Parallelism in the Reverse Mode

Jochen Benary*

Abstract

In the basic form of the reverse mode for calculating derivatives, the amount of memory needed to record the intermediate values can become excessively large for problems of practical interest. If sequential checkpointing schemes are used, the memory requirement can be dramatically reduced, but the run time may be significantly increased. Implementing suitable checkpointing schemes on multiprocessor systems can decrease the run time to its theoretical minimum. Among the many possible scheduling strategies, we develop one that minimizes resource requirements. We present different communication structures that depend on the memory architecture of the multiprocessor system and the available resources. We also estimate the limits of the complexity and the memory requirements of the problem function.

1 Introduction

Our starting point is a problem $P(f, \nabla f, g, (b1, \ldots, b_n))$ whose solution requires, in addition to the problem function $f : \mathcal{R}^\mu \to \mathcal{R}^\nu$, its derivative ∇f. It is assumed that the computation of the problem function f can be divided into a sequence of modules of the same complexity. Hence, the function f is computed by an iteration of length n according to the description

$$s_i = (x_i, r_i) = M(i, x_{i-1}, r_{i-1}, g, b_i) \ , \qquad\qquad i = 1, \ldots, n \ ,$$

with $f(x_0) = r_n$. In this context the vector x_0 contains the independent variables, r_0 is a constant initial value for the evaluation of f, and r_n contains the dependent variables. The pair $s_i = (x_i, r_i)$ are intermediate variables describing the "state" after the ith iteration step. The components of g are global constant data, which are identical in all the iteration steps. The vectors b_i are constant data for the ith step, which may vary from step to step.

Example. Let the differential equation $\dot{x}(t) = \mathbf{A} \cdot x$ be a description of some (physical) problem. Then the Euler discretization with constant increment h yields the iteration $x_i = x_{i-1} + h \cdot \mathbf{A} \cdot x_{i-1}$. For a set of given (measured) values b_i, the problem P is to find an $\tilde{x}$ that minimizes the function $f(\tilde{x}) = \sum_{i=1}^{n} (x_i - b_i)^2$ with $x_0 = \tilde{x}$ and $r_0 = 0$. In other words, we wish to find an $\tilde{x}$ for which the x_i approximate the b_i. The iteration algorithm to compute $f(\tilde{x})$ is

$$\begin{aligned} (x_i, r_i) &= (\ x_{i-1} + h \cdot \mathbf{A} \cdot x_{i-1} \ , \ r_{i-1} + (x_i - b_i)^2 \ ) \\ &= M(i, x_{i-1}, r_{i-1}, (\mathbf{A}, h), b_i) \ , \end{aligned}$$

and the minimization problem P can be solved by some gradient method.

In the basic form of the reverse mode for calculating derivatives, one must save (at least) all those intermediate values that occur in a nonlinear context during the evaluation of the underlying problem function. For this purpose one must derive from the original module M

*Institute of Scientific Computing, Technical University Dresden, D-01069 Dresden, Germany

- a module $\bar{M}$ that calculates one iteration step including the (partial) record $\mathcal{R}_i$ of the intermediates

$$(x_i, r_i, \mathcal{R}_i) = \bar{M}(i, x_{i-1}, r_{i-1}, g, b_i) \ , \qquad\qquad i = 1, \ldots, n \ ,$$

- and a module M' that calculates the derivative in a reverse mode for one iteration step

$$(\frac{\delta f}{\delta x_{i-1}}) = M'(i, \frac{\delta f}{\delta x_i}, \mathcal{R}_i, g) \ , \qquad\qquad i = n, \ldots, 1 \ .$$

Then one follows the scheme presented in Figure 1.

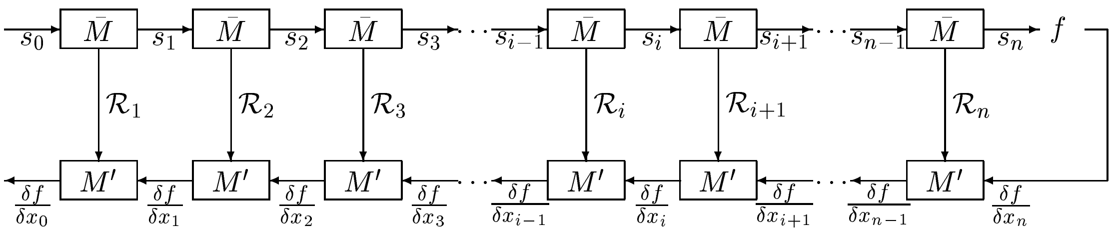

FIG. 1. *Gradient calculation in the reverse mode*

The amount of memory needed for this recording is proportional to the total number of steps n and can become so large that applying the reverse mode in its basic form is almost impossible. This problem can be solved by strategies that split the record in parts and calculate them backwards (see Figure 2). This procedure involves repeated forward evaluations from suitably selected checkpoints. In other words, when the record $\mathcal{R}_i$ is needed, the iteration is done until step $i-1$ without any recording and then one iteration step with recording is performed. To avoid starting from state s_0 every time, intermediate states $s_j = (x_j, r_j)$ are stored at suitably selected indices j as temporary starting points. Such states s_j are called *snapshots*. Griewank [2] has proposed a modification of the reverse mode based on this idea, which limits the growth in both spatial and temporal complexity to a multiple of $\log_4 n$ compared with the original function evaluation.

