# H-Revolve: A Framework for Adjoint Computation on Synchronous Hierarchical Platforms

Julien Herrmann, Guillaume Pallez

## To cite this version:

Julien Herrmann, Guillaume Pallez. H-Revolve: A Framework for Adjoint Computation on Synchronous Hierarchical Platforms. ACM Transactions on Mathematical Software, Association for Computing Machinery, 2020, 10.1145/3378672 . hal-02080706v2

HAL Id: hal-02080706

https://hal.inria.fr/hal-02080706v2

Submitted on 28 Feb 2020

# H-Revolve: A Framework for Adjoint Computation on Synchronous Hierarchical Platforms

JULIEN HERRMANN, Inria & Labri, Univ. of Bordeaux, Talence, France
GUILLAUME PALLEZ (AUPY), Inria & Labri, Univ. of Bordeaux, Talence, France

We study the problem of checkpointing strategies for adjoint computation on synchronous hierarchical platforms, specifically computational platforms with several levels of storage with different writing and reading costs. When reversing a large adjoint chain, choosing which data to checkpoint and where is a critical decision for the overall performance of the computation. We introduce H-Revolve, an optimal algorithm for this problem. We make it available in a public Python library along with the implementation of several state-of-the-art algorithms for the variant of the problem with two levels of storage. We provide a detailed description of how one can use this library in an adjoint computation software in the field of automatic differentiation or backpropagation. Finally, we evaluate the performance of H-Revolve and other checkpointing heuristics though an extensive campaign of simulation.

Additional Key Words and Phrases: Adjoint computation, automatic differentiation, deep learning, revolve, hierarchical memory

**ACM Reference Format:**
Julien Herrmann and Guillaume Pallez (Aupy). 2020. H-Revolve: A Framework for Adjoint Computation on Synchronous Hierarchical Platforms. *ACM Trans. Math. Softw.* 1, 1, Article 1 (January 2020), 25 pages. https://doi.org/10.1145/3378672

## 1 INTRODUCTION

Computation of adjoint is at the core of many scientific applications, from climate and ocean modeling [Adcroft et al. 2008] to oil refinery [Brubaker 2016]. In addition, the structure of the underlying dependence graph is also at the basis of the retropropagation step of machine learning [Kukreja et al. 2018a].

With that many applications come a wide variety of software (ADtool, Adol-C, Tapenade, etc). Each software for adjoint computations needs to implement different steps: the algorithm library based on parameters of the system; the checkpointing techniques, parallelisation of the code and overall front-end. Due to the abundance of algorithmic problems arising from these implementations, many of the recent algorithmic advances for adjoint computations are still not implemented in the latest software and suboptimal strategies are used [Pringle et al. 2016]. The *Devito Project* [Louboutin et al. 2018] proposes to implement an API to deal with the implementation of checkpointing strategies (shared-memory parallelism, vectorization etc). They rely on libraries outputing the algorithms to be used. In this work we present such a library for hierarchical storage systems.

Authors' addresses: Julien Herrmann, Inria & Labri, Univ. of Bordeaux, Talence, France, julien.herrmann@inria.fr; Guillaume Pallez (Aupy), Inria & Labri, Univ. of Bordeaux, Talence, France, guillaume.pallez@inria.fr.

0098-3500/2020/1-ART1 $15.00
https://doi.org/10.1145/3378672

Storage has been one of the key issue with the computation of adjoints. The computation of adjoints has always been a trade-off between recomputations and memory requirements [Griewank 1992]. Adjoint computations are now larger than ever and need always more storage. As an example, MITGcm executes approximatively 350k timesteps [Walther and Narayanan 2016] to simulate flows in the atmosphere or the ocean corresponding to 1 year of physical time. These computations are executed on supercomputers such as Mira (US based supercomputer) [Schanen et al. 2016] or Archer (UK based supercomputer) [Walther and Narayanan 2016]. To improve the performance of supercomputers, multiple level of storage are now available, such as RAM, NVRAM, Burst-Buffers, PFS, etc. As an example, each node of Summit, the current fastest supercomputer in the world according to the TOP500 list, is connected to 96GB of High-Bandwidth memory, then to 512 GB of SDRAM, then to 800GB of NVRAM (shared), and finally they have access to 250PB of storage (shared) [ORNL [n.d.]].

