(0, \varphi_{n-3}), \varphi_n)$. One actually wants to show , however, that one can carry out a smaller shift such that $Tr(\varphi_n, \varphi_n + \varphi_{n-3}) = \mathcal{S}(Tr(\varphi_{n-2}, \varphi_{n-1}), \varphi_{n-1})$. By using Corollary 3.2 again and the definition (3.19) of $l_M(r_i)$, one can show that

$$Tr(\varphi_{n-2}, \varphi_{n-1}) = \mathcal{S}(Tr(0, \varphi_{n-3}), \varphi_{n-2}) \ .$$

Combining these two results, one finds that

$$\begin{aligned} Tr(\varphi_n, \varphi_n + \varphi_{n-3}) &= \mathcal{S}(Tr(0, \varphi_{n-3}), \varphi_n) \\ &= \mathcal{S}(\mathcal{S}(Tr(\varphi_{n-2}, \varphi_{n-1}), -\varphi_{n-2}), \varphi_n) \\ &= \mathcal{S}(Tr(\varphi_{n-2}, \varphi_{n-1}), \varphi_n - \varphi_{n-2}) \\ &= \mathcal{S}(Tr(\varphi_{n-2}, \varphi_{n-1}), \varphi_{n-1}) \ . \end{aligned}$$

Putting the pieces $Tr(\varphi_n, \varphi_n + \varphi_{n-3})$ and $Rec(\varphi_{n-1}, \varphi_n, \varphi_n + \varphi_{n-3})$ back together yields (3.22). Combining (3.21) and (3.22), the corollary is proven. ■

Now it is assumed that the resource distributions defined by the resource intervals $\mathbf{r}_i(l)$ within the area $Te(0, \varphi_{n-3}, \varphi_{n-1})$ are instantaneously reversible. Since one can shift these areas by φ_{n-1} or φ_n as in Corollary 3.4, the instantaneous reversibility is true for the union of the areas $Te(\varphi_{n-1}, \varphi_n, \varphi_{\varphi_n + \varphi_{n-3}})$ and $Te(\varphi_n, \varphi_n + \varphi_{n-3}, \varphi_{n+1})$. Hence, the final proof of Lemma 3.3 can be obtained by proving that all resource positions defined by the resource intervals $\mathbf{r}_i(l)$ with $l \in [\varphi_n, \varphi_{n+1})$, which are not included in $Te(\varphi_{n-1}, \varphi_n, \varphi_{\varphi_n + \varphi_{n-3}}) + Te(\varphi_n, \varphi_n + \varphi_{n-3}, \varphi_{n+1})$, are instantaneously reversible.

Proof of Lemma 3.3:
Again, the range of $l \in [\varphi_n, \varphi_{n-1})$ will be divided into the three subranges

- $l \in [\varphi_n, \varphi_n + \varphi_{n-3})$,
- $l \in [\varphi_n + \varphi_{n-3}, \varphi_n + \varphi_{n-2})$, and
- $l \in [\varphi_n + \varphi_{n-2}, \varphi_n + \varphi_{n-1}) = [\varphi_n + \varphi_{n-2}, \varphi_{n+1})$.

First, $l \in [\varphi_n, \varphi_n + \varphi_{n-3})$ is considered. The resource positions r_i for $i = [0, 2]$, defined by the formula for the resource distribution (Definition 3.1), are given by $r_0 = 0$, $r_1 = \varphi_{n-2}$ and $r_2 = \varphi_{n-1}$. The resource interval $\mathbf{r}_2(l)$ is defined by $\mathbf{r}_2(l) = [r_2, r_2]$ because $l_M(r_2) = l_M(\varphi_{n-1}) = \varphi_{n-2} + \varphi_n \geq \varphi_n + \varphi_{n-3}$, which is the largest possible value for l within the considered interval. Using the result of Lemma 3.1, one finds that

$$\begin{aligned} 2r_1 - r_0 &= 2\varphi_{n-2} &&= \varphi_{n-1} + \varphi_{n-4} &&\leq l \\ 2r_2 - r_1 &= 2\varphi_{n-1} - \varphi_{n-2} &&= \varphi_{n-1} + \varphi_{n-3} &&\leq l \ . \end{aligned}$$

A similar result can be obtained for the lower bound of the resource interval $\mathbf{r}_1(l)$. Only the distance to r_2 is the point of interest here, because the largest distance between r_0 and r_1 already satisfies the inequality (3.5), as shown above. Thus,

$$\begin{aligned} 2r_2 - \big(r_1 - l + l_M(r_1)\big) &= 2\varphi_{n-1} - \big(\varphi_{n-2} - l + \varphi_{n-3} + \varphi_{n-1}\big) \\ &= 2\varphi_{n-1} - \varphi_{n-2} + l - \varphi_{n-3} - \varphi_{n-1} \\ &= \varphi_{n-1} - \varphi_{n-2} + l - \varphi_{n-3} \\ &= l \ . \end{aligned}$$

Hence, for all possible combinations of the resource distributions $r_i \in \mathbf{r}_i(l)$ for $i = [0, 2]$ and $l \in [\varphi_n, \varphi_n + \varphi_{n-3})$, the inequality (3.5) is true.

Secondly, $l \in [\varphi_n + \varphi_{n-3}, \varphi_n + \varphi_{n-2})$ is assumed. Again, the resource interval defining resource positions r_i for $i = [0, d]$ has to be defined first. Since $d = 3$, the possible resource positions are given by $r_0 = 0$, $r_1 = \varphi_{n-1}$, $r_2 = \varphi_{n-1} + \varphi_{n-3}$ and $r_3 = \varphi_n$. Applying the proof of Lemma 3.1 for the upper bounds of the resource intervals, one obtains

$$\begin{aligned} 2r_1 - r_0 &= 2\varphi_{n-1} &&= \varphi_n + \varphi_{n-3} &&\leq l \\ 2r_2 - r_1 &= 2(\varphi_{n-1} + \varphi_{n-3}) - \varphi_{n-1} &&= \varphi_n + \varphi_{n-5} &&\leq l \\ 2r_3 - r_2 &= 2\varphi_n - (\varphi_{n-1} + \varphi_{n-3}) &&= \varphi_n + \varphi_{n-4} &&\leq l \ . \end{aligned}$$

Using the definition (3.19) of $l_M(r_i)$, the intervals $\mathbf{r}_1(l)$ and $\mathbf{r}_3(l)$ only consist of one resource position r_1 and r_3 respectively. Hence, one has to prove inequality (3.5) for all possible distances between a resource position out of the interval $\mathbf{r}_2(l)$ and the resource position r_3. Inequality (3.5) can be shown as

$$\begin{aligned} 2r_3 - \big(r_2 - l + l_M(r_2)\big) &= 2\varphi_n - \big(\varphi_{n-1} + \varphi_{n-3} - l + \varphi_{n-1} + (\varphi_{n-2} + \varphi_{n-4})\big) \\ &= 2\varphi_n - \varphi_{n-1} - \varphi_{n-3} + l - \varphi_{n-1} - \varphi_{n-2} - \varphi_{n-4} \\ &= 2\varphi_n - 2\varphi_{n-1} + l - 2\varphi_{n-2} \\ &= l \ . \end{aligned}$$

Thus, $2r_3 - \big(r_2 - l + l_M(r_2)\big) \leq l$ and therefore, inequality (3.5) is true for all possible resource distributions defined by the resource intervals $\mathbf{r}_i(l)$ for $i = [0, 3]$ and $l \in [\varphi_n + \varphi_{n-3}, \varphi_n + \varphi_{n-2})$.

Finally to complete the proof, $l \in [\varphi_n + \varphi_{n-2}, \varphi_{n+1})$ is considered. As in the two previous cases, the upper bound of the resource intervals $\mathbf{r}_i(l)$ for $i = [0, 2]$ is considered. They are given by $r_0 = 0$, $r_1 = \varphi_{n-1}$ and $r_2 = \varphi_n$. Stressing the proof of Lemma 3.1 once again, one finds that

$$\begin{array}{rclclcl} 2r_1 - r_0 & = & 2\varphi_{n-1} & = & \varphi_n + \varphi_{n-3} & \leq & l \\ 2r_2 - r_1 & = & 2\varphi_n - \varphi_{n-1} & = & \varphi_n + \varphi_{n-2} & \leq & l \ . \end{array}$$

Since $\mathbf{r}_2(l)$ contains only one resource position, one has to prove that $2r_2 - (r_1 - l + l_M(r_1)) \leq l$, which is true because of

$$\begin{aligned} 2r_2 - \big(r_1 - l + l_M(r_1)\big) &= 2\varphi_{n-1} - \big(\varphi_{n-2} - l + \varphi_{n-3} + \varphi_{n-1}\big) \\ &= 2\varphi_{n-1} - \varphi_{n-2} + l - \varphi_{n-3} - \varphi_{n-1} \\ &= \varphi_{n-1} - \varphi_{n-2} + l - \varphi_{n-3} \\ &= l \ , \end{aligned}$$

which is less than or equal to l. Thus, all resource position distributions defined by the resource intervals $\mathbf{r}_i(l)$ for $i = [0, 2]$ satisfy the inequality (3.5).

For all three parts, it is shown that the first d (with $d = 2$ or $d = 3$) resource intervals satisfy the inequality (3.5). Furthermore, by Corollary 3.4 it was shown, that for each l a λ can be found, such that the resource distribution intervals for λ can be shifted by φ_n or φ_{n-1} to a subset of the resource distribution intervals for l (more precise to the last $n - d$ resource distribution intervals of $\mathfrak{R}(l)$). Using these two facts recursively, the lemma is proven. ■

3.5 Properties of Instantaneously Reversible Distributions

The structure and the properties of instantaneously reversible distributions can be explored further by analysing the intervals $[a_i, b_i)$. The size of the intervals will be analysed first. Using results about the size of the intervals, one can derive a visualisation of the computation of the instantaneously reversible resource distributions (Definition 3.1).

Theorem 3.1
Let the interval definition for the computation of an instantaneously reversible resource distribution be given by (3.1). Then the size of these intervals can be obtained by

$$(3.23) \qquad b_i - a_i = \begin{cases} \varphi_{n-i-1} & \text{if } a_{i-1} = \tilde{b}_{i-1} \ , & \text{(a)} \\ \varphi_{n-i-2} & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \text{ and } l < \tilde{b}_{i-1} \ , & \text{(b)} \\ \varphi_{n-i-1} & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \text{ and } l \geq \tilde{b}_{i-1} \ . & \text{(c)} \end{cases}$$

□

The cases (3.23(a)), (3.23(b)) and (3.23(c)) are identical to the cases distinguished in (3.1). The proof is done by induction over i.

Proof of Theorem 3.1:

Induction base case ($i = 1$): In order to compute this induction base case, one has to look for the case $i = 0$ first. The interval is initially given by $[a_0, b_0) = [\varphi_n, \varphi_{n+1})$. Hence, one obtains $b_0 - a_0 = \varphi_{n-1}$. Using these interval limits, one can compute for $i = 1$ the value $\tilde{b}_0$ by $\tilde{b}_0 = b_0 - \varphi_{n-2} = \varphi_{n+1} - \varphi_{n-2} = \varphi_n + \varphi_{n-3}$. Since $\tilde{b}_0 = \varphi_n + \varphi_{n-3} > \varphi_n = a_0$, the case (3.23(a)) for $i = 1$ cannot happen. The two other cases yield

$$\begin{array}{lllcll} & \begin{array}{ll} a_1 = a_0 & = \varphi_n \\ b_1 = \tilde{b}_0 & = \varphi_n + \varphi_{n-3} \end{array} & \Longrightarrow & b_1 - a_1 = \varphi_{n-3} = \varphi_{n-i-2} \ , & \text{(b)} \\ (3.24) & & & & \\ & \begin{array}{ll} a_1 = \tilde{b}_0 & = \varphi_n + \varphi_{n-3} \\ b_1 = b_0 & = \varphi_{n+1} \end{array} & \Longrightarrow & b_1 - a_1 = \varphi_{n-1} = \varphi_{n-i-1} \ . & \text{(c)} \end{array}$$

Since not all possibilities of (3.23) are covered, one has to look at the next value for i.

Induction base case ($i = 2$): First one obtains that $\tilde{b}_1$ is given by $\tilde{b}_1 = b_1 - \varphi_{n-3}$. From (3.24(b)) follows $\tilde{b}_1 = b_1 - \varphi_{n-3} = \varphi_n + \varphi_{n-3} - \varphi_{n-3} = a_1$. Hence, the only choice is (3.23(a)). This means that $a_2 = a_1 = \varphi_n$, $b_2 = b_1 = \varphi_n + \varphi_{n-3}$ and, therefore, $b_2 - a_2 = \varphi_{n-3} = \varphi_{n-i-1}$. For the case (3.24(c)) one can compute $\tilde{b}_1 = b_1 - \varphi_{n-3} = \varphi_{n+1} - \varphi_{n-3} = \varphi_n + \varphi_{n-2}$. For the choice (3.23(b)), it follows that $a_2 = \tilde{b}_1 = \varphi_n + \varphi_{n-3}$ and $b_2 = b_1 = \varphi_n + \varphi_{n-2}$ and, hence, $b_1 - a_1 = \varphi_{n-4} = \varphi_{n-i-2}$. For the choice (3.23(c)) it follows that $a_2 = \tilde{b}_1 = \varphi_n + \varphi_{n-2}$, as well as $b_2 = b_1 = \varphi_{n+1}$, such that $b_1 - a_1 = \varphi_{n-3} = \varphi_{n-i-1}$. Overall, one obtains that

$$(3.25) \qquad b_2 - a_2 = \begin{cases} b_1 - a_1 = \varphi_{n-3} & \text{if } a_1 = \tilde{b}_1 \ , & \text{(a)} \\ \tilde{b}_1 - a_1 = \varphi_{n-4} & \text{if } a_1 \neq \tilde{b}_1 \text{ and } l < \tilde{b}_1 \ , & \text{(b)} \\ b_1 - \tilde{b}_1 = \varphi_{n-3} & \text{if } a_1 \neq \tilde{b}_1 \text{ and } l \geq \tilde{b}_1 \ . & \text{(c)} \end{cases}$$

Having this induction base case, one can establish an induction assumption.

Induction assumption: Now it is assumed that

$$(3.26) \qquad b_i - a_i = \begin{cases} b_{i-1} - a_{i-1} = \varphi_{n-i-1} & \text{if } a_{i-1} = \tilde{b}_{i-1} \ , & \text{(a)} \\ \tilde{b}_{i-1} - a_{i-1} = \varphi_{n-i-2} & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \text{ and } l < \tilde{b}_{i-1} \ , & \text{(b)} \\ b_{i-1} - \tilde{b}_{i-1} = \varphi_{n-i-1} & \text{if } a_{i-1} \neq \tilde{b}_{i-1} \text{ and } l \geq \tilde{b}_{i-1} \ . & \text{(c)} \end{cases}$$

is true. Then the induction hypothesis can be given as

$$(3.27) \qquad b_{i+1} - a_{i+1} = \begin{cases} b_i - a_i = \varphi_{n-i-2} & \text{if } a_i = \tilde{b}_i \ , & \text{(a)} \\ \tilde{b}_i - a_i = \varphi_{n-i-3} & \text{if } a_i \neq \tilde{b}_i \text{ and } l < \tilde{b}_i \ , & \text{(b)} \\ b_i - \tilde{b}_i = \varphi_{n-i-2} & \text{if } a_i \neq \tilde{b}_i \text{ and } l \geq \tilde{b}_i \ . & \text{(c)} \end{cases}$$

In order to prove this hypothesis, one has to go through all possible combinations in (3.26) and (3.27).

First the case (3.26(a)) is assumed. The interval $[a_i, b_i)$ is given by $[a_{i-1}, b_{i-1})$, and $\tilde{b}_i$ is given by $b_i - \varphi_{n-i-2} = b_{i-1} - \varphi_{n-i-2}$. Since $a_i = a_{i-1} = b_{i-1} - \varphi_{n-i-1}$ and $\tilde{b}_i = b_{i-1} - \varphi_{n-i-2} > b_{i-1} - \varphi_{n-i-1} = a_i$, one can conclude that case (3.27(a)) cannot follow the case (3.26(a)). For the case (3.27(b)) one obtains

$$b_{i+1} - a_{i+1} = \tilde{b}_i - a_i = b_{i-1} - \varphi_{n-i-2} - a_{i-1} = \varphi_{n-i-1} - \varphi_{n-i-2} = \varphi_{n-i-3}$$

and for the case (3.27(c))

$$b_{i+1} - a_{i+1} = b_i - \tilde{b}_i = b_{i-1} - (b_{i-1} - \varphi_{n-i-2}) = \varphi_{n-i-2} \ .$$

Now the case (3.26(b)) is assumed. The interval $[a_i, b_i)$ is given by $[a_{i-1}, \tilde{b}_{i-1})$, and $\tilde{b}_i$ is given by $b_i - \varphi_{n-i-2} = \tilde{b}_{i-1} - \varphi_{n-i-2} = b_{i-1} - \varphi_{n-i-1} - \varphi_{n-i-2} = b_{i-1} - \varphi_{n-i}$. The left interval limit a_i can be calculated using (3.26(b)) by $a_i = a_{i-1} = \tilde{b}_{i-1} - \varphi_{n-i-2} = b_{i-1} - \varphi_{n-i} = \tilde{b}_i$. That means that assuming case (3.26(b)), only case (3.27(a)) can happen. Now, the size of the interval $[a_{i+1}, b_{i+1})$ can be computed by

$$b_{i+1} - a_{i+1} = b_i - a_i = b_i - (b_i - \varphi_{n-i-2}) = \varphi_{n-i-2} \ .$$

Finally, the case (3.26(c)) is assumed. The interval $[a_i, b_i)$ is given by $[\tilde{b}_{i-1}, b_{i-1})$, and $\tilde{b}_i$ is given by $b_i - \varphi_{n-i-2} = b_{i-1} - \varphi_{n-i-2}$. Since $a_i = \tilde{b}_{i-1} = b_{i-1} - \varphi_{n-i-1} < \tilde{b}_i = b_{i-1} - \varphi_{n-i-2}$, the case (3.26(a)) cannot occur. For the case (3.27(b)) one obtains

$$b_{i+1} - a_{i+1} = \tilde{b}_i - a_i = b_{i-1} - \varphi_{n-i-2} - (b_{i-1} - \varphi_{n-i-1}) = \varphi_{n-i-3}$$

and for the case (3.27(c))

$$b_{i+1} - a_{i+1} = b_i - \tilde{b}_i = b_{i-1} - (b_{i-1} - \varphi_{n-i-2}) = \varphi_{n-i-2} \ .$$

Thus, for all possible combinations of (3.26) and (3.27) the induction hypothesis is shown, which completes the proof. ■

The resource distribution definition (Definition 3.1) can be visualised by a decision graph as shown in Figure 3.13. Thereby, every possible choice in (3.1) and (3.2) is assigned to a vertex of the graph, visualised by the upper label. Two additional vertices, the start vertex S and the end vertex E, are needed.

Due to the exclusion of some combination of (3.26) and (3.27), (compare to the proof of Theorem 3.1) not all possible edges exist within the graph. The lower label of the vertices in Figure 3.13 represents the action to be done with the resource position r_i, if the vertex is reached.

The addition of two to the current resource positions at the end vertex E is equivalent to the assignment of l to the resource position r_{n-1}, i.e. $r_{n-1} - r_{n-2} = 2$ for any resource distribution $\mathfrak{r}(l)$. This can be clarified by checking the first resource distributions up to $l = 4$ explicitly (proof of Lemma 3.1 Formula (3.7)). For all step numbers larger than four, one uses the technique of shifting the resource distribution as done in the proof of Corollary 3.2.