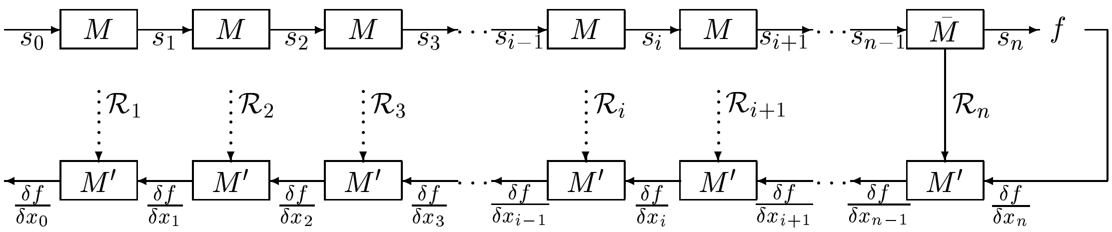

FIG. 2. *Partial calculation of the record*

But the problem remains that the auxiliary calculations may represent the bulk of the computational effort and thus slow the actual adjoint computation considerably. Therefore, it seems natural to employ a certain number of auxiliary processes for the repeated forward sweeps while one designated process performs the reverse sweep without any interruption.

Let t_M be the time needed to calculate one iteration step by using the module M. Then the scheme in Figure 2 yields the following turnaround time needed to compute a derivative:

$$
\begin{aligned}
T &= (n-1)\cdot t_M + t_{\bar{M}} + n\cdot t_{M'} = (n-1)\cdot t_M + \beta\cdot t_M + n\cdot\alpha\cdot t_M \\
&= ((n-1)+\beta+n\cdot\alpha)\cdot t_M \ ,
\end{aligned} \tag{1}
$$

where $t_{M'} = \alpha \cdot t_M$ and $t_{\bar{M}} = \beta \cdot t_M$ are the time needed to execute module $\bar{M}$ and M', respectively. We emphasize that this turnaround time becomes smaller than that of the sequential algorithm in Figure 1. Because the computation of the derivative is usually more time consuming than the simple forward step, we may assume $\alpha \geq 1$. Possibly, M' computes first a record and then the derivative, and $\bar{M}$ transfers only the corresponding state and environment data. Hence we demand only $\beta > 0$.

2 The Evaluation Strategy

In this section we deduce a strategy for the use of multiple processors in derivative calculation in which the turnaround time needed is given by the equation (1). As mentioned above, the basic idea is to have one process that calculates the derivative "backwards" and several processes that deliver the parts of the record just in time. The aim is to find the maximum number of iteration steps that can be done with a given number of processors.

