Revolve-Based Adjoint Checkpointing for Multistage Time Integration*

Hong Zhang[1] and Emil Constantinescu[1]

Mathematics and Computer Science Division
Argonne National Laboratory, Lemont, IL 60439
{hongzhang,emconsta}@anl.gov

Abstract. We consider adjoint checkpointing strategies that minimize the number of recomputations needed when using multistage timestepping. We demonstrate that we can improve on the seminal work based on the Revolve algorithm. The new approach provides better performance for a small number of time steps or checkpointing storage. Numerical results illustrate that the proposed algorithm can deliver up to two times speedup compared with that of Revolve and avoid recomputation completely when there is sufficient memory for checkpointing. Moreover, we discuss a tailored implementation that is arguably better suited for mature scientific computing libraries by avoiding central control assumed in the original checkpointing strategy. The proposed algorithm has been included in the PETSc library.

Keywords: Adjoint checkpointing, · multistage timestepping · Revolve.

1 Introduction

Adjoint computation is commonly needed in a wide range of scientific problems such as optimization, uncertainty quantification, and inverse problems. It is also a core technique for training artificial neural networks in machine learning via backward propagation. The adjoint method offers an efficient way to calculate the derivatives of a scalar-valued function at a cost independent of the number of the independent variables. In the derivative computation, the chain rule of differentiation is applied starting with the dependent variables and propagating back to the independent variables; therefore, the computational flow of the function evaluation is reversed. But the intermediate information needed in the reverse computation may not be available and must be either saved beforehand or recomputed. When the storage is insufficient for all the intermediate information, one can checkpoint some selected values and recompute the missing information as needed. This approach gives rise to the adjoint checkpointing problem,

* This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under Contract DE-AC02-06CH11357 at Argonne National Laboratory.

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

which aims to minimize the recomputation cost, usually in terms of number of recomputed time steps, given limited storage capacity. Griewank and Walther proposed the first offline optimal checkpointing strategy [4] that minimizes the number of recomputations when the number of computation steps is known a priori. The algorithm was implemented in a software called REVOLVE and has been widely used in automatic differentiation tools such as ADtool, ADOL-C, and Tapenade. Many follow-up studies have addressed online checkpointing strategies for cases where the number of computation steps is unknown [5, 9, 10], and multistage or multilevel checkpointing strategies [2, 1, 7] have been developed for heterogeneous storage systems (e.g., devices with memory and disk). A common assumption used in developing these algorithms is that memory is considered to be limited and the cost of storing/restoring checkpoints is negligible.

While the problem has been well studied in the context of reversing a sequence of computing operations explicitly, time-dependent problems are often used to model the general stepwise evaluation procedures [4] because of their common sequential nature. In addition, time-dependent differential equations are ubiquitous in scientific simulations, and in their adjoint computation, a time step can be considered as the primitive operation in the sequence to be reversed.

In this paper, we show that the classical REVOLVE algorithm can be improved when multistage time integration methods such as Runge–Kutta methods are used to solve differential equations. A simple modification to the algorithm and the checkpointing settings can lead to fewer recomputations. Performance of the proposed algorithm is demonstrated and compared with that of REVOLVE.

2 Revisiting optimality of classic checkpointing strategy for multistage methods

Here we discuss the adjoint method applied to functionals with ordinary differential equation (ODE) constraints such as $\dot{\boldsymbol{u}} = F(\boldsymbol{u})$. Previous studies assume implicitly or explicitly that only the solution is saved if it is marked as a checkpoint, as opposed to saving the intermediate stages for the corresponding time step. However, the optimal scheduling based on this assumption will not always work best for all timestepping methods, especially multistage methods.

Multistage schemes, for example, Runge–Kutta methods, have been popular in a wide range of applications; their adjoint counterparts are implemented by FATODE [12] and have recently been used by the open source library PETSC [3, 11]. An s-stage explicit Runge–Kutta (ERK) method is

$$\begin{aligned} \boldsymbol{U}_i &= \boldsymbol{u}_n + \sum_{j=1}^{i-1} h_n\, a_{ij}\, F(\boldsymbol{U}_j), \quad i = 1, \cdots, s, \\ \boldsymbol{u}_{n+1} &= \boldsymbol{u}_n + \sum_{i=1}^{s} h_n\, b_i\, F(\boldsymbol{U}_i). \end{aligned} \tag{1}$$

