Profiling checkpointing schedules in adjoint ST-AD

Laurent Hascoët * Jean-Luc Bouchot * Shreyas Sunil Gaikwad †
Sri Hari Krishna Narayanan ‡ Jan Hückelheim ‡

Abstract

Checkpointing is a cornerstone of data-flow reversal in adjoint algorithmic differentiation. Checkpointing is a storage/recomputation trade-off that can be applied at different levels, one of which being the call tree. We are looking for good placements of checkpoints onto the call tree of a given application, to reduce run time and memory footprint of its adjoint. There is no known optimal solution to this problem other than a combinatorial search on all placements. We propose a heuristics based on run-time profiling of the adjoint code. We describe implementation of this profiling tool in an existing source-transformation AD tool. We demonstrate the interest of this approach on test cases taken from the MITgcm ocean and atmospheric global circulation model. We discuss the limitations of our approach and propose directions to lift them.

1 Introduction

Source-transformation algorithmic differentiation (ST-AD) in its adjoint mode transforms a *primal* code that evaluates some original function into an *adjoint* code that computes its gradient. It is well known [9] that the most efficient implementation of the adjoint code must progress backwards of the original computation, progressively using values originating from the primal execution. The amount of values used grows linearly with the run time of the primal code and, since they are used in the reverse of their production order, their management (*data-flow reversal*) is a key issue that requires a delicate trade-off between storage and recomputation.

This work focuses on one particular setting, where data-flow reversal is primarily done through a stack and the memory cost of this stack is mitigated through a classical storage/recomputation trade-off known as *checkpointing*. Tuning this trade-off is difficult, and optimal tuning exists only in specific cases [8]. In our setting, checkpointing can be applied at every procedure call. When the granularity of procedure calls is not fine enough, checkpointing can also be applied at user-designated procedure fragments, provided they *could* be turned into procedures. Potential checkpointing locations therefore end up being the nodes of the call graph, possibly extended to procedure fragments, and at run time they are nested as the nodes of the call tree. In this work, we examine the question of finding a subset of the checkpointing locations that will result in good enough execution time given a limited storage budget. Optimality, although desirable, has already been shown to be NP-hard [15]. We will rather explore heuristics, based on profiling results, to decide which checkpoint locations should be activated or inhibited to achieve a better performance.

In section 2, we describe in more detail the principle of checkpointing in the setting of our stack-based data-flow reversal, and illustrate its costs and benefits. Section 3 contrasts with checkpointing in other settings and relates this work with other profiling support in existing AD tools. Section 4 describes the information appropriate to guide the choice of activated checkpoints, and how to gather it from a profiled run. Section 5 discusses implementation of this profiling in an existing source-transformation AD tool, and section 6 applies it to two realistic test-cases taken from the MITgcm code suite. We will show how the developer can achieve a significant performance gain by exploiting the profiling results. In section 7, we come back to some limitations of our proposed approach and discuss how they could be overcome, before concluding in section 8.

2 Our checkpointing model / setting

In our setting, data-flow reversal is achieved by storing intermediate values of the primal execution. Consequently, the adjoint code basically obeys a two-sweeps structure:

1. a first, "*forward*", sweep runs the primal code, augmented to organize the storage of whatever intermediate values or derivatives will be needed in the second sweep.

2. a second, "*backward*", sweep propagates the gradients in reverse order, retrieving the intermediate values from the first sweep when needed.

*INRIA, Sophia-Antipolis, France
†Oden Institute for Computational Engineering and Sciences, The University of Texas at Austin, USA
‡Argonne National Laboratory, Lemont, IL, USA

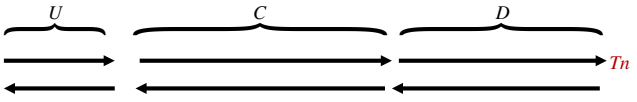

Figure 1: Sketch of execution for the adjoint of a code arbitrarily split as a sequence of three parts U, C, and D. Thick arrows to the right stand for forward sweeps, running the primal code together with storing intermediate values on the stack. Thick arrows to the left stand for backward sweeps, that retrieve values from the stack and use them while propagating gradients backwards. Time goes top-down, and inside each line time follows the arrows direction.

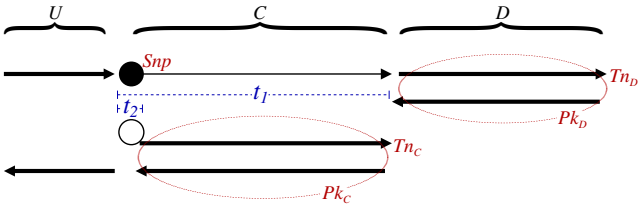

Figure 2: Sketch of execution for the adjoint of the same code than in Fig. 1, this time checkpointing code part C. Time goes top-down, and inside each line time follows the arrows direction. The thin arrow to the right stands for the extra execution of the primal C, and the black and white bullets stand for respectively the writing and reading of the *snapshot*, i.e. a set of variables sufficient for identical duplicate execution of C.

Naturally, the appropriate storage structure for that is a stack. Note that the intermediate values that are not used in the derivative computation, such as those appearing only in linear computations, need not be stored. This can reduce the stack size significantly. AD tools implement several kinds of data-flow analysis to detect many opportunities to reduce the stack usage. Still, stack size generally grows linearly with run-time and the checkpointing technique, which trades recomputation for peak stack size, is unavoidable.

Figure 1 sketches execution of the adjoint code, in the base case where no checkpointing is active. Code fragment C, a candidate for future checkpointing, is singled out from upstream and downstream code U and D, only for future comparison with the case where checkpointing is applied to C. At the turn point between the forward and backward sweeps, the stack reaches its maximum size Tn.