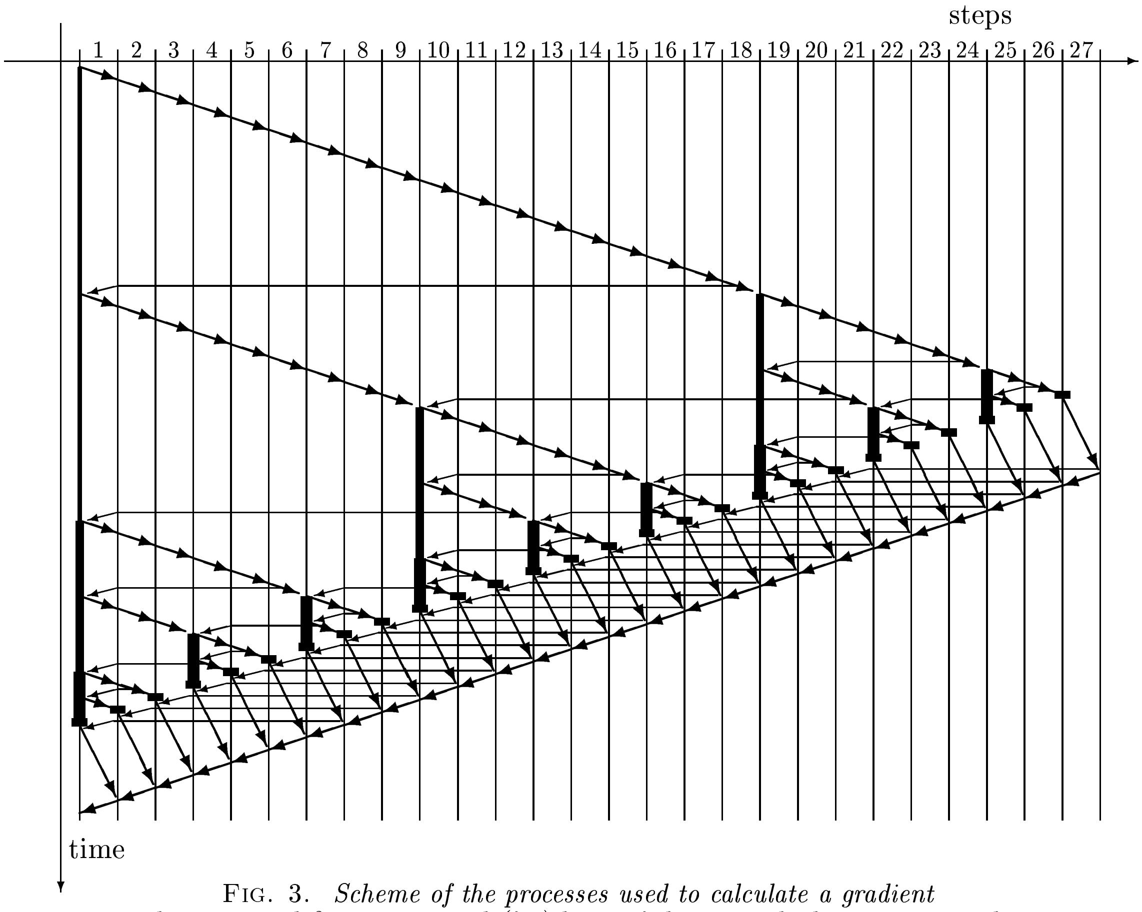

FIG. 3. *Scheme of the processes used to calculate a gradient with $\alpha = 1$ and $\beta = 6$. Vertical (fat) lines of the same thickness are snapshots belonging to the same level. Thick arrows represent one step of M, $\bar{M}$, or M'. Concatenated arrows show the run of one processor.*

2.1 Distribution of the Snapshots

We start with the following concept.

1. It was assumed that the computation of the problem function f can be divided into a sequence of modules of the same complexity. From this it follows that the values of t_M, $t_{M'}$, and $t_{\bar{M}}$ are constant.
2. Snapshots are used not only to compute records but also to compute snapshots. Hence, the snapshots can be assigned to different levels. The lowest level indicated by 0 contains all snapshots that are used to compute records directly. Level 1 contains all snapshots that are used to compute snapshots of level 0. In general, level i contains all snapshots that are used to compute snapshots of level $i-1$. A snapshot may occur at various levels.
3. To compute the snapshots of the level $i-1$ or the records, a certain number p_i of processors are used, and we will assume that they are used exclusively for this task. As we will see later, this assumption leads to a rather simple and robust implementation of the schedule, with a low overhead of communication control. Because no loops occur in the transmission scheme and the processor use, problems such as deadlock are avoided.

The question is, How many processors must be assigned to the different sequences to obtain a maximum iteration length? Figure 3 illustrates the concept and gives an idea of what we obtain: use the minimal processor number feasible at each level.

THEOREM 2.1. *Assuming the concept mentioned above, an optimal strategy consists in assigning* $p_{\mathcal{R}} = \left\lceil \frac{\beta}{\alpha} \right\rceil$ *processors to the "recording" level* 0 *and one processor to each higher level. If* n_p *processors are available, we have* $p_{\mathcal{S}} = n_p - 1 - p_{\mathcal{R}}$ *processors to produce snapshots, and the maximum iteration length* n_{max} *is given by*

$$n_{max}(n_p) = \Delta_{\mathcal{R}} \cdot \Delta^{p_{\mathcal{S}}} \ , \tag{2}$$

with $\Delta_{\mathcal{R}} = \lfloor \alpha \cdot p_{\mathcal{R}} + 1 - \beta \rfloor$ *and* $\Delta = 2 + \lfloor \alpha \rfloor$.

Proof. To show this, we execute the following steps: find the minimal number of processors needed, calculate the greatest distance and estimate the distance for more processors. Level 0 is considered first. The snapshots are suitably selected states s_{k_l} and can be ordered with decreasing indices $k_1, k_2, \ldots$. This sequence is denoted by $\mathcal{S}_0 = (\mathcal{S}_{0,1}, \mathcal{S}_{0,2}, \ldots)$ with $\mathcal{S}_{0,l} = s_{k_l}$. Since we wish to produce a record as fast as possible, its calculation starts from the nearest snapshot. This process, starting from a snapshot of $\mathcal{S}_0$ and producing a record $\mathcal{R}_j$, is called a *propagation* and can be identified by the number $n-j$. The *length* of a propagation is the number of iteration steps it contains. Let λ be the length of a propagation. Then the execution time of this propagation is obtained by $(\lambda - 1) \cdot t_M + t_{\bar{M}}$. The *distance* between two propagations j_1 and j_2 is the number of records $\mathcal{R}_j$ between their results, that is $j_2 - j_1$. The time between the moments the results of the propagations j_1 and j_2 are needed is given by $(j_2 - j_1) \cdot t_{M'}$. As mentioned above, a number p_0 of processors are available exclusively used at this level to produce the records. Let the propagation j_1 be executed immediately before the propagation j_2 by the same processor, let all the other propagations between j_1 and j_2 be executed by the other processors, and let λ_{j_2} be the length of the propagation j_2. Then the propagation j_2 is called *critical* if

(1) $(j_2 - j_1) \cdot t_{M'} \geq (\lambda_{j_2} - 1) \cdot t_M + t_{\bar{M}}$

and

(2) this inequality does not apply for any $\hat{\lambda}_{j_2} > \lambda_{j_2}$.