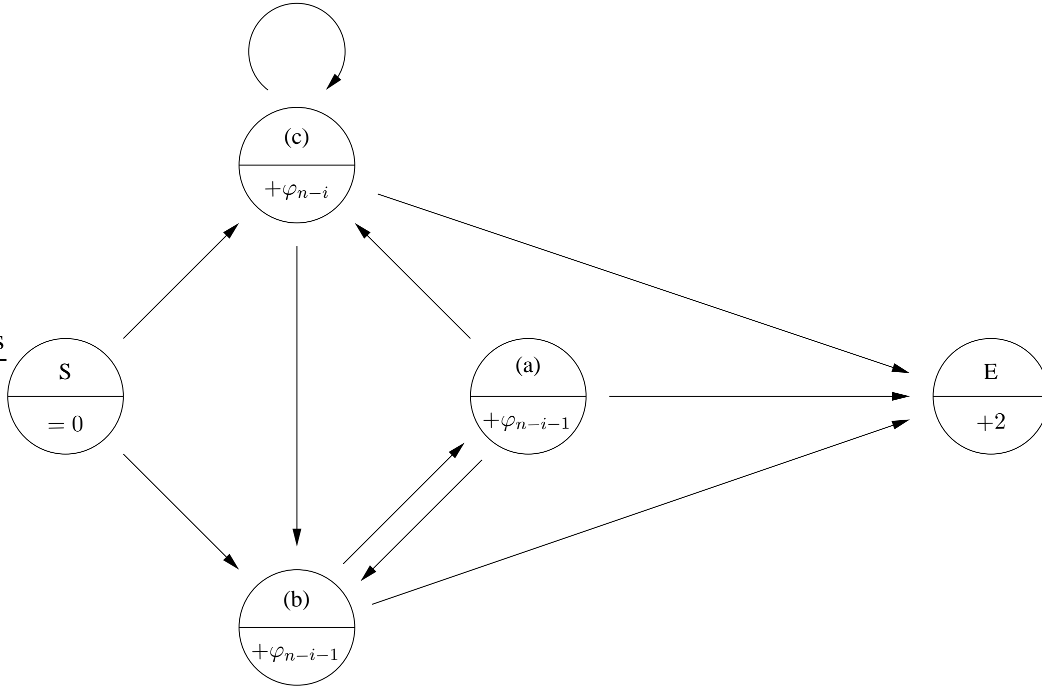


Figure 3.13: Graph for the computation of instantaneously reversible resources distributions (Defini- tion 3.1.) – decision graph

Example 3.2. Let the number of resources be given by $n = 5$. It follows that the step numbers which can be reversed lie between $l \in [\varphi_5, \varphi_6) = [8, 13)$. A path can be given by an n-tuple $\mathfrak{p} = (v_0, v_1, \ldots, v_{n-1})$ of vertices, which lie on the path. Now the resource distribution $\mathfrak{r}(l)$ and the corresponding paths are given by

$$\begin{array}{lclcl} \mathfrak{r}(8) & = & \{0,3,5,6,8\} & \Longrightarrow & \mathfrak{p} = (S, \text{(b)}, \text{(a)}, \text{(b)}, E) \\ \mathfrak{r}(9) & = & \{0,3,5,7,9\} & \Longrightarrow & \mathfrak{p} = (S, \text{(b)}, \text{(a)}, \text{(c)}, E) \\ \mathfrak{r}(10) & = & \{0,5,7,8,10\} & \Longrightarrow & \mathfrak{p} = (S, \text{(c)}, \text{(b)}, \text{(a)}, E) \\ \mathfrak{r}(11) & = & \{0,5,8,9,11\} & \Longrightarrow & \mathfrak{p} = (S, \text{(c)}, \text{(c)}, \text{(b)}, E) \\ \mathfrak{r}(12) & = & \{0,5,8,10,12\} & \Longrightarrow & \mathfrak{p} = (S, \text{(c)}, \text{(c)}, \text{(c)}, E) \ . \end{array}$$

Overall, one can conclude the following theorem.

Theorem 3.2

Let $\mathfrak{r}(l)$ be a resource distribution computed by (3.1) and (3.2) for a given step number l and for n resources. Then there exists a unique path through the decision graph, which starts at the vertex S *and ends at the vertex* E *after n steps. Furthermore, the computation of the resource distribution is equivalent to the action described by the path through the decision graph.* □

As already described in Example 3.2, a path is given by an n-tuple $\mathfrak{p} = (v_0, v_1, \ldots, v_{n-1})$ of vertices. The computed resource position sequence is defined by $\mathfrak{s}(\mathfrak{p}) = (w_0, w_1, \ldots, w_{n-1})$. For all paths through the graph, one knows that $v_0 =$ S, $v_{n-1} =$ E, $w_0 = 0$.

Example 3.3. The resource distribution for $l = 23$ is given by

$$\mathfrak{r}(23) = \{r_0, r_1, r_2, r_3, r_4, r_5, r_6\} = \{0, 8, 13, 18, 20, 21, 23\} \ .$$

The corresponding path and the corresponding resource position sequence then are given by

$$\begin{array}{lllll} \mathfrak{p} & = & (v_0, v_1, v_3, v_4, v_5, v_6) & = & \big(\mathrm{S}, (\mathrm{b}), (\mathrm{a}), (\mathrm{c}), (\mathrm{b}), (\mathrm{a}), \mathrm{E}\big) \quad \text{and} \\ \mathfrak{s}(\mathfrak{p}) & = & (w_0, w_1, w_3, w_4, w_5, w_6) & = & \big(0, 8, 13, 18, 20, 21, 23\big) \ . \end{array}$$

One should note, that $\mathfrak{r}(23) = \mathfrak{s}(\mathfrak{p})$.

The example suggests that the reverse direction of the Theorem 3.2 is also true. This assumption is established in the next theorem.

Theorem 3.3
Let $\mathfrak{p} = (v_0, v_1, \ldots, v_{n-1})$ be a path through the decision graph (Figure 3.13), and let the corresponding resource position sequence $\mathfrak{s}(\mathfrak{p})$ be given by $\mathfrak{s}(\mathfrak{p}) = (w_0, w_1, \ldots, w_{n-1})$. Then there exists a unique step number l, such that the resource position sequence is equal to the resource distribution for l computed by (3.1) and (3.2), i.e.

$$\mathfrak{r}(l) = \mathfrak{s}(\mathfrak{p}) \ .$$

□

Proof:
The proof in done by induction over the path length.

Induction base case ($n = 3$ and $n = 4$): For both cases one can show that the assertion of the theorem is true. For $n = 3$ one has

$$\begin{array}{rcl} \mathfrak{p} = (S, (\mathrm{b}), E) & \longrightarrow & \mathfrak{s}(\mathfrak{p}) = (0, 1, 3) = \mathfrak{r}(3) \\ \mathfrak{p} = (S, (\mathrm{c}), E) & \longrightarrow & \mathfrak{s}(\mathfrak{p}) = (0, 2, 4) = \mathfrak{r}(4) \end{array}$$

For $n = 4$ one has

$$\begin{array}{rcl} \mathfrak{p} = (S, (\mathrm{b}), (\mathrm{a}), E) & \longrightarrow & \mathfrak{s}(\mathfrak{p}) = (0, 2, 3, 5) = \mathfrak{r}(5) \\ \mathfrak{p} = (S, (\mathrm{c}), (\mathrm{b}), E) & \longrightarrow & \mathfrak{s}(\mathfrak{p}) = (0, 3, 4, 6) = \mathfrak{r}(6) \\ \mathfrak{p} = (S, (\mathrm{c}), (\mathrm{c}), E) & \longrightarrow & \mathfrak{s}(\mathfrak{p}) = (0, 3, 5, 7) = \mathfrak{r}(7) \ . \end{array}$$

Induction assumption: It is assumed now that for any $m < n$ and any path $\mathfrak{p}$ through the graph of Figure 3.13 of the length m, there exists an l, such that the resource distribution $\mathfrak{r}(l) = \{r_0, r_1 \ldots, r_{m-1}\}$, computed by (3.1) and (3.2), is equal to the resource position sequence $\mathfrak{s}(\mathfrak{p}) = (w_0, w_1, \ldots, w_{m-1})$ of the path $\mathfrak{p}$. Further, $l \in [\varphi_m, \varphi_{m+1})$, and $l = r_{m-1} = w_{m-1}$.

Induction hypothesis: Let $\mathfrak{p} = (v_0, v_1, v_2, \ldots, v_{n-1})$ be a path through the graph of Figure 3.13, and let the corresponding resource position sequence $\mathfrak{s}(\mathfrak{p})$ be given by

$$\mathfrak{s}(\mathfrak{p}) = (w_0, w_1, \ldots, w_{n-1}) \ .$$

Then there exists a unique step number l, such that the resource position sequence is equal to the resource distribution for l, computed by (3.1) and (3.2).

Any path through the graph of Figure 3.13 starts with one of the following sequences:

(a) $(S, (\mathrm{b}), (\mathrm{a}), (\mathrm{b}), \ldots)$ or $(S, (\mathrm{b}), (\mathrm{a}), (\mathrm{c}), \ldots)$,

(b) $(S, (\mathrm{c}), (\mathrm{b}), \ldots)$ or $(S, (\mathrm{c}), (\mathrm{c}), \ldots)$.

Therefore, the path $\mathfrak{p} = (v_0, v_1, \ldots, v_{n-1})$ can be rewritten as $\mathfrak{p} = (S, (\mathrm{b}), (\mathrm{a}), v_3, \ldots, v_{n-1})$, assuming case (a), or the path $\mathfrak{p}$ can be rewritten as $\mathfrak{p} = (S, (\mathrm{c}), v_3, \ldots, v_{n-1})$, if the case (b) assumed. Since the vertices (b) and (c) are only one step away from the vertex S, a sub path $\mathfrak{P}$ of the length $m = n - d$ can be defined by $\mathfrak{P} = (S, v_{d+1}, \ldots, v_{n-1})$. Using case (a), $d = 2$ while using case (b), $d = 1$.

Now case (a) is assumed. The resource position sequence for the path $\mathfrak{P} = (\nu_0, \nu_1, \ldots, \nu_{n-3})$ is defined by $\mathfrak{s}(\mathfrak{P}) = (\omega_0, \omega_1, \ldots, \omega_{n-3})$. By the induction assumption, this sequence is equivalent to the resource distribution for $\lambda = \varrho_{n-3}$, i.e. $\mathfrak{s}(\mathfrak{P}) = \mathfrak{r}(\lambda) = (\varrho_0, \varrho_1, \ldots, \varrho_{n-3})$. Since $\mathfrak{P}$ is a subpath of $\mathfrak{p}$ one can rewrite $\mathfrak{p}$ by

$$\begin{aligned}\mathfrak{p} &= (v_0, v_1, v_2, v_3, \ldots, v_{n-1}) \\ &= (v_0, v_1, v_2, \nu_0, \nu_1, \ldots, \nu_{n-3}) \\ &= (S, (\mathrm{b}), (\mathrm{a}), \nu_0, \nu_1, \ldots, \nu_{n-3}) \ .\end{aligned}$$

Now, the resource position sequence $\mathfrak{s}(\mathfrak{p})$ for the path $\mathfrak{p}$ can be computed by using the resource position sequence $\mathfrak{s}(\mathfrak{P})$ for the path $\mathfrak{P}$, i.e.

$$\begin{aligned}\mathfrak{s}(\mathfrak{p}) &= (w_0, w_1, w_2, w_3, \ldots, w_{n-1}) \\ &= (w_0, w_1, w_2 + \omega_0, w_2 + \omega_1, \ldots, w_2 + \omega_{n-3}) \\ &= (0, \varphi_{n-2}, \varphi_{n-1}, \varphi_{n-1} + \omega_1, \varphi_{n-1} + \omega_2, \ldots, \varphi_{n-1} + \omega_{n-3}) \ .\end{aligned}$$

The induction assumption yields

$$\mathfrak{s}(\mathfrak{p}) = (0, \varphi_{n-2}, \varphi_{n-1}, \varphi_{n-1} + \varrho_1, \varphi_{n-1} + \varrho_2, \ldots, \varphi_{n-1} + \varrho_{n-3}) \ .$$

Since $\mathfrak{s}(\mathfrak{p})$ represents the resource distribution $\mathfrak{r}(\lambda)$ with $\lambda = \varrho_{n-3} \in [\varphi_{n-2}, \varphi_{n-1})$ shifted by φ_{n-1}, one obtains (using Corollary 3.1) that for $l = \lambda + \varphi_{n-1}$ the resource distribution $\mathfrak{r}(l)$ is given by

$$\mathfrak{r}(l) = \{0, \varphi_{n-2}, \varphi_{n-1}, \varphi_{n-1} + \varrho_1, \varphi_{n-1} + \varrho_2, \ldots, \varphi_{n-1} + \varrho_{n-3}\} = \mathfrak{s}(\mathfrak{p}) \ .$$

For the case (b) an equivalent result can be obtained. The resource position sequence for the path $\mathfrak{P} = (\nu_0, \nu_1, \ldots, \nu_{n-2})$ is now defined as $\mathfrak{s}(\mathfrak{P}) = (\omega_0, \omega_1, \ldots, \omega_{n-2})$. By using the induction assumption this sequence is equivalent to the resource distribution for $\lambda = \varrho_{n-2}$, i.e. $\mathfrak{s}(\mathfrak{P}) = \mathfrak{r}(\lambda) = (\varrho_0, \varrho_1, \ldots, \varrho_{n-2})$. Again, since $\mathfrak{P}$ is a subpath of $\mathfrak{p}$, one can rewrite $\mathfrak{p}$ by

$$\begin{aligned}\mathfrak{p} &= (v_0, v_1, v_2, v_3, \ldots, v_{n-1}) \\ &= (v_0, v_1, \nu_0, \nu_1, \ldots, \nu_{n-2}) \\ &= (S, (\mathrm{c}), \nu_0, \nu_1, \ldots, \nu_{n-2}) \ .\end{aligned}$$

Once more, the resource position sequence $\mathfrak{s}(\mathfrak{p})$ for the path $\mathfrak{p}$ can be computed by using the resource position sequence $\mathfrak{s}(\mathfrak{P})$ for the path $\mathfrak{P}$, i.e.

$$\begin{aligned}\mathfrak{s}(\mathfrak{p}) &= (w_0, w_1, w_2, w_3, \ldots, w_{n-1}) \\ &= (w_0, w_1 + \omega_0, w_1 + \omega_1, \ldots, w_1 + \omega_{n-2}) \\ &= (0, \varphi_{n-1}, \varphi_{n-1} + \omega_1, \varphi_{n-1} + \omega_2, \ldots, \varphi_{n-1} + \omega_{n-2}) \ .\end{aligned}$$

Again, the induction assumption yields

$$\mathfrak{s}(\mathfrak{p}) = (0, \varphi_{n-1}, \varphi_{n-1} + \varrho_1, \varphi_{n-1} + \varrho_2, \ldots, \varphi_{n-1} + \varrho_{n-2}) \ .$$

Since $\mathfrak{s}(\mathfrak{p})$ represents the resource distribution $\mathfrak{r}(\lambda)$ with $\lambda = \varrho_{n-2} \in [\varphi_{n-1}, \varphi_n)$ shifted by φ_{n-1}, one obtains by using Corollary 3.1 that for $l = \lambda + \varphi_{n-1}$, the resource distribution $\mathfrak{r}(l)$ is given by

$$\mathfrak{r}(l) = \{0, \varphi_{n-1}, \varphi_{n-1} + \varrho_1, \varphi_{n-1} + \varrho_2, \ldots, \varphi_{n-1} + \varrho_{n-2}\} = \mathfrak{s}(\mathfrak{p}) \ .$$

■

3.6 An Algebraic View onto Instantaneously Reversible Distributions

The Definition 3.1 can be used to obtain another algebraic description of natural numbers. All resource positions are defined by a sum of Fibonacci numbers. Using the last resource r_{n-1} of each instantaneously reversible resource distribution, one gets a definition for the step number as the sum

$$l = 2 + \sum_{i=1}^{m} n_i \varphi_i \ .$$

The numbers n_i can be written as a word $\mathfrak{n}(l)$ defined by $\mathfrak{n}(l) = n_0\, n_1\, \ldots\, n_m$ over the alphabet $\{0, 1, 2\}$, i.e., each single word entry n_i is limited by two. This can be verified by exploring the structure of the decision graph (Figure 3.13). Since at each vertex of the graph the index i increases by one, the word entry 2 is obtained if the path contains the vertices (a) and (c) in that order, i.e.

$$\mathfrak{p} = (\ldots, (\mathrm{a}), (\mathrm{c}), \ldots) \implies \mathfrak{n}(l) = \ldots 2 \ldots \ .$$

A word entry 0 appears if a path goes through the vertex (c), i.e. the path contains one of the following vertex orders:

$$\begin{aligned}\mathfrak{p} &= (\ldots, (\mathrm{c}), (\mathrm{b}), \ldots) &\implies \mathfrak{n}(l) &= \ldots 01 \ldots \\ \mathfrak{p} &= (\ldots, (\mathrm{c}), E) &\implies \mathfrak{n}(l) &= \ldots 0 \ .\end{aligned}$$

The vertex (c) can only be reached via the vertex (a) and can only be left via the vertices (b) or E. Since the path between reaching and leaving the vertex (c) consisted only of the vertex

(c), a word entry 2 is always followed by a word entry 0. Between these two entries, there may be a couple of 1s, i.e.

$$\begin{aligned}\mathfrak{p} &= (\ldots,(\mathrm{a}),(\mathrm{c}),\underbrace{(\mathrm{c})}_{k-\text{times}},(\mathrm{b}),\ldots) &\Longrightarrow\quad \mathfrak{n}(l) &= \ldots 2\underbrace{1}_{k-\text{times}}01\ldots\\ \mathfrak{p} &= (\ldots,(\mathrm{a}),(\mathrm{c}),\underbrace{(\mathrm{c})}_{k-\text{times}},E) &\Longrightarrow\quad \mathfrak{n}(l) &= \ldots 2\underbrace{1}_{k-\text{times}}0\end{aligned}$$

Furthermore, the vertex (a) within the decision graph (Figure 3.13) can only be reached via the vertex (b). With the exception of the final vertex E, the vertex (a) can be left in the direction of the vertex (b) which yields a 2, or the path can ”jump” between the vertex (b) and back to the vertex (a) again which yields always two 1 entries. Therefore, a path between (b) and (c) only can look like

$$\mathfrak{p} = (\ldots,(\mathrm{b}),\underbrace{(\mathrm{a}),(\mathrm{b})}_{k-\text{times}},(\mathrm{a}),(\mathrm{c}),\ldots) \quad\Longrightarrow\quad \mathfrak{n}(l) = \ldots 1\underbrace{11}_{k-\text{times}}2\ldots$$

with $k \in \mathbb{N}$. Thus, in front of a word entry 2 there has to be an odd number of 1s. All properties of a word $\mathfrak{n}(l)$ can be summarised by

1. A word $\mathfrak{n}(l)$ consists of 0, 1 and 2.

2. A sequence always starts with a 1.

3. A 0 stands at the end of a sequence or is followed by a 1.

4. A 2 is always followed by any number of 1s and one 0.

5. The number of 1s between a 0 and a 2 or between the sequence start and a 2 is always odd.

6. Between two 0s there is at least one 1 and exactly one 2.

Example 3.4. First, equivalent to Example 3.2, all words for the number of resources $n = 5$ are given by

$$\begin{aligned}\mathfrak{p} &= (S,(\mathrm{b}),(\mathrm{a}),(\mathrm{b}),E) &\Longrightarrow\quad \mathfrak{n}(8) &= 111\\ \mathfrak{p} &= (S,(\mathrm{b}),(\mathrm{a}),(\mathrm{c}),E) &\Longrightarrow\quad \mathfrak{n}(9) &= 120\\ \mathfrak{p} &= (S,(\mathrm{c}),(\mathrm{b}),(\mathrm{a}),E) &\Longrightarrow\quad \mathfrak{n}(10) &= 1011\\ \mathfrak{p} &= (S,(\mathrm{c}),(\mathrm{c}),(\mathrm{b}),E) &\Longrightarrow\quad \mathfrak{n}(11) &= 1101\\ \mathfrak{p} &= (S,(\mathrm{c}),(\mathrm{c}),(\mathrm{c}),E) &\Longrightarrow\quad \mathfrak{n}(12) &= 1110\ .\end{aligned}$$