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

The discrete adjoint of ERK is

$$\begin{aligned} \boldsymbol{\lambda}_{s,i} &= h_n \boldsymbol{f}_{\boldsymbol{u}}^T(\boldsymbol{U}_i) \left(b_i \boldsymbol{\lambda}_{n+1} + \sum_{j=i+1}^{s} a_{ji} \boldsymbol{\lambda}_{s,j} \right), \quad i = s, \cdots, 1 \\ \boldsymbol{\lambda}_n &= \boldsymbol{\lambda}_{n+1} + \sum_{j=1}^{s} \boldsymbol{\lambda}_{s,j}, \end{aligned} \tag{2}$$

where $\boldsymbol{\lambda}$ is the adjoint variable that carries the sensitivity information.

To perform an adjoint step of an ERK scheme, one needs all the stage values from the corresponding forward time step according to the sensitivity equation (2). When using REVOLVE, this is achieved by implementing an action named *youturn*, which takes a forward step followed immediately by an adjoint step. Figure 1a shows that an adjoint step in the reverse run is always preceded by a forward step. Ideally if one checkpointed the solution at every time step, $m-1$ recomputations would still be required in order to adjoin m time steps. If one checkpointed the stage values instead of the solution for all the time steps, however, no recomputation would be needed in the ideal case, and fewer recomputations may be expected for other cases.

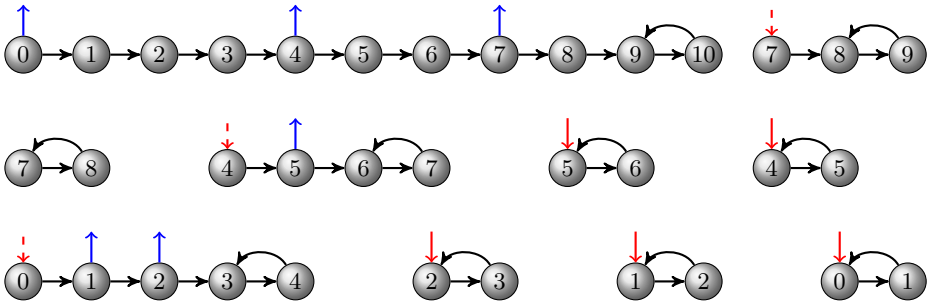
Based on this observation, we extend the existing optimal offline checkpointing scheme to the case where both the solution and stage values are saved. Although saving more information at each time step yields fewer allowed checkpoints, we show that the extended schemes may still outperform the original schemes in certain circumstances, depending on the total number of time steps to be adjoined.

3 Modified offline checkpointing scheme

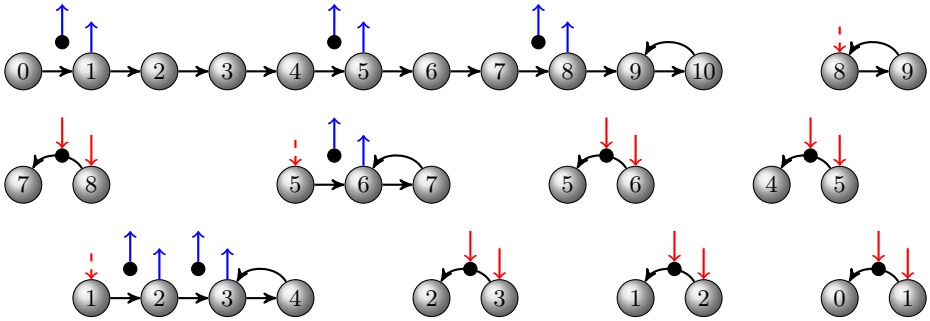
For convenience of notation, we associate the system solution at each time step with an index. The index of the system solution starts with 0 corresponding to the initial condition and increases by 1 for each successful time step. In the context of adaptive timestepping, a successful time step refers to the last actual time step taken after several attempted steps to determine a suitable step size. Therefore, the failed attempts are not indexed. The adjoint integration starts from the final time step and decreases the index by 1 after each reverse step until reaching 0.

An optimal reversal schedule that is generated by `Revolve` yields a minimal number of recomputations for a given number of 10 time steps and 3 checkpoints is shown in Figure 1a. During the forward integration, the solutions at time index 0, 4 and 7 are copied into checkpoints 0, 1, and 2, respectively. After the final step $9 \rightarrow 10$ is finished, the adjoint sensitivity variables are initialized, and the adjoint calculation starts to proceed in the backward direction. The solution and stage values at the last time step are accessible at this point, so the first adjoint step can be taken directly. To compute the adjoint step $9 \rightarrow 8$, one can obtain the forward solution at 9 and the stage values by restoring the checkpoint 2

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37



(a) Checkpoint only of solutions.



