Extending the Binomial Checkpointing Technique for Resilience*

Andrea Walther[1] Sri Hari Krishna Narayanan[2]

Abstract

In terms of computing time, adjoint methods offer a very attractive alternative to compute gradient information, required, e.g., for optimization purposes. However, together with this very favorable temporal complexity result comes a memory requirement that is in essence proportional with the operation count of the underlying function, e.g., if algorithmic differentiation is used to provide the adjoints. For this reason, checkpointing approaches in many variants have become popular. This paper analyzes an extension of the so-called binomial approach to cover also possible failures of the computing systems. Such a measure of precaution is of special interest for massive parallel simulations and adjoint calculations where the mean time between failure of the large scale computing system is smaller than the time needed to complete the calculation of the adjoint information. We describe the extensions of standard checkpointing approaches required for such resilience, provide a corresponding implementation and discuss first numerical results.

1 Introduction

The usage of adjoint methods allows the computation of gradient information within a time that is only a very small multiple of the time needed to evaluate the underlying function itself. However, as soon as the considered process is nonlinear, the memory requirement to compute the adjoint information is in principle proportional to the operation count of the underlying function, see, e.g., [1, Sec. 4.6]. In Chap. 12 of the same book, several checkpointing alternatives to reduce this high memory complexity are discussed. Checkpointing strategies use a small number of memory units (checkpoints) to store the system state at distinct times. Subsequently, the recomputation of information that is needed for the adjoint computation but not available is performed using these checkpoints in an appropriate way. Several checkpointing techniques have been developed all of which seek an acceptable compromise between memory requirement and runtime increase.

For this paper, we assume that the evaluation of the function of interest has a time-step structure given by

$$x_i = F_i(x_{i-1}, u_{i-1}), \quad i = 1, \ldots, l \ , \tag{1.1}$$

for a given x_0, where $x_i \in \mathbb{R}^n$, $i = 0, \ldots, l$, denote the state of the considered system and $u_i \in \mathbb{R}^m$ the control. The operator $F_i : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}^n$ defines the time step to compute the state x_i. The process to compute x_l for a given x_0 is also called forward integration. To optimize a specific criterion or to obtain a desired state, the cost functional

$$J(x(u), u) = J(x, u)$$

measures the quality of $x(u) = (x_1, \ldots, x_l)$ and $u = (u_1, \ldots, u_l)$, where $x(u)$ depends on the control u. For applying a derivative-based optimization method, one could use an adjoint integration of the form

$$\begin{aligned} &\bar{u}_l = 0 \ , \bar{x}_l \text{ given} \\ &(\bar{x}_{i-1}, \bar{u}_{i-1}) = \bar{F}_i(\bar{x}_i, \bar{u}_i, x_{i-1}, u_{i-1}), \ i = l, \ldots, 1, \end{aligned} \tag{1.2}$$

where the operator $\bar{F}_i$ denotes the adjoint time step. Subsequently or concurrently to the adjoint integration, the desired derivative information $J_u(x(u), u)$ can be reconstructed from $\bar{x}$. As can be seen, the information of the forward integration (1.1) is needed for the adjoint computation (1.2). To provide this information within only a limited amount of memory, we use the binomial checkpointing approach proposed in [2, 3] as a basis to develop a checkpointing approach that can also handle a failure of the computing system. This includes a foreseen suspension, where the application should suspend itself gracefully after completing the set number of forward or adjoint time steps. However, also an unforeseen failure has to be covered, where the application is killed externally because of machine failure or expired time allocation. This scenario occurs when running the MIT General Circulation Model (MITgcm, [4, 5]) on ARCHER, a UK based supercomputer. MITgcm executes for around 351,000 timesteps to simulate 1 year of physical time. This requires around 24 hours of wall clock computation time. Because the mean time between failure (MTBF) of ARCHER is lower than 24 hours, administrative policies require applications execute for a fixed time allocation such as 6 hours at a time before they are suspended. Typically, an application can be restarted from suspension when checkpoints containing intermediate data are available and the application is aware of its position in the overall computation.