Checkpointing reduces the peak stack size by allowing for one recomputation of a chosen fragment of the primal code, here C as shown in Fig. 2. At the cost of this duplicate execution of primal fragment C, which does not write values to the stack, the stack size reaches two local peaks Tn_D and Tn_C, both hopefully smaller than Tn. On the one hand, Tn_D contains intermediate values from U, D, but not from C, plus a so-called "*snapshot*" needed to run C again and which we hope is smaller than the intermediate values in C. On the other hand, Tn_C only contains intermediate values from U and C. Notice that Tn_C does not contain the snapshot, which has been read to prepare for the forward sweep of C and immediately deleted. Symbols Pk_D, Pk_C, and Snp in Fig. 2 denote stack sizes that will be discussed in section 4. It is worth observing on Fig. 2 that the reading and writing of the snapshot blend nicely with the general stack behavior, which allows us to store the snapshot and the other intermediate values onto the same stack. Whatever efficient implementation, or cache level optimization, exists for the stack will be equally beneficial for snapshots. Also notice that *some* values in the snapshot may have been already stored in U, not too deep in the stack. Although some data-flow analysis can detect this and therefore slightly reduce either the snapshot or the stack after U, our experiments have shown that the benefit is very often minor. As discussed in section 7, this sort of extra improvement, implemented in the AD tool we are using here, can be the cause for inaccuracy in the profiling predictions.

Notice that on the figures, the position and length of the arrows do not represent the stack size, but only their correspondence with the primal code. In other words Tn_C may well be larger than Tn_D.

On large codes, checkpointing can and must be applied repeatedly, recursively inside fragments U, C, and D. With a well-balanced choice of the nested checkpoints, i.e. the choice of C and of all other checkpoints nested inside U, C, and D, one can achieve logarithmic growth – with respect to the primal run time – of both the peak stack size and of the number of times a given primal code fragment is repeatedly run. Notice however that the control structure of the primal code restricts the choice of the nested checkpoints. Basically since C will be re-executed, it must have unique entry and exit points. It must also be "*reentrant*", i.e. contains no adverse side-effect such as an isolated `send` or `receive`, `malloc` or `free`, file I-O, etc. Obvious checkpointing candidates C's are all the (reentrant) procedure calls, but one may also define at will additional checkpoints inside procedure bodies, essentially when these additional checkpoints *could* be turned into reentrant procedure calls. In this work, we will assume that one has defined a large enough collection of nested potential checkpoints, and our goal is to use profiling to selectively inhibit or activate these checkpoints to reach

good performance in run time and in peak stack size.

At one extreme, activating all the potential checkpoints (i.e. actually performing checkpointing on them) will severely affect run time due to primal code recomputations. Conversely, inhibiting all potential checkpoints can easily lead to excessive stack memory use. One can look for an optimal trade off for instance: for a given fixed stack memory budget, look for the checkpoints activation that minimizes primal code recomputation time. Previous works [15] have shown that a general answer to this combinatorial problem is NP-hard, possibly resorting to Integer Programming approaches [13]. We will here explore a greedy approach, progressively improving our subset of activated checkpoints. Still, the information returned by profiling could also be fed to optimal search tools.

The combined questions we consider are: what information can we extract from run-time profiling of the adjoint code, and how can we use it to guide selection of potential checkpoints towards a good enough trade off?

3 Related approaches

In the so-called "*recompute-all*" model, primal values are made available by repeatedly running the primal code from a chosen initial point. This approach trades computation time for memory space. The alternative "*store-all*" model stores the needed primal values on a stack, sparing the extra recomputation time at the cost of storage space. On realistic codes that run for long times, both models require a similar notion of checkpointing, and the question of the optimal choice of checkpoints is the same in both models. The tool TAF [5] ("recompute-all" model) allows one to fine-tune the checkpointing locations and the variables they must store, through user directives. This manual tuning is (by principle) more powerful than any automated approach, but is labor-intensive. It could as well benefit from some profiling.

In some "store-all" models, the values stored on the stack are the intermediate values of variables computed by the primal code, or at least the subset of those that are indeed used in the partial derivatives accumulated during gradient computation. In other models [16] they can be the partial derivatives directly. When storing intermediate values, one can choose to store them upon each primal instruction whose adjoint uses the value. This "*store-on-use*" has the risk of storing the same value several times. The alternative "*store-on-kill*", which we follow in the sequel, is to store the value only when it is going to be overwritten by the next primal instruction.

The present work considers checkpointing essentially at the level of the call tree. In the restricted setting of a time-stepping loop, several checkpointing schedules have been studied. In particular the well-known *binomial checkpointing* [8] is optimal, in its application case. These checkpointing strategies on loops differ from ours as they amount to choosing the extent of the nested checkpoints over the sequences of time steps, whereas our setting operates on a collection of fixed potential checkpoints on which we search a good subset. Given the nested time-step sequences that binomial checkpointing has found optimal, inhibiting checkpointing on any such sequence can only lead to a less efficient scheme. Conversely we do not consider in our setting the option to enlarge or shrink the code fragments C to which checkpointing may be applied. Binomial checkpointing, or its refinements dedicated to very large snapshots such as H-revolve [11], are independent from the present checkpointing on call trees. We will see in section 6.2 how they may complement each other.

Profiling is one answer to improve performance when no static analysis can guarantee optimality at compile time. Machine learning frameworks such as PyTorch, TensorFlow, and JAX offer automatic differentiation capability as well as profiling. In JAX, a common use of profiling is to understand the GPU or TPU memory usage of a program, for example if trying to debug an out-of-memory problem [1]. A profiler trace can be created by instrumenting the code with the `jax.profiler.start_trace()` and `jax.profiler.stop_trace()` methods. The trace can be viewed later using a graphical interface. As opposed to the approach presented here, the profiling information is not used to select a differentiation strategy.