In other words, λ_{j_2} is the maximum length, so that propagation j_2 can be executed immediately after propagation j_1 and produces the record $\mathcal{R}_{j_2}$ in time. Criticality depends on the distance between the snapshots of the sequence and the number of available processors. Therefore, the number of processors p_0 working on level 0 determines the greatest possible distance $\Delta_0(p_0)$ between two snapshots in the sequence $\mathcal{S}_0$, and we get

$$\mathcal{S}_{0,l} = s_{k_l} \quad \text{with} \quad k_l = n - l \cdot \Delta_0(p_0) \ ,$$

with n the number of iteration steps. In other words, we obtain a uniform distribution of snapshots on the recording level, which is an indirect consequence of our assumption that a fixed number of processors is assigned to this level. As we will see later, this property is inherited by the higher levels.

To obtain the minimal number $p_\mathcal{R}$ of processors needed, we look at one particular processor. The time between the completion of one record and the next produced by this processor is given by $p_\mathcal{R} \cdot t_{M'}$ because the $p_\mathcal{R} - 1$ records between them are produced by the other processors. During this time the processor must calculate this next record (i.e., $p_\mathcal{R} \cdot t_{M'} \geq t_{\bar{M}}$):

$$p_\mathcal{R} = \left\lceil \frac{\beta}{\alpha} \right\rceil \ . \tag{3}$$

In addition, the processor may execute some simple forward steps. Let $\Delta_\mathcal{R} = \Delta_0(p_\mathcal{R})$ be the greatest distance between two snapshots in $\mathcal{S}_0$. Then $\Delta_\mathcal{R}$ must satisfy $p_\mathcal{R} \cdot t_{M'} \geq t_{\bar{M}} + (\Delta_\mathcal{R} - 1) \cdot t_M$, which yields

$$\Delta_\mathcal{R} = \lfloor \alpha \cdot p_\mathcal{R} + 1 - \beta \rfloor \leq 1 + \lfloor \alpha \rfloor \ . \tag{4}$$

Thus, $\Delta_\mathcal{R} - 1$ is the number of steps M that can be taken by one processor after finishing the calculation of one record and before starting the next execution of $\bar{M}$.

If $p_\mathcal{R} + 1$ processors are available at level 0, we have

$$\Delta_0(p_\mathcal{R}{+}1) \leq \Delta_\mathcal{R} \cdot (2 + \lfloor \alpha \rfloor) \ . \tag{5}$$

To show this, we start with the case $p_\mathcal{R} = 1$. Then $\alpha \geq \beta$, and the critical propagation is the first propagation from a snapshot. The preceding propagation done by the same processor must be $\Delta_\mathcal{R} + 1$ records apart because only $\Delta_\mathcal{R}$ propagations can be executed by the other processor. Thus

$$(\Delta_\mathcal{R} + 1) \cdot t_{M'} \geq (\Delta_0(p_\mathcal{R} + 1) - 1) \cdot t_M + t_{\bar{M}} \qquad \text{and}$$

$$\Delta_0(p_\mathcal{R} + 1) = \lfloor (\Delta_\mathcal{R} + 1) \cdot \alpha + 1 - \beta \rfloor \leq \Delta_\mathcal{R} \cdot \lfloor \alpha \rfloor + \Delta_\mathcal{R} - 1 + 1 \leq \Delta_\mathcal{R} \cdot (2 + \lfloor \alpha \rfloor) \ .$$

The other possibility is $p_\mathcal{R} > 1$. Here, too, the critical propagation is the first propagation from a snapshot. In this case the preceding propagation done by this processor must be $p_\mathcal{R} + 1$ records apart. Therefore

$$(p_\mathcal{R} + 1) \cdot t_{M'} \geq (\Delta_0(p_\mathcal{R} + 1) - 1) \cdot t_M + t_{\bar{M}} \qquad \text{and}$$

$$\Delta_0(p_\mathcal{R} + 1) = \lfloor (p_\mathcal{R} + 1) \cdot \alpha + 1 - \beta \rfloor \leq \Delta_\mathcal{R} + \lfloor \alpha \rfloor + 1 \leq \Delta_\mathcal{R} \cdot (2 + \lfloor \alpha \rfloor) \ .$$

Next, the levels i with $i > 0$ are considered. Analogous to the records, the snapshots of $\mathcal{S}_0$ need not be available permanently but only at a certain time. In a similar way as above we get a sequence of snapshots $\mathcal{S}_1 = (\mathcal{S}_{1,1}, \mathcal{S}_{1,2}, \ldots)$ at one level higher. This process can be continued until the sequence at the highest level contains only the initial state s_0. At every level i we get

(6) $$\mathcal{S}_{i,l} = s_{k_l} \ , \quad \text{with} \quad k_l = n - l \cdot \Delta_i(p_i) \ ,$$

where $\Delta_i(p_i)$ is the maximum distance between two snapshots of the sequence $\mathcal{S}_i$ under the assumption that a fixed number of processors p_i is assigned to the level i. Thus, the maximum number of iterations is limited by the number of processors.