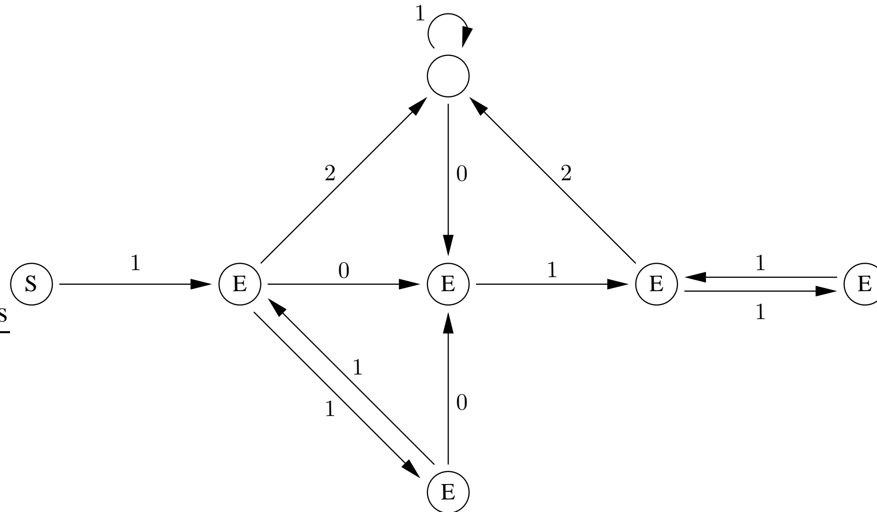
To show all properties, one has to look for longer words e.g.

$$\begin{array}{rclrclrcl}
\mathfrak{n}(34) & = & 111111, & \mathfrak{n}(42) & = & 1011111, & \mathfrak{n}(50) & = & 1110111, \\
\mathfrak{n}(35) & = & 111201, & \mathfrak{n}(43) & = & 1011120, & \mathfrak{n}(51) & = & 1110120, \\
\mathfrak{n}(36) & = & 111210, & \mathfrak{n}(44) & = & 1012011, & \mathfrak{n}(52) & = & 1111011, \\
\mathfrak{n}(37) & = & 120111, & \mathfrak{n}(45) & = & 1012101, & \mathfrak{n}(53) & = & 1111101, \\
\mathfrak{n}(38) & = & 120120, & \mathfrak{n}(46) & = & 1012110, & \mathfrak{n}(54) & = & 1111110, \\
\mathfrak{n}(39) & = & 121011, & \mathfrak{n}(47) & = & 1101111, & \mathfrak{n}(55) & = & 1111111, \\
\mathfrak{n}(40) & = & 121101, & \mathfrak{n}(48) & = & 1101201, & \mathfrak{n}(56) & = & 1111120, \\
\mathfrak{n}(41) & = & 121110, & \mathfrak{n}(49) & = & 1101210, & \mathfrak{n}(57) & = & 1112011.
\end{array}$$

A special case is given if the word consists of m 1s. Then the word represent the known formula for Fibonacci numbers, namely

$$l = 2 + \sum_{i=1}^{m} \varphi_i = \varphi_{m+2} \quad . \tag{3.28}$$

Now, a final deterministic automata can be built which only accepts words represented by the six properties. In Figure 3.14 such a final deterministic automata is displayed. The analysis



PSfrag replacements

Figure 3.14: Automata excepting a word $\mathfrak{n}(l)$.

of any given word starts at the vertex labelled with S. If for every entry within the word an appropriate edge can be found, and if the automata stops at a vertex labelled with E, then the word is correct and represents an instantaneously reversible distribution.

The smallest number, which can be displayed using a word of length m, is given by $101\ldots1$. Using (3.28), this word is equivalent to the step number

$$l = \varphi_m + 2 + \sum_{i=1}^{m-2} \varphi_i = 2\varphi_m \quad .$$

The largest number, which can be displayed using the six properties, is $121\ldots10$. The step number for this word is given by

$$\begin{aligned} l &= \varphi_{m-1} + 2 + \sum_{i=2}^{m} \varphi_i &&= \varphi_m + \varphi_{m-1} + 2 + \sum_{i=1}^{m-1} \varphi_i - \varphi_1 \\ &= \varphi_{m+1} + 2 + \sum_{i=1}^{m-1} \varphi_i - 1 &&= 2\varphi_{m+1} - 1 \end{aligned}$$

Thus, if the step number l is given by $l \in [2\varphi_m, 2\varphi_{m+1})$, then a word representing $\mathfrak{r}(l)$ has the length m.

Chapter 4

Implementation of Parallel Reversal Schedules

vor ort hat der plan nicht nur zahlen
Pension Volkmann in "vor ort" (1985)

The developed theory for parallel reversal schedules does not contain much information about the usage and the implementation of such schedules on parallel computers. The main challenge is finding an optimal allocation of the computational work to processors of the computer. Further, one has to be careful about the location of the information about the checkpoints, the traces, the adjoints and other needed data structures within the computer memory. Depending on the type of parallel computer used, this information is made available to processors needing this information, or this information has to be protected against other processors. There exist two major implementation strategies for the implementation of parallel reversal schedules, independent of the computer type. For each strategy there are four possibilities on how to handle the preparing and the reversing step. Depending on the implementation strategy and substrategy, the optimality regarding the resource needed may be violated. This problem is discussed in [LW02] also. Further on, the implementation of the forward computation for an online constructed parallel reversal schedule will be described.

4.1 Programming models

There are two major programming models which can be used on parallel computers. The first one is the shared memory programming model. Its main advantage is that the user need not care about transferring data within the computer memory. Hence, the transfer time can be ignored for the implementation. It might be done internally by the operating system. The main task concerning the data within the parallel computer is to protect the data against uncoordinated data access.

For the shared memory programming model, few approaches exist. One is based on the join fork mechanism, mostly used on UNIX platforms. The main idea behind this approach is to create child processes by duplicating the parent process. This may lead to unnecessary

memory usage, since data that is not needed, may also be copied. Another problem is the synchronisation. This is mostly done using signals. Furthermore, for the data exchange special data structures are needed. An example for such a programming technique is given in [Bra98]. Using the programming language extension OpenMP [OMP00], this approach can be simplified for the programmer. A second approach is to use threads for shared memory programming. This technique is available on most computer platforms. The main problems using threads are the synchronisation and the data protection. The realisation can be done by using special libraries like the ptheard library [NBF96].

Another programming model assumes distributed memory. Here, each part of the memory is assigned to a fixed processor. Hence, one has to worry about how to transfer the data between the processors and how the transfer time influences the algorithm. The most commonly used tools are message passing libraries such as PVM [PVM94] or MPI [MPI95, MPI97]. By implementing a reversal schedule using the distributed memory model, one can distinguish critical and noncritical communication.

If a communication is classified as critical, then the data written by one process at the end of one advancing, preparing or reversing step at a time t is needed for the beginning of the next advancing, preparing or reversing step by another process at the same time t. This is independent of the type of data required. If the communication is noncritical, the time between writing and reading of the data set is at least as long as the minimum of the time an advancing, preparing or reversing step lasts. The data transfer can be carried out asynchronously. Additional temporary memory for the communication might also be needed for the distributed memory programming model, since most message passing libraries can only send connected memory regions.

Henceforth, only the distribute memory programming model is discussed and later used. One reason for this is the portability. The support and the runtime behaviour of distribute memory programs running on shared memory or distribute shared memory computers (such as the Tera/Cray vector computers or the SGI Origin series) are very good. On the other hand, the support for running shared memory programs on distribute memory computers (such as a workstation cluster) is still in its infancy and still at a point of research.

The next section discusses the implementation of offline constructed parallel reversal schedules for the distributed memory programming models [LW02]. As mention in Section 3.3 these schedule implementations are needed to compose the reverse computation for online constructed parallel reversal schedules. This and the implementation for the forward computation (compare Section 3.2) is discussed in Section 4.3 in more detail.

4.2 Offline Constructed Parallel Reversal Schedules for Distributed Memory Programming Models

Two basic approaches exist to implement parallel reversal schedules using a distributed memory programming model. Both approaches assume that a pool of available processors exists.

4.2.1 Checkpoint Oriented Implementation Approach

The first possibility of implementing a parallel reversal schedule scheme using a distributed memory programming model is to assign each processor to a fixed checkpoint. This approach, shown in Figure 4.1, is called checkpoint oriented. It works in the following way: A processor receives a checkpoint and stores it. At a predetermined time, the processor starts the forward computation up to a state where the next checkpoint is written. The current state is sent to another available processor. After doing so, the processor waits until it will restart the forward computation at the state stored in the assigned checkpoint. Again, this forward computation is done up to the state where the next checkpoint is sent. The state number is smaller than the previous one. Hence, the numbers of advancing steps computed is monotonically decreasing. The processor continues switching between the forward computation and waiting until the checkpoint can be vacated after serving as a starting point for the preparing step. Using this implementation approach, all occurring communication is critical. This implementation cannot satisfy the optimal requirement profile (grey line in Figure 4.1), because of the idle times.

4.2.2 Process Oriented Implementation Approach

The second possibility of implementing a parallel reversal schedule scheme using a distributed memory programming model is called process oriented. This approach is shown in Figure 4.2. A processor receives a checkpoint and starts the forward computation at a predefined time, until the processor reaches a final state. Such a final state can be defined as the state where the preparing step starts, the state where the preparing step ends and the reversing step starts, or the state where the reversing step ends. During the computation the processor writes, stores and sends needed checkpoints. The time when a checkpoint is sent or received by another processor, respectively, can be any time between the time when the checkpoint is written, which is the earliest time, and the time the computation using this checkpoint starts, which defined the latest time.

The determination of the time when the checkpoint data is sent defines the need of storage and the optimality of an implementation. If the checkpoint is sent just when it is written, then no additional memory is needed to store the checkpoint. Since in that case the receiving processor has to wait before, within or at the end of the forward computation, this kind of implementation cannot satisfy the optimal requirement profile. This profile equals the resource requirement of a checkpoint oriented implementation, shown in Figure 4.1. On the other hand, if the checkpoint is sent at the latest point of time, one processor has to possibly store up to three checkpoints temporarily. This is caused by the fact that the computation of the sub-schedules S^1 starts, before the computation of the sub-schedules S^2 ends. But, thereby, this kind of the process oriented implementation can fulfil the optimality, if the checkpoint written at state $i = 0$ is needed before the processor which does the advancing from the state $i = 0$ up to state $i = l$ reaches its final state.

The upper bound of three checkpoints is established in the following lemma.

Lemma 4.1
Suppose the number of steps to be reversed is l with $l \in (\varphi_n, \varphi_{n+1}]$, The schedule is imple-

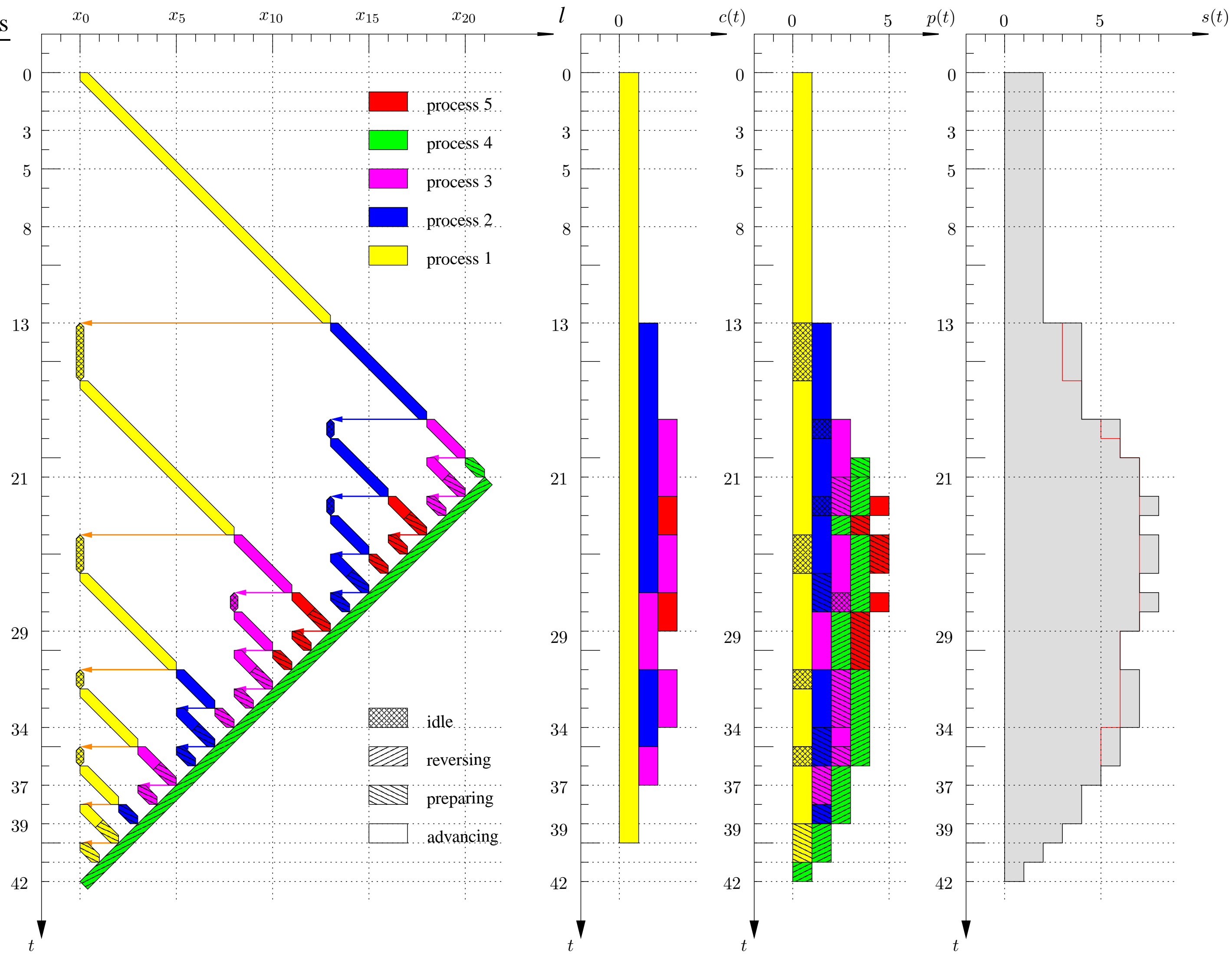


Figure 4.1: General implementation strategies (checkpoint oriented).

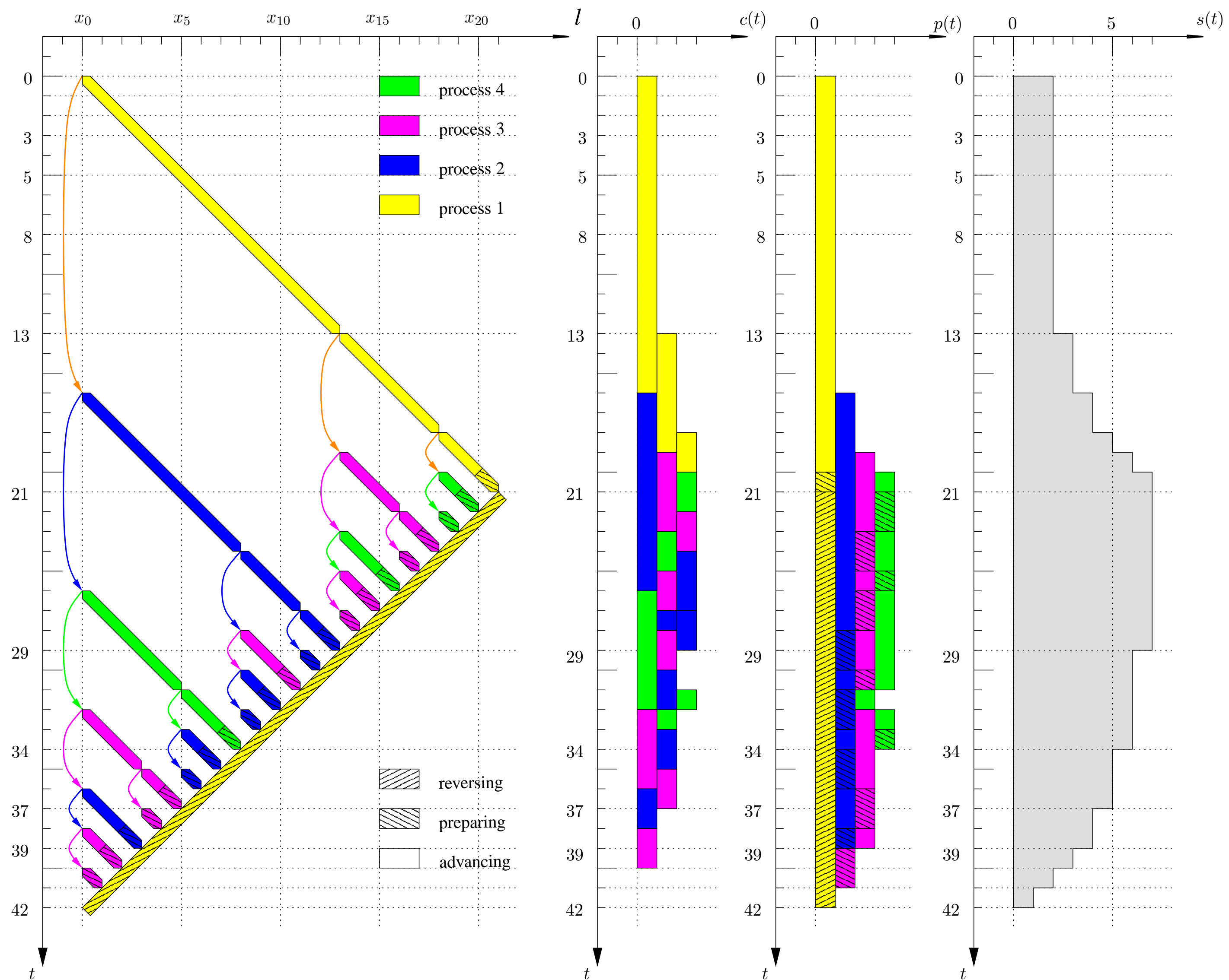


Figure 4.2: General implementation strategies (process oriented).