(b) Checkpoint of both solutions and stage values (denoted by dots).

Fig. 1: From left to right, top to bottom: the processes controlled by (a) REVOLVE and (b) modified REVOLVE. The up arrow and down arrow stand for the "store" operation and "restore" operation, respectively. When a stack is used for holding the checkpoints, the arrows with solid lines correspond to push and pop operations. The down arrow with a dashed line indicates to read the top element on the stack without removing it. Adapted from [11].

and recomputing two forward steps. The checkpoint 2 can be discarded after the adjoint step $8 \to 7$ so that its storage can be reused in the following calculation.

This schedule results from calling the REVOLVE routine repeatedly and implementing the actions determined. The return value of REVOLVE indicates the calling program to perform one of the actions including *advance*, *takeshot* (*store*), *restore*, *firsturn*, and *youturn*, which are explained in [4] and briefly summarized in Table 1.

The main modification we made is to change every checkpoint position by adding 1 to the index and save the stage values, which are used to compute the solution. For example, if the original `revolve` algorithm determines that the solution at time index i should be checkpointed, we will save the solution at time index $i+1$ and the stage values for the time step $i \to i+1$ as a combined checkpoint with index $i+1$. The actions prescribed by REVOLVE are essentially mapped to a series of new but similar actions to guarantee the optimality for

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

Table 1: REVOLVE nomenclature.

REVOLVE actions

restore	copy the content of the checkpoint back to the solution
takeshot(store)	copy the solution into a specified checkpoint
advance	propagate the solution forward for a number of time steps
youturn	take one forward step and then one reverse step for adjoint
firsturn	take one reverse step directly for adjoint

REVOLVE parameters

`check`	number of checkpoint being stored
`capo`	beginning index of the time step range currently being processed
`fine`	ending index of the time step range currently being processed
`snaps`	upper bound on number of checkpoints taken

Table 2: Mapping the REVOLVE output to new actions.

REVOLVE action	New action in modified REVOLVE
restore to solution i	copy checkpoint to solution $i+1$ and the stages
store solution i	copy solution $i+1$ and the stages into a specified checkpoint
advance from i to j	propagate the solution from $i+1$ to $j+1$
youturn	take one reverse step directly

the new checkpointing settings. Table 2 enumerates the mapping we conduct in the modified scheme.

The adjoint of every time step except the last one always starts from a "restore" operation, followed by recomputations from the solution restored. Since the positions of all checkpoints are shifted one time step forward, one fewer recomputation is taken in the recomputation stage before computing an adjoint step. This observation leads to the following proposition for this modified REVOLVE algorithm.

Proposition 1. *Assume a checkpoint is composed of stage values and the resulting solution. Given s allowed checkpoints in memory, the minimal number of extra forward steps (recomputations) needed for the adjoint computation of m time steps is*

$$\tilde{p}(m,s) = (t-1)\,m - \binom{s+t}{t-1} + 1, \tag{3}$$

where t is the unique integer (also known as repetition number [4]) satisfying $\binom{s+t-1}{t-1} < m \leq \binom{s+t}{t}$.

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

Proof. Griewank and Walther proved in [4] that the original REVOLVE algorithm would take

$$p(m, s) = t\,m - \binom{s+t}{t-1} \tag{4}$$

extra forward steps. According to the observation mentioned above, one can save $m-1$ extra forward steps by using the modified scheme. We can prove by contradiction that no further savings are possible.

If a schedule that satisfies the assumption takes fewer recomputations than (3), one can move all the checkpoints backward by one step and change the content of the checkpoints to be solutions only. The resulting scheme will cause $m-1$ extra recomputations by construction, thus the total number of recomputations will be less than $t\,m - \binom{s+t}{t-1}$. This contradicts with the optimality result proved in [4].

We remark that the modification can also be applied to the online algorithms in [9, 5, 10] and to the multistage algorithms in [8]. The saving in recomputations is always equal to the total number of steps minus one, given the same amount of allowable checkpoints.