Because $\alpha \geq 1$, the minimal number of processors needed is one. With one processor the maximum distance $\Delta_i(1)$ is

(7) $$\Delta_i(1) = \Delta \cdot \Delta_{i-1}(p_{i-1}) \qquad \text{with} \qquad \Delta = 2 + \lfloor \alpha \rfloor \ .$$

This follows from the fact that, in this case, the critical propagations are always the ones that start as the second propagation (in time) from a given snapshot. Thus, $\Delta_{i-1}(p_{i-1}) \cdot t_{M'} \geq (\Delta_i(1) - 2\Delta_{i-1}(p_{i-1})) \cdot t_M$ must be valid. If the first condition of critical propagations holds for the first two propagations of a snapshot, it necessarily holds for the subsequent pairs of propagations of this snapshot as well. The distance between the last propagation of a snapshot and the first of the next one is 2, which corresponds to $2\Delta_{i-1}(p_{i-1}) \cdot t_{M'} \geq (\Delta_i(1) - \Delta_{i-1}(p_{i-1})) \cdot t_M$.

If two processors are available, one processor can execute the last $1 + \lfloor \alpha \rfloor$ propagations, as shown before. For executing further propagations, the processors must act by turns. Hence, the third propagation of a snapshot could be a critical propagation. From the corresponding inequality $2\Delta_{i-1}(p_{i-1}) \cdot t_{M'} \geq (\Delta_i(2) - 3\Delta_{i-1}(p_{i-1})) \cdot t_M$, there follows as an upper boundary

(8) $$\Delta_i(2) \leq 3 + 2\lfloor \alpha \rfloor \ .$$

The equation (2) is derived from the recurrence (7) with $p_0 = p_{\mathcal{R}}$ and $p_i = 1$ for $i = 1, \ldots, p_{\mathcal{S}}$. If at any level we use one processor more than required, the number of levels decreases by one, and the corresponding factor $\Delta_{\mathcal{R}}$ or Δ must be replaced by $\Delta_0(p_{\mathcal{R}} + 1)$ or $\Delta_k(2)$, respectively. The inequality (5) or (8) implies that $n_{max}(n_p)$, as defined in (2), is the maximum. □

Since α and β are constant for a certain problem function, the number $p_{\mathcal{R}}$ of processors needed to generate the partial records is constant, too. Therefore, the formula (2) implies that the number of processors grows with $\mathcal{O}(\log_{\Delta} n)$ as a function of the length of the iteration n. As we will see, this holds for the memory needed as well. Comparing this result with the symmetrical scheduling strategy of [2], we may conclude that the logarithmic growth of the run time in the sequential scheduling method is replaced by the logarithmic growth of the processor requirement in our parallel scheduling scheme.

In the underlying concept we have assumed that each processor is permanently assigned to one level. But at the beginning and the end of the evaluation of a derivative not all processors are involved in the computation (see Figure 3). During this time they could work at other levels. In this sense the strategy presented here is only suboptimal.

2.2 Use of the Resources

To derive the strategy presented here, we have assumed constant values for the times t_M, $t_{M'}$, and $t_{\bar{M}}$. In most cases, this assumption does not hold for an actual task. Hence, some safety margin in the utilization ratio of the processors is advisable. As it will be shown, this is true for the processors generating snapshots.

The average utilization ratio of one of those processors calculating the records can be found by looking at a time interval of duration $\Delta_{\mathcal{R}} \cdot t_{M'}$. During this period the processors must calculate $\Delta_{\mathcal{R}}$ records. Therefore, one processor must work for $\sum_{i=0}^{\Delta_{\mathcal{R}}-1}(i \cdot t_M + t_{\bar{M}})/p_{\mathcal{R}}$ on average. Then the average utilization ratio is

$$\eta_0 = \frac{\sum_{i=0}^{\Delta_{\mathcal{R}}-1}(i \cdot t_M + t_{\bar{M}})}{p_{\mathcal{R}} \cdot \Delta_{\mathcal{R}} \cdot t_{M'}} = \frac{\Delta_{\mathcal{R}} + 2\beta - 1}{2p_{\mathcal{R}} \cdot \alpha} \ . \tag{9}$$

Setting $\beta = p_{\mathcal{R}} \cdot \alpha - \epsilon$ with $\epsilon \in [0, \alpha)$, we get

$$\frac{p_{\mathcal{R}} - 1}{p_{\mathcal{R}}} < \eta_0 \leq 1 \ .$$

If $\epsilon = 0$, then $\Delta_{\mathcal{R}} = 1$ and $\eta_0 = 1$. Using one more processor than required, we get $\eta_0 = 1 - 1/(p_{\mathcal{R}} + 1)$ in this case.