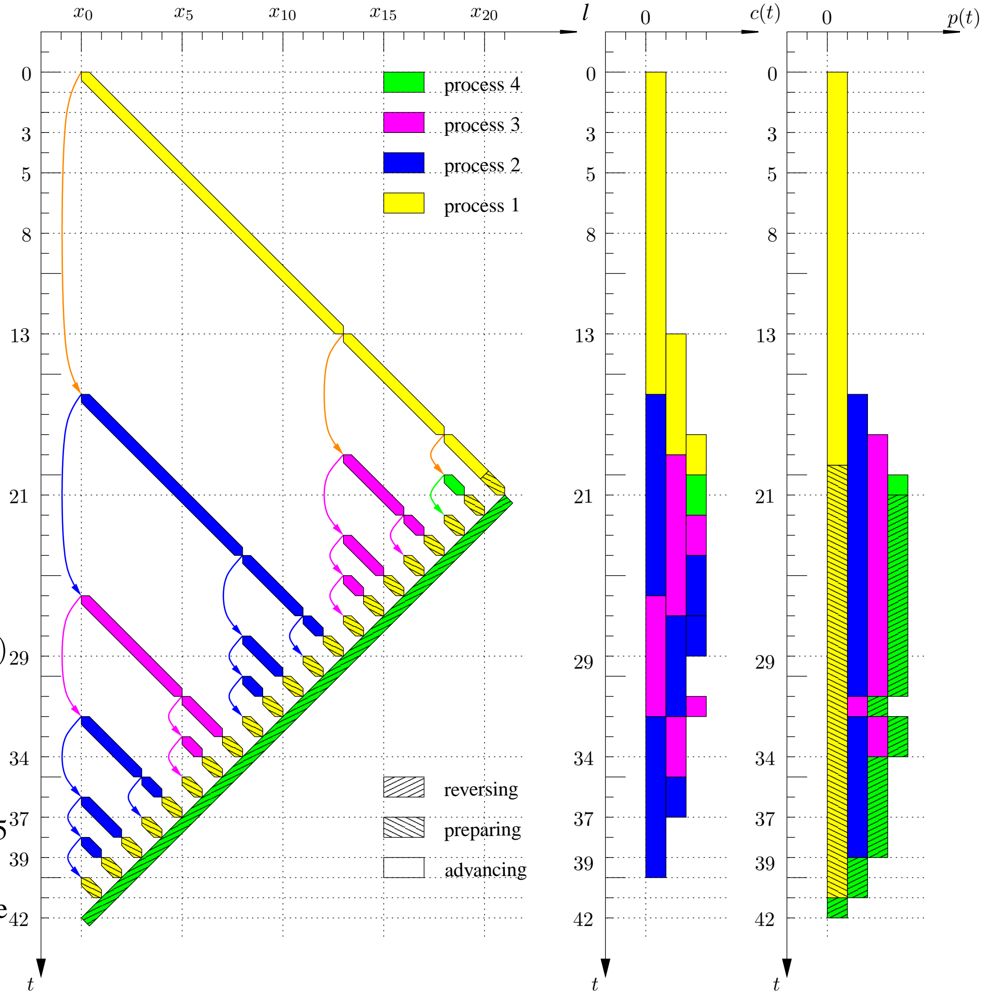


Figure 4.3: One fixed process carries out the tracing and one fixed process carries out reversing.

mented in a process oriented manner. Each process may store as many checkpoints as required, i.e. the checkpoints will be sent as late as possible. Then each processor has to store at most three checkpoints at any time. □

Proof:
One only has to show that the process started first satisfies this property. For all other processes, one applies the claim to the sub-schedules. To prove that lemma one defines the number λ by $\lambda = l - \varphi_{n-1}$, hence $\lambda \in (\varphi_{n-2}, \varphi_n]$. As shown in [Wal99], the times $t_W(i)$, where the ith checkpoint is written, are recursively defined by $t_W(1) = 0$, $t_W(2) = \lambda$ and $t_W(i) = t_W(i-1) + \varphi_{n-2i+4}$. The maximal number of checkpoints written is limited above by $\lceil \frac{n}{2} \rceil + 1$. The time when a checkpoint is needed/read is defined by $t_R(i) = t_W(i) + 2\varphi_{n-2i+1}$ [Wal99].

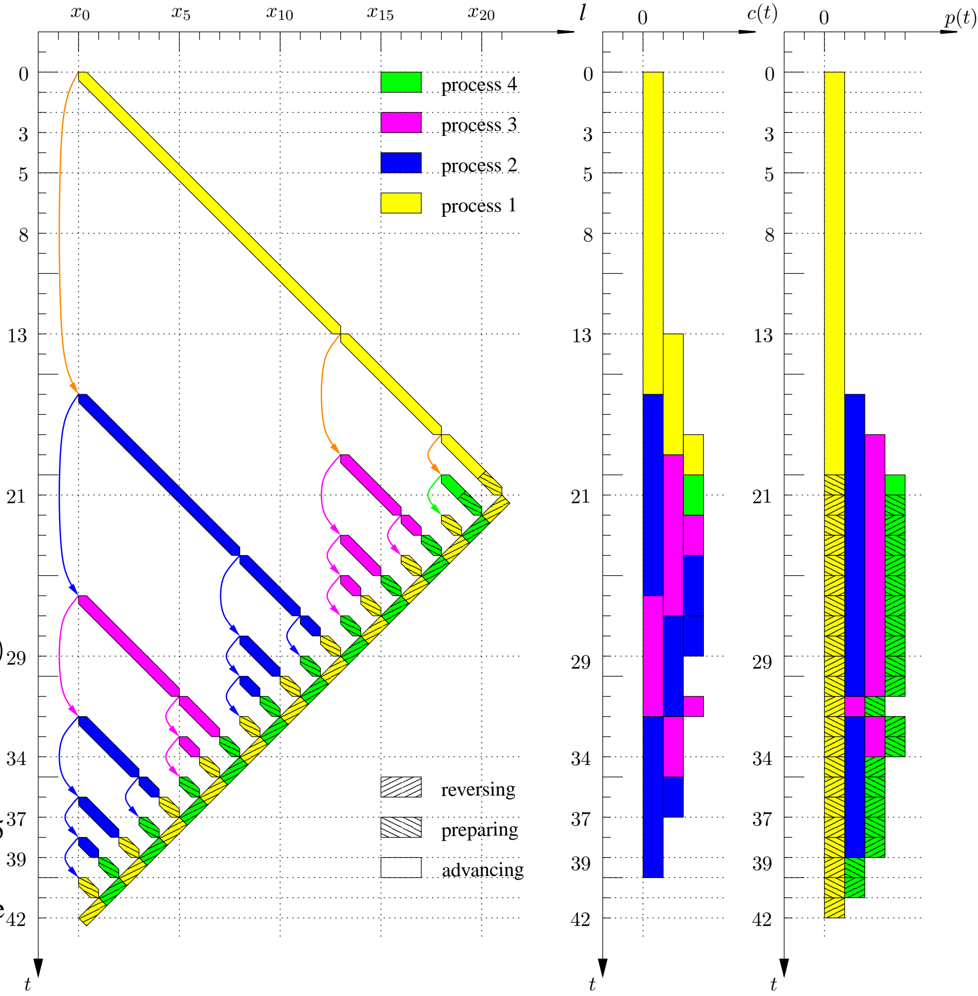


Figure 4.4: Two processes carry out tracing and reversing alternately.

Hence, the lemma is proven if the inequality $t_R(i) < t_W(i+3)$ holds for any i. One has $t_W(i+3) = t_W(i) + \varphi_{n-2i-2} + \varphi_{n-2i} + \varphi_{n-2i+2}$. Furthermore, one obtains

$$\begin{array}{lll} & t_R(i) & < \quad t_W(i+3) \\ \Longleftrightarrow & t_W(i) + 2\varphi_{n-2i+1} & < \quad t_W(i) + \varphi_{n-2i-2} + \varphi_{n-2i} + \varphi_{n-2i+2} \\ \Longleftrightarrow & 0 & < \quad 2\varphi_{n-2i-2} \end{array}$$

which is true for all $i \leq \lceil \frac{n}{2} \rceil + 1$. This completes the proof. ■

There are three ways to carry out the preparing and the reversing step, namely, the processor

- stops before the trace has to be written, stores the data in a checkpoint and sends it to a special tracing process (Figure 4.4 and Figure 4.3);

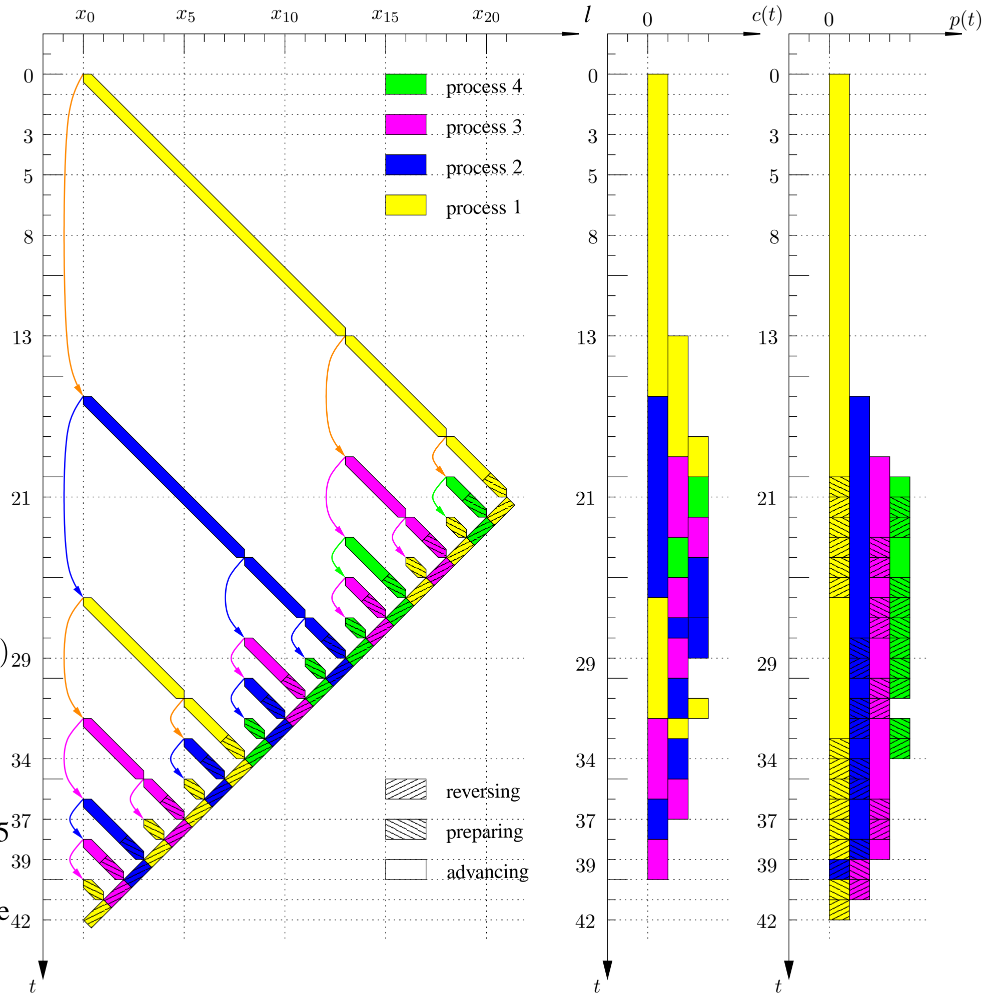


Figure 4.5: Processes carry out the tracing and the reversing by themselves.

- carries out the writing of the trace and sends it to a reversing process (Figure 4.2);
- carries out the writing of the trace and reversing by itself (Figure 4.5) and sends the reversing result to the next reversing process.

Using the first approach, the computation of a preparing step and the computation of a reversing step can be assigned to a fixed processor each (Figure 4.3), or can be combined into one task alternately done by two processors (Figure 4.4). If the computation time $\bar{\tau}$ of a reversing step is smaller than the computation time $\hat{\tau}$ of a preparing step, then there are more than two processors needed to do the special preparing and reversing computation.

The decision about which approach is to be used, the checkpoint oriented or the process oriented, depends on the properties of the problem to be reversed. The checkpoint oriented approach might be easier to implement, because the writing and sending times of a checkpoint

are identical. The disadvantage is that more processors are needed than for an optimal schedule.

The size of the data to be sent forms one criteria which implementation strategy should be applied. For large traces, an implementation where no trace segments are sent is preferable (Figure 4.5 or Figure 4.4). If the result of the reversing step is relatively large compared to the size of the result of a preparing step, then a fixed reversing processor reduces the communication cost (Figure 4.2 or Figure 4.3). This invalidates the previous approach if both the trace segments and the result of the reversing step are relatively large and have approximately the same size.

Another problem may occur if the time $\hat{\tau}$ one preparing step needs and/or the time $\bar{\tau}$ one reversing step needs does not equal one. The preparing or reversing process is not able to interrupt the computation. If then a checkpoint has to be sent from this preparing or reversing process to another process, then the problem occurs.

There are two solutions. The first possibility is to send the checkpoint after the preparing or reversing step is finished. This leads to an increase in time, i.e. one will need more than the minimal time to complete the function reversal. The second solution is to send the checkpoint before the computation of the preparing or reversing step starts.

If, using the second approach, the receiving processor will not stay in the pool of available processors, one may need more processors for the execution of the parallel reversal schedule than the defined optimal number of processors. Otherwise, if the processor is used before computation using the received checkpoint starts, then the checkpoint has to be sent to another processor. This causes additional communication.

4.3 Online Constructed Parallel Reversal Schedules for Distributed Memory Programming Models

As stated in Section 3.3, the reverse computation of an offline constructed parallel reversal schedule and an online constructed parallel reversal schedule look almost like the same. Therefore one can use the results of the previous section to implement the appropriate part, i.e the computation after the vertex was reached. Within this section the implementation of the forward computation will be discussed in detail.

Two processes are needed to carry out the actual computational work. One process starts the computation and checks whether or not the vertex of the program reversal is reached at the end of each advancing step. The second process starts the same computation at the time when the first process reaches the step number two. From then on, the two processes do the same computation in parallel with a displacement of two advancing steps. The second process is responsible for storing the checkpoints. Hence, if a checkpoint has to be written, then the current state of the second process represents the starting point for the checkpoint writing.

Using a shared memory programming model, only one process is needed to carry out the checkpoint handling. The second process can store the checkpoint into the shared memory or delete the checkpoint from it at the appropriated time.

A different situation occurs for a distributed memory programming model. Here again, the second process could store the checkpoints for itself, but this would lead to an enormous

imbalance of memory use. Therefore, it seems preferable to distribute the checkpoints among other processors. Storing a checkpoint then means that the checkpoint data is sent to another process after it was written by the second process. In order to obtain the correct schedule, the processes storing checkpoints have to be notified by at least one of the two computing processes when the checkpoints has to be deleted.

At the returning point of the computation, i.e. at the beginning of the program reversal, a data structure containing all information of the reversal schedule used in the reverse computation can be built. This data structure can be realised by a list of actions one processor has to carry out, as described in Section 2.2. Using a distributed memory programming model, one also has to be careful about the checkpoint location while building up the schedule data structure. The checkpoint that one process needs for the computation is not necessarily stored in the connected memory. Hence, additional communication has to be done during the program reversal. This may cause additional communication or processors, if the execution time $\bar{\tau}$ and $\hat{\tau}$ are not equal to one (compare previous section).

4.4 TOPAS – User interface

In order to realise the discussed implementation strategies, a test implementation was written using the programming language C. This test implementation is called TOPAS, which stands for "Time Optimal Parallel Reversal Schedules". As already stated, the parallel programming model chosen is the distributed memory programming model. For availability and performance reasons the communication is realised using the *Message Passing Interface – MPI* [MPI95]. TOPAS consists of more than 70 routines implemented on more than 12000 lines of C code. So far, the names of the routines described below are fixed. In further versions of TOPAS this could be changed to a declaration of function pointers.

Compared with the packages REVOLVE and REVOLVE++, the program TOPAS does not just tell the user what to do, i.e. which function has to be called, it also calls the required functions itself. This is essential, because in a parallel environment the call of a function must be done at a specified time and at a specified location, i.e processor. Therefore, the execution of the same parallel reversal schedule may run in a different manner if the environment changes, e.g. if the reversal schedule is executed on computers with a different number of processors or the reversal schedule is executed on computers with a different amount of memory per processor. Thus, TOPAS calls functions whose interfaces are predefined by TOPAS. In the following, these user interfaces will be described.

4.4.1 Predefined function

First of all, the two main functions provided by TOPAS are described. These are the functions `TOPAS_RTC_forward` and `TOPAS_RTC_reverse`, whose interfaces are given by

```
TOPAS_el *TOPAS_RTC_forward(
   int *, int *, void *, TOPAS_el *,
   int (*func)(int, const void *));
```

and

```
TOPAS_el *TOPAS_RTC_reverse(
   int*, int*, void *, TOPAS_el *, int,
   int (*func)(int, const void *, const void *));  .
```

The first two parameters are the time count t and the step number l. These are both input and output parameters. The third variable is optional. Depending on whether one wants to use global parameters or not, the user has to choose whether to compile the program using the preprocessor flag `USE_PARAM` or not. If the parameter flag is used, then one has to assign the global parameter information to the third argument. This data will be passed through the complete program up to the user defined routines. (The user defined routines will be described below.) Therefore, the user may use a pointer pointing to any data structure that contains the global parameter information. The next argument of type `TOPAS_el` is a list of pointers to checkpoint data structure, trace data structure or reverse information data structure. The latter will also be called adjoints data structure. An item of the `TOPAS_el` type list element contains, among others, a pointer of type `void *`, pointing to a data structure that contains the checkpoint, trace or adjoint information and an element of type `enum TOPAS_T_e={..., CP = 3002, TR = 3003, AJ = 3004, ...}`, which holds the information about the type of the pointer. The last argument of the functions `TOPAS_RTC_reverse` and `TOPAS_RTC_forward` are pointers to the function checking the restart criterion for the forward computation and reversal criterion for the reverse computation, respectively. The argument of this function is the step number l and the current state x_l contained in a checkpoint. The third argument of the restart criterion is the current adjoint information $\bar{x}_l$. The restart criterion argument of the function `TOPAS_RTC_reverse` is optimal, depending on whether the last but one argument is one or the last but one argument is zero. That means that either the restart criterion must be checked during the reverse computation or it does not have to be.

In order to handle the lists of type `TOPAS_el` the two functions,

```
TOPAS_el *TOPAS_RTC_insert_el(void *, TOPAS_el *);
```

and

```
void *TOPAS_RTC_search_el(enum TOPAS_T_e, int, TOPAS_el *);
```

were implemented. The first function inserts a user defined data structure into the `TOPAS_el` list, and the second function searches a user defined data structure in the list. The function `TOPAS_RTC_insert_el` returns the changed list which was given as the second input argument. The first input argument is a pointer to the data one wants to insert into the list. The second function searches for an element which type is given by the first argument and which step number is given by the second argument. This element is returned if it is found.

The function `TOPAS_RTC_reverse` can also be used to compute program reversals with offline constructed reversal schedules. The input list (fourth argument) consists of just one element, i.e. the checkpoint for the state x_0. The second argument, i.e. the step number, contains the final step number. If the function is used with theses inputs, then it builds up the schedule data structure and runs the reversal. The last two arguments are irrelevant in that case.

Both functions `TOPAS_RTC_forward` as well as `TOPAS_RTC_reverse`, the input arguments are checked of consistency. Thus, it is checked, for instance, whether the input list contains the checkpoints x_{l-2} and x_l or not. If the check fails, then the program terminates with an error.

4.4.2 User defined function

The program TOPAS calls for a couple of routines which have to be provided by the user.

First of all, the user has to define data structures which store the information about a checkpoint, a trace and an adjoint. An example of such a data structure that contains the checkpoint information might be given by `struct my_cp{int l; double *x; double *u;};` where `l` denotes the step number of the checkpoint, the variable `*x` is a pointer to an array of `doubles` containing the state x_l, and `*u` containing the current control u_l at this state. The names of the data structures can be chosen arbitrarily.

Two functions which have to be implemented by the user are the functions

```
int TOPAS_USER_reversal_criterion(int, void *);
```

and

```
int TOPAS_USER_restart_criterion(int, void *, void *);
```

checking the reversal criterion of the forward computation and of the reversal criterion of the reverse computation, respectively. The first two arguments are a step number l and a pointer to a data structure containing checkpoint information. The restart criterion needs, as an additional argument, a pointer to a data structure containing the adjoint information. The functions are always called with the current step number l, the appropriate checkpoint x_l, and, if restart criterion function is called, with the adjoint state $\bar{x}_l$. If a restart or reversal criterion is satisfied, then the corresponding function must return a value larger than zero; if not, then the functions must return a value equals zero.