*This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by a grant from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

[1]Universität Paderborn

[2]Argonne National Laboratory

An additional aspect that has to be taken into account is the actual location where checkpoints are stored. Checkpoints stored in memory can be lost on failure. For the sake of resilience or because future supercomputers may be memory constrained, checkpoints may have to necessarily be stored to disk. Therefore, the access time to read or write a checkpoint is not negligible in contrast to the assumption frequently made for the development of checkpointing approaches. There are a few contributions to extend the available checkpointing techniques to a hierarchical checkpointing, see, e.g., [6, 7, 8]. However, to derive a first checkpointing technique that incorporates resilience we ignore this hierarchical nature and assume throughout that the writing or reading process for a checkpoint is performed asynchronously such that it does not interfere with the adjoint computation.

This paper has the following structure. In Sect. 2, we describe the functionality of the software revolve [3] that we will use as a starting point for a checkpointing approach that can also cover resilience. The extensions of revolve that are required for this purpose are described in Sect. 3. First results with respect to the temporal complexity will be given in Sect. 4. Finally, we draw conclusions and give an outlook of future work in Sect. 5.

2 Binomial Checkpointing using revolve

For large scale applications like the MITgcm, because of limited storage or the time to store checkpoints, it is only possible to store a very limited number of intermediate states as checkpoints. Hence, one obvious question is where to place these checkpoints during the forward integration to minimize the amount of required recomputations. It was shown in [3] that a checkpointing scheme based on binomial coefficients yields for a given number of checkpoints the minimal number of time steps to be recomputed.

To apply such an optimal checkpointing strategy, one can use the software revolve that provides a data structure r to steer the checkpointing process and the storage of all information required for this purpose. Then, the forward integration as well as the corresponding adjoint computation is performed within a do-while-loop of the structure in Fig. 1, where steps and snaps denote the number l of time steps of the forward simulation and the number c of checkpoints, respectively.

Hence, the routine revolve determines the next action to be performed which must by supported by the application being differentiated. For example, this action may be the execution of a part of the forward integration based on the routine forward(x,u), where x represents the state of the system and u the control, one

```
...
r=new Revolve(steps,snaps)
do
    whatodo = r->revolve()
    switch(whatodo)
        case advance: for r->oldcapo < i ≤ r->capo
                          forward(x,u)
        case takeshot: store(x,xstore, r->check)
        case firsturn:  eval_J(x,u)
                        init(bu,bx)
                        adjoint(bx,bu,x,u)
        case youturn:  adjoint(bx,bu,x,u)
        case restore:   restore(x,xstore, r->check)
while(whatodo <> terminate)
...
```

Figure 1: revolve algorithm with calls to the application interface

step of the actual adjoint computation performed in the routine adjoint(bx,bu,x,u), where bx denotes the adjoint state and bu the adjoint control. For the checkpointing approach, it is required to store the current state as a checkpoint, i.e., execute the routine store(x,xstore, r->check), where xstore represents an array to store system states and r->check is the number of the entry, where the checkpoint has to be stored. Finally, to recompute the required intermediate information a checkpoint has to be read, i.e., restore(x,xstore, r->check) has to be performed.

It is important to note that this checkpointing approach is completely independent from the method that is actually used to provide the adjoint information. As can be seen, as soon as an adjoint computation is available only a very limited effort is needed to combine this with binomial checkpointing to reduce the memory requirement. We have to stress that revolve provides a so-called serial checkpointing which means that only one forward time step or one adjoint step is performed at each stage of the adjoint computation. This is in contrast to so-called parallel checkpointing techniques where several forward time steps might be performed in parallel even in conjunction with one adjoint step.

Up to now it was always assumed that for the serial checkpointing the computation of the forward time step and the adjoint step are free of failures. In reality, the computation of the forward step or the adjoint step may be performed heavily in parallel, i.e., may be evaluated on a large scale computer system. This is precisely the situation where we have to take resilience into account and therefore an appropriately adapted extension of binomial checkpointing approach is required.

3 Binomial Checkpointing for Resilience

The ability to recover from a possible failure poses two additional challenges for the checkpointing scheme. First, the distance between two checkpoints should not be too large such that a restart of the computation is not too costly. Hence, there has to be an additional bound on the distance of two checkpoints in terms of the number of time steps that are performed before the next checkpoint is set. Second, since a failure may also occur during the adjoint computation also the adjoint state has to be checkpointed. Due to the nature of the adjoint computation, only one adjoint state is required to restart the adjoint computation. However, since the adjoint state itself is assumed to be large it is not possible to checkpoint every adjoint state computed. Therefore, we also have to define a distance between these so-called adjoint checkpoints. This distance corresponds to the number of adjoint steps performed after which the current adjoint state is stored again. Hence, two new variables were introduced in the data structure provided by revolve, namely resilience_distance for the maximal number of forward time steps between two checkpoints and adjoint_distance for the number of adjoint steps performed before the current adjoint state is stored again.