Now, for the processor producing the snapshots of $\mathcal{S}_{i-1}$, we look at the time interval $\Delta_{i-1}(p_{i-1}) \cdot t_{M'}$. During this time it must execute all propagations that start from one snapshot of $\mathcal{S}_i$. The time needed is $\sum_{i=1}^{\Delta-1} i \cdot \Delta_{i-1}(p_{i-1}) \cdot t_M$. Therefore, the utilization ratio is

$$\eta_i = \frac{\sum_{i=1}^{\Delta-1} i \cdot \Delta_{i-1}(p_{i-1}) \cdot t_M}{\Delta_{i-1}(p_{i-1}) \cdot t_{M'}} = \frac{1 + \lfloor\alpha\rfloor}{2\alpha} \tag{10}$$

and can be estimated from

$$\frac{1}{2} < \eta_i \leq \frac{\alpha + 1}{2\alpha} \ .$$

Only some of the snapshots of a sequence must be available at a given time. Every snapshot $\mathcal{S}_{i,j}$ is also contained in the sequence $\mathcal{S}_{i-1}$ as snapshot $\mathcal{S}_{i-1,j\frac{\Delta_i}{\Delta_{i-1}}}$. Hence, the latest moment t_E this snapshot is needed is when the direct computation of the record starts. The first moment the snapshot $\mathcal{S}_{i,j-k}$ is needed can be expressed as

$$\begin{aligned} t_B &= t_E + t_{\bar{M}} + (k-1) \cdot \Delta_i \cdot t_{M'} + t_{M'} - t_{\bar{M}} - (\Delta_i - 1) \cdot t_M \\ &= t_E + ((k-1) \cdot \Delta_i + 1) \cdot t_{M'} - (\Delta_i - 1) \cdot t_M \ . \end{aligned}$$

Both snapshots are needed simultaneously if $t_B < t_E$. It follows that

$$k < \frac{\Delta_i(p_i) - 1}{\Delta_i(p_i)} \cdot \frac{\alpha + 1}{\alpha} \ .$$

Since

$$1 < \frac{\Delta_i(p_i) - 1}{\Delta_i(p_i)} \cdot \frac{\alpha + 1}{\alpha} < 2 \qquad \text{and} \qquad \frac{\Delta_{\mathcal{R}} - 1}{\Delta_{\mathcal{R}}} \cdot \frac{\alpha + 1}{\alpha} \leq 1 \ ,$$

the number ν_i of snapshots needed at the same moment is

$$\nu_i = \begin{cases} 1 & \text{if} \quad i = 0 \quad \text{and} \quad p_0 = p_{\mathcal{R}} \\ 2 & \text{otherwise} \end{cases} \ . \tag{11}$$

The results of this section will be used to obtain the algorithms proposed in the next one.

3 Some Remarks on the Implementation of the Strategy

Two basic architectures can be distinguished for multiprocessor systems of the MIMD type: shared– and distributed–memory architecture. The advantage of the shared–memory architecture is that the data themselves need not be transferred. Hence, the assumption that the transfer time can be ignored is appropriate on such architectures. The snapshots that must be available at the same moment can be stored in a field of the length $2p_{\mathcal{S}}$ in the common memory. This length arises from the fact that $\nu_i = 2$ if $i > 0$, $\nu_0 = 1$ and $\nu_{p_{\mathcal{S}}} = 1$, since $\mathcal{S}_{p_{\mathcal{S}}}$ contains only the snapshot of the initial state. Moreover, storage is needed for the $p_{\mathcal{R}}$ records. If we denote the size of a snapshot and a record as $M_{\mathcal{S}}$ and $M_{\mathcal{R}}$, respectively, and the size of memory needed by the modules M, $\bar{M}$, and M' as M_M, $M_{\bar{M}}$, and $M_{M'}$, respectively, we can estimate the required memory as

$$M_{shared} = p_{\mathcal{S}} \cdot (M_M + 2M_{\mathcal{S}}) + p_{\mathcal{R}} \cdot (M_{\bar{M}} + M_{\mathcal{R}}) + M_{M'} \ . \tag{12}$$

Since $p_{\mathcal{S}}$ depends on the iteration length logarithmically, we find that the memory needed grows with an order of $\mathcal{O}(\log_\Delta n)$.

```
snapshot(n,i,Δ,I₁ⁱ,I₂ⁱ)
   Index_i=EMPTY;
   target_index=I₁ⁱ;
   for j=1 ... ⌈n/Δ^i⌉
      wait until Index_i != EMPTY;
      source_index=Index_i; Index_i=EMPTY;
      for l=Δ ...1
         wait until Index_{i-1} == EMPTY;
         S_{i,j} = State[source_index];
         produce S_{i-1,j·Δ-l};
         State[target_index] = S_{i-1,j·Δ-l};
         Index_{i-1} = target_index;
         if (target_index == I₁ⁱ) target_index=I₂ⁱ;
         else target_index=I₁ⁱ;
      end
      wait until Index_{i-1} == EMPTY;
      Index_{i-1} = source_index;
   end
end
```

FIG. 4. *Producing snapshots in a shared–memory architecture*

Figure 4 outlines the algorithm to produce snapshots of $\mathcal{S}_{i-1}$. The values $\mathtt{I}_1^i$ and $\mathtt{I}_2^i$ are the indices of the elements of the common field (here referred to as `State`) where the produced snapshots can be stored. The variable $\mathtt{Index}_i$ is used to control the communication and contains the index of that field element that contains the present snapshot of $\mathcal{S}_i$. The value `EMPTY` indicates that no snapshot is available at the moment. The processes that produce the records work in a similar way.