The choice of using modified or original algorithms clearly results from the tradeoff between recomputation and storage. Given a fixed amount of storage capacity, the choice depends solely on the total number of time steps since the recomputations needed can be determined by these two factors according to Proposition 1. For example, we suppose there are 12 allowed checkpoints if we save only the solution, which means 6 checkpoints are allowed if we add one stage to the checkpoint data and 4 checkpoints for adding two stages. Figure 2 shows the relationship between the recomputations and the total number of steps for these different options. In this example, the crossover point at which saving the stages together with the solution leads to fewer recomputations than saving only the solution occurs when 224 and 41 steps are taken for the two illustrated scenarios (saving one additional stage and two additional stages, respectively). Furthermore, the number of stages saved, determined by the timestepping method, has a significant impact on the range of number of time steps in which the former option is more favorable; for methods with fewer stages, the range is generally larger (compare 224 with 41).

Consequently, one can choose whether or not to save stage values in order to minimize the recomputations for discrete adjonit calculation. The optimal choice depends on the number of steps, the number of stages of the timestepping algorithm, and the memory capacity.

Moreover, when using the modified REVOLVE algorithm, we can save fewer stages than a timestepping method actually has, if the last stage of the method is equal to the solution at the end of a time step. For instance, the Crank–Nicolson method, which can be cast as a two-stage Runge–Kutta method, requires storing only one stage with the solution corresponding to the solutions at the beginning and the end of a time step. Many classic implicit Runge–Kutta methods have such a feature that enables further improvement of the performance.

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

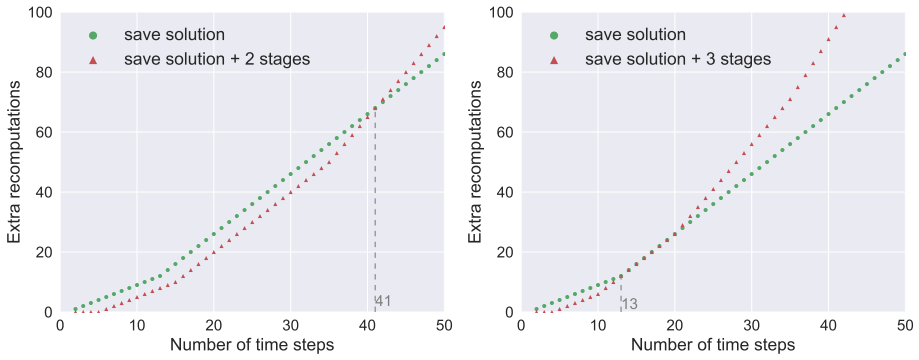


Fig. 2: Comparison in terms of recomputation effort for including different numbers of stages (0,1,and 2) in the checkpoint data. It is assumed that up to 12 solution vectors or stage vectors can be stored in both forward integration and adjoint integration. Adapted from [11].

4 Using Revolve in the discrete adjoint calculation

The Revolve library is designed to be an explicit controller for conducting forward integration and adjoint integration in time-dependent applications. The interface requires the user to provide procedures such as performing a forward and backward step and saving/restoring a checkpoint. Thus, incorporating Revolve in other simulation software such as PETSc can be intrusive, or even infeasible, especially when the software has its own adaptive time step control and an established framework for time integration. To mitigate intrusion, we use Revolve in a different way in the sense that its role becomes more of a "consultant" rather than a "controller." Algorithm 1 describes the workflow for the adjoint calculation with checkpointing. The parameters `capo`, `fine`, and `check` are updated internally by Revolve. See Table 1 for descriptions of these parameters. A counter `stepstogo` for the number of steps to advance is computed from these variables. We insert calls to Revolve only before a forward step is taken when `stepstogo` is not zero, so as to preserve the original integration process, which is represented by `forwardSweep`. This trick is also applied to the adjoint sweep. One can verify that the resulting schedule is equivalent to the one generated by calling Revolve repeatedly (the "controller" mode) based on the following observations:

- If Revolve asks to store a checkpoint (`takeshot`), it will return either *advance* or *youturn* (*firsturn*) in the next call.
- In the adjoint sweep, it is always required to restore a checkpoint and recompute one or more steps from this point.