4 A profiling model for checkpointing

We will distinguish static and dynamic checkpoint locations. We call static a location in the source code, and more precisely in the primal source code. This is where the end-user may conveniently decide to (de-)activate a checkpoint e.g. through a differentiation directive. Dynamic checkpoints are the run-time occurrences when the execution pointer reaches a static checkpoint. There can be several dynamic checkpoints for each static one. It is in principle possible to apply different checkpointing choices to each dynamic occurrence, but this is impractical and leads to a more involved implementation. We therefore choose to attach the profiling information that we seek to each static checkpoint, although it will be collected and accumulated at run-time on each of its dynamic occurrences.

As a starting configuration that we will profile and progressively improve upon, we will activate all potential checkpoints. One reason is that this configuration

gives profiling an easier access to useful information. The other reason is that it is less likely to fail at run-time by lack of memory, although it may take longer. It is therefore a safer starting point when the structure of the primal code is not well known. As the improvement process goes on, the checkpointing configuration is modified according to profiling results, and a new profiling run is in theory needed for each configuration in order to determine the next configuration.

For some current configuration, for every potential checkpoint X which is currently activated, we expect profiling to give us the following key figures about inhibiting X:

- the associated run-time benefit $\Delta t(X)$, i.e. the run-time difference when X is switched from activated to inhibited. It is always negative (run-time decreases).
- the associated memory cost $\Delta Pk(X)$, i.e. the peak stack size difference when X is switched from activated to inhibited. It is in general positive except for some extreme cases.

Let us look back at Fig. 2, now assuming we are in the general case where there can be nested checkpoints in the adjoint codes for C and D (and U). Just view C and D as sections of the primal code, in which there may be any number of checkpointed sections X, one of them coinciding with C. Let us call *round trip* the sequence of a forward sweep and its associated backward sweep. For instance the rightmost part of Fig. 2 is the round trip on D, also written $\overline{D}$, the sequence of its forward sweep $\overrightarrow{D}$ and backward sweep $\overleftarrow{D}$. The bottom two arrows of the central part of Fig. 2 form the round trip on C. A round trip can include nested checkpoints and for instance the round trip on $U;C;D$ is exactly Fig. 1 (if C is inhibited) or Fig. 2 (if C is activated).

Since we now assume that there may be nested checkpoints inside D, we must distinguish peak stack sizes Tn_D and Pk_D: as defined already, Tn_D is the (local) peak reached at the turn point at the end of $\overrightarrow{D}$, whereas Pk_D is the (global) maximum peak stack size attained during the round trip on D, i.e. at the turn point after $\overrightarrow{D}$ but also at the turn point of any checkpoint N nested in D. Nothing prevents $\overrightarrow{N}$ from storing more intermediate values than D and therefore $Pk_D \geq Tn_D$ in general. In the sequel we will need to keep track of both Tn_D and Pk_D. The same remark applies to C.

Profiling is performed on the adjoint code corresponding to the current checkpointing configuration. We want profiling to return, for each static checkpoint X that is currently active, the cost/benefits $\Delta t(X)$ and $\Delta Pk(X)$. Only as an intermediate information, profiling will also return $\Delta Tn(X)$. As cost/benefits can be evaluated independently for each X, we will in the sequel drop the (X) to keep notation light.

We will accumulate cost/benefit figures bottom-up and right-to-left on the tree of nested dynamic checkpoints or in other words, bottom-up on the tree of nested round trips. Following Fig. 2, the key recursive operation is therefore to compute these figures for the round trip on the code sequence $C;D$, given these figures for the round trips on C and on D. Notation-wise, we will specify as an index the round trip on which a cost/benefit is computed. Therefore our goal is, given Δt_C, ΔTn_C, ΔPk_C about the round trip on C and likewise Δt_D, ΔTn_D, ΔPk_D about the round trip on D, to compute Δt_{CD}, ΔTn_{CD}, and ΔPk_{CD} about the round trip on $C;D$. We keep in mind that this computation must be made for any active checkpoint X, and therefore in particular for C, but also for any other static checkpoint that may occur, any number of times, in C, in D, or in both. In addition we will also need to keep track of the basis times and stack sizes when none of the currently activated checkpoints is inhibited i.e. regarding notation, compute t_{CD}, Tn_{CD}, Pk_{CD} from t_C, Tn_C, Pk_C, t_D, Tn_D, and Pk_D.

For the base case of a piece of code that contains no activated checkpoint, profiling can easily return the basis times and stack sizes. Note that Pk and Tn are then equal. Δ values are all zero for all X.

For the recursive case, it is easy to verify on Fig. 2 that, about run time:

$$t_{CD} = t_1 + t_D + t_2 + t_C \tag{4.1}$$

where t_1 is the extra time to take the snapshot and run the primal C, and t_2 is the time to read the snapshot. Then about stack sizes:

$$Tn_{CD} = Tn_D \tag{4.2}$$

$$Pk_{CD} = max(Pk_D, Pk_C) \tag{4.3}$$

For each X, cost/benefit results are the changes in the value of t_{CD}, Tn_{CD}, and Pk_{CD}, from the reference case (equations 4.1, 4.2, 4.3) with no additional checkpoint inhibited, to the case where only X is switched to inhibited. These Δ values are computed differently whether X coincides with C or not. If $X \neq C$ then the effect of inhibiting checkpoint X does not affect C and the resulting computation scheme remains as in Fig. 2. Therefore:

$$\Delta t_{CD} = \Delta t_D + \Delta t_C \tag{4.4}$$

$$\Delta Tn_{CD} = \Delta Tn_D \tag{4.5}$$

$$\Delta Pk_{CD} = max \begin{pmatrix} Pk_D + \Delta Pk_D \\ Pk_C + \Delta Pk_C \end{pmatrix} - Pk_{CD} \tag{4.6}$$