In this work we present a strategy to schedule adjoint computations on a hierarchical storage system. We implement it into a library called H-Revolve along with other algorithms from the literature for platforms with two levels of storage.

We make the following contributions:

- We present an algorithm taking full advantage of the storage hierarchy of a computing system with an unbounded number of storage levels (Section 3);
- Then we present an optimality statement for an algorithm used with two level of storage from the literature which has the advantage of being easier to implement with the existing tools (Section 3.2);
- We present a library that implements different algorithms for adjoint computations on machines with multiple levels of storage;
- Finally we show on a set of simulations the advantages of our algorithm compared to current implementations in various frameworks.

We start by introducing the problem and related work in Section 2.

## 2 FRAMEWORK

### 2.1 The Adjoint Computation problem

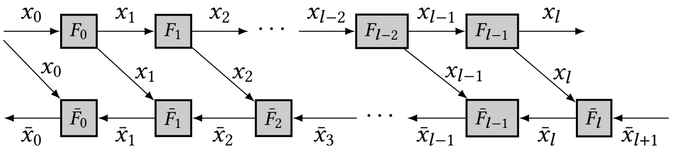


Fig. 1. The data dependencies in the Adjoint Computation (AC) chain.

**Definition 1** (Adjoint Computation [Griewank and Walther 2000; Stumm and Walther 2009]). An adjoint computation with $l$ time steps can be described by the following set of equations:

$$F_i(x_i) = x_{i+1} \text{ for } 0 \leq i < l \tag{1}$$

$$\bar{F}_i(x_i, \bar{x}_{i+1}) = \bar{x}_i \text{ for } 0 \leq i \leq l \tag{2}$$

The dependencies between these operations[1] are represented by the graph $\mathcal{G} = (V, E)$ depicted in Figure 1.

[1] In some approaches [Griewank 1992], an extra $F_l$ operation is included. The algorithm with and without this operation are essentially the same.

The $F$ computations are called *forward* steps. They have an execution cost of $u_f \in \mathbb{R}^+$. The $\bar{F}$ computations are called *backward* steps, they have an execution cost of $u_b \in \mathbb{R}^+$. If $\bar{x}_{l+1}$ is initialized appropriately, then at the conclusion of the adjoint computation, $\bar{x}_0$ will contain the gradient with respect to the initial state ($x_0$).

## 2.2 Computing Platform

In this work, we consider a computing platform with computational buffers and a hierarchic storage system. The computational buffers are used as input and output storage for the computations. The hierarchic storage system consists of $K$ levels of storage with increasing access costs where data can be stored for later usage.

**Definition 2** (Hierarchical Platform)**.** We consider a platform with $K + 1$ storage locations:

- *Buffers:* there are two buffers, the top buffer and the bottom buffer. The top buffer is used to store a value $x_i$ for some $i$, while the bottom buffer is used to store a value $\bar{x}_i$ for some $i$. For a computation ($F$ or $\bar{F}$) to be executed, its input values have to be stored in the buffers. Let $\mathcal{B}^\top$ and $\mathcal{B}^\perp$ denote the content of the top and bottom buffers. In order to start the execution of the chain, $x_0$ must be stored in the top buffer, and $\bar{x}_{l+1}$ in the bottom buffer[2]. Hence without loss of generality, we assume that at the beginning of the execution, $\mathcal{B}^\top = \{x_0\}$ and $\mathcal{B}^\perp = \{\bar{x}_{l+1}\}$.
- *K levels of storage:* each level has $\mathbf{c}_k$ slots of storage where the content of a buffer can be stored. The time to write from buffer to the level $k$ of storage is $\mathbf{w}_k$. The time to read from the level $k$ of storage to buffer is $\mathbf{r}_k$. Note that we assume that data moves are synchronous, meaning that moving data can not be performed at the same time as a computation (see Section 2.5 for more details). We assume that the levels of storage represent a hierarchy where each level is "further away" compared to the processor than the previous one. Hence, we assume that for every $k$, $\mathbf{w}_k \leq \mathbf{w}_{k+1}$ and $\mathbf{r}_k \leq \mathbf{r}_{k+1}$. Let $\mathcal{M}_k$ be the set of $x_i$ and $\bar{x}_i$ values stored in the level $k$ of storage at any time. The storage is empty at the beginning of the execution ($\forall k, \mathcal{M}_k = \emptyset$).