Next, the functions for the advancing steps F_i, for the preparing step $\hat{F}_i$ and reversing step $\bar{F}_i$ have to be implemented. The declaration of the advancing step function

```
void TOPAS_USER_forward(void *, void *);
```

requires, beside a checkpoint pointer for the first argument, a pointer for the input of the global parameters. Again, one has to compile TOPAS with the `USE_PARAM` preprocessor flag. If the flag is not used, then the second argument is omitted. The pointer to the checkpoint data structure has to be of the type as it was defined by the user, i.e. using the example above, the pointer must be of type `my_cp`. The declaration of the preparing step function is given by

```
void TOPAS_USER_trace(void *, void *, void *);
```

and

```
void TOPAS_USER_trace_f(void *, void *, void *);   .
```

The two different function declaration are needed to give the user the opportunity to use different preparing step functions during the forward computation (TOPAS_USER_trace_f) and during the reverse computation (TOPAS_USER_trace). As in the advancing step function declaration, the first argument is a pointer to the checkpoint data. The second argument is a pointer to the data structure keeping the trace information z_i. The last argument of the preparing step defines the global parameter input. Therefore, if this information is not needed, then it can be omitted. The function declaration

```
void TOPAS_USER_reverse(void *, void *, void *);
```

defines one reverse step $\bar{F}_i$. The first argument is an input parameter and is a pointer to the trace information z_i. The second argument is an input and output argument and is declared by a pointer to the adjoint data $\bar{x}_i$. Again, the third argument is the global parameter input for the current state and is an optional argument. The first reverse step of the reverse computation has to do the initialisation of the adjoint data $\bar{x}_i$. Therefore, an additional reverse function is declared by

```
void TOPAS_USER_init_return(void *, void *, void *);
```

which is only called at the beginning of each reverse computation. The declaration of the argument lists of both functions is identical.

The function names are slightly different than the names used before. If one wants to use the original names, one has to adapt TOPAS, or one has to write wrapper functions for forward(..),prepare(..)and reverse(..).

A further type of user defined functions needed in order to run TOPAS is the communication routine for the exchange of user defined data between the processors. The interfaces for sending and receiving checkpoints, trace and adjoint informations are defined by

```
int TOAPS_USER_send_cp(void *, int, int);
int TOAPS_USER_recv_cp(void *, int, int);
int TOAPS_USER_send_tr(void *, int, int);
int TOAPS_USER_recv_tr(void *, int, int);
int TOAPS_USER_send_aj(void *, int, int);
int TOAPS_USER_recv_aj(void *, int, int);   ,
```

respectively. The first argument of all routines is a pointer to the data one wants to send or receive. The second argument defines the receiver or the sender of the data to whom the data is sent or for whom the data is received. The last argument is the time t, given in counted computing cycles. This time may be used to identify the messages, e.g. it could be used for the tag argument in the appropriated *MPI*-function call.

The last type of required user functions are functions for the memory management. The declaration is given by

```
void *TOPAS_USER_malloc_cp(int);
void *TOPAS_USER_malloc_tr(int);
```

```
void *TOPAS_USER_malloc_aj(int);
void TOPAS_USER_free_cp(void *);
void TOPAS_USER_free_tr(void *);
void TOPAS_USER_free_aj(void *);
void *TOPAS_USER_copy_cp(void *);
void *TOPAS_USER_copy_tr(void *);
void *TOPAS_USER_copy_aj(void *);   .
```

The semantics of each single function can easily extracted from the function name. The input argument of the allocation function is the step number of the current checkpoint, trace or adjoint information. This argument can be used by the user to identify the data. The functions `TOPAS_USER_free_...` are used to delete checkpoints, trace or adjoint informations, if they are not needed anymore. Before TOPAS calls a function whose output is a new checkpoint, trace or adjoint information, an allocation function is called, such that the pointer to the data structure is always valid. Examples for such functions are `TOAPS_USER_recv_tr` or `TOPAS_USER_trace`.

Chapter 5

Numerical Results

This chapter presents the numerical results. The discussed parallel reversal schedules were implemented using a distributed memory programming model. Again, the program skeleton TOPAS, which is the test realisation of the parallel reversal schedules, is written in C. Once more, the communication between the processes is carried out using the MPI. The analysis of the program behaviour was mainly done using *Vampir* [NA$^+$96].

First, only the implementation of the schedules was tested. The results are presented in the next section. Following that section, numerical results for an optimal control problem are presented.

5.1 Schedules

The basic structure of a program using the program skeleton can be divided into three parts. One part is responsible for the computation, while the reversal criterion of the forward computation is not satisfied (i.e. the forward computation is carried out). Another part of the program skeleton generates the data structure containing all information for running the program reversal, i.e. builds a data structure containing the reversal schedule information. A third part is responsible for all computation after the reversal criterion is fulfilled (i.e. the reverse computation is carried out). This reverse computation is performed until a restart criterion is satisfied or the step number $l = 0$ is reached. Hence, the basic behaviour of the program skeleton can be given by:

do
  while (reversal criterion is not fulfilled) **do**
    forward computation
  done
  generate reversal schedule data structure
  while (restart criterion is not fulfilled) **and** ($l \neq 0$) **do**
    reverse computation using the parallel reversal schedule data structure
  done
while ($l \neq 0$).

The behaviour of the program skeleton can be seen in Figure 5.1. The values for $\bar{\tau}$ and $\hat{\tau}$ were set to one. The reversal criterion of the forward computation was satisfied when the step number 48, 40 and 32 was reached. The reversal was demanded at the step number 32 and 24. On the left of Figure 5.1, the schedule computed is shown. The next figure illustrates the measured runtime behaviour of the program. There, one can see the six most time consuming phases of the computation, namely three forward computations (labelled with ①) from $l = 0$ to $l = 48$, from $l = 32$ to $l = 40$ and from $l = 24$ to $l = 32$, as well as the three reverse computations (labelled with ②) from $l = 48$ to $l = 32$, from $l = 40$ to $l = 24$, and from $l = 32$ to $l = 0$. The part labelled with ③ in Figure 5.1 illustrates the generation of the reversal schedule data structure. Since his part does not need much time (the runtime of all three schedule generation parts together is just two percent of the overall runtime of the example), this time can be neglected.

As stated in Section 3.2, two processes are needed to accomplish the forward computation. For the first occurring forward computation, these are the processes number zero and two. For the second forward computation, these are the processes four and five. For the third forward computation, these are the processes one and three (middle grep rectangles in Figure 5.1). For the first forward computation, the processes were chosen arbitrarily, while for all other forward computation the processes keeping the checkpoints for the intermediated states x_{l-2} and x_l were chosen.

Since a distributed memory programming model was used, the other processes were used to store the checkpoints during the forward computation. At the end of each advancing step, the computing process (e.g. process zero) sends a synchronisation signal to the processes keeping checkpoints (black lines in middle diagram of Figure 5.1).

The other parts of the computing time-line in Figure 5.1, i.e. the part where more than two processes were used for the computation (middle grep rectangles) illustrate the reverse computation. Again, at the end of each computed step, a time synchronisation is done. Also, the checkpoints, traces and adjoint are sent if necessary.

The second diagram from the right of Figure 5.1 shows the summarised processor need over the whole computation time. The middle grep areas again illustrate the processor need of the user functions `forward(..)`, `prepare(..)` and `reverse(..)`. Therefore, the middle grep area should be identical with the processor resource profile of the parallel reverse schedule (compare resource profile $p(t)$ in Figure 3.8).

The right most diagram shows what happens during the time when schedules data structure is built up. Since every processor builds its own schedule data structure, no communication is needed within this part of the program, i.e. no black line is shown in the diagram.

5.2 Example: Formula One Car Model

For the purposes of testing the implementation of TOPAS, an optimal control problem, namely the simulation of an automobile, was chosen (compare [WL01]). The aim is to minimise the time the car needs to travel on the specified road.

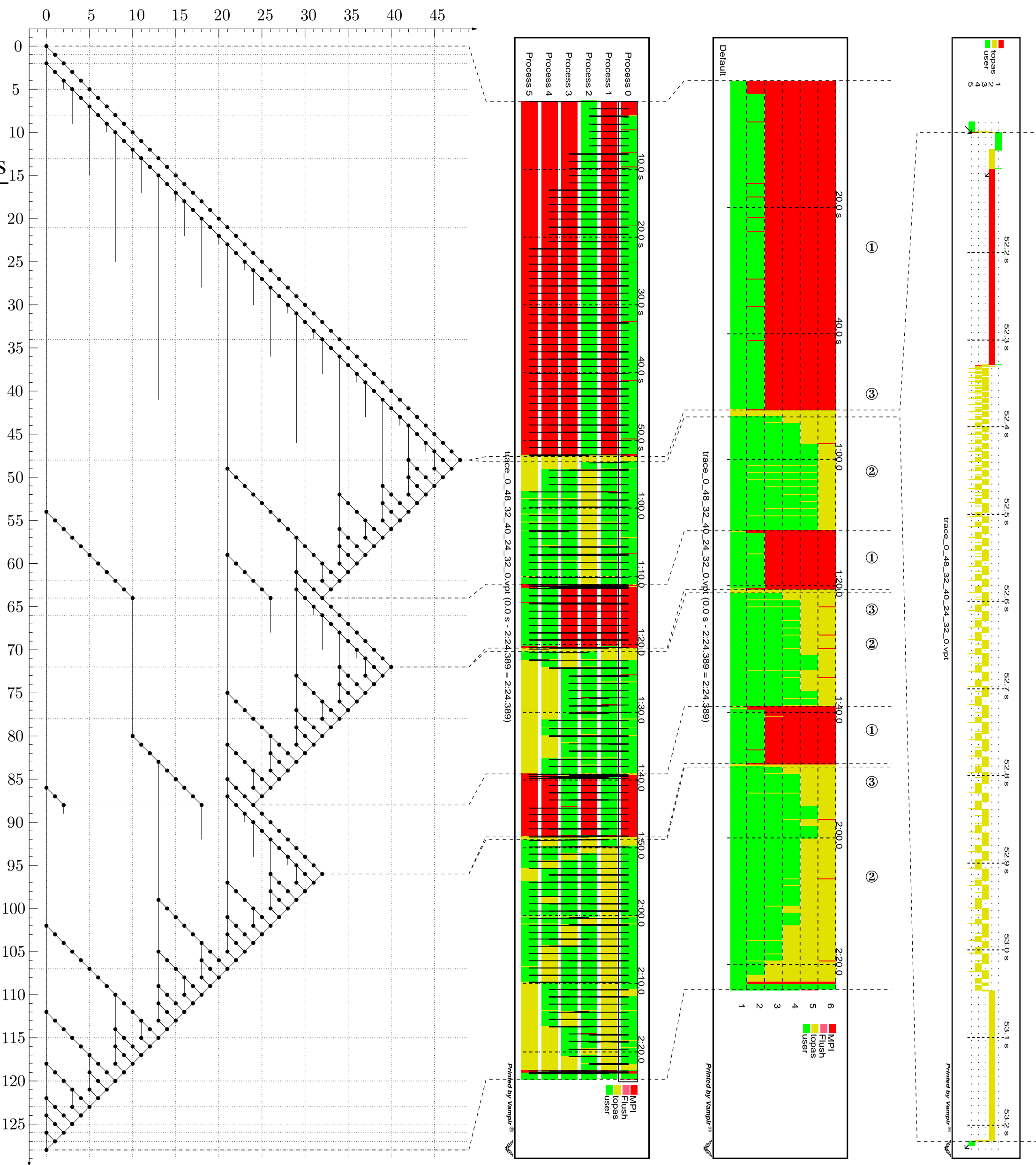


Figure 5.1: Parallel schedules and measured runtime behaviour of an example computation with restarts.

PSfrag replacements

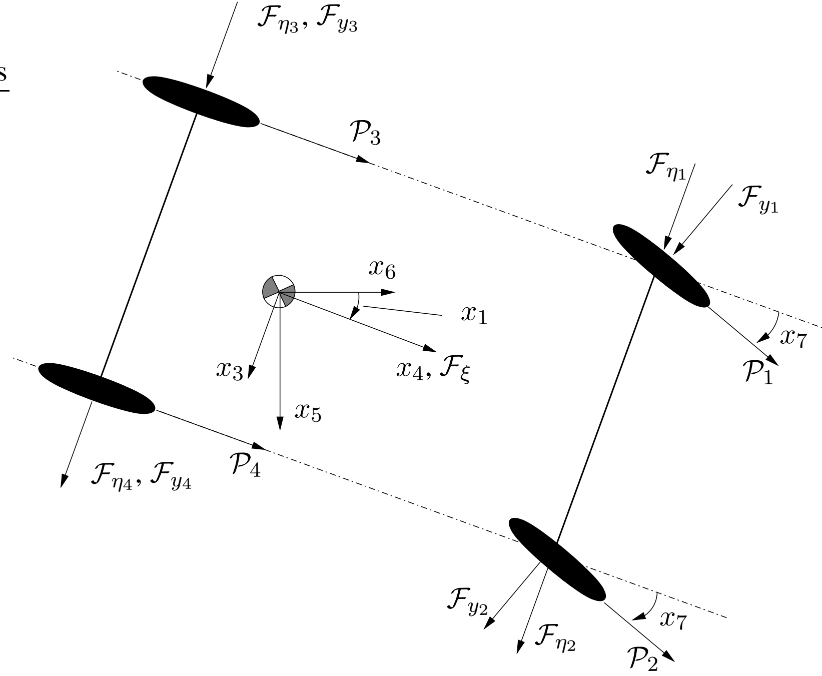


Figure 5.2: Visualisation of the car model values.

5.2.1 Car model

The car model is a simplified model of a Formula one car, i.e. it describes a go-kart model (Figure 5.2). That means that the modelled car only has the lateral and longitudinal velocity as well as the yaw angle as a degree of freedom. Furthermore, the car has a rigid suspension of all axis and tyres and a body rolling about a fixed roll axis. Hence, the model-defining system of ordinary differential equations is given by

$$
\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= \frac{(\mathcal{F}_{\eta_1} + \mathcal{F}_{\eta_2})l_f - (\mathcal{F}_{\eta_3} + \mathcal{F}_{\eta_4})l_r}{I} \\
\dot{x}_3 &= \frac{\mathcal{F}_{\eta_1} + \mathcal{F}_{\eta_2} + \mathcal{F}_{\eta_3} + \mathcal{F}_{\eta_4}}{M} - x_2 x_4 \\
\dot{x}_4 &= \frac{\mathcal{F}_\xi - \mathcal{F}_\alpha}{M} + x_2 x_3 \\
\dot{x}_5 &= x_4 \sin(x_1) + x_3 \cos(x_1) \\
\dot{x}_6 &= x_4 \cos(x_1) - x_3 \sin(x_1) \\
\dot{x}_7 &= u_1 \quad .
\end{aligned}
\tag{5.1}
$$

This model was adapted from [HMK96] and can also be found in [All97]. The vector $\mathbf{x} := (x_1, x_2, x_3, x_4, x_5, x_6, x_7)^{\mathrm{T}}$ defines the current state of the car and includes values for the lateral

and longitudinal velocity of the centre of gravity of the car (x_3 and x_4), as well as the lateral and longitudinal position of the car (x_5 and x_6), among others. A complete listing of all system variables can be found in Appendix A.2. The control vector consists of two values, namely the steering rate and the longitudinal force, and is defined by $\mathbf{u}(s) := (u_1(s), u_2(s))^{\mathrm{T}}$. The value s is the integration parameter. Using this notation, the car model ODE system (5.1) can be rewritten by

$$\dot{\mathbf{x}}(s) = f\big(\mathbf{x}(s), \mathbf{u}(s), s\big) \ . \tag{5.2}$$

Besides fixed model parameter such as the mass M or the length of the car (given by l_f and l_r), other model parameter such as the tyre forces $\mathcal{F}_{\eta_i}$ or the aerodynamic drag force $\mathcal{F}_\alpha$ have to be computed. The tyre forces are calculated by using the so-called Magic Tyre Formula, which can originally be found in [PB92] or [PB97]. The formula used in this computation was adapted from [HMK96]. Using the Magic Tyre Formula, a cyclic dependency of the tyre parameters occurs. The computation can be done iteratively using an initial guess.

5.2.2 Cost function

In order to judge the quality of a driven line, the cost function

$$J(t_f) \quad = \quad \int_0^{t_f} \big(1 + g_t(\mathbf{x}, t)\big) \ dt$$

is used. The final time t_f, which is the time the car needs to drive through the course, is unknown. This parameter is the one to be minimised. Therefore, the time dependent integration of the cost function has to be changed to a distance integration along the road centre line by using the scaling factor $S_{CF}(\mathbf{x}, s)$. Then, the cost function is defined by

$$J(s_f) \quad = \quad \int_0^{s_f} S_{CF}(\mathbf{x}, s)\big(1 + g_s(\mathbf{x}, s)\big) \ ds \ . \tag{5.3}$$

The derivation of the scaling factor $S_{CF}(\mathbf{x}, s)$ can be found in [SCS00] or [CSS00]. The scaling factor $S_{CF}(\mathbf{x}, s)$ is computed using the state values $\mathbf{x}$ of the car and their derivatives $\dot{\mathbf{x}}$, as well as the closest point of the road centre line to the car position. Another part of the cost function is the penalty function $g_s(\mathbf{x}, s)$, judging the quality of the driven line with respect to the road boundaries. As long as the car is within the predefined road boundaries, the function $g_s(\mathbf{x}, s)$ returns zero. If the car leaves the road, then the penalty function returns a value larger than zero. For the numerical results presented here, it returns the distance between the car position and the road boundary squared.

Merging the cost function definition (5.3) and the definition of the car model (5.2), one can express the optimal control problem of steering a car by

$$\begin{array}{rrcll} \text{Minimise} & J(\mathbf{x}) & = & J\big(s_f\big) & \\ \text{such that} & \dot{\mathbf{x}}(s) & = & f\big(\mathbf{x}(s), \mathbf{u}(s), s\big) & s \in [0, s_f], \\ & \mathbf{x}(0) & = & x_0, & \\ & \mathbf{u}(s) & \in & \mathbb{R}^2 \ . & \end{array} \tag{5.4}$$

The state $\mathbf{x}(s)$ has values in $\mathbb{R}^p$ with $p = 7$, while the control space has the dimension $q = 2$.

5.2.3 The Forward Computation

For the implementation of the test problem, the optimal control problem has to be discretised. In order to do so the four-stage Runge-Kutta scheme

$$
\begin{aligned}
l_1 &= s^{i-1}, & k_1 &= f\left(x^{i-1},\, u(l_1)\right), \\
l_2 &= s^{i-1} + \frac{h}{2}, & k_2 &= f\left(x^{i-1} + \frac{h}{2}k_1,\, u(l_2)\right), \\
l_3 &= s^{i-1} + \frac{h}{2}, & k_3 &= f\left(x^{i-1} + \frac{h}{2}k_2,\, u(l_3)\right), \\
l_4 &= s^{i-1} + h, & k_4 &= f\left(x^{i-1} + hk_3,\, u(l_4)\right), \\
& & x^i = x^{i-1} &+ \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
\end{aligned}
\tag{5.5}
$$

for $i = 1, \ldots, l$ was used for the forward computation. The algorithm (5.5) defines one advancing step F_i. The starting conditions were chosen by an initial state vector x_0 and a starting position of the integration variable $s_0 = 0$. The integration was carried along a given road centre line, using a step size h of twenty centimetres. Since an implementation of an online constructed parallel reversal schedule is used, one can use a reversal criterion which stops the forward computation after a distance of 130 meters. This distance, however, is not measured along the road centre. Rather, the real distance is used, i.e. the value x_6 of the car model. Furthermore, the evaluation of the cost function was done using the trapeze rule. Since the computational effort to compute one Runge-Kutta step is very small, ten Runge-Kutta steps F_i were glued together to one advancing step `forward(..)`. This avoids the possibility that the communication dominates the computation. Furthermore, the merging of the single Runge-Kutta steps yields smaller reversal schedules, i.e. it yields schedules for a step number of about 65, instead of a step number of approximately 650.