If on the other hand $X = C$, execution when inhibiting X will follow the scheme of Fig. 1. There is no duplicate execution of the primal C so that time benefit becomes:

$$\Delta t_{CD} = \Delta t_D + \Delta t_C - t_1 - t_2 \tag{4.7}$$

The stack space used by $\overrightarrow{C}$ now accumulates with that used by the round trip $\overline{D}$. Stack size after $\overrightarrow{C}$ in Fig. 1 is indeed $Tn_C + \Delta Tn_C$. On top of it is accumulated the stack space specifically used by $\overrightarrow{D}$ and $\overline{D}$ which are respectively $Tn_D + \Delta Tn_D - Snp$ and $Pk_D + \Delta Pk_D - Snp$, where Snp is the stack size measured just after taking the snapshot.

$$\Delta Tn_{CD} = Tn_C + \Delta Tn_C + \Delta Tn_D - Snp \tag{4.8}$$

$$\Delta Pk_{CD} = max \begin{pmatrix} Tn_C + \Delta Tn_C + Pk_D + \Delta Pk_D - Snp \\ Pk_C + \Delta Pk_C \end{pmatrix} - Pk_{CD} \tag{4.9}$$

All the times and stack sizes needed can be measured while running with C activated. Note that ΔTn_{CD} and ΔPk_{CD} can be negative in the extreme case where Tn_C is less than SNP, or in other words when the snapshot for checkpointing C is larger than the collection of intermediate values stored during $\overrightarrow{C}$. In this extreme case, checkpointing C is never beneficial.

5 Implementation

We implemented our profiling tool inside the source-transformation AD tool Tapenade [10].

5.1 Installing callbacks Section 4 tells us which elementary data must be collected from the profiled run in order to compute the estimated cost/benefits. These are, for each active checkpoint:

- the run times t_1 and t_2.
- the stack size Snp immediately after taking the snapshot.

and in addition every local peak stack size Tn, measured at each turn point. Collecting this data is implemented by installing a number of callbacks in the differentiated code. These callbacks are inserted automatically by Tapenade when invoked with the `-profile` command-line option. Figure 3 sketches the instrumented adjoint code produced by Tapenade, corresponding to Fig 2. The added callbacks (highlighted in red) all belong to a new `adProfileAdj` package. Calls occur in the forward sweep before writing a snapshot and before and after calling the duplicate primal C, and symmetrically in the backward sweep before reading the snapshot and before and after calling the round trip $\overline{C}$ which is $\overrightarrow{C}; \overleftarrow{C}$. Last but not least, a callback to `adProfileAdj_turn()` is inserted at each turn point between a forward and a backward sweep.

```
< code for U→ >
CALL adProfileAdj_SNPWrite("C", 42)
CALL PUSH(snapshot)
CALL adProfileAdj_beginAdvance("C", 42)
CALL C()
CALL adProfileAdj_endAdvance("C", 42)
< code for D→ >
```

```
CALL adProfileAdj_turn()
```

```
< code for D← >
CALL adProfileAdj_SNPRead("C", 42)
CALL POP(snapshot)
CALL adProfileAdj_beginReverse("C", 42)
CALL C̄()
CALL adProfileAdj_endReverse("C", 42)
< code for U← >
```

Figure 3: Adjoint code with profiling callbacks inserted, corresponding to the code sketch of Fig 2. Forward sweep on top, turn point in the middle, backward sweep below. Except for `turn`, callbacks take as argument the called function name and the corresponding line number in the primal code, for later reference.

The profiling mechanism requires minor additional adaptations to blend well with sophisticated structures of the adjoint code such as binomial adjoint (for time-stepping loops) and two-phases adjoint (for fixed-point loops).

5.2 Callback implementation Implementation of the profiling callbacks maintains an internal stack of the activated checkpoints encountered, to retrieve correspondence from forward sweep to backward sweep, between the callbacks on a given dynamic checkpoint. Timing t_1 is deduced from measurements between `SNPWrite` and `endAdvance`, t_2 between `SNPRead` and `beginReverse`. The final computation of the cost/benefits for the round trip $C; D$ is entirely performed during `endReverse`, when all the needed values t, Tn, Pk, and all Δt, ΔTn, ΔPk are known for C and D. At the end of the profiled run, a readable summary of the cost/benefits is displayed.

Complexity of the profiling process lies in that of callback `endReverse`, that implements all equations 4.1 to 4.9, and that is called once per run-time procedure call (or encounter of a manual checkpoint). In each `endReverse`, equations 4.1, 4.2, and 4.3 need to be evaluated only once, whereas the subsequent equations need to be evaluated once per static checkpoint X. The total complexity is therefore the product of the size of the call tree times the number of static checkpoints. Memory-wise, the data stored by profiling occupies a size at most proportional to the number of static checkpoints times the depth of the call tree.

5.3 Inhibiting / activating checkpoints Bearing in mind that calls are always checkpointed by default, the functionality for inhibition of potential checkpointing locations was already present in the AD tool Tapenade. End-users could play with it but our impression is that very few did so, possibly due to the absence of a profiling tool to guide them. The user interface of Tapenade provides two ways to inhibit checkpointing on calls:

- On the Tapenade command-line, one can use option `-nocheckpoint "foo foo2"`, which will prevent checkpointing on every call to procedures `foo` and `foo2`.

- In the primal source, one can place pragma `$AD NOCHECKPOINT` before definition of a procedure `foo`, which inhibits checkpointing on every call to `foo` (equivalent to the command line option). Alternatively one can place the same pragma before any individual call to `foo`, thus selectively inhibiting this particular call.