The checkpointing scheme using the modified Revolve algorithm is depicted in Algorithm 2. A main difference from Algorithm 1 is that the call to Revolve is lagged because the positions of all the checkpoints have been shifted. We note

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

Algorithm 1 Adjoint checkpointing for a sequence of m time steps.

```
Initialize global variables capo ← 0, fine ← m, check ← −1, snaps ← s
Initialize global variable stepstogo ← 0   ▷ steps to recompute in the adjoint sweep
Initialize REVOLVE with capo, fine, check, and snaps
state ← FORWARDSWEEP(0, m, state)                                   ▷ forward sweep
adjstate ← ADJOINTSTEP(adjstate)                                   ▷ reverse last step directly
for i ← m − 1 to 1 do                                              ▷ adjoint sweep
    whatodo ← REVOLVE(check,capo,fine,snaps)
    Assert(whatodo=restore)                ▷ always start from restoring a checkpoint
    state, restoredind ← RESTORE(check)    ▷ get the index of the restored checkpoint
    state ← FORWARDSWEEP(restoredind, i − restoredind, state)   ▷ recompute from
the restored solution
    adjstate ← ADJOINTSTEP(adjstate)
end for
function FORWARDSWEEP(ind, n, state)   ▷ advance n steps from the ind-th point
    for i ← ind to ind + n − 1 do ▷ REVOLVE returns youturn/firsturn at the end
        REVOLVEFORWARD(state)
        state ← FORWARDSTEP(state)
    end for
    return state
end function
function REVOLVEFORWARD(state)                  ▷ query REVOLVE and take actions
    if stepstogo = 0 then
        oldcapo ← capo
        whatodo ← REVOLVE(check,capo,fine,snaps)
        if whatodo = takeshot then
            STORE(state, check)                                     ▷ store a checkpoint
            oldcapo ← capo
            whatodo ← REVOLVE(check,capo,fine,snaps)
        end if
        if whatodo = restore then
            RESTORE(state, check)                                   ▷ restore a checkpoint
            oldcapo ← capo
            whatodo ← REVOLVE(check,capo,fine,snaps)
        end if
        Assert(whatodo=advance || whatodo=youturn || whatodo=firsturn)
        stepstogo ← capo − oldcapo
    else
        stepstogo ← stepstogo − 1
    end if
end function
```

that reducing the counter `stepstogo` by one in the loop for adjoint sweep reflects the fact that one fewer recomputation is needed for each adjoint step.

Both Algorithms 1 and 2 are implemented under the `TSTrajectory` class in PETSc, which provides two critical callback functions `TSTrajectorySet()` and `TSTrajectoryGet()`. The former function wraps REVOLVEFORWARD in FOR-

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

Algorithm 2 Adjoint checkpointing using the modified REVOLVE algorithm.

```
Initialize global variables capo ← 0, fine ← m, check ← −1, snaps ← s
Initialize global variable stepstogo ← 0
Initialize REVOLVE with capo, fine, check, and snaps
state ← FORWARDSWEEP(0, m, state)
adjstate ← ADJOINTSTEP(adjstate)
for i ← M − 1 to 1 do
    restoredind ← REVOLVEBACKWARD(state)          ▷ always restore a checkpoint
    state ← FORWARDSWEEP(restoredind, i − restoredind, state)
    adjstate ← ADJOINTSTEP(adjstate)
end for
function FORWARDSWEEP(ind, n, state)
    for i ← ind to ind + n − 1 do
        state ← FORWARDSTEP(state)
        REVOLVEFORWARD(state)
    end for
    return state
end function
function REVOLVEBACKWARD(state)
    whatodo ← REVOLVE(check,capo,fine,snaps)
    Assert(whatodo = restore)
    state, restoredind ← RESTORE(check)
    stepstogo ← max(capo − oldcapo − 1, 0)    ▷ need one less extra step since stage values are saved
    return restoredind
end function
```

WARDSWEEP. The latter function wraps all the statements before ADJOINTSTEP in the `for` loop. This design is beneficial for preserving the established workflow of the timestepping solvers so that the impact to other PETSc components such as TSADAPT (adaptor class) and TSMONITOR (monitor class) is minimized.

PETSc uses a redistributed package [1] that contains a C wrapper of the original C++ implementation of REVOLVE. The parameters needed by REVOLVE can be passed through command line options at runtime. Additional options are provided to facilitate users monitoring the checkpointing process. Listing 1.1 exhibits an exemplar output for `-ts_trajectory_monitor -ts_trajectory_view` when using modified REVOLVE to reverse 5 time steps given 3 allowable checkpoints.

By design, PETSc is responsible for the manipulation of checkpoints. A stack data structure with push and pop operations is used to conduct the actions decided by the checkpointing scheduler. The `PetscViewer` class manages deep copy between the working data and the checkpoint, which can be encapsulated in either sequential or parallel vectors that are distributed over different processes.