Internally, only the maximal distance between two checkpoints, i.e., the value of resilience_distance interferes with the optimal binomial checkpointing approach. To limit the number of time steps between two consecutive checkpoints, first the distance required for the optimal, i.e., binomial checkpointing, is computed. This number is then compared with the value of resilience_distance. If the value of resilience_distance is smaller than the number of time steps chosen by revolve, only resilience_distance steps are performed despite the fact that this might lead to a suboptimal checkpointing schedule. This comparison is performed each time a checkpoint has to be stored. Therefore, revolve implements the optimal checkpointing approach whenever possible. An analysis of this possibly suboptimal checkpointing approach is presented in Sect. 4.

If a failure actually occurs, one faces two possibilities: First the failure happens during the forward integration. Then, the computation can just restart at the last checkpoint stored. Second, the failure happens after the start of the adjoint computation. Then it may happen that the checkpoint distribution at the time of the failure differs from the one when the last adjoint checkpoint was written. As a small example to illustrate this we consider the case, when there are $l = 100$ time steps performed during the forward integration, five checkpoints can be stored, the resilience distance is 30 and an adjoint checkpoint is stored every 10 adjoint steps. Assume now, that the failure happens directly after the computation of the 57th adjoint step. Hence, the adjoint states 59, 58, and 57 are lost. However, because the adjoint state is stored every 10 adjoint steps, the 60th adjoint state is available and the adjoint computation can be restarted at this point. Using the original revolve version, the following states would serve as checkpoints when storing the last, i.e., 60th, adjoint state:

$$0 \quad 45 \quad 54 \quad 56 \quad 58$$

As can be seen, there are more than 30 time steps between the first and the second checkpoint violating the bound of 30 for resilience. Therefore, the correspondingly adapted version of revolve would yield the checkpoint distribution

$$0 \quad 30 \quad 60 \quad 61 \quad 62$$

Hence, one clearly sees the influence on the resilience distance on the distribution of the checkpoints since now there are no more than 30 time steps between the checkpoints. At that stage of the adjoint computation, the states 61 and 62 are still stored as checkpoints. Therefore, they still occur here in the checkpoint list. They will be reset within the next actions to earlier states. Due to this realocation of the checkpoints, the following states serve as checkpoints at the failure:

$$0 \quad 30 \quad 44 \quad 51 \quad 56$$

Therefore, to recover the three adjoint steps that were lost due to the failure, it is not possible to just restart revolve at the current checkpoint distribution. To handle this situation, we developed two new routines to extract the current checkpointing distribution and the checkpointing distribution that was available when the last adjoint checkpoint was written:

```
r->get_last_checkpoint_distribution(...)
r->get_checkpoint_distribution_stored(...)
```

such that this information can be extracted and stored before a failure happens. To restart the adjoint computation after a failure, the stored information is read. In our example, two specific files are used to store this data. One of the files stores the state of the system such as the actual values contained in the checkpoints. In the example below, this file is named state.txt. The other one stores the information required by revolve itself. Therefore, it is named revolve.txt in the example below. If the files do not exist, the computation just starts and the usual initialization of revolve can be used. If such files exist they contain all the data required to restart the adjoint computation with binomial checkpointing at the last adjoint state saved as a checkpoint if the failure

```
...
r=new Revolve(steps,snaps)
r->set_resilience_distance(distance_resil)
r->set_adjoint_distance(distance_adjoint)
if(fileExists("state.txt") && fileExists("revolve.txt")){
        init_status(...)
        restart_revolve_internals(...)}
do
    whatodo = r->revolve()
    switch(whatodo)
        case advance: for r->oldcapo < i ≤ r->capo
                          forward(x,u)
        case takeshot: store(x,xstore, r->check)
        case firsturn:  eval_J(x,u)
                        init(bu,bx)
                        adjoint(bx,bu,x,u)
                        store_adjoint(bx,bu)
        case youturn:  adjoint(bx,bu,x,u)
        case youturn_with_check:  adjoint(bx,bu,x,u)
                                  store_adjoint(bx,bu)
        case restore:   restore(x,xstore, r->check)
while(whatodo <> terminate)
...
```