5.4 Inhibiting checkpointing on externals Since external routines are not shown to the AD tool, Tapenade provides a way for the end-user to specify information about these externals that are useful for a better differentiation, through a special-purpose specification file. These are typically about types and read/write status of their arguments, or about the differentiable dependency of their outputs on their inputs. The end-user is also required to write the derivative code for these externals. For a non-checkpointed call to an external, the AD tool can produce a better code by knowing which of the arguments are used in a "*non-linear*" way by the external, or in other words which of the values *returned* by the external must be, when later overwritten, preserved on the stack for future use in the backward sweep. If this information is not provided to the AD tool, the conservative information is built from the read/write status, possibly causing stack storage of more values than necessary. We thus extended the mini-language of the specification file to designate these "non-linear" arguments.

Symmetrically, the differentiated code for a non-checkpointed external `foo` consists of two procedures `foo_fwd` and `foo_bwd` (instead of a single `foo_b` in the classical checkpointed case). The end-user is required to provide their implementation. One consequence of not checkpointing is that `foo_fwd` may be required by the differentiation context to take care of preserving some of its input values, on the stack, and `foo_bwd` must restore these values from the stack at the symmetric location. This extra task for the end-user is easy enough and worth the effort. However, in cases where the user can not do so, the conservative fallback option is to let the AD tool store what is needed immediately before the call to `foo_fwd` and restore just after `foo_bwd`. This will be triggered by another option in the external specification file. Interaction with the end-user thus occurs in two phases:

1. Before differentiation, the end-user can use the specification file to provide the non-linearly used arguments of the external procedures. The user also has the option to accept that the hand-written `foo_fwd` and `foo_bwd` take care of storing and restoring some of their arguments.

2. After differentiation, the AD tool emits messages that tell the user which differentiated external procedures must be provided, specifying which derivatives (of which output with respect to which input) must be computed, and whether they must come in the `foo_b` form or in the `foo_fwd`, `foo_bwd` form. When the user has accepted to do so, messages also specify the input arguments that must be preserved in case they are overwritten in `foo_fwd`.

5.5 Asynchronous stack offloading to files Searching for a good checkpointing scheme is all the more important for large applications that need a huge amount of memory to store the stack. Until recently Tapenade kept this stack in RAM, and this has been noted as an unfortunate limitation by several users. We implemented a new stack mechanism in Tapenade, that triggers offloading of the deeper parts of the stack to files. It is a variation on virtual memory: when the stack size grows close to a user-defined given limit, the parts of the stack that were pushed early, and therefore that will be popped late, are stored to files and the associated main memory becomes available again. The reciprocal mechanism occurs during the backward sweep. These deep parts of the stack, that are sub-

ject to this file write/read mechanism, are in general "dormant" i.e. disjoint from the stack top where all push/pop activity occurs. Therefore Tapenade offers the option of performing the file write/read operations asynchronously, using Posix `pthreads`. When a spare thread is available, this reduces the overhead of the file offloading mechanism.

For instance the application described in section 6.3 makes use of this stack offloading mechanism, with a stack peak that goes over 4 Gb. while never using more main memory than a user-defined limit of 256 Mb.

6 Application

We test our profiling capability on test cases from the Massachusetts Institute of Technology General Circulation Model (MITgcm) [14, 2]. The MITgcm is widely used by the climate science community to simulate planetary atmosphere and ocean circulations, and its dynamical core has been successfully used to also perform ice stream and ice shelf simulations, and thus develop synchronous ice-ocean coupling capabilities as well as its adjoint [12, 7].

MITgcm has always been developed to be compatible with AD tools, specifically TAF [5], enabling the generation of tangent-linear and adjoint models. It is thus a key component of the Estimating the Circulation and Climate of the Ocean (ECCO) state estimate: a data assimilation product widely employed by the physical oceanography research community. The MITgcm has recently been differentiated using Tapenade [4], which is what allows us to perform the profiling experiments discussed next.

6.1 Test `streamice`: The `halfpipe_streamice` test case uses the package `streamice` to simulate the flow of land ice in a valley. The simulation runs for 80 timesteps on a 40×20 horizontal grid. The individual time steps are more complex than a typical MITgcm simulation since they contain fixed point iteration based solvers to solve the non-linear stress balance. Their adjoint is built according to Christianson's two-phases algorithm [3, 6].

Adjoint AD of `streamice` was first performed with OpenAD in 2016 [6], and a few years later with Tapenade. Both tools apply the same strategy for the adjoint of the fixed-point loop. In both cases the end-user chose not to apply binomial checkpointing to the outer time-stepping loop. The structure of the adjoint codes is therefore similar. Typically, run-time performance of an adjoint code is measured by the ratio with the run-time of the primal. The adjoint code by Tapenade yielded a 7.0 ratio, which is fair but not as good as with OpenAD. Likewise, on most MITgcm test cases, the ratio obtained with Tapenade is not as good as the average ratio obtained with TAF. By default, Tapenade activates checkpointing at each procedure call, whereas OpenAD doesn't. Checkpointing in TAF is carefully hand-tuned, with methods that have not been published. Certainly, there is room for improvement from the default checkpointing scheme of Tapenade.