In addition to the offline checkpointing scheme, PETSc supports online checkpointing and multistage checkpointing schemes provided by the REVOLVE

[1] https://bitbucket.org/caidao22/pkg-revolve.git

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

package, and the modification proposed in this paper has been applied to these schemes as well. Therefore, checkpoints can be placed on other storage media such as disk. For disk checkpoints, the `PetscViewer` class offers a variety of formats such as binary and HDF5 and can use MPI-IO on parallel file systems for efficiency.

```
TSTrajectorySet: stepnum 0, time 0. (stages 1)
TSTrajectorySet: stepnum 1, time 0.001 (stages 1)
Store in checkpoint number 0 (located in RAM)
Advance from 0 to 2
TSTrajectorySet: stepnum 2, time 0.002 (stages 1)
TSTrajectorySet: stepnum 3, time 0.003 (stages 1)
Store in checkpoint number 1 (located in RAM)
Advance from 2 to 4
TSTrajectorySet: stepnum 4, time 0.004 (stages 1)
TSTrajectorySet: stepnum 5, time 0.005 (stages 1)
First turn: Initialize adjoints and reverse first step
TSTrajectoryGet: stepnum 5, stages 1
TSTrajectoryGet: stepnum 4, stages 1
Restore in checkpoint number 1 (located in RAM)
Advance from 2 to 3
Skip the step from 2 to 3 (stage values already checkpointed)
Forward and reverse one step
TSTrajectoryGet: stepnum 3, stages 1
Restore in checkpoint number 1 (located in RAM)
Forward and reverse one step
Skip the step from 2 to 3 (stage values already checkpointed)
TSTrajectoryGet: stepnum 2, stages 1
Restore in checkpoint number 0 (located in RAM)
Advance from 0 to 1
Skip the step from 0 to 1 (stage values already checkpointed)
Forward and reverse one step
TSTrajectoryGet: stepnum 1, stages 1
Restore in checkpoint number 0 (located in RAM)
Forward and reverse one step
Skip the step from 0 to 1 (stage values already checkpointed)
TSTrajectoryGet: stepnum 0, stages 1
TSTrajectory Object: 1 MPI processes
  type: memory
  total number of recomputations for adjoint calculation = 2
```

Listing 1.1: Monitoring the checkpointing process in PETSc

5 Algorithm performance

To study the performance of our algorithm, we plot in Figure 3 the actual number of recomputations against the number of time steps and compare our algorithm with the classic Revolve algorithm. For a fair comparison, the same number of checkpointing units is considered.

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

Figure 3 shows that our modified algorithm outperforms REVOLVE. For 30 checkpointing units and 300 time steps, modified REVOLVE clearly takes fewer recomputations than does REVOLVE, with the maximum gap being 40 recomputations. For 60 checkpointing units and 300 time steps, the savings with modified REVOLVE are 186 recomputations. If the recomputation cost of a time step is fixed, the result implies a speedup of approximately $1.5X$ in runtime for the adjoint calculation. As the number of time steps further increases, REVOLVE is expected to catch up with modified REVOLVE and eventually surpass modified REVOLVE. Furthermore, when the number of time steps is small, which means there is sufficient memory, no recomputation is needed with modified REVOLVE; however, REVOLVE requires the number of recomputations to be at least as many as the number of time steps minus one.

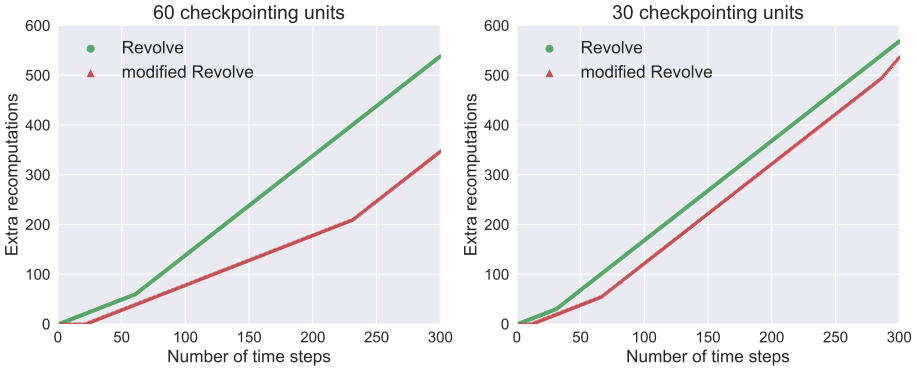


Fig. 3: Performance comparison between REVOLVE and modified REVOLVE. The plotted data is computed for time integration methods with two stages.

