The Implementation and Testing of Time-minimal and Resource-optimal Parallel Reversal Schedules

Uwe Lehmann[1] and Andrea Walther[2]

[1] Center for High Performance Computing
Technical University Dresden, D–01062 Dresden, Germany
lehmann@zhr.tu-dresden.de
[2] Institute of Scientific Computing
Technical University Dresden, D–01062 Dresden, Germany
awalther@math.tu-dresden.de

Abstract. For computational purposes such as the computation of adjoint, applying the reverse mode of automatic differentiation, or debugging one may require the values computed during the evaluation of a function in reverse order. The naïve approach is to store all information needed for the reversal and to read this information backwards during the reversal. This technique leads to an enormous memory requirement, which is proportional to the computing time. The paper presents an approach to reducing the memory requirement without increasing the wall clock time by using parallel computers. During the parallel computation, only a fixed and small number of memory pads called checkpoints is stored. The data needed for the reversal is recomputed piecewise by starting the evaluation procedure from the checkpoints. We explain how this technique can be used on a parallel computer with distributed memory. Different implementation strategies will be shown, and some details with respect to resource-optimality are discussed.

1 Introduction

For many applications nonlinear vector functions have to be evaluated by a computer program. For several purposes, e.g. the calculation of adjoints, debugging or interactive control, one may need to reverse the program execution. Throughout this article we assume that the program execution to be reversed can be split into parts, called steps. Furthermore, it is presumed that all steps need the same computation time. Typical examples of such vector functions are iterations or time-dependent evolutions as occur in crash tests and meteorological or oceanographical simulations.

The computing resources needed for the reversal are computing time, memory and computing power, i.e. the number of processors used. The usual way to implement the reversal of a program execution is the following approach. One records a complete execution log, called trace, during the evaluation of the given function [1]. For each arithmetic operation, this execution log contains a code

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 1049–1058, 2002.
© Springer-Verlag Berlin Heidelberg 2002

and the addresses of the arguments. Subsequently, the execution log is read backwards to perform the reversal of the program evaluation. Obviously this basic method leads to a memory requirement that is proportional to the runtime of the program execution. Hence, for real-world applications with several thousand steps, the memory requirement can be too large to fit in any computer memory.

To avoid the enormous amount of memory required by the basic approach, checkpoint strategies were proposed and developed for one-processor machines [2, 3]. This checkpointing method naturally increases the computing time because instead of storing all the information required, some of the data is recomputed repeatedly. Using parallel computers, the increased runtime of the reversal can be traded for an increase in computing power [3]. Here, more than one processor is applied to perform the reversal. One important application of such a parallel reversal is given by real time computation, because in this case any increase in temporal complexity has to be avoided.

2 Theoretical Issues

2.1 Definitions and Assumptions

Suppose a given function $F : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ with $x \longmapsto F(x)$ can be decomposed into l parts F_i with $0 \leq i \leq l-1$, such that $F_i : \mathbb{R}^{n_i} \longrightarrow \mathbb{R}^{n_{i+1}}$ with $x_i \longmapsto x_{i+1} = F_i(x_i)$. These parts F_i are called advancing steps, or steps for short. The argument x_i and the result x_{i+1} define states where i or $i+1$ is called the state counter. The function F can be viewed as an evolution comprising l steps F_i. Each x_i represents an intermediate state of the evolution process. Usually, these x_i are quite large vectors of constant size.

The goal is to reverse the function F starting at state x_l down to state x_0 as illustrated by Fig. 1. Performing the overall reversal, one would like to achieve an optimal usage of time and computer resources. For every advancing step F_i, a corresponding $\hat{F}_i$ exists. This so-called preparing step not only evaluates F_i but also assembles data needed for the later reversal during the computation of the advancing step F_i. The data is stored in a special data structure called the trace. The trace is needed by the sequential reversal function $\bar{F}_i : \mathbb{R}^{\bar{n}_{i+1}} \longrightarrow \mathbb{R}^{\bar{n}_i}$ with $\bar{x}_{i+1} \longmapsto \bar{x}_i = \bar{F}_i(\bar{x}_{i+1}, x_i)$.