In the case of a distributed–memory architecture the data transfer time influences the computational time. This fact can be taken into account by choosing appropriate values for t_M, $t_{\bar{M}}$, and $t_{M'}$. In this case the upper boundary of the memory needed by the different processes is

$$M_{distributed} = \max\{M_M + 2M_{\mathcal{S}}, M_{\bar{M}} + M_{\mathcal{S}} + M_{\mathcal{R}}, M_{M'} + M_{\mathcal{R}}\}\ . \tag{13}$$

A substantial part of the communication time is the time needed to transfer the data across the communication network. Hence, asynchronous sending and receiving are useful; these can be achieved by multithreading. Then the redundancy of the utilization ratio of the processors reduces the influence of the transfer time. But additional memory is needed in the communication thread as interim storage. The estimate of the memory here is

$$M_{distributed} = \max\{M_M + 3M_{\mathcal{S}}, M_{\bar{M}} + 2M_{\mathcal{S}} + 2M_{\mathcal{R}}, M_{M'} + 2M_{\mathcal{R}}\}\ . \tag{14}$$

The growth of all needed memory has the order $\mathcal{O}(\log_\Delta n)$, too. But the more interesting result is that the amount of memory needed by one processor is independent of the iteration length in this case.

snapshot(n,i,Δ)

receive $\mathcal{S}_{i,1}$;
j=1;
loop
  copy $\mathcal{S}_{i,j}$; →
  if (j == $\left\lceil \frac{n}{\Delta^i} \right\rceil$) break;
  j = j + 1;
  receive $\mathcal{S}_{i,j}$;
end
	for j=1 ... $\left\lceil \frac{n}{\Delta^i} \right\rceil$
	  for l=Δ ...1
	    produce $\mathcal{S}_{i-1,j\cdot\Delta-l}$;
	    copy $\mathcal{S}_{i-1,j\cdot\Delta-l}$; →
	  end
	  copy $\mathcal{S}_{i,j}$; →
	end
		for j=1 ... $\left\lceil \frac{n}{\Delta^i} \right\rceil$
		  for l=Δ ...1
		    send $\mathcal{S}_{i-1,j\cdot\Delta-l}$;
		  end
		  send $\mathcal{S}_{i,j}$;
		end

end

Fig. 5. *Producing snapshots in a distributed–memory architecture*

Figure 5 depicts the algorithm to produce snapshots of $\mathcal{S}_{i-1}$. On the left and on the right the communication thread is shown, while in the middle the computational part is presented. The copy operations are the synchronization points.

4 First Results and Further Work

DAP [1] is a first version of a software system that implements the strategy on a homogeneous multiprocessor system using PVM [4]. The use of PVM implies the asynchronous communication as mentioned above. DAP is implemented as a preprocessing system. Hence, the user need specify only these parts that are specific to the given problem. The specification is written in C and contains

- the definition of the data structures, above all the structures that participate in the communication and describe the states, the records, and the derivatives as well as the environment and global data;
- globally used functions and variables;
- the modules M, $\bar{M}$, and M', as mentioned in the first section;
- a module that prepares the environment data; and

• the module that handles the original problem that requires the derivatives.

Based on this specification the executables are generated automatically.

To illustrate the results presented here, we used DAP to handle the rather academic problem function

$$(x_i, r_i) = (\ x_{i-1} + \frac{1}{n^2} \cdot (i \cdot \mathbf{A} + (n-i) \cdot \mathbf{B}) \cdot x_{i-1}\ ,\ r_{i-1} + \frac{1}{2}(x_i - b_i)^2\ )\ ,$$

where $\mathbf{A}$ and $\mathbf{B}$ are randomly generated square matrices of dimension μ with $\|\mathbf{A}\|_1 < 1$ and $\|\mathbf{B}\|_1 < 1$, respectively. The two matrices form the global data. The b_i are the unit vectors e_i for $i = 1, \ldots, \mu$ and the null vector otherwise. The size $M_{\mathcal{S}}$ of a snapshot as well as the size $M_{\mathcal{G}}$ in the given example is $(8\mu + 12)$ bytes. To clarify the influence of the communication, we generated records of about $\mu \cdot (8\mu + 13)$ bytes. The tasks were executed on a Parsytec–PowerXplorer with 16 nodes and 32 megabytes of local memory. The PVM version used was PowerPVM [5]. The time needed to compute the gradient was measured for various dimensions and numbers of iteration steps. Figure 6 shows the ratio of the turnaround time to the time needed to evaluate the problem function only.