Figure 2: revolve algorithm with adjoint checkpoints

happened in the adjoint computation. If the failure occurred in the first forward integration, the computation is restarted at the last state stored as checkpoint. For these purposes, we provide the additional routine

restart_revolve_internals(...)

to set up the data structure of revolve and the checkpoint distribution if required correspondingly. Hence, the adjoint computation with checkpointing as illustrated above has to be adapted as shown in Fig. 2. As can be seen, in addition to the required initializations only one new action is introduced, namely the execution of one adjoint step in combination with the storage of the current adjoint state. Hence, if one has already set up the adjoint computation with the original binomial checkpointing, it requires remarkably few changes to adapt it for resilience purposes.

4 A First Complexity Study

To illustrate the deviation of the extended checkpointing approach from the optimal binomial checkpointing, we consider the adjoint computation for the following small

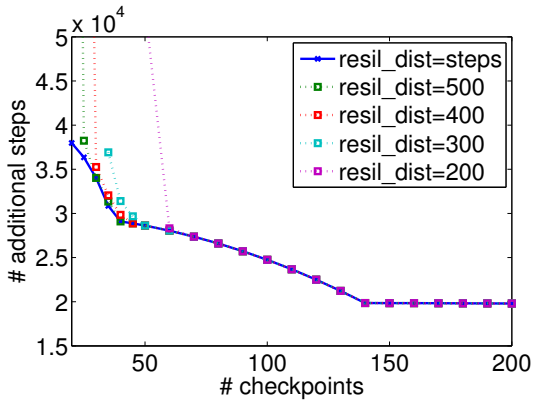

Figure 3: Additional time steps needed for adjointing 10,000 steps

academic test case

$$\begin{aligned} &\min J(x,u) \quad \text{with} && J(x,u) \equiv x_2(1), \\ \text{s.t. } &x_1'(t) = 0.5x_1(t) + u(t), && x_1(0) = 1 \\ &x_2'(t) = x_1(t)^2 + 0.5u(t)^2, && x_2(0) = 0 \\ &t \in [0,1]\ . \end{aligned}$$

Since for this optimization problem, the adjoint can be derived analytically yielding

$$\begin{aligned} \lambda_1'(t) &= -0.\lambda_1(t) - 2 * x_1(t)\lambda_2(t) && \lambda_1(1) = 0 \\ \lambda_2'(t) &= 0 && \lambda_2(1) = 1 \end{aligned}$$

it is possible to verify the correctness of the adjoint computation also for the checkpointing with resilience, i.e., with the restart using the information stored in the additional files. We tested and verified the binomial checkpointing with resilience for up to 100,000 time steps and failures occurring at numerous different places. As a representative observation, Fig. 3 illustrates the additional recomputations needed as a solid line for 10,000 steps and a varying number of checkpoints. The additional recomputations needed by the binomial checkpointing approach that incorporates resilience are illustrated with dotted lines for the resilience distances of 200, 300, 400, and 500 steps. Here, one has to note that the number of checkpoints denoted with c and the resilience distance denoted with d can not be chosen completely independent from each other. Because d is the maximal number of steps between two consecutive checkpoints, it must hold for the computation to be adjointed comprising of l time steps that

$$l \leq d \cdot c\ .$$

This bound limits the resilience distance from below for a very small number of checkpoints as illustrated