Now we demonstrate the performance with a real example. In the experiment, we consider the gradient calculation using an adjoint method to solve a PDE-constrained optimization problem. The objective is to minimize the discrepancy between the simulated result and observation data (reference solution):

$$\underset{\boldsymbol{U}_0}{\text{minimize}} \, \|\boldsymbol{U}(t_f) - \boldsymbol{U}^{ob}(t_f)\|^2 \tag{5}$$

subject to the Gray-Scott equations [6]

$$\begin{aligned} \dot{\mathbf{u}} &= D_1 \nabla^2 \mathbf{u} - \mathbf{u}\mathbf{v}^2 + \gamma(1 - \mathbf{u}) \\ \dot{\mathbf{v}} &= D_2 \nabla^2 \mathbf{v} + \mathbf{u}\mathbf{v}^2 - (\gamma + \kappa)\mathbf{v}, \end{aligned} \tag{6}$$

where $\boldsymbol{U} = [\mathbf{u}\ \mathbf{v}]^T$ is the PDE solution vector. The settings of this example follow [11]. The PDE is solved with the method of lines. The resulting ODE is solved by using the Crank–Nicolson method with a fixed step size 0.5. A

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

centered finite-difference scheme is used for spatial discretization on a uniform grid of size 128×128. The computational domain is $\Omega \in [0, 2]^2$. The time interval is $[0, 25]$. The nonlinear system that arises at each time step is solved by using a Newton-based method with line search, and the linear systems are solved by using GMRES with a block Jacobi preconditioner.

Figure 4 shows that the runtime decreases as the allowable memory for checkpointing increases for both schemes and that the best performance achieved by modified REVOLVE is approximately $2.2X$ better than that by REVOLVE. When memory is sufficient, modified REVOLVE requires no recomputation. Thus the estimated speedup of modified REVOLVE over REVOLVE would be approximately $2X$ provided the cost of the forward sweep is comparable to the cost of the adjoint sweep. Another important observation is that the experimental results, despite being a bit noisy, roughly match with the theoretical predictions. The observable oscillation in timing can be attributed mostly to the fact that the cost of solving the implicit system varies across the steps, which is not uncommon for nonlinear dynamical systems.

6 Conclusion

With the abstraction of a sequence of time steps, the classic algorithm REVOLVE provides optimal checkpointing schedules for efficient adjoint computation in many scientific computations. However, it may yield suboptimal performance when directly applied to multistage time integration methods.

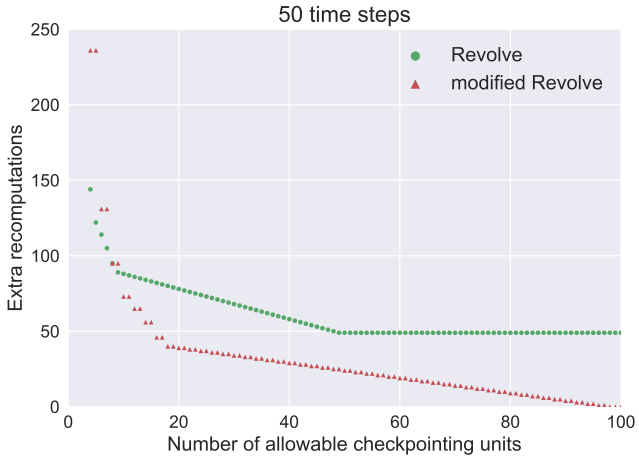
In this paper we have considered checkpointing strategies that minimize the number of recomputations under two assumptions: (1) the stage values of a multistage method can be saved as part of a checkpoint, and (2) a stage vector has the same size (therefore the same memory cost) with a solution vector. By extending REVOLVE and redefining the content of a checkpoint, we derived a modified REVOLVE algorithm that provides better performance for a small number of time steps. The performance has been studied numerically. The results on some representative test cases show that our algorithm can deliver up to $2X$ speedup compared with REVOLVE and avoid recomputation completely when there is sufficient memory for checkpointing.