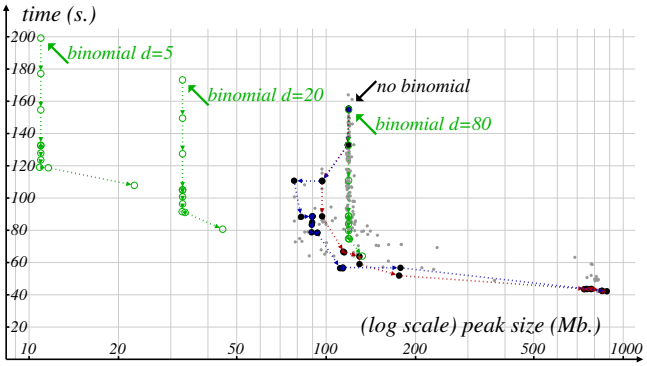

Figure 4: Tuning the adjoint of `streamice` by profiling. Each checkpointing configuration is shown as a dot whose coordinates are peak stack size and run time of the adjoint. The red and blue dotted lines show the evolution of performance of the adjoint by repeatedly following the suggestions made by our profiling tool. The red line follows a "run-time gain first" strategy, the blue line a "careful on memory" strategy. Time measurements are averaged on 5 runs. Smaller gray dots show the performance of 250 random checkpointing configurations. Green dots show combination with binomial checkpointing, discussed in section 6.2

Fig. 4 summarizes this search for a better checkpointing configuration. The initial configuration with all checkpoints activated (indicated as "*no binomial*") has an average run-time of 155.4 seconds and a peak stack size of 119.1 megabytes. The number of static checkpoint locations is 85, yielding an enormous 2^{85} number of possible configurations to explore.

We use our profiling tool to explore only the most profitable changes. Incidentally, we compare adjoint run-time with and without profiling and find a negligible 1.1 second overhead for profiling. The suggestions returned by the profiled adjoint code are sorted in 3 categories:

- Static checkpoints whose inhibition give a run-time gain (as always), and a stack peak gain. Those are rare cases where the snapshot is larger than the amount of intermediate values stored in the forward sweep. This is generally due to overestimation of the AD data-flow analysis on arrays, where activity for a single cell of an array implies activity of the whole array. This sort of profiling result means

that the checkpoint should be inhibited without hesitation.

- Static checkpoints whose inhibition give no change of the stack peak. This happens when the effect of inhibiting the checkpoint indeed modifies the size of one local peak of the stack, but this peak is smaller than the global peak and therefore the change is hidden. The checkpoint is a good candidate for inhibition, as this will gain some run time without changing the global peak size.
- Static checkpoints whose inhibition will increase the stack peak size. Decision to inhibit it results from a trade off decision, comparing with the run-time expected benefit and possibly the available budget in memory space.

As these suggestions allow for several alternative decisions, the end-user may apply different strategies. Two example strategies are presented in Fig. 4: one (red line) follows the suggestions with the highest run-time gain first, but deferring as much as possible suggestions with a high memory cost. The other (blue line) follows suggestions with the lowest memory cost first, regardless of run-time gain. For both strategies, only one or a few suggestions are selected and applied jointly, then differentiation is performed and the adjoint code is run again, yielding new suggestions. This process is repeated until no suggestion remains, which is when all 85 checkpoints are inhibited. In real life, the end user will probably stop the process before that stage, as the memory cost may go over the available memory budget. Fig. 4 shows with black dots the time and memory performance of the successive adjoint codes after each step. Gray dots show the performance of 250 randomly chosen checkpointing configurations. We observe that they are mostly situated above and right of the red and blue lines, indicating that the profiling suggestions, combined with these strategies, take performance close to the front of optimal configurations.

From the results in Fig. 4, we find a few checkpointing configurations that offer a significant time gain for a relatively small memory cost, or even sometimes a small gain. For instance one dot on the red line exhibits a stack peak of 176.0 Mb, not too much above the initial 119.1 Mb, and a time of 51.9 seconds, a significant gain from the initial 155.4 s. resulting in a slowdown ratio of 2.4. If memory is really scarce, another dot on the blue line exhibits a stack peak of 111.41 Mb, even less than the initial 119.1 Mb, and a time of 56.49 s. and therefore a still very good ratio of 2.6. Unless memory is very abundant, it is unlikely that the end-user pushes the process to its extreme, costing 880.6 Mb, even if time becomes as short as 42.2 s.

6.2 Binomial checkpointing A natural question is the comparison and interaction between our checkpointing on the call tree and binomial checkpointing on time-stepping loops. Let us introduce binomial checkpointing into the `streamice` test case by adding directive

```
$AD BINOMIAL-CKP
```

on the time-stepping loop. Although the test case features a limited number of time steps (80), our experiments clearly capture the effect of binomial checkpointing and its interaction with our profiling method. Binomial checkpointing is tuned by choosing the maximum number d of "before time-step" snapshots that can be stored together in memory. Figure 4 shows (in green) experiments with an extreme $d = 80$, a more reasonable $d = 20$, and a quite realistic $d = 5$. For each d, dotted arrows show the evolution of performance when following profiling suggestions (highest run-time gain first) until no suggestion remains. The $d = 80$ case is interesting mostly as it makes the link with the experiment with no binomial ckeckpointing: allowing d to be as high as the total number of time-steps is equivalent to discarding binomial checkpointing altogether and activating checkpointing on the time-stepping loop body, which happens here to be a single procedure call. Therefore all green dots of the $d = 80$ evolution are equivalently checkpointing configurations of the experiment with *no* binomial checkpointing (i.e. black and small gray dots), but they belong to a subset where checkpointing remains activated on the time-stepping loop body. Evolution of the green dots is therefore restricted and cannot reach performance as good as the red and blue evolutions. Notice that the red and blue evolutions almost immediately choose to inhibit checkpointing on the time-stepping loop body.

Binomial checkpointing controls memory usage through the choice of d. A smaller d reduces the memory used to store snapshots, at the cost of a higher number of duplicate execution of time-steps. The green dots for $d = 20$ and $d = 5$ demonstrate that. For all three d, our profiling tool provides profitable suggestions, decreasing run-times by a factor 1.8 to 2.4 at almost no memory cost.