5.2.4 The Reverse Computation

In order to calculate the discrete adjoints, there are two possible ways. The first possibility is to adjoin the continuous model equations and, after that, discretise the continuous adjoint equations. The second possibility works the other way around. First, the continuous model equation is discretised and after that Algorithmic Differentiation or hand coding is used to adjoin the discretised model equations. Both ways do not lead to the same result in general (see e.g. [Hag00] or [GW01]). This fact has to be take into account, if one has to come to a decision of which way to choose.

The example presented uses the second way, i.e. doing first the discretisation and second the adjoint. The adjoining of the four-stage Runge-Kutta scheme used for the forward computation

leads to the following algorithm for the calculation of the reversing step $\bar{F}_i$

$$\begin{aligned}
&\bar{x}_i = \bar{x}_{i+1} \\
&m_j = \frac{\partial}{\partial x} f\left(k_j, l_j, u(l_j)\right) \qquad n_j = \frac{\partial}{\partial u} f\left(k_j, l_j, u(l_j)\right) \qquad j = 4, \ldots, 1 \\
&\bar{k}_4 = \frac{h}{6}\bar{x}_{i+1} \qquad \bar{k}_3 = \frac{h}{3}\bar{x}_{i+1} + h\bar{k}_4 m_4 \\
&\bar{k}_2 = \frac{h}{3}\bar{x}_{i+1} + \frac{h}{2}\bar{k}_3 m_3 \qquad \bar{k}_1 = \frac{h}{6}\bar{x}_{i+1} + \frac{h}{2}\bar{k}_2 m_2 \\
&\bar{u}^i = \bar{u}^i + \bar{k}_4 n_4 \qquad \bar{x}^i = \bar{x}^i + \bar{k}_4 m_4 \\
&\bar{u}^i = \bar{k}_3 n_3 \qquad \bar{x}^i = \bar{x}^i + \bar{k}_3 m_3 \\
&\bar{u}^i = \bar{u}^i + \bar{k}_2 n_2 \qquad \bar{x}^i = \bar{x}^i + \bar{k}_2 m_2 \\
&\bar{u}^i = \bar{k}_1 n_1 \qquad \bar{x}^i = \bar{x}^i + \bar{k}_1 m_1
\end{aligned} \tag{5.6}$$

for $i = l, .., 1$. As can be seen, the evaluation of the adjoint state vector $\bar{x}^i$ is done in a reverse order. The initial values $\bar{\mathbf{x}}^l$ for the adjoint of the state vector and $\bar{\mathbf{u}}^l$ for the adjoint control vector are defined by

$$\bar{\mathbf{x}}^l = \frac{\partial}{\partial \mathbf{x}} J(\mathbf{x}^l) \qquad \text{and} \qquad \bar{\mathbf{u}}^l = 0 \ .$$

The formulae (5.6) define one reversing step $\bar{F}_i$. As done for the forward computation, ten reversing steps $\bar{F}_i$ are glued together to the function `reverse(..)`.

The functions k_j and l_j for $j = 1, .., 4$ in algorithm (5.6) are identical with the functions k_j and l_j in algorithm (5.5). This gives an indication of how the corresponding preparing step $\hat{F}_i$, respectively, the function `prepare(..)` can be implemented.

The first possibility is just to store the intermediate values of the functions k_j. Then, the reversing step has to compute all the Jacobians m_j and n_j.

Another implementation strategy is to do the calculation of the partial Jacobians m_j and n_j during the preparing step and then to store the results. This can be done because the Jacobians only depend on the value of k_j and l_j of the advancing step F_i. This leads to a larger trace and a larger $\hat{\tau}$ than for the first approach. Then, the reversing step only has computed the last statements of algorithm (5.6).

Besides the trace size, the runtime behaviour of the two implementation approaches can be a criterion about which approach is used. For the example presented, the first approach leads to $\hat{\tau}$ equals one and $\bar{\tau}$ equals four, while the second approach has the runtime behaviour of $\hat{\tau}$ equals four and $\bar{\tau}$ equals one. For the first approach, no online construction of parallel reversal schedules has been developed yet. The second approach is implementation by using τ processors for the preparing step. These processors carry out the computations one after the other. The additional resource need still agrees with resource need given by (2.2).

At the beginning of this section, the order of adjoining and discretising was discussed. Since the results of the first adjoining and then discretising do not equate with the results of first discretising and then adjoining in general, one should note that when using the four-stage Runge-Kutta scheme (5.5), the results do. Further explanation can be found in [Hag00].

5.2.5 The Computation of the Jacobians m_j and n_j

The adjoining of the described problem can be done by hand as done above (compare (5.6)) or can be done by using a Algorithmic Differentiation tool. If the forward integration will be adjoined by hand one has to adjoin the whole car model as well. This error-prone hand-coding can be avoided by using the technique of Algorithmic Differentiation.

Algorithmic or Automatic Differentiation (AD for short) offers the opportunity to provide derivative information for a given code segment. The basic idea behind AD is the systematic application of the chain rule of differentiation to the statements within the code segment. For a comprehensive discussion of AD see [Gri00].

The application of the chain rule can be performed at two stages. The first possibility is the application before compile time. AD tools using this so called source-to-source approach generate new derivative program code from of original code. An example for such a tool is ADIC. The second possibility to obtain derivative information is the application of AD during the runtime. This is usually done by operator overloading. One tool using this technique for the programming languages C and C++ is ADOL-C, detailed explained in [GJ$^+$99].

For the reverse computation, AD was applied to the Runge-Kutta step (5.5). The operator overloading AD tool ADOL-C was used. As explained in the previous section the Jacobians are computed during the preparing step. Therefore the trace data sent is only the result of the Jacobian computation, which are small vectors or matrices. Thus, one needs not care about the internal trace representation of the ADOL-C package. This ADOL-C internal trace data is always kept locally.

5.2.6 Optimisation

The optimisation of the optimal control problem (5.4) was done using a nonlinear conjugated gradients method as can be found, for instance, in [NW99]. In order to carry out the line search, a method using cubic polynomials as described in [DS96] was used. Different to the described method, the polynomial was built by using two points and the directional derivatives at these two points. These directional derivatives can be obtained by using the forward mode of AD for the function evaluation of J, instead of using the normal function evaluation.

5.2.7 Results

Since the memory requirement is one of the major challenges for the program reversal (compare Chapter 1), this memory requirement will be investigated first. The comparison of the memory requirement of the full-logging approach described in Section 2.2 and Figure 2.2 and the parallel approach is done in Table 5.1. The values for the memory need are computed for 650 advancing steps. Thereby, one checkpoint has the size of 7 double values, while the size of the data obtained by one preparing step is given by 63 double values. The size of a double value is set to 64 bits, as it is on most computers. The maximal memory need of the parallel approach is composed by the size of three checkpoints (compare Lemma 4.1) and the size of the data one preparing step requires. During the forward computation, the checkpoints were distributed between the processors equally. Since six processors were used and schedules of

	full-logging approach	parallel approach
maximal number values	40950	651
memory required in kB	327.6	5.2
in %	100.0	1.6

Table 5.1: Memory requirement for one function reversal.

the length of about 62 were utilised, the resource needed was given by ten. Therefore, each processor has to store a maximum of two checkpoints during the forward computation at the same time. This means that the memory requirement during the forward computation can be disregarded, compared to the memory requirement during the reverse computation.

The maximal memory requirement during the reverse computation of the parallel approach is less than two percent of the maximal initial memory, which was needed for the full-logging approach at the computation vertex.

An online constructed parallel reversal schedule was used to compute the adjoint of the forward integration of the ODE system with respect to a given road shape. In order to carry out the time measurement shown in Table 5.2, the program was slightly changed. Thus, the

		full-logging approach	parallel approach
T3E	in sec.	20.27	18.91
	in %	100.0	93.3
Origin 3800	in sec.	6.71	6.04
	in %	100.0	90.0

Table 5.2: Runtime of one function reversal

discretisation step size was changed from twenty centimetres to one centimetre. Furthermore, not 10 but 200 Runge-Kutta steps were glued together to one advancing step `forward(..)`. Thereby, the behaviour of the program with respect to the used online constructed reversal schedule was the same, but the computational work to be done for one advancing was 200 times larger. Since the original problem is very small, this was necessary to get the computation time measurable. Furthermore, the computation time of all computed advancing and preparing steps was set to the same value, such that the full-logging approach and the parallel approach would get competitive. The computations were measured on a Cray T3E and on a SGI Origin3800. As it can be seen in Table 5.2, the runtime of the program reversal has just a slight improvement. This may be due to the less memory each computing node has to manage [SNB00].

The result of the computed optimal control problem can be seen in Figure 5.3 and Figure 5.5. Figure 5.3 shows the car position after the 1st, 3rd and 7th optimisation step. The road shape was an S-curve defined by the road centre line given by

$$f(x) = \frac{17}{4}\arctan\left(\frac{3}{34}(x-40)\right) + 6$$

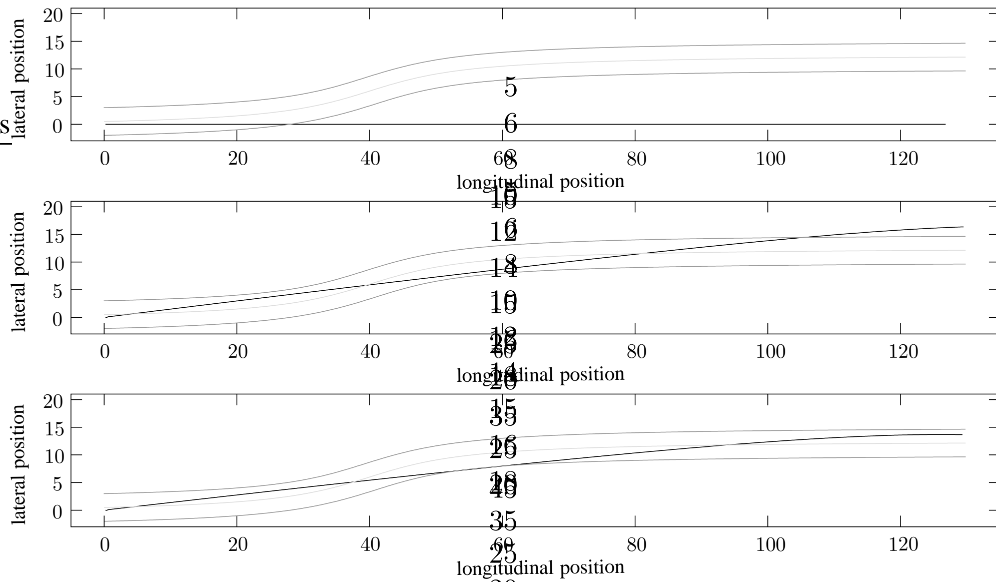


Figure 5.3: Driven car line after the 1st, 3rd and 7th optimisation step.

and the road width was given by three meters. Again, the reversal criterion was satisfied if the longitudinal position x_6 reached a value equals to or larger than 130 meters. After seven optimisation steps no visible change of the driven car line takes place. This can be verified by a look at the cost function value in Figure 5.4.

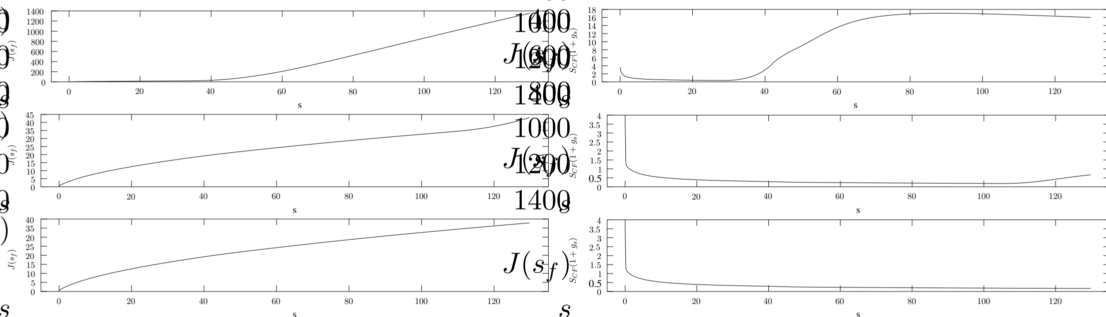


Figure 5.4: Cost function $J(s_f)$ (left) and the integrand $S_{CF}(\mathbf{x},s)(1+g_s(\mathbf{x},s))$ of the cost function (right) after the 1st, 3rd and 7th optimisation step.

The computed adjoined values of the control along the road centre line are shown in Figure 5.5. The left column of diagrams shows the adjoint car steer rate $\bar{u}_1$, and the right column of diagrams shows the adjoint car longitudinal force $\bar{u}_2$ plotted over the distance measured on the road centre line. Again, the values after the 1st, 3rd and 7th optimisation step were visualised.

If, during the optimisation, the car must stay on the road, then one might use the restart of the forward computation. Additional to the target line criterion, the reversal can be initiated

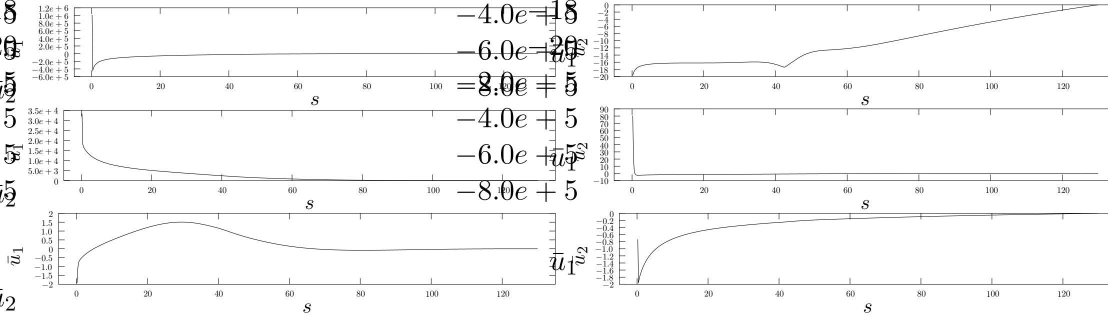


Figure 5.5: Adjoint control values $\bar{u}_1$ (left) and $\bar{u}_2$ (right) for the 1st, 3rd and 7th optimisation step.

if the distance between the car and the road centre line is too large. The reverse computation updates the control until the forward computation starts again, for instance, if the car is close enough to the road centre line. Thus, the car goes forth and back until it reaches the target line. The result of this bidirectional computation is usually a nonoptimal but valid driven car trajectory, i.e. the car always stays on the road. This driven car line can then be improved by computing further optimisation steps using online constructed parallel reversal schedules with no restart.

5.3 Example: Lipschitz tracking

An example which uses the bidirectional computing feature of the implementation of TOPAS is the Lipschitz tracking problem. The Lipschitz tracking problem is defined by a number of values y_i^* given at the sampling points x_i for $i = 0, 1, \ldots, n$. The values y_i^* are called data points and are assumed to be generated recursively so that $y_i^* = F_i(y_{i-1}^*)$ with each F_i being not invertible or ill conditioned. The sampling points are ordered, such that $x_0 < x_1 < \ldots < x_{n-1} < x_n$. The aim is the construction of a piecewise linear continuous function $\Psi(x)$ whose derivative is bounded. That means that the function $\Psi(x)$ has a globally bounded Lipschitz constant. The objective is to minimise the L_2-norm of the discrepancies $y_i^* - \Psi(x_i)$ for $i = 0, \ldots, n$. Moreover, it is assumed that the values y_i^* are generated one by one and the approximation has to be optimal at each stage. One example for $i = 2, \ldots, 12$ is given in Figure 5.6.

The data points y_i^*, for example, could be measured values one wants to follow with a Lipschitz bounded piecewise linear control, or the values y_i^* obtained by a complex computer simulation F with $y_i^* = F(y_i^*)$, which is not invertible.

With $y_i = \Psi(x_i)$ the optimisation problem can now be defined by

$$\text{(5.7)} \qquad \text{Minimise} \qquad g(\mathbf{y}) = \frac{1}{2}\sum_{i=0}^{n}\left(y_i - y_i^*\right)^2$$

$$\text{(5.8)} \qquad \text{such that} \qquad \mu_i \geq \left|f_i(y_i, y_{i-1})\right|$$

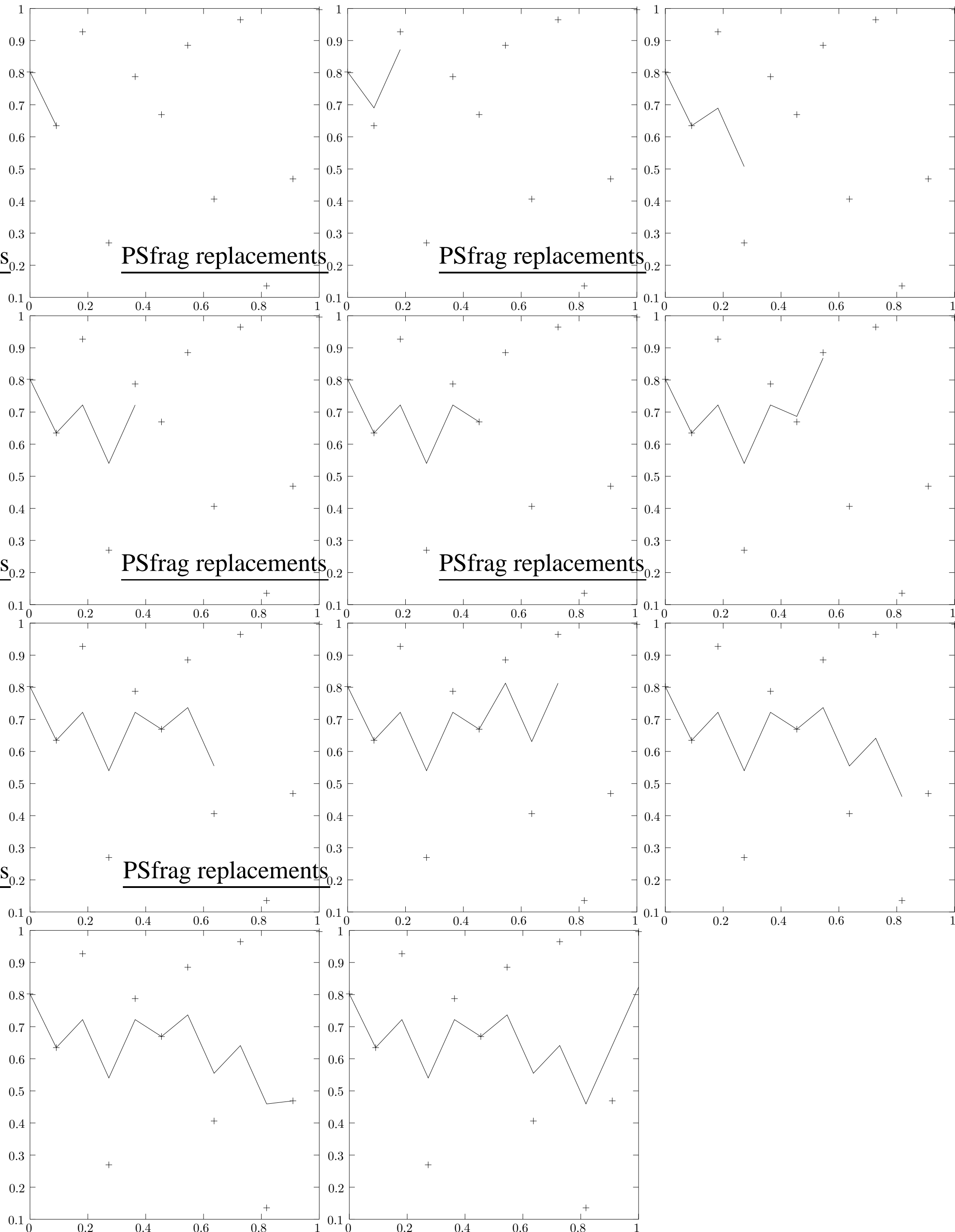


Figure 5.6: Optimal solutions of the Lipschitz tracking problem for $n = 11$ points using the Logistic map to generate the data values y_i^* and $\mu = 2$.

for $i = 1, 2, \ldots, n$ and with $\mathbf{y} \in \mathbb{R}^{n+1}$, $f_i(y_i, y_{i-1}) = y_i - y_{i-1}$ and $\mu(x_i - x_{i-1}) \equiv \mu_i$ for a given $0 < \mu$.