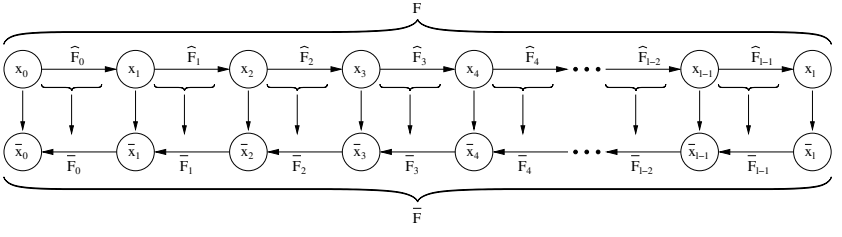


Fig. 1. The Reversal of the Function F

The naïve way to reverse F is to store all information one needs for the reversal onto a global trace. Thus the naïve reversing approach reads: Perform l preparing steps and after that perform l reversing steps as shown in Fig. 1. For real-world problems it may happen that the complete trace of F does not fit into memory. However, it is probably possible to keep a few intermediate states x_i at any time. The states kept in memory can be used as checkpoints in order to perform the desired reversal. In the remainder of this section, some basics of such checkpointing strategies are presented.

As in [4], it is assumed that the time t_i that one advancing step F_i takes is equal to a constant number $t \in \mathbb{R}$ that can be normalised to $t = 1$. Such evolutions F with $t_i = 1$ are said to have uniform step cost. Furthermore, let the value $\hat{t}_i$ be the time needed to perform one preparing step $\hat{F}_i$, and $\bar{t}_i$ the time needed to perform one reversing step $\bar{F}_i$. For simplicity it is assumed in this paper that $t_i = \hat{t}_i$, i.e. $\hat{t}_i = \hat{t} = 1$. If an evolution F has uniform step costs and if there exists a number $\bar{t} \in \mathbb{N}$, such that $\bar{t}_i = \bar{t}$ for all i, then the evolution is called a uniform evolution. Using a parallel schedule, the minimal time to reverse a uniform evolution that comprises l steps is given by

$$t_{\min} = l + l\bar{t} = (\bar{t} + 1)l \ . \tag{1}$$

Reversal schedules achieving this property are called time-minimal. Let $\varrho = c + p$ be the number of resources where c is the maximal number of checkpoints that can be stored at any time and p the maximal number of available processors. Then the maximal number of steps l_ϱ that can be reversed in minimal time with ϱ resources satisfies

$$l_\varrho = \begin{cases} \varrho & \text{if } \varrho \leq 2 \\ l_{\varrho-1} + \bar{t}\, l_{\varrho-2} & \text{else} \end{cases} \ . \tag{2}$$

The proof for this formula is given in [4]. When $\bar{t} = 1$, the formula (2) recursively yields the ϱ-th Fibonacci numbers. The direct non-recursive approximation

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

shows the exponential behaviour of the maximal number of steps l_ϱ that can be reversed for a given ϱ.

Vice versa, the formula (2) can be used to determine the minimal resources needed for the reversal of a given number of steps l. For example: Let the evolution comprise l steps with $l \in (f_\varrho, f_{\varrho+1}]$. Here, f_ϱ denotes the ϱ-th generalised Fibonacci number. Then according to formula (2), the number of required computer resources equals ϱ. For that purpose, it is assumed that each resource can perform an advancing, preparing or reversing step, or can act as a checkpoint. This property is called processor-checkpoint-convertibility in [3].

2.2 Building optimal parallel reversal schedules

For $l = 8$ steps, an optimal reversal schedule is displayed on the left of Fig. 2. The vertical axis in Fig. 2 can be thought of as the time or computational axis.

The horizontal axis denotes the state counter. All vertical solid lines denote checkpoints. The diagonal lines from the top left to the bottom right represent the execution of the step F_i if it is solid or $\hat{F}_i$ if dotted. Both functions compute the state x_{i+1} using the state x_i as an input value (horizontal axis) within one computational time step (vertical axis). The dashed lines represent the reverse steps $\bar{F}_i$ running from state x_{i+1} back to state x_i within one time unit. The three other diagrams show the resources needed. The value c^j denotes the checkpoints, p^j the processors, and s^j the computing resources, i.e. the sum of checkpoints and processors. This function is also called resource profile because it describes the resource requirement at any time of the reversal process.