The different levels of storage represent the storage locations of the architecture where each level is closer to the computational resource that the next one. Examples of such levels of storage in current architecture include Cache, RAM, NVRAM, Buffers, Disks, Tape [ORNL [n.d.]].

## 2.3 Optimization problem

The core of the Adjoint Computation (AC) problem is the following: after the execution of a forward step, its output is kept in the top buffer only. If it is not saved in one level of storage before the next forward step, it is lost and will have to be recomputed when needed for the corresponding backward step. A trade-off must be found between the cost of storage accesses and that of recomputations.

In accordance to the scheduling literature, we use the term *makespan* for the total execution time. A schedule that executes an AC chain is a sequence of operations from Table 1 and respects the input constraints. It is *Optimal* when it minimizes the makespan, meaning that there is no other schedule with a strictly smaller makespan.

We assume that at the beginning of the execution, each level of storage is empty, while the top buffer contains the data $x_0$, and the bottom buffer contains the data $\bar{x}_{l+1}$. The general hierarchical AC problem is named $\textsc{HierProb}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$ and can be described as in Problem 1.

**Problem 1** ($\textsc{HierProb}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$)**.** We want to minimize the makespan of the AC problem with the following parameters:

[2] Note that this value may only be known only after the forward integration.

| | Operation | Input | Action | Cost |
|---|---|---|---|---|
| $F_i$ | Executes one forward step $F_i$ (for $i \in \{0, \ldots, l-1\}$). This operation takes a time cost$(F_i) = u_f$. | $\mathcal{B}^\top = \{x_i\}$ | $\mathcal{B}^\top \leftarrow \{x_{i+1}\}$ | $u_f$ |
| $F_{i \to i'}$ | Denotes the sequence of operations $F_i; F_{i+1}; \ldots; F_{i'}$. | $\mathcal{B}^\top = \{x_i\}$ | $\mathcal{B}^\top \leftarrow \{x_{i'}\}$ | $(i'-i+1)u_f$ |
| $\bar{F}_i$ | Executes the backward step $\bar{F}_i$ ($i \in \{0, \ldots, l\}$). This operation takes a time cost$(\bar{F}_i) = u_b$. | $\mathcal{B}^\top = \{x_i\}$, $\mathcal{B}^\perp = \{\bar{x}_{i+1}\}$ | $\mathcal{B}^\perp \leftarrow \{\bar{x}_i\}$ | $u_b$ |
| $W_i^k$ | Writes the value $x_i$ of the top buffer into the level $k$ of storage. This operation takes a time cost$(W_i^k) = \mathbf{w}_k$. | $\mathcal{B}^\top = \{x_i\}$ | $\mathcal{M}_k \leftarrow \mathcal{M}_k \cup \{x_i\}$ | $\mathbf{w}_k$ |
| $R_i^k$ | Reads the value $x_i$ from the level $k$ of storage, and puts it into the top buffer. This operation takes a time cost$(R_i^k) = \mathbf{r}_k$. | $x_i \in \mathcal{M}_k$ | $\mathcal{B}^\top \leftarrow \{x_i\}$ | $\mathbf{r}_k$ |
| $D_i^k$ | Discard the value $x_i$ from the level $k$ of storage. This operation is considered free: it can be done by overwriting another data on top of it. | $x_i \in \mathcal{M}_k$ | $\mathcal{M}_k \leftarrow \mathcal{M}_k \backslash \{x_i\}$ | 0 |