The optimisation problem (5.7) and (5.8) can also be interpreted as an optimal control problem with a constrained control. The control is given by the slope of the straight line, i.e. values

$$u_i = \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \ ,$$

whose absolute values are bounded by μ. The cost function one wants to minimise is again given by (5.7). This is a convex quadratic optimisation problem with box constraints. However, the objective Hessian is dense with respect to the u_i but diagonal with respect to the y_i. Therefore, one sticks with the original formulation.

The Kuhn-Tucker theorem for the considered QP-problem yields the existence of the Lagrange multipliers λ_i such that

$$\sum_{i=0}^{n} \lambda_i \nabla f_i(y) = \nabla g(y) \tag{5.9}$$

$$\text{with} \quad \lambda_i = 0 \quad \text{if} \quad |f_i| < \mu_i \tag{5.10}$$

$$\text{and} \quad \lambda_i f_i < 0 \quad \text{if} \quad |f_i| = \mu_i \ . \tag{5.11}$$

Because the problem is strictly convex, these first order necessary conditions are already sufficient for optimality. The Kuhn-Tucker condition (5.9) can be given in detail by

$$\sum_{i=0}^{n} \lambda_i \nabla f_i(y) = \begin{pmatrix} -\lambda_1 \\ \lambda_1 - \lambda_2 \\ \vdots \\ \lambda_{n-1} - \lambda_n \\ \lambda_n \end{pmatrix} = \begin{pmatrix} y_0 - y_0^* \\ y_1 - y_1^* \\ \vdots \\ y_{n-1} - y_{n-1}^* \\ y_n - y_n^* \end{pmatrix} = \nabla g(y) \ .$$

Hence, the ith component is given by $\lambda_{i-1} - \lambda_i = y_{i-1} - y_{i-1}^*$ with $\lambda_1 = -(y_0 - y_0^*)$ and $\lambda_n = y_n - y_n^*$. Thus the kth Lagrange multiplier λ_k can be computed forward, starting form the sampling point x_0 or the multiplier can be computed backward, starting from x_n by

$$\lambda_k = -\sum_{i=0}^{k-1} (y_i - y_i^*) \qquad \text{or} \qquad \lambda_k = \sum_{i=k}^{n} (y_i - y_i^*) \ ,$$

respectively. If for an interval $[x_{i-1}, x_i]$ the constrain $|f_i(y_i, y_{i-1})| \leq \mu_i$ is not active, i.e. $|f_i(y_i, y_{i-1})| < \mu_i$, then the problem separates in that $\lambda_i = 0$. Now an algorithm can be defined which computes the optimal solution stepwise.

First, it is assumed that the piecewise linear function is optimal for all sampling points x_i with $i = 0, \ldots, k-1$. That means that for the intervals with $|f_i| = |y_i - y_i^*| < \mu_i$, one knows

that $\lambda_i = 0$. Now the new data point y_k^* at the sampling point x_k is entered. The first estimate for the value y_k is chosen by

$$y_k = \begin{cases} y_k + \mu_k & \text{if} \qquad 1 < \dfrac{y_k^* - y_k}{\mu_k} \\ y_k - \mu_k & \text{if} \qquad -1 > \dfrac{y_k^* - y_k}{\mu_k} \\ y_k^* & \text{else.} \end{cases}$$

Considering the third case, the Lagrange multiplier $\lambda_k = y_k - y_k^*$ satisfies the optimality criterion (5.10), and, hence, one can proceed with the next sampling point x_{k+1}. For the first two cases, one has to pull up a segment of the piecewise linear function Ψ, if the Lagrange multiplier λ_k is larger than zero, or one has to pull down a segment of the piecewise linear function Ψ, if the Lagrange multiplier λ_k is smaller than zero.

Pulling a segment of the piecewise linear function Ψ up or down means that one has to find an interval $[x_{j-1}, x_j]$ with

1. $u_j < \mu$ if the segment $[x_j, x_k]$ has to be pulled up (i.e. $\lambda_k > 0$), or
2. $u_j > \mu$ if the segment $[x_j, x_k]$ has to be pulled down (i.e. $\lambda_k < 0$).

If such an interval was found, one has to move the segment by

$$\Delta y = \frac{1}{k-j} \sum_{i=j}^{k} (y_i - y_i^*)$$

i.e. all values y_i for $i = j, \ldots, k$ must be updated in the corresponding direction. If $|y_k - y_{k-1} - \Delta y|$ is larger than μ_k, then the segment can only be moved by a value Δy such that $|y_k - y_{k-1} - \Delta y| = \mu_k$. Then the search for an interval $[x_{j-1}, x_j]$ with a smaller index j and the properties 1 or 2, respectively, must be continued and the values y_i for $i = j, \ldots, k$ have to be updated again.

One sweep for a new value y_k^* decreases the value of the objective function $g(\mathbf{y})$. Thereby, one decreases the value of the objective function by shifting a whole segment of the piecewise linear function Ψ. The segment is bounded by two intervals $[x_{i-1}, x_i]$ with $|f_i| < \mu_i$ on each side. This shift can be taken by a move through a set S of feasible points of the subspace $\mathbb{R}^k \subset \mathbb{R}^{n+1}$. The dimension of this feasible set S is given by the number of constraints, which are not active, i.e. the number of intervals $[x_{i-1}, x_i]$ with $\lambda_i = 0$ plus one. The boundaries of the feasible set S are defined by the sizes of all possible shifts of all segments within the piecewise linear function Ψ which do not change the activity constraints of the intervals. If now a sweep activates the constraint for a certain interval $[x_{j-1}, x_j]$, then one move to the boundary of the feasible set S and its dimension decreases by one. If a sweep deactivates the constraint for a certain interval $[x_{j-1}, x_j]$, then the feasible set S is extended by a new direction. The value of the objective function is minimal if $\mathbf{y}$ is restricted to this feasible set S. Since there is only a finite number of feasible sets, and since at each sweep the value of the objective function is minimal within the feasible set, the algorithm stops after a finite number of steps.

The reversal schedules come into play if the data points y_i^* are computed by an iterative process which is not invertible. The computation of the Lagrange multipliers λ_j then needs the data points y_i^* in a reverse order. In that case, the advancing step `forward(..)` and the preparing step `prepare(..)` are identical. Both compute the data point $y_i^* = F(y_{i-1}^*)$. The reversing step `reverse(..)` does nothing except provide the data point y_i^*.

A one dimensional example for such a function F is the logistic map, which was used in order to test the optimisation algorithm. Hence, the data points y_i^* are computed by

PSfrag replacements

$$y_i^* = F(y_{i-1}^*) = 4y_{i-1}^*(1 - y_{i-1}^*) \ .$$

1

A first test case was computed using 11 data points, as can be seen in Figure 5.7. The movement of the algorithm through the data points y_i^*, can be seen in Figure 5.7. The upper value represents the index k, while the lower value illustrates how far the algorithm has to go backwards, i.e. the values illustrate the index j. These two indices define the behaviour of the bidirectional schedule used.

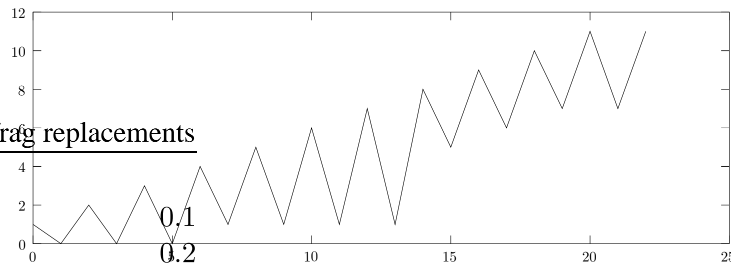
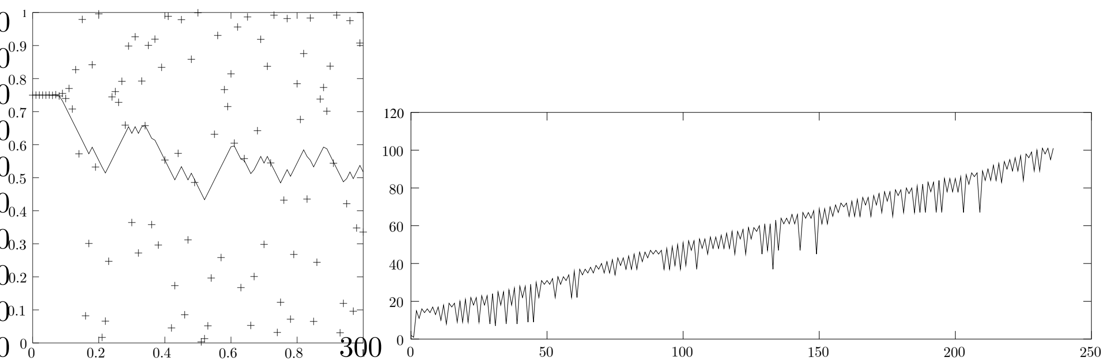


Figure 5.7: Movement of the optimisation algorithm through the data points y_i^*.

The final solution of a larger data point set ($n = 100$) is shown in Figure 5.8(a). Again,



(a) Final optimal solution.

(b) Movement of the optimisation algorithm through the data points y_i^*.

Figure 5.8: Lipschitz tracking problem for $n = 100$ points using the Logistic map for the data values y_i^* and $\mu = 2$.

the data points are computed using the logistic map. The start value for the logistic map was chosen near the stationary point of the logistic map. That means that the data points stay near the stationary point at the beginning, and then they start to get a chaotic behaviour. This is the reason that the movement of the optimisation algorithm through the data point is very small at the beginning and then starts to get larger. For both computations the bound for the Lipschitz number was given by $\mu = 2$.

Chapter 6

Summary and Outlook

6.1 Summary

For the solution of several computational tasks, the reversal of a computer function F is needed. The main problem of reversing the computer program which evaluates the function F is the enormous memory requirement especially, if the function F is ill-conditioned or difficult to reverse. This memory requirement grows linearly with the length of the function evaluation, if all values needed for function reversal are stored. One way to avoid the memory requirement is to recompute of all values used for the function reversal. This leads to an enormous increase in the time needed to complete the program reversal, compared with the runtime of the reversal if all values are stored. The runtime using the latter approach grows linearly with respect to the original duration of the function evaluation.

The present work investigated parallel reversal schedules for program reversion. Reversal schedules are one way to find an optimal medium between complete recalculation and complete storing of the values needed for the reversal of the function F.

In the beginning, a brief introduction was given about the classes of functions which were explored. Within this work, it was supposed that the function F can be divided into a sequence of l advancing steps, such that the function forms an evolutionary system. Each advancing step uses a current state of the evolutionary system to compute a new state. Reversal schedules use checkpointing to find the optimal medium between complete recalculation and complete storing. A checkpoint is an intermediate state of the evolutionary system which is used to start recomputations and which is saved into the computer memory for a certain time while the computation proceeds. Furthermore, it was assumed that all advancing steps of the evolutionary system require the same time to evaluate, or the runtime of each advancing step is uniquely bounded for all advancing steps. Only little is known about the reversal schedules, if the length l of the evolutionary system is not known priori.

In the second part, reversal schedules for single-processor computers were discussed for both cases, if the length of the function F is known priori and if the length of the function is not known. The former case leads to the offline construction of serial reversal schedules, while the latter case leads to the online construction of serial reversal schedules. The outcome of

the first case is that for the reversal of an evolutionary function in which c checkpoints can be stored at the same time, the runtime using an offline constructed optimal serial reversal schedule increases by a factor of $\sqrt[c]{l}$. Measurements for example problems done so far have shown that the introduced technique for the online construction of serial reversal schedules leads to only a small increase in the runtime of the complete program reversal, compared to the runtime of an offline constructed serial reversal schedule. Furthermore, the theory of reversal schedules on parallel computer was introduced in a nutshell. Only the theory of offline constructed reversal schedules was discussed.

One major part of this work was the extension of the offline construction of parallel reversal schedules for the online scenario. First, a possibility to define instantaneously reversible resource distributions for any given step number l was discussed. A resource distribution was defined to be a set of different intermediate states of the evolutionary system. Thereby, the number of resources within a resource distribution for the step number l was bounded by the maximal number of resources needed to reverse l advancing step, using offline constructed parallel reversal schedules. An important property was that one instantaneously reversible resource distribution for a given step number l can easily be converted into the instantaneously reversible resource distribution for the next step number $l + 1$, within the time of one advancing step, by changing one or more intermediate states. The way in which the resource distributions are converted to the next step number defines the way in which the forward computation is carried out using online constructed parallel reversal schedules. Having the conversion property, the structure of the occurring set of resource distributions for all step numbers up to a step number l was investigated next. Two options for finding recursive substructures in the resource distributions were discussed. One substructure is associated with the step number, while the other substructure is associated with the time of the forward computation. Using these substructures, the definition of instantaneously reversible resource distribution was extended from an exact position of the resource to intervals in which the resources must lie, if a resource distribution has to be instantaneously reversible. This interval definition was used to join the offline and the online construction of parallel reversal schedules. This allows one to restart the forward computation during the reverse computation, such that all arising resource distribution are instantaneously reversible. Thus, an algorithm was defined, which converts a given instantaneously reversible resource distribution (defined by using the intervals), either to an instantaneously reversible resource distribution for the step number $l + 1$, or to an instantaneously reversible resource distribution for the step number $l - 1$ on a parallel computer. One can use this algorithm to switch between the forward computation and the reverse computation at any time. Overall, this leads to schedules for dynamic bidirectional simulations on multi-processor machines. Furthermore, different possibilities for the presentation of instantaneously reversible resource distributions were discussed.

Subsequently, the implementation of parallel reversal schedules was discussed in detail. The preferred programming model was the distributed memory model. It was discovered that the checkpoint oriented implementation concept cannot keep the resource optimality, by definition. It was furthermore discussed under which condition the resource requirement of the process oriented programming model is optimal, and how many checkpoints have to be stored by one processor in that case.

Finally, this work contains a documentation of the testing of the theoretical results. Therefore, a program skeleton was written in C. This test implementation, called TOPAS, was written using the MPI for the communication. The basic structure of such a program skeleton realising parallel reversal schedules was discussed. Further on, the TOPAS package was used to compute the solution of two small optimal control problems, namely the steering of a Formula One car and a Lipschitz tracking problem. Considering the Formula One car problem, the resource need of the implementation and the runtime behaviour were discussed in addition to the actual optimal control result, i.e. the optimal steering sequence. Therefore, the optimal control problem was solved on two different parallel computers. The Lipschitz tracking problem was solved by an algorithm, which uses a bidirectional schedules.

6.2 Outlook

During the reversal of a program using a parallel reversal schedule, the number of actual used resources changes. Most parallel computers are unable to dynamically handle the change of resource need while a computation is running. This is particularly the case if the number of used processes changes. Since a process without a processor does not make much sense with respect to the parallel reversal schedules, the size of the pool of available processors must be the same or larger than the number of processes currently used during the execution of a parallel reversal schedule. Difficulties occur if one must add a processor to the processors available. Thus, one should investigate how to get a sufficient efficiency for a fixed number of processors while computing a program reversal using parallel reversal schedules. In many cases, the program which one wants to reverse and which evaluates the evolutionary system is running in parallel already. For the forward computation, this problem can be easily solved, since the number of needed processors does not change. Thus, each half of the available processors carries out one forward computation. The reverse computation seems to be much more complicated. If in addition to the parallel execution of the reversal schedule, one wants to execute the advancing, preparing and reversing step in parallel, then the parallelisation of the single advancing, preparing and reversing step must be very dynamical and might be changed at the end of each computed step. In this context, one should investigate which parallel programming model should be used for the implementation of the parallel reversal schedule and which parallel programming model should be used for the implementation of the advancing, preparing and reversing step.

A continuative task is the construction of parallel reversal schedules online if the runtime of the preparing and the reversing step are not one. The case $\hat{\tau} > 1$ can easily be handled if no restart is needed. The preparing steps start at the end of each forward computation, i.e. with a timely distance of $\tau = 1$. The results of the preparing steps are obtained in a vector or pipeline-like manner. Therefore, $\hat{\tau} - 1$ additional processors are needed. On the other hand, this pipelined computation of the preparing steps causes problems if a restart is needed. At the restart point, there are $\hat{\tau} - 1$ processors carrying out the preparing step, which can usually not be interrupted. This leads either to a delay of the restart, or one has to construct special parallel reversal schedules handling the restart. The case $\bar{\tau} > 1$ seems to be much harder to handle. From the theory of the offline constructed parallel reversal schedules [Wal99], one knows that

the resource need decreases for a fixed step number l and a fixed $\hat{\tau}$, if $\bar{\tau}$ increases. Thus, the resource needed of the online constructed parallel reversal schedules should decrease as well.

A further generalisation is that the construction of parallel reversal schedules for nonuniform steps cost, i.e. the subfunctions F_i need different amount of time. Almost nothing is known about the offline and online construction of parallel reversal schedules for nonuniform step cost. Furthermore, there is an additional adaptivity in the problem, namely whether the step costs of the single steps is known priori or not.

A further challenge is the construction of parallel schedules for the computation of second order derivative information for optimal control problems, in order to use Newton-like methods. One efficient way to compute the Newton direction for the discrete optimal control problem is Pantoja's algorithm [Chr99, CBB00] in its direct formulation. In addition to the forward and the reverse computation of the discussed program reversal, another forward computation is needed for Pantoja's algorithm. This second forward computation needs information obtained during the reverse computation in the reverse order, i.e. the indices are running from l to 0 during the reverse computation, but the second forward computation needs the data in the natural order from 0 to l. Again, a full-logging approach seems to be possible, but the required reverse state information has the size of a normal state information x_i, squared. One possible solution is to perform the first forward computation in a full-logging manner. For the reverse computation and the second forward computation, one uses parallel reversal schedules which were discussed in this thesis. Another possibility could be the use of an indirect formulation of Pantoja's algorithm [Chr01]. In doing so, the second forward computation is no longer needed. Using an indirect Pantoja-like algorithm, however, one must beware of the algorithm stability.

A further topic which is left over is the transfer of the test implementation TOPAS to a robust program package. Right now, TOPAS consists of a couple of files written in C containing more than 12000 lines of code. These files must be adapted to each problem. Especially when the user must write his own communication routines. The goal is to write a package consisting of three libraries associated with the three major part of the implementation. Using predefined interfaces, these libraries should be exchangeable by user written libraries. Furthermore, the libraries should be able to use both kinds of parallel computer architecture, the distribute and the shared memory computer architecture. If the libraries are used on a distribute memory computer, then a tool has to be written, which creates the communication routines for the checkpoints or for the traces automatically for example. The user should only have to provide a file describing the data structure. A first approach for the automatic communication code generation was done in [Ben95].