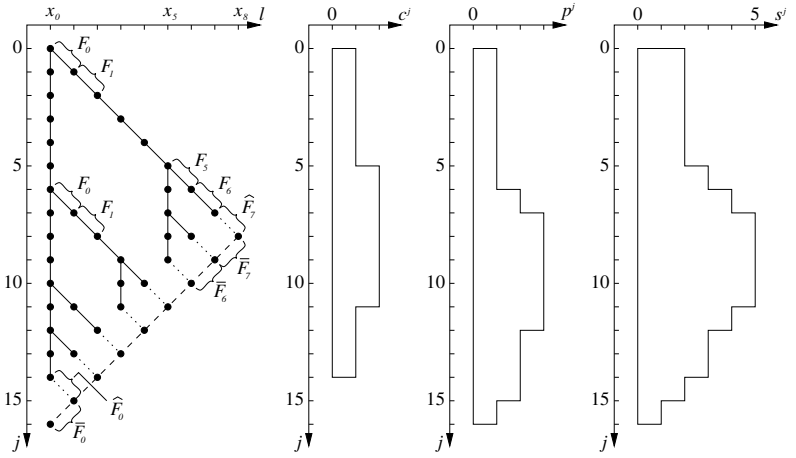


Fig. 2. An optimal schedule for $l = 8$ steps with $\bar{t} = \hat{t} = 1$

The recursive construction instruction of an optimal reversal schedule can be derived from the proof of (2) given in [4]. The first forward integration of the l steps is divided into parts corresponding to the Fibonacci-numbers (left hand diagram in Fig. 3). Hence, a schedule S^0 for l steps with $l \in (f_\varrho, f_{\varrho+1}]$ will be built from the schedule S^1 for $f_{\varrho-1}$ steps and from schedule S^2 for $l - f_{\varrho-1}$ steps as shown in Fig. 3. This is recursively done for all sub-schedules down to trivial single step reversal. By inspection of Fig. 4 we note that, the resources cannot be allocated statically to the various sub-schedule reversals, but must migrate between them during the execution of these subtasks. First the startup computation (dark grey area in the resource profile of Fig. 4) requires two resources. Then as the right and left sub-schedules commence being executed at $j = 5$ and $j = 6$, they both have increasing resource demands until the right sub-schedule begins to wind down at $j = 9$. It then passes resources to the left sub-schedule, which is still winding up. This behaviour is independent of the resource type (checkpoint, processor).

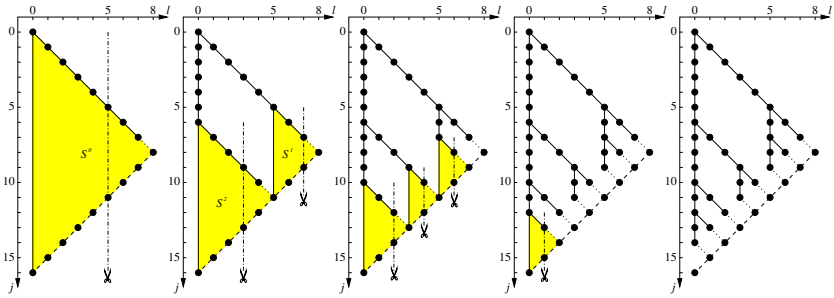


Fig. 3. Recursive construction of an optimal schedule for $l = 8$ steps and $\hat{t} = \bar{t} = 1$

3 Implementation Strategies

This section presents several possible strategies to implement reversal schedules on parallel computers. Furthermore, the strategy used for the numerical example introduced in Sect. 4 is discussed in detail.

3.1 Programming model

The programming model assumes a 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 or MPI. By implementing a reversal schedule using the distributed memory model, one can distinguish critical and non-critical communication. If a communication is classified as critical, the data written by one process at the end of one advancing, preparing or reversing step at a time j is needed for the beginning of the next advancing, preparing or reversing step by another process at the same time j. This is independent of the type of data required. If the communication is non-critical, the time between writing and reading of the data set is at least as long as the minimum of the time of 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.

3.2 User provided routines

To use the provided reversal schedule program skeleton, the user has to write three major computing routines [5, 6]. First an advancing routine is needed, which will be referred to as forward(..). This routine has to compute for a given state i the next state $i+1$. The second user routine does the same, but additionally stores all tracing data required for the reversing while it does the advancing.