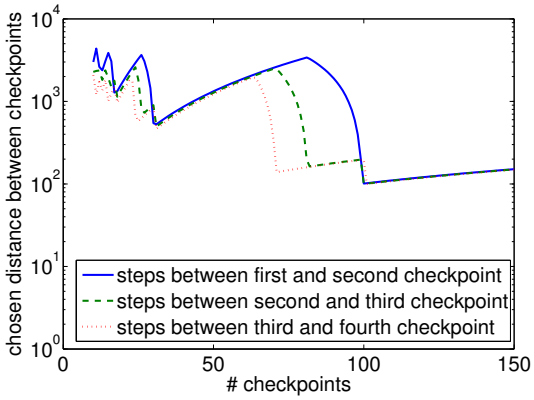

Figure 4: Distance of the first four checkpoints chosen by revolve

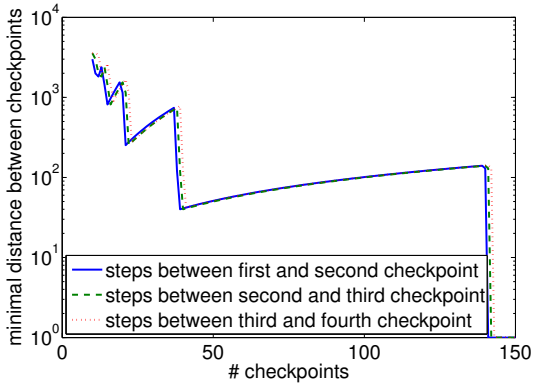

Figure 5: Alternative distance of the first four checkpoints

also in Fig. 3. If l is close to the upper bound $d \cdot c$ a lot of recomputations have to be performed since this corresponds to the strategy of complete recomputation for a large part of the forward integration. This explains the very high number of additional time steps required for a c, d- combination where the product of both values is close to l. On the other hand, it can be seen for this example that the binomial checkpointing with resilience only interferes with the optimality of the binomial checkpointing if the number of checkpoints is less than 0.6 % of the computed intermediate states.

At first sight this might be a little surprising, especially because the resilience distance is considerably smaller than the distance of the checkpoints chosen by the optimal binomial checkpointing approach as illustrated in Fig. 4 for the first four checkpoints to compute the adjoint of 10,000 steps with a varying number of checkpoints. However, as analyzed in [3], the checkpointing strategy implemented in revolve is in most cases only one out of a whole variety of choices that would lead to a minimal number of step recomputations. The approach realized in revolve was taken, because it also minimizes the number of times a checkpoint is written [3, Prop. 2]. If one wants to minimize the distance between two consecutive checkpoints, for example for resilience, a completely different strategy for setting the next checkpoint has to be taken as described in detail in the remainder of this section.

For a given number of time steps l the adjoint of which has to be computed and a given number of checkpoints c, let r be the unique integer such that

$$\beta(c, r-1) < l \leq \beta(c, r) \equiv \binom{c+r}{c}$$

holds. It was shown in [3], that then no time step is executed more than $r+1$ times. For given values of l, c, and hence r, the proof of [3, Prop. 1] yields that the next checkpoint has to be set to a state $\hat{l}$ that fulfills

$$\begin{aligned} \max\{\beta(c, r-2), l - \beta(c-1, r)\} \;&\leq\; \hat{l} \\ &\leq\; \min\{\beta(c, r-1), l - \beta(c-1, r-1)\} \end{aligned}$$

to minimize the overall number of recomputations. The application of this rule in a recursive way then leads to a checkpointing strategy to compute the adjoint with the least possible number of recomputations. Hence, if one wants to minimize the distance between two checkpoints, the state

$$\hat{l} = \max\{\beta(c, r-2), l - \beta(c-1, r)\} \tag{4.3}$$

has to be chosen as next checkpoint. For the very small example of Sect. 3, i.e., $l = 100$ and $c = 5$, one obtains for the computation of the 60th adjoint state then the checkpoint distribution

$$0 \quad 30 \quad 44 \quad 55 \quad 58$$

Hence, this checkpointing distribution fits to the resilience distance of 30. For the larger example considered above, the effects on the checkpointing distribution are illustrated in Fig. 5. As can be seen, the number of time steps between two consecutive checkpoints can be drastically reduced. One can observe in the Figs. 4 and 5 that the distance between two consecutive checkpoints decreases in most cases but unfortunately not always. This observation makes it difficult to explain the surprisingly good complexity result for the binomial checkpointing with resilience. However, for special cases it is possible to justify the good complexity. For this purpose, assume that the identity $l = \beta(c, r-1)+1$

l	10,000	
r	3	≤ 2
c	38,...,139	≥ 140
l	100,000	
r	3	≤ 2
c	83,...,445	≥ 446
l	1,000,000	
r	3	≤ 2
c	180,...,1412	≥ 1413

Table 1: Checkpoint numbers for $r \in \{2, 3\}$

for $r \geq 3$ holds. Then, using the Eq. (4.3) recursively yields that

$$\begin{aligned} l &= \beta(c, r-1) + 1 = \beta(c, r-2) + \beta(c-1, r-1) + 1 \\ &= \beta(c, r-2) + \beta(c-1, r-2) + \beta(c-2, r-1) + 1 \\ &= \sum_{\gamma=0}^{c} \beta(\gamma, r-2) + 1. \end{aligned}$$