Appendix A

Glossary

A.1 General Notations

$\vec{F}$ and $\vec{F}_i$	Function or subfunction in terms of a mathematical mapping
F_i	Advancing step, i.e. computer function evaluating the vetcor function $\vec{F}_i$
$\bar{F}_i$	Reversing step, i.e. computer function evaluating the reversal of the advancing step F_i
$\hat{F}_i$	Preparing step, i.e. computer function evaluating the advancing step F_i and assembling all data required for the reversing step $\bar{F}_i$.
φ_n	The nth Fibonacci number defined by $\varphi_0 = 1$, $\varphi_1 = 1$ and $\varphi_{n+1} = \varphi_n + \varphi_{n-1}$,
r_i and ϱ_i	Position of the ith resource
l	Current step number
l_I and λ_I	Inner end point of the live time of a resource defined by (3.17) page 43.
l_M and λ_M	The joining step number of a resource defined by (3.19) page 47.
l_O and λ_O	Outer end point of the live time of a resource defined by (3.8) page 38.
$\tilde{b}_i$ and $\tilde{\beta}_i$	Shortage for $b_i - \varphi_{n-i}$ and $\beta_i - \varphi_{m-i}$ (page 30 and page 34)
τ	Time of one advancing step F_i
$\bar{\tau}$	Time of one reversing step $\bar{F}_i$
$\hat{\tau}$	Time of one preparing step $\hat{F}_i$
$\mathbf{r}$	Intervals of possible resource positions (see Section 3.4 at page 47)
$\mathfrak{r}(\cdot)$	Resource distribution (see Section 3.1 at page 30)
$\mathfrak{S}(\cdot,\cdot)$	Set of all possible resource positions (see Section 3.2 at page 38 Corollary 3.2)
$\mathfrak{R}(\cdot)$	Resource distribution intervals (see Section 3.4 at page 50 Corollary 3.4)
$Tr(\cdot,\cdot)$	Triangle area containing resource positions (see Section 3.2 at page 40)
$Te(\cdot,\cdot,\cdot)$	Tetragon area containing resource positions (see Section 3.2 at page 40)
$Rec(\cdot,\cdot,\cdot)$	Rectangle area containing resource positions (see Section 3.2 at page 39)
$\mathcal{S}(\cdot,\cdot)$	Shift of an area of resources in the down-rightward direction (see Section 3.2 at page 40)
$\Phi.$ and $\bar{\Phi}.$	Sums of Fibonacci numbers (see Section 3.4 at page 41 Definition (3.13))

A.2 Notations for Car Model Problem

x_1	Car yaw angle (relative to the inertial reference axes)
x_2	Car yaw rate
x_3	Lateral velocity of the centre of gravity
x_4	Longitudinal velocity of the centre of gravity
x_5	Lateral position of the centre of gravity (relative to the inertial reference axes)
x_6	Longitudinal position of the centre of gravity (relative to the inertial reference axes)
x_7	Car steer angle
u_1	Car steer rate (as a control action)
u_2	Car longitudinal force
$\mathcal{F}_{\eta_i}$	Car lateral force of the ith tyre where i if defined by

(A.1)	Front left tyre	...	$i = 1$
	Front right tyre	...	$i = 2$
	Rear left tyre	...	$i = 3$
	Rear right tyre	...	$i = 4$

$\mathcal{F}_{\xi}$	Car longitudinal force
$\mathcal{F}_{\alpha}$	Car aerodynamic drag force
M	Car mass
l_f	Distance from the centre of gravity to the front axis
l_r	Distance from the centre of gravity to the rear axis
$\mathcal{F}_{y_i}$	Car lateral tyre force of the ith tyre (i defined by (A.1))
$\mathcal{P}_i$	Car longitudinal tyre force of the ith tyre (i defined by (A.1))
$\mathcal{F}_{z_i}$	Tyre normal force of the ith tyre (i defined by (A.1))
α_f	Front slip angle
α_r	Rear slip angle

List of Figures

1.1 Dependencies and temporal order of the frames of a MPEG-decoded video stream. 11
1.2 Evaluation of vector function $\vec{F}$. 11
1.3 The reversal of a function F consisting of 5 steps. 13

2.1 The evaluation and reversal of a function F. 17
2.2 Full-logging approach to calculate the reversal of function F. 19
2.3 Serial reversal schedules for step number $l = 8$. 20
2.4 Online constructed serial reversal schedules (reversal criterion is satisfied at $l = 10$) for three checkpoints. 22
2.5 A schedule for $l = 8$ steps with $\hat{\tau} = \bar{\tau} = 1$ using the bisection strategy. 24
2.6 Recursive construction of an optimal schedule for $l = 8$ steps and $\hat{\tau} = \bar{\tau} = 1$. . . 26
2.7 Recursive construction of the resource profile for an optimal schedule for $l = 8$ steps and $\hat{\tau} = \bar{\tau} = 1$. 27

3.1 Resource distributions for $l \in [0, 34]$. 31
3.2 Sufficient criterion of reversibility. 32
3.3 Proof sketch for Corollary 3.1. 33
3.4 Changing of the resource positions ϱ_{m-2} and ϱ_{m-1} to the resource positions r_{n-3}, r_{n-2} and r_{n-1} 37
3.5 Forward computation for an unknown number of steps. 38
3.6 Proof sketch for Corollary 3.2. 40
3.7 Proof sketch for Corollary 3.2. 41
3.8 One forward and backward computation for an instantaneous reversal. 44
3.9 Resource distribution of the reverse calculation ($\times$) mirrored onto resource distribution of the forward computation ($\square$). 45
3.10 Life cycle of the resource position r_i. 46
3.11 Intervals of instantaneous reversible resource distributions. 47
3.12 Proof sketch for Corollary 3.4. 50
3.13 Graph for the computation of instantaneously reversible resources distributions (Definition 3.1.) – decision graph 57
3.14 Automata excepting a word $\mathfrak{n}(l)$. 62

4.1 General implementation strategies (checkpoint oriented). 68

4.2 General implementation strategies (process oriented). 69
4.3 One fixed process carries out the tracing and one fixed process carries out reversing. 70
4.4 Two processes carry out tracing and reversing alternately. 71
4.5 Processes carry out the tracing and the reversing by themselves. 72

5.1 Parallel schedules and measured runtime behaviour of an example computation with restarts. 81
5.2 Visualisation of the car model values. 82
5.3 Driven car line after the 1st, 3rd and 7th optimisation step. 88
5.4 Cost function $J(s_f)$ (left) and the integrand $S_{CF}(\mathbf{x}, s)(1+g_s(\mathbf{x}, s))$ of the cost function (right) after the 1st, 3rd and 7th optimisation step. 88
5.5 Adjoint control values $\bar{u}_1$ (left) and $\bar{u}_2$ (right) for the 1st, 3rd and 7th optimisation step. 89
5.6 Optimal solutions of the Lipschitz tracking problem for $n = 11$ points using the Logistic map to generate the data values y_i^* and $\mu = 2$. 90
5.7 Movement of the optimisation algorithm through the data points y_i^*. 93
5.8 Lipschitz tracking problem for $n = 100$ points using the Logistic map for the data values y_i^* and $\mu = 2$. 93

Bibliography

[All97] JONATHON ALLEN. *Computer Optimisation of Cornering Line*. Master's thesis, Cranfield University, School Of Mechanical Engineering, September 1997.

[Ben73] CHARLES H. BENNETT. *Logical Reversibility of Computation*. IBM Journal of Research and Development, vol. 17, pp. 525–532, 1973.

[Ben95] JOCHEN BENARY. *DAP – Dresdener Adjugierten Parallelisierungsprojekt*. Technical Report IOKOMO–05–1995, Technische Universit¨at Dresden, December 1995.

[Ben96] JOCHEN BENARY. Parallelism in the reverse mode. In MARTIN BERZ, CHRISTIAN H. BISCHOF, GEORGE F. CORLISS, AND ANDREAS GRIEWANK, eds., *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings of the Second International Workshop on Computational Differentiation, Santa Fe, New Mexico, USA, February 12-14, 1996, pp. 137–147. SIAM Philadelphia, February 1996.

[Bra98] BERTRAM BRACHER. *Einsatz von Depot4 zur unterstütenden Kodetransformation beim Automatischen Differenzieren (Parallelisierung beim Rückwärstlauf)*. Diplomarbeit, Technische Universit¨at Dresden, Fakult¨at f¨ur Mathematik und Naturwissenschaften, Fachrichtung Mathematik, Institut f¨ur Wissenschaftliches Rechnen, March 1998.

[CBB00] BRUCE CHRISTIANSON AND MICHEAL BARTHOLOMEW-BIGGS. Globalisation of Pantoja's Optimal Control Algorithm. In Corliss et al. [CF$^+$00], pp. 225–130.

[CF$^+$00] GEORGE CORLISS, CHRISTÈLE FAURE, ANDREAS GRIEWANK, LAURENT HASCOËT, AND UWE NAUMANN, eds. *Automatic Differentiation of Algorithms: From Simulation to Optimisation*, Selected papers from the Third International Conference on Automatic Differentiation, June 19-23, 2000, Cote d'Azur, France, Heidelberg, Berlin, June 2000. Springer.

[Chr99] BRUCE CHRISTIANSON. *Cheap Newton Steps for Optimal Control Problem: Automatic Differentiation and Pantoja's Algorithm*. Optimisation Methods and Software, vol. 10, No. 5, pp. 729–743, 1999.

[Chr01] BRUCE CHRISTIANSON. *A Self-Stabilizing Pantoja-Like Indirect Algorithm for Optimal Control*. Optimisation Methods and Software, vol. 16, pp. 131–149, 2001.

[CSS00] DANIELE CASANOVA, ROBIN S. SHARP, AND PAT SYMONDS. *Minimum Time Manoeuvring: The Significance of Yaw Inertia*. Vehicle System Dynamics, vol. 34, pp. 77–115, 2000.

[DS96] JOHN E. DENNIS JR. AND ROBERT B. SCHNABEL. *Numerical Methods for Unconstrained Optimisation and Nonlinear Equations*, volume 16 of *Classics in Applied Mathematics*. SIAM – Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[GJ+99] ANDREAS GRIEWANK, DAVID JUEDES, HRISTO MITEV, JEAN UTKE, OLAF VOGEL, AND ANDREA WALTHER. *ADOL-C, A package for the automatic differentiation of algorithms written in C/C++*. Technical report, Technische Universit¨at Dresden, March 1999. Updated version of [GJU96].

[GJU96] ANDREAS GRIEWANK, DAVID JUEDES, AND JEAN UTKE. *ADOL-C, A package for the automatic differentiation of algorithms written in C/C++*. ACM Transactions on Mathematical Software, vol. 22, No. 2, pp. 131–167, 1996.

[Gri92] ANDREAS GRIEWANK. *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*. Optimisation Methods and Software, vol. 1, pp. 35–54, 1992.

[Gri00] ANDREAS GRIEWANK. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Frontiers in Applied Mathematics. SIAM – Society for Industrial and Applied Mathematics, Philadelphia, 2000.

[GW00] ANDREAS GRIEWANK AND ANDREA WALTHER. *Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation*. ACM Transactions on Mathematical Software, vol. 26, No. 1, pp. 19–45, 2000.

[GW01] ROLAND GRIESSE AND ANDREA WALTHER. *Evaluating Gradients in Optimal Control – Continuous Adjoints versus Automatic Differentiation*. Technical Report IOKOMO–11–2001, Technische Universit¨at Dresden, December 2001.

[Hag00] WILLIAM W. HAGER. *Runge-Kutta Methods in Optimal Control and the Transformed Adjoint System*. Numerische Mathematik, vol. 87, pp. 247–282, 2000.

[HHG02] PATRICK HEIMBACH, CHRIS HILL, AND RALF GIERING. Automatic Generation of Efficient Adjoint Code for a Parallel Navier-Stokes Solver. In Sloot et al. [ST+02], pp. 1019–1028.

[HL+00] GUNDOFT HAASE, ULRICH LANGER, EWALD LINDNER, AND WOLFRAM MÜHLHUBER. Optimal Sizing of Industrial Structural Mechanics Problems Using AD. In Corliss et al. [CF+00], pp. 181–188.

[HMK96] J.P.M. HENDRIKX, T.J.J. MEIJLINK, AND R.F.C. KRIENS. *Application of Optimal Control Theory to Inverse Simulation of Car Handling*. Vehicle System Dynamics, vol. 26, pp. 449–461, 1996.

[HP98] PETER HILTON AND JEAN PETERSEN. *A Fresh Look at old Favourites: The Fibonacci and Lucas sequences Revisited.* Australian Mathematical Society Gazette, vol. 25, No. 3, pp. 146–160, 1998.

[KGW00] WOLFRAM KLEIN, ANDREAS GRIEWANK, AND ANDREA WALTHER. Differentiation Methods for Industrial Strength Problems. In Corliss et al. [CF+00], pp. 3–23.

[Knu97] DONALD E. KNUTH. *The art of computer programming: Fundamental algorithms*, volume 1 of *Addison-Wesley series in computer science and information processing, World student series edition*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

[LW02] UWE LEHMANN AND ANDREA WALTHER. The Implementation and Testing of Time-minimal and Resource-optimal Parallel Reversal Schedules. In Sloot et al. [ST+02], pp. 1049–1058.

[MPG02] MOVING PICTURE EXPERTS GROUP. Moving Picture Experts Group Homepage. 2002. `http://mpeg.telecomitalialab.com/`.

[MPI95] MESSAGE PASSING INTERFACE FORUM. *MPI: A Message-Passing Interface Standard*. International Journal of Supercomputer Application, vol. 8, No. 3/4, pp. 165–416, June 1995.

[MPI97] MESSAGE PASSING INTERFACE FORUM. *MPI-2: Extensions to the Message-Passing Interface*. University of Tennessee, Knoxville, Tennessee, July 1997.

[NA+96] WOLFGANG E. NAGEL, ALFRED ARNOLD, MICHAEL WEBER, HANS-CHRISTIAN HOPPE, AND KARL SOLCHENBACH. *VAMPIR: Visualisation and Analysis of MPI Resources*. Supercomputer 63, vol. 12, No. 1, pp. 69–80, January 1996. The article was published as special issue of the journal Supercomputer (Supercomputer 63).

[NBF96] BRADFORD NICHOLS, DICK BUTTLAR, AND JACQUELINE PROULX FARRELL. *Pthreads Programming*. O'Reilly, Sebastopol, CA,USA, 1996.

[NW99] JORGE NOCEDAL AND STEPHEN J. WRIGHT. *Numerical Methods for Unconstrained Optimisation and Nonlinear Equations*. Springer Series in Operations Research. Springer, Heidelberg, 1999.

[OMP00] OPENMP ARCHITECTURE REVIEW BOARD. OpenMP: Simple, Portable, Scalable SMP Programming. 2000. `http://www.openmp.org/`.

[PB92] H.B. PACEJKA AND E. BAKKER. The Magic Formula Tyre Model. In H.B. PACEJKA, editor, *Tyre Models for Vehicle Dynamics Analysis*, volume 21 of *Supplement to Vehicle Systems Dynamics*, pp. 1–18, 1992.

[PB97] H.B. PACEJKA AND I.J.M. BESSELINK. The Magic Tyre Model with transient properties. In F. BÖHM AND H.-P.WILLUMEIT, eds., *Tyre Models for Vehicle Dynamics Analysis*, volume 27 of *Supplement to Vehicle Systems Dynamics*, pp. 234–249, 1997.

[PVM94] AL GEIST, ADAM BEGUELIN, JACK DONGARRA, WEICHENG JIANG, ROBERT MANCHEK, AND VAIDY SUNDERAM. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computing. The MIT Press, Cambridge, Massachusetts, London, England, 1994. http://www.epm.ornl.gov/pvm/pvm_home.html.

[RdB99] MICHEL RONSSE AND KOEN DE BOSSCHERE. *RecPlay: A Fully Integrated Practical Record/Replay System*. ACM Transactions on Computer Systems, vol. 17, No. 2, pp. 133–152, May 1999.

[RDG93] NICOLE ROSTAING, STÉPHANE DALMAS, AND ANDRÉ GALLIGO. *Automatic Differentiation in Odyssée*. Tellus, vol. 45A, pp. 558–568, 1993.

[RLG98] JUAN MARIO RESTREPO, GARY K. LEAF, AND ANDREAS GRIEWANK. *Circumventing Storage Limitations in Variational Data Assimilation Studies*. SIAM Journal on Scientific Computing, vol. 19, No. 5, pp. 1586–1605, September 1998.

[SCS00] ROBIN S. SHARP, DANIELE CASANOVA, AND PAT SYMONDS. *A Mathematical Model for Driver Steering Control, with Design, Tuning and Performance Results*. Vehicle System Dynamics, vol. 33, pp. 289–326, 2000.

[SNB00] STEPHAN SEIDL, WOLFGANG E. NAGEL, AND HOLGER BRUNST. The Future of HPC at SGI: Early Experiences with SGI SN-1. In B. J. JESSON, editor, *Proceedings of Sixth European SGI/Cray Workshop*, Manchester, 2000.

[Sne93] JAN L.A. VAN DE SNEPSCHEUT. *What computing is all about*. Texts and Monographs in Computer Science. Springer Verlag, Berlin, 1993.

[ST+02] PETER M. A. SLOOT, C. J. KENNETH TAN, JACK J. DONGARRA, AND ALFONS G. HOEKSTRA, eds. *Proceedings of the International Conference on Computational Science, Amsterdam, April 21-24, 2002, Part II*, Lectures Notes in Computer Science – ICCS 2002, Heidelberg, Berlin, April 2002. Springer.

[Ste02] JULIA STERNBERG. *Adaptive Umkehrschemata für Schrittfolgen mit nicht uniformen Kosten*. Diplomarbeit, Technische Universität Dresden, Fakultät für Mathematik und Naturwissenschaften, Fachrichtung Mathematik, Institut für Wissenschaftliches Rechnen, July 2002.

[Wal99] ANDERA WALTHER. *Program Reversal Schedules for Single- and Multi-processor Machines*. Dissertation, Technische Universität Dresden, Fakultät für Mathematik und Naturwissenschaften, December 1999.

[WG01] ANDREA WALTHER AND ANDREAS GRIEWANK. *Bounding the number of processes and checkpoints in time-minimal parallel reversal schedules*. Technical Report IOKOMO–03–2001, Technische Universit¨at Dresden, April 2001.

[WL01] ANDREA WALTHER AND UWE LEHMANN. *Adjoint Calculation using Time-Minimal Program Reversals for Multiprocessor Machines*. Technical Report IOKOMO–09–2001, Technische Universit¨at Dresden, November 2001.

Versicherung

Hiermit versichere ich, daß ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

I affirm that I have written this dissertation without any inadmissible help from any third person and without recourse to any other aids; all sources are clearly referenced. The dissertation has never been submitted in this or similar form before, neither in Germany nor in any foreign country.

Die vorgelegte Dissertation habe ich am Zentrum für Hochleistungsrechnen / Institut für Wissenschaftliches Rechnen der Technischen Universität Dresden unter der wissenschaftlichen Betreuung von Herrn Prof. Andreas Griewank Ph.D. und Herrn Prof. Dr. Wolfgang E. Nagel angefertigt.

I have written this dissertation at the Center for High Performance Computing / Institute of Scientific Computing, Technical University Dresden, under the scientific supervision of Prof. Andreas Griewank Ph.D. and Prof. Dr. Wolfgang E. Nagel.

Dresden, den 10. Februar 2003