We interpret the almost vertical shape of the green curves by the fact that each configuration systematically uses d "time-stepping" snaphots for any activation pattern of the other checkpoints. Observe that with binomial checkpointing, round trips on each time-step always occur separately. Therefore, the memory cost of inhibiting checkpoints inside a time-step is not multiplied by the number of time steps. Profiling suggestions reflect that, by finding lower memory costs (and benefits) to inhibiting checkpoints that are contained in a time-stepping loop with binomial checkpointing. As

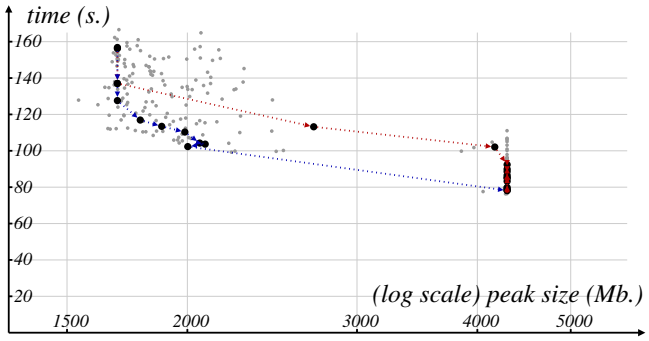

Figure 5: Tuning the adjoint of `tutorial_global_oce_biogeo` by profiling. Each checkpointing configuration is shown as a dot whose coordinates are peak stack size and run time of the adjoint. The red and blue lines show the evolution of performance guided by profiling, respectively with a "run-time gain first" strategy, and a "careful on memory" strategy. Gray dots show the performance of 250 random checkpointing configurations.

long as this cost is small compared to that of the d "time-stepping" snaphots, it remains almost invisible. It only becomes visible when almost all checkpointing locations are inhibited.

6.3 Test `tutorial_global_oce_biogeo` : The MITgcm test case `tutorial_global_oce_biogeo` is a global simulation on a $128 \times 64 \times 15$ grid. It also simulates the biogeochemistry aspects of the ocean, for example, the carbon uptake and how it is affected by the sea surface temperature anomalies. We simulate 10 steps so that the runtime is similar to the `halfpipe_streamice` test. There is no binomial checkpointing as a default. Figure 5 shows the improvement in performance obtained by applying the suggestions from profiling, with the same two strategies as in Fig 4.

From this experiment it becomes clear that making an informed choice of checkpointing is critical. Indeed, one can easily reduce by 35% the compute time, yet avoiding the massive memory consumption of an extreme "no checkpoint" strategy.

7 Further research

Experience raises a few questions about our approach. One issue is that profiling is run on one given checkpointing configuration, and changing this configuration can lead to another regime with different profiling results, although often only slightly. Therefore it may be safer to re-run differentiation and profiling each time one checkpoint is inhibited. This can prove impractical and we often chose to switch several checkpoints at a time, possibly missing a better configuration. In other words, to which extent can we use one set of profiling suggestions for several checkpoints at a time, or re-use this set even if some more checkpoints are inhibited? This may relate to relative position in the tree of nested round trips.

Alternatively, one can probably extract enough information from the initial profiling run, enabling some separate optimal search tool to directly find the front of "optimal" configurations. This could avoid the need for re-profiling, but would require storage of profiling data proportional to the size of the (dynamic) tree of nested checkpoints, which can be large. In comparison, our approach only stores at a given time data proportional to the depth of the tree of nested checkpoints, times the number of static checkpoints.

Profiling predictions can also be inaccurate due to optimization of the adjoint code, which the AD tool performs using static data-flow analysis. This leads in many cases to inexact predictions, often too optimistic in favor of inhibiting a checkpoint. To understand the issue, recall that an activated checkpoint C causes the forward sweep $\overrightarrow{C}$ to be followed immediately by $\overleftarrow{C}$. A good AD tool will take advantage of this to remove some "*adjoint-dead*" code at the junction of the two sweeps, such as intermediate values pushed then immediately popped from the stack, as well as primal computations that are not needed for the upcoming backward sweep. Taking the snapshot before running the primal C also has the consequence of possibly reducing the set of values that will be stored during $\overrightarrow{D}$ and $\overrightarrow{C}$ (found by the "TBR" analysis) because the snapshot already stored these values. The benefits of these data-flow analysis is invisible at profiling time. All in all, time and memory performance of the configuration with an activated checkpoint C are not easily modeled, or equivalently inhibiting C often brings improvements that are not exactly those deduced from our profiling measurements. An obvious answer to this disturbing issue would be to de-activate these data-flow analysis in the AD tool, at least during the iterative profiling process. However, one would then optimize a derivative code which is not the same as the final one, with a chosen checkpointing configuration that may not be the most appropriate.

To validate the discussion above, we have run two series of profiling experiments on the `streamice` test case, that are identical except that one series (*optim-on*) is performed with the standard Tapenade whereas the other (*optim-off*) uses a Tapenade with a deactivated "adjoint-dead" optimization. We could not deactivate the "TBR" analysis, as this changes the adjoint code too radically. In each series of experiments, we have considered the following cost/benefit predic-

	exact	good	acceptable	inexact
optim-on	24	9	17	28
optim-off	29	17	11	21

Table 1: Exactness of memory cost predictions when activating or deactivating the "*adjoint-dead*" adjoint code optimization. In both settings, we count the number of predictions that are "exact", not exact but "good" (estimation error less than 0.01Mb), not good but "acceptable" (estimation error less than 0.1Mb).

tions: on the one hand each prediction corresponding to each suggestion followed along the "careful on memory" itinerary (blue line of Fig. 4), and on the other hand all predictions, followed or not, made by the initial profiling step i.e. on the initial configuration with all potential checkpoints activated. For technical reasons, predictions made about procedures called at several static locations are not considered. For each prediction, we inhibit the corresponding checkpoint, then differentiate and run, thus measuring the actual peak stack change for comparison with the predicted one. Our results shown in Table 1 mostly corroborate our tentative explanation as we observe better predictions on the *optim-off* experiment. Still, this deserves more investigation.