According to Eq. (4.3), for this specific choice the checkpoints are set to the states $\hat{l}_i, i = 0, \ldots, c-1$, with

$$\hat{l}_i = \sum_{\gamma=c-i+1}^{c} \beta(\gamma, r-2), \qquad i = 0, \ldots, c-1 \; .$$

Furthermore, the inequalities

$$\beta(\gamma, r-2) \geq \beta(\gamma - 1, r-2)$$

hold for all $\gamma \in \{1, \ldots, c\}$. Therefore in this case, $\beta(c, r-2)$ is the maximal distance between two consecutive checkpoints and provides one possibility to determine a resilience distance that does not interfere with the optimal binomial checkpointing. To examine this distance more closely, we make the following observation. Since the overall adjoint computation should not be too expensive, it is desirable to have $r \in \{2, 3\}$. In these cases, the forward integration needed for the computation of the adjoint needs three to four times the time to perform one forward calculation. For numerous cases, this is still doable but reaches almost the limit of acceptable runtimes. To illustrate the relation between l, c, and r, Table 1 shows the number of checkpoints required to reach $r = 2$ and $r = 3$, respectively. As can be seen, already a reasonable small number of checkpoints yields $r \leq 3$. For $r = 3$, one has

$$\beta(c, r-2) = \beta(c, 1) = \binom{c+1}{c} = c + 1$$

and therefore the number of checkpoints plus one equals the minimal resilience distance for $l = \beta(c, r-1)+1$ that does not interfere with the optimality of the binomial checkpointing. This is also true for a larger range of l. However, there is much more difficult to show this property since the difference to the value $\beta(c, r-1)$ may lead to a different checkpoint distribution. Furthermore the case $r = 2$ is more complicated to analyze since then one has to consider the case

$$\begin{aligned} \hat{l} &= l - \beta(c-1, r) = \max\{1, l - \beta(c-1, r)\} \\ &= \max\{\beta(c, r-2), l - \beta(c-1, r)\}. \end{aligned}$$

5 Conclusions and Future Work

We have presented a modified binomial checkpointing algorithm that supports the restart of the adjoint computation after a failure of the computing system. The modified algorithm maintains the optimality of binomial checkpointing while limiting the maximum distance between successive forward and adjoint checkpoints. The required changes were integrated in the software package revolve for binomial checkpointing and will be made available on the web site of revolve as stated at the tool list web site on www.autodiff.org.

We plan to apply the resilient binomial checkpointing algorithm to compute the adjoint of the MITgcm. It has previously been differentiated by OpenAD [9] using the original binomial checkpointing algorithm. This requires us to examine the additional data that must be checkpointed in order to support restart of the application. This includes OpenAD's `tape` data structures that are used to hold intermediate values as well as global data structures that are used outside the time stepping loop. We plan to use a modified version of OpenAD's template mechanism for revolve to support the restart of the computation.

References

[1] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, 2008.

[2] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[3] A. Griewank and A. Walther. Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, 2000.

[4] Alistair Adcroft, Chris Hill, and John Marshall. Representation of topography by shaved cells in a height coordinate ocean model. *Monthly Weather Review*, 125(9):2293–2315, 1997.

[5] Alistair Adcroft, Jean-Michel Campin, Chris Hill, and John Marshall. Implementation of an atmosphere ocean general circulation model on the expanded spherical cube. *Monthly Weather Review*, 132(12):2845–2863, 2004.

[6] G. Aupy, J. Herrmann, P. Hovland, and Y. Robert. Optimal multi-stage algorithm for adjoint computation. *SIAM Journal on Scientific Computing*, 2016. to appear.

[7] Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver nek5000. *Procedia Computer Science*, 80:1147 – 1158, 2016. International Conference on Computational Science, ICCS 2016.

[8] P. Stumm and A. Walther. Multi-stage approaches for optimal offline checkpointing. siam journal of scientific computing. *SIAM Journal of Scientific Computing*, 31(3):1946–1967, 2009.

[9] Jean Utke, Uwe Naumann, Mike Fagan, et al. OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes. *ACM Trans. Math. Softw.*, 34(4):18:1–18:36, 2008.