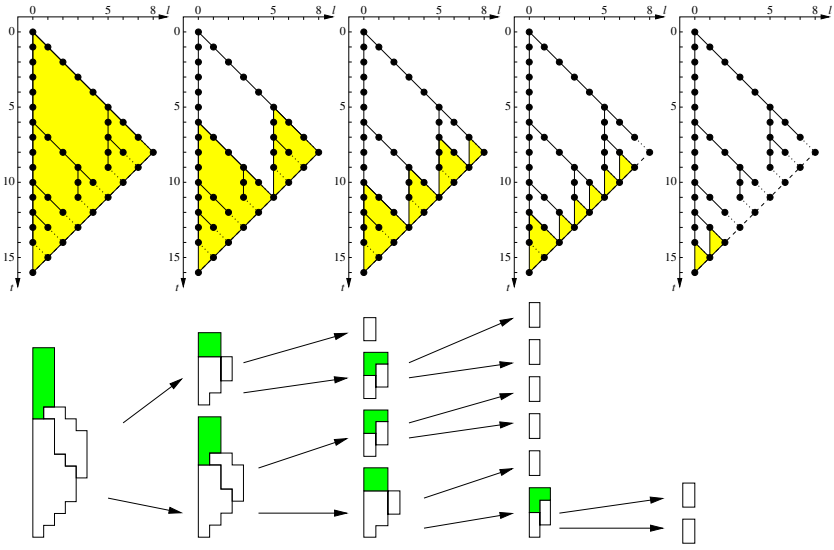


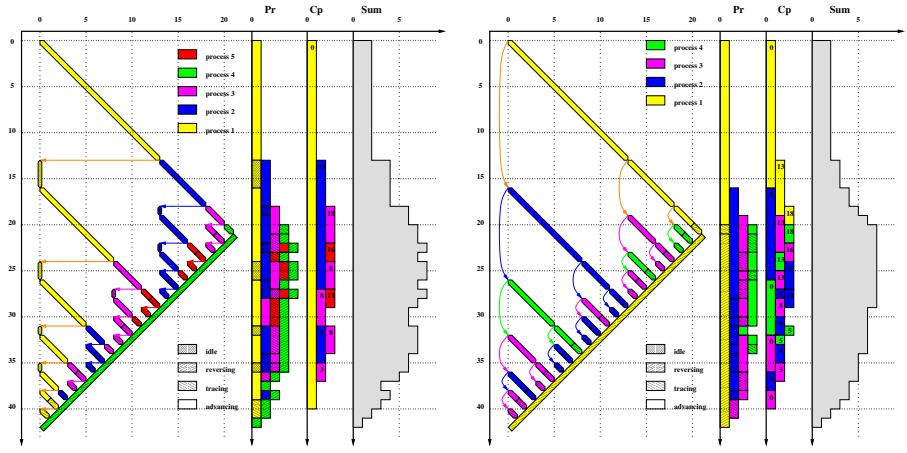
Fig. 4. Recursive construction of the resource profile for an optimal schedule for $l = 8$ steps and $\hat{t} = \bar{t} = 1$

Hence, it computes the state $i + 1$ for a given state i and the trace for this computation. This routine is referred to as preparing(..). The third user function is reverse(..), which carries out the reversing step. The input for this routine is the trace for a computation from state i to state $i + 1$. If reverse(..) is called for the first time, i.e. it is called at state l, it carries out the required initialisation of so-called adjoint values for the reversal computation. Otherwise, a second input argument is required, namely the values of the reversal computed so far. These values represent the reversal from state l down to state $i + 1$.

3.3 Schedules for distributed memory programming models

There are two ways to implement a parallel reversal schedule in a distributed memory style. In both cases it will be assumed that a pool of available processors exists. For simplification $\bar{t}$ is assumed to be 1.

First, one may assign each checkpoint to a fixed processor. This approach, shown in Fig. 5(a), is called checkpoint oriented. A processor receives a checkpoint, stores it and at a predetermined time the processor carries out the advancing up to the state where the next checkpoint has to be written. Once it reaches the state, the processor writes the checkpoint and sends it to the next available processor. Then the processor goes back, i.e. waits at its assigned checkpoint, which it has received, until called upon to advance once more. The number of steps in each advance monotonically decreases until the checkpoint can be vacated after serving as starting point for the preparing step. When using this implementation, all communication is critical. Because of the idle time, this