It is visible in figures 4 and 5 that some relatively better checkpointing configurations are not attained by following the two strategies that we experimented with. Visually there are a few gray dots below-left of the dotted lines. We mostly interpret this as a consequence of the few inexact profiling predictions that we just mentioned. It also indicates that, in addition to the red and blue dotted lines on the figures, there may exist better strategies for selecting the next profiling suggestion to follow.

As already mentioned, checkpointing can also be applied at the level of time-stepping loops, for which the well-known binomial checkpointing [8] provides an optimal schedule. On large codes, we need to apply both "kinds" of checkpointing: (optimal) binomial checkpointing on time-stepping loops, and call tree checkpointing (tuned by profiling) on the rest of the code. Unfortunately we have no unifying framework that encompasses the two. Nevertheless, interactions exist between both: the same stack that we used throughout this work can also accommodate the snapshots of binomial checkpointing. Given a memory budget for this stack, the choice exists to use it to allow for more snapshots for binomial checkpointing, or to inhibit more call-tree checkpoints in the body of the time-steps. Since a cost model exists for both options, the profiling tool could suggest which option is more profitable, thus making one step towards blending the two approaches.

8 Conclusion

The memory/recomputation trade off known as checkpointing is a key ingredient for adjoint (or reverse) differentiation of large applications. Checkpointing can and must be applied at many levels in the target application, in particular at the nodes of the call tree or on iterative loops. The choice of the locations that deserve checkpointing or not has a major impact on the final performance, in run time and memory usage, of the adjoint code. Yet very little guidance is provided to the end-user. We proposed a profiling algorithm that predicts the performance cost/benefit of checkpointing for each node of the call tree. We experimented with greedy strategies that exploit these predictions, progressively improving performance of the adjoint code by adapting its checkpointing schedule. On the two large test cases that we studied, both from the MITgcm test suite, we obtain significant performance gains, up to a two- or three-fold speedup for a minor memory cost. Profiling for better checkpointing appears as very promising.

The profiling algorithm that we proposed is a prototype. It is now available as a component of the AD tool Tapenade. It could be worth exploring more sophisticated strategies to find almost optimal checkpointing schedules, directly instead of step by step, from a similar profiling pass. It also seems profitable to guide the end-user decision better based on profiling results, or to further automate the process. In particular the interplay between call tree checkpointing and binomial checkpointing should be quantified as well, and integrated in the decision strategy.

Acknowledgments

References

[1] Profiling JAX programs. `https://jax.readthedocs.io/en/latest/profiling.html`, 2024. Online; accessed 13 May 2024.

[2] Alistair Adcroft, Jean-Michel Campin, Ed Doddridge, Stephanie Dutkiewicz, Constantinos Evangelinos, David Ferreira, Mick Follows, Gael Forget, Baylor Fox-Kemper, Patrick Heimbach, Chris Hill, Ed Hill, Helen Hill, Oliver Jahn, Jody Klymak, Martin Losch, John Marshall, Guillaume Maze, Matt Mazloff, Dimitris Menemenlis, Andrea Molod, and Jeff Scott. *MITgcm User Manual*. Available at https://mitgcm.readthedocs.io.

[3] B. Christianson. Reverse accumulation and attractive fixed points. 1994. Original article can be found at: http://www.informaworld.com/smpp/title content=t713645924 db=all Copyright Taylor and Francis

/ Informa. [Originally issued in 1992 as UH Technical Report 258].

[4] Shreyas Sunil Gaikwad, Sri Hari Krishna Narayanan, Laurent Hascoët, Jean-Michel Campin, Helen Pillar, An Nguyen, Jan Hückelheim, Paul Hovland, and Patrick Heimbach. Mitgcm-ad v2: Open source tangent linear and adjoint modeling framework for the oceans and atmosphere enabled by the automatic differentiation tool tapenade, 2024.

[5] R. Giering and T. Kaminski. Recipes for Adjoint Code Construction. *ACM Transactions on Mathematical Software*, 24(4):437–474.

[6] D. N. Goldberg, S. H. K. Narayanan, L. Hascoët, and J. Utke. An optimized treatment for algorithmic differentiation of an important glaciological fixed-point problem. *Geoscientific Model Development*, 9(5):1891–1904, 2016.

[7] D.N. Goldberg, K. Snow, P. Holland, J.R. Jordan, J.-M. Campin, P. Heimbach, R. Arthern, and A. Jenkins. Representing grounding line migration in synchronous coupling between a marine ice sheet model and a z-coordinate ocean model. *Ocean Modelling*, 125:45–60, 2018.

[8] 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. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[9] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[10] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013.

[11] Julien Herrmann and Guillaume Pallez (Aupy). H-revolve: A framework for adjoint computation on synchronous hierarchical platforms. *ACM Trans. Math. Softw.*, 46(2), jun 2020.

[12] James R. Jordan, Paul R. Holland, Dan Goldberg, Kate Snow, Robert Arthern, Jean-Michel Campin, Patrick Heimbach, and Adrian Jenkins. Ocean-forced ice-shelf thinning in a synchronously coupled ice-ocean model. *Journal of Geophysical Research: Oceans*, 123(2):864–882, 2018.

[13] J. Lotz, U. Naumann, and S. Mitra. *Mixed Integer Programming for Call Tree Reversal*, pages 83–91. 2016.

[14] J. Marshall, A. Adcroft, C. Hill, L. Perelman, and C. Heisey. A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers. *J. Geophys. Res. Oceans*, 102:5753–5766, 1997.

[15] U. Naumann. DAG reversal is NP-complete. *Journal of Discrete Algorithms*, 7(4):402–410, 2009.

[16] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4):18:1–18:36, 2008.