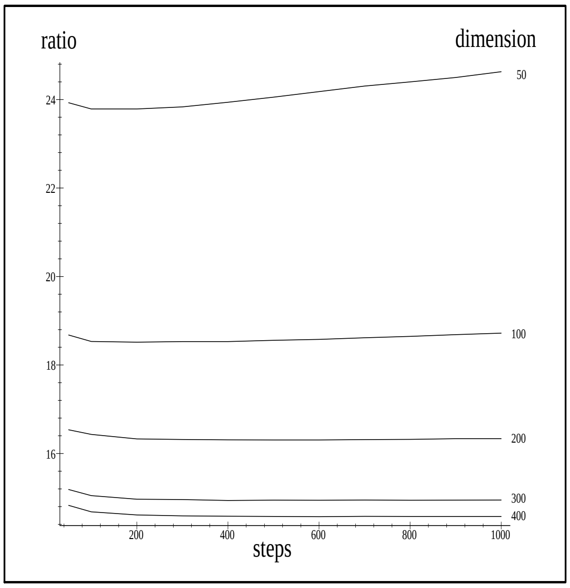

FIG. 6. *Ratio of the run time of the multiprocessor scheduling to the simple function evaluation*

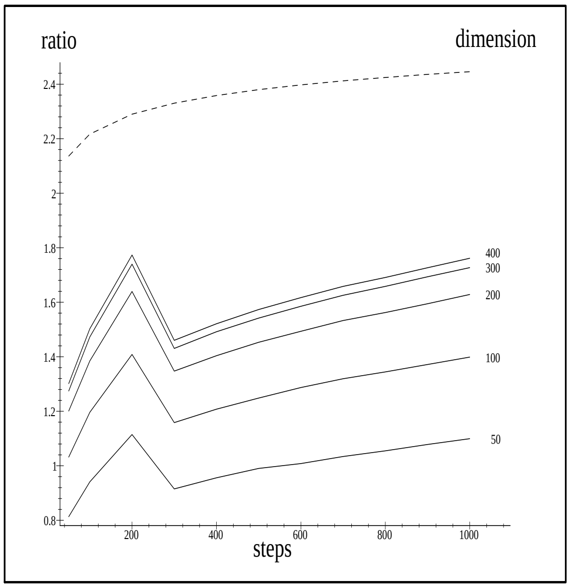

FIG. 7. *Ratio of the run time of the single–processor scheduling to the multiprocessor scheduling*

In addition, the scheduling method of [2] in its symmetrical form was implemented to compare the results of the time measuring with the single–processor scheduling. Figure 7 depicts this ratio in dependency on the dimension μ and the number of iteration steps n. The dashed curve at the top shows the ratio that could be reached in theory and that could be computed by the function $r(n) = 2 + \frac{\log_4 n}{\alpha+1}$. The main reason for the lower efficiency in Figure 7, as well as for the nonlinearity in Figure 6, is the big contribution of the communication, especially the data transfer time, to the run time (from about 67% in case of $\mu = 50$ to about 49% in case of $\mu = 400$). To get better run-time values, we can reduce the influence of the communication. This reduction can be done in two ways: decreasing the amount of data transferred across the communication network by the user, and using the machine–specific communication software instead of PVM by the implementor. On the other hand, the dimension and the iteration length were selected in such a way that the single–processor scheduling could run on one node of the Parsytec–PowerXplorer without any swapping or similar system actions. If we limit to a certain number d the number of states that can be stacked, the time grows not with logarithmic order but with $\mathcal{O}(\sqrt[d]{n})$. In this case, the advantages of the multiprocessing strategy would become clearer.

Some topics for further investigation are the following.

1. We have assumed here that the computation of the problem function f can be divided into a sequence of modules of the same complexity. This configuration is unlikely in practical problems. The values of t_M, $t_{\bar{M}}$, and $t_{M'}$ will be distributed within some interval in a problem–dependent way instead. If the communication schedule is to be determined beforehand, suitable estimates of α and β must be obtained. Otherwise, we can try to use the various values of t_M that are provided by the first execution of an iteration step, to adjust the communication scheme to the problem function.
2. We have investigated the problem "Given turnaround time and iteration length, find minimal resource requirements." Another interesting problem is "Given resource limits and iteration length, find the turnaround time needed."
3. The software system DAP implements the strategy presented here. A bottleneck consists in the supply of the iteration steps with the corresponding values b_i. In general, these data are located on the file system. Since the access to the file system is time consuming, it makes sense to use some selected processors to compute the environment data to be supplied to the other processors on request. An important question is what strategy should be used for storing and freeing these data.
4. In the current version, only the programming language C can be used to describe the problem. In addition, the modules $\bar{M}$ and M' must be worked by hand. In the future, a Fortran version should be implemented. Furthermore, automatic generation of the modules $\bar{M}$ and M' should be possible. Initially, ADOL–C [3] will be used for this purpose.

Acknowledgments

This project benefited from discussions with Bertram Bracher and Andreas Griewank, who originally suggested the approach. Furthermore, Bertram Bracher transferred DAP to the Parsytec–PowerXplorer and carried out the timings.

References

[1] J. Benary: *DAP – Dresdener Adjungierten Parallelisierungsprojekt*, Preprint IOKOMO-05-1995, TU Dresden, Dresden, 1995

[2] A. Griewank: *Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation*, Optimization Methods and Software, 1992, Vol. 1, pp. 35-54

[3] A. Griewank, D. Juedes, J. Utke: *ADOL–C, A Package for the Automatic differentiation of Algorithms Written in C/C++*, ACM TOMS, to appear 1996

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam: *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, 1994

[5] Th. Frach, G. Viehöver: *PowerPVM for PARIX 1.2-PPC, Version 1.0 beta*, PowerStone GmbH, Herzogenrath, 1994