(a) checkpoint oriented (b) process oriented

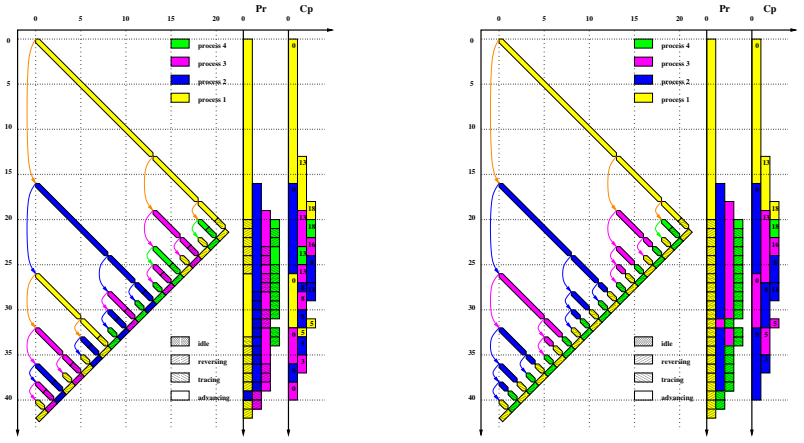
Fig. 5. General implementation strategies

scheme can not satisfy the optimal requirement profile [3] if more than three steps have to be reversed.

The second way to assign the processes to the given schedules is shown in Fig. 5(b). It is called process oriented. The computing process receives a checkpoint and carries out the advancing until it reaches its final state. The final state can be the state where the last advancing step ends and the preparing step starts, the state where the preparing step ends and the reversing step starts or the state where the reversing step ends. Along the way, the process writes, stores and sends the needed checkpoints. After the processors have finished the assigned computation they return back to the pool of available processors. Except at the final state, all communication is non-critical. The process oriented implementation fulfils the optimality if the following is true for all schedules and sub-schedules: The checkpoint written at state $i = 0$ is needed before the processor which does the advancing from state $i = 0$ up to state $i = l$ reaches the final state $i = l$. As already mentioned above, 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 preparing process (Fig. 6(b));
- carries out the writing of the trace and sends it to a reversing process (Fig. 5(b));
- carries out the writing of the trace and reversing by itself (Fig. 6(a)) and sends the reversing result to the next reversing process.

In the first case, the task preparing and reversing can be assigned to fixed processes or might be combined to one task and must be carried out alternately by



(a) Processes carry out the preparing and the reversing by themselves

(b) Two processes carry out preparing and reversing alternately

Fig. 6. General implementation strategies

two processes (Fig. 6(b)). 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 (Fig. 6(a) or Fig. 6(b)). If the result of the reversing step is relatively large compared to the size of the result of a preparing step, a fixed reversing processor reduces the communication cost (Fig. 5(b)) This invalidates the previous approach if both trace and adjoint are relatively large.

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. On the other hand, using a process oriented approach for implementing an optimal reversal schedule, one processor has to store up to three checkpoints temporarily. This is caused by the fact that the sub-schedule S^1 starts before the sub-schedule S^2. The upper bound of three checkpoints is established in the following lemma.

Lemma 1. *Suppose the number of steps to be reversed is l with $l \in (f_\varrho, f_{\varrho+1}]$, where f_ϱ is the ϱ-th Fibonacci number. The schedule is implemented in a process oriented manner. Any process may store as many checkpoints as required, i.e. the checkpoints will be sent as late as possible. Then any 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 r by $r = l - f_{\varrho-1}$, hence $r \in (f_{\varrho-2}, f_{\varrho}]$. As shown in [3], the times $t_W(i)$, where the i-th checkpoint is written, are recursively defined by $t_W(1) = 0$, $t_W(2) = r$ and $t_W(i) = t_W(i-1) + f_{\varrho-2i+4}$. The maximal number of checkpoints written is limited above by $\lceil \frac{\varrho}{2} \rceil + 1$. The time when a checkpoint is needed/read is defined by $t_R(i) = t_W(i) + 2f_{\varrho-2i+1}$ [3]. Hence, the lemma is proven if the inequality $t_R(i) < t_W(i+3)$ holds for any i. One has $t_W(i+3) = t_W(i) + f_{\varrho-2i-2} + f_{\varrho-2i} + f_{\varrho-2i+2}$. Furthermore, one obtains