In addition, the implementation of our algorithm is tailored to fit into the workflow of mature scientific computing libraries. Integrating our algorithm into existing frameworks avoids using it as a centralized controller over the entire workflow; thus, it becomes less intrusive to the application codes. The proposed algorithm has been successfully employed in the PETSC library.

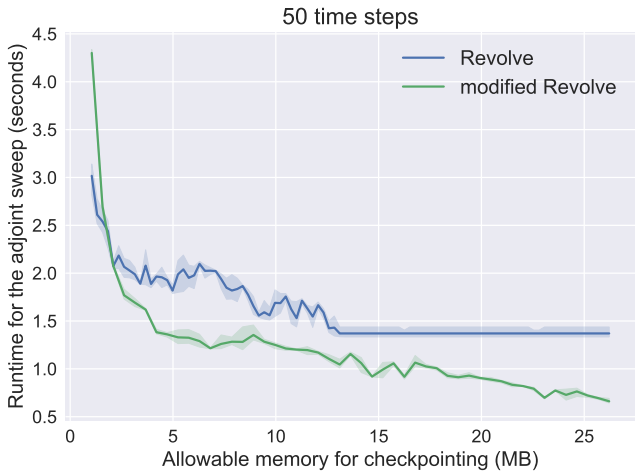
References

1. Aupy, G., Herrmann, J.: Periodicity in optimal hierarchical checkpointing schemes for adjoint computations. Optimization Methods and Software **32**(3), 594–624 (2017). https://doi.org/10.1080/10556788.2016.1230612

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37



(a) Theoretical predictions.



(b) Runtime for adjoint vs. checkpointing memory consumption.

Fig. 4: Comparison of the timing results of the adjoint calculation with the theoretical predictions for REVOLVE and modified REVOLVE. The Crank–Nicolson method is used for time integration. It consists of two stages, but only the first stage needs to be saved since the second stage is the same as the solution.

2. Aupy, G., Herrmann, J., Hovland, P., Robert, Y.: Optimal multistage algorithm for adjoint computation. SIAM Journal on Scientific Computing **38**(3), C232–C255 (2016)
3. Balay, S., Abhyankar, S., Adams, M., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W., Kaushik, D., Knepley, M., McInnes, L.C., Rupp, K., Smith, B., Zampini, S., Zhang, H., Zhang, H.: PETSc Users Manual. Tech. Rep. ANL-95/11 - Revision 3.7, Argonne National Laboratory (2016)

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37

4. Griewank, A., Walther, A.: Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Transactions on Mathematical Software **26**(1), 19–45 (2000). https://doi.org/10.1145/347837.347846
5. Heuveline, V., Walther, A.: Online checkpointing for parallel adjoint computation in PDEs: application to goal-oriented adaptivity and flow control. In: Euro-Par 2006 Parallel Processing, Lecture Notes in Computer Science, vol. 4128. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11823285
6. Hundsdorfer, W., Ruuth, S.J.: IMEX extensions of linear multistep methods with general monotonicity and boundedness properties. Journal of Computational Physics **225**(2007), 2016–2042 (2007). https://doi.org/10.1016/j.jcp.2007.03.003
7. Schanen, M., Marin, O., Zhang, H., Anitescu, M.: Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver Nek5000. Procedia Computer Science **80**, 1147–1158 (2016). https://doi.org/10.1016/j.procs.2016.05.444
8. Stumm, P., Walther, A.: MultiStage Approaches for Optimal Offline Checkpointing. SIAM Journal on Scientific Computing **31**(3), 1946–1967 (2009). https://doi.org/10.1137/080718036
9. Stumm, P., Walther, A.: New algorithms for optimal online checkpointing. SIAM Journal on Scientific Computing **32**(2), 836–854 (2010). https://doi.org/10.1137/080742439
10. Wang, Q., Moin, P., Iaccarino, G.: Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. SIAM Journal on Scientific Computing **31**(4), 2549–2567 (2009). https://doi.org/10.1137/080727890
11. Zhang, H., Constantinescu, E.M., Smith, B.F.: PETSc TSAdjoint: a discrete adjoint ODE solver for first-order and second-order sensitivity analysis. CoRR **abs/1912.07696** (2020), `http://arxiv.org/abs/1912.07696`
12. Zhang, H., Sandu, A.: FATODE: a library for forward, adjoint, and tangent linear integration of ODEs. SIAM Journal on Scientific Computing **36**(5), C504–C523 (oct 2014). https://doi.org/10.1137/130912335

ICCS Camera Ready Version 2021
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-77961-0_37