$$\begin{array}{lll} & t_R(i) & < t_W(i+3) \\ \iff & t_W(i) + 2f_{\varrho-2i+1} & < t_W(i) + f_{\varrho-2i-2} + f_{\varrho-2i} + f_{\varrho-2i+2} \\ \iff & 0 & < 2f_{\varrho-2i-2} \end{array}$$

which is true for all $i \leq \lceil \frac{\varrho}{2} \rceil + 1$. □

4 Example

The implementation of the reversal schedules was tested with the simulation of a simple Formula 1 racing car model. As described in [6], the parallel reversal schedule was used to compute the adjoint of the forward integration of an ODE system with respect to a given road shape. The forward integration was carried out using a four-stage Runge-Kutta scheme and for the adjoint calculation its adjoint Runge-Kutta scheme was used [7]. The computations were measured on a Cray T3E and on a SGI Origin3800. A parallel reversal schedule for $l = 55$ steps was and hence five processors were utilised. A sixth processor (master) was used to organise the program run. In Table 1, the memory requirements and the runtimes are listed for naïve approach and the parallel approach. Compared to the naïve approach, only two percent of the initial memory was needed using the parallel schedules. As the theory has shown, the runtime stays almost the same in all test cases. The slight improvement in computing time may be due to less memory traffic [8].

Table 1. Memory requirement and Runtime

Approach	Values	Memory	Time (T3E)	Time (Origin3800)
Naïve	266010	2128.10 kB ≙ 100.0 %	20.27 s ≙ 100.0 %	6.71 s ≙ 100.0 %
Parallel	5092	40.70 kB ≙ 1.9 %	18.91 s ≙ 93.3 %	6.04 s ≙ 90.0 %

5 Discussions, Conclusions and Further Work

The reversal of evaluation programs may cause problems due to the memory requirement being proportional to the computation time. A theory to reduce

the memory requirement and time-minimal parallel reversal schedules were presented. Various possibilities for implementing these parallel reversal schedules were discussed. Thereby, some pitfalls which may cause the loss of resource optimality were shown. Furthermore, we discussed the problem characteristics that determine the choice of the implementation strategy.

As shown by the numerical example, the parallelisation of the reversal process was carried out in time. However, in addition the function may also be parallelised in space. The parallel evaluation of the function has to be autonomous and independent of external influences. In order to use a parallel computer efficiently, the parallelisation in space has to distribute the computational work dynamically between the processors. This is necessary because the number of available processors for the function evaluation will change during the reversal. Thus a repartitioning of the problem may occur, which is generally difficult. Therefore the implementation effort for such two level parallelisation is significant larger.

The advantage of using a distributed memory programming model is that one may apply a shared memory parallelisation within the user functions for the parallelisation in space. Again, the user has to take care while implementing the user functions. The reason for this is that most of the commonly used message passing libraries (MPI, PVM) are not thread-safe by definition. Hence, the parallel execution of shared memory parallelised code can start after the function was called and must end when leaving this function. Possibly improved message passing libraries will remove such a restriction in future.

References

1. van de Snepscheut, J.: What computing is all about. Texts and Monographs in Computer Science. Springer Verlag, Berlin (1993)
2. Griewank, A.: Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optimisation Methods and Software **1** (1992) 35–54
3. Walther, A.: Program Reversal Schedules for Single- and Multi-processor Machines. PhD thesis, TU Dresden, Fakultät für Mathematik und Naturwissenschaften (1999)
4. Walther, A.: An upper bound on the number of time steps to be reversed in parallel. Technical Report IOKOMO–03–2001, Technische Universität Dresden (2001)
5. Benary, J.: DAP – Dresdener Adjugierten Parallelisierungsprojekt. Technical Report IOKOMO–05–1995, Technische Universität Dresden (1995)
6. Walther, A., Lehmann, U.: Adjoint calculation using time-minimal program reversals for multiprocessor machines. Technical Report IOKOMO–09–2001, Technische Universität Dresden (2001)
7. Hager, W.: Runge-Kutta methods in optimal control and the transformed adjoint system. Numer. Math. **87** (2000) 247–282
8. Seidl, S., Nagel, W.E., Brunst, H.: The future of HPC at SGI: Early experiences with SGI SN–1. In Jesson, B.J., ed.: Proceedings of Sixth European SGI/Cray Workshop, Manchester (2000)