Table 1. Operations performed by a schedule.

| | | Initial state: |
|---|---|---|
| AC chain: | size $l$ | |
| Steps Costs: | $u_f, u_b$ | |
| Storage of level $\forall k \in \{1, ..., K\}$: | $\mathbf{c}_k, \mathbf{w}_k, \mathbf{r}_k$ | $\mathcal{M}_k = \emptyset$ |
| Buffers: | $\mathcal{B}^\top, \mathcal{B}^\perp$ | $\mathcal{B}^\top = \{x_0\}$, $\mathcal{B}^\perp = \{\bar{x}_{l+1}\}$ |

### 2.4 Variants of the hierarchical AC problem, and literature survey

There already exist many studies on different versions of the hierarchical storage problem. We present here the two main versions and several works related to them.

*2.4.1 $K = 1$, $\boldsymbol{w}_1 = \boldsymbol{r}_1 = 0$; Revolve.* This is the most known variant of the problem and has been widely studied. In this variant of the problem, the platform considered has a single level of storage ($K = 1$) and negligible access costs ($\mathbf{w}_1 = \mathbf{r}_1 = 0$). This problem has been heavily studied in the community of automatic differentiation. Grimm et al. [Grimm et al. 1996] showed the optimality of a binomial approach. Those theoretical results were later extended and implement by Griewank and Walther [Griewank and Walther 2000] under the name Revolve. Revolve [Griewank and Walther 2000] takes a length $l$ and a number of checkpoints $\mathbf{c}_1$ and returns an optimal solution sequence to MemoryProb$(l, \mathbf{c}_1)$ (Problem 2 below). We define as Time-Revolve$(l, \mathbf{c}_1)$, the makespan of an optimal solution to MemoryProb$(l, \mathbf{c}_1)$.

**Problem 2** (MemoryProb$(l, \mathbf{c}_1)$)**.** We want to minimize the makespan of the AC problem with the following parameters:

| | | Initial state: |
|---:|---|---|
| AC chain: | size $l$ | |
| Steps: | $u_f, u_b$ | |
| Storage level 1: | $\mathbf{c}_1, \mathbf{w}_1 = \mathbf{r}_1 = 0$ | $\mathcal{M}_1 = \emptyset$ |
| Buffers: | $\mathcal{B}^\top, \mathcal{B}^\perp$ | $\mathcal{B}^\top = \{x_0\}, \mathcal{B}^\perp = \{\bar{x}_{l+1}\}$ |

**Definition 3** (Time-Revolve$(l, \mathbf{c}_1)$).
Given $l \in \mathbb{N}$, and $\mathbf{c}_1 \in \mathbb{N}$, Time-Revolve$(l, \mathbf{c}_1)$ is the execution time of an optimal solution to MemoryProb$(l, \mathbf{c}_1)$.

We then give some results known for Time-Revolve$(l, \mathbf{c}_1)$ and needed for this work. Theorem 1 gives the explicit form for Time-Revolve$(l, \mathbf{c}_1)$.

**Definition 4** ($\beta$). Define $\beta$ the function

$$\beta : x, y \mapsto \binom{x+y}{x} \tag{3}$$

**Theorem 1** ([Grimm et al. 1996, Theorem 2]). *Let $l \in \mathbb{N}$ and $\mathbf{c}_1 \in \mathbb{N}$. The explicit form for Time-Revolve$(l, \mathbf{c}_1)$ is:*

$$\textit{Time-Revolve}(l, \mathbf{c}_1) = (l+1) \cdot (t+1)u_f - \beta(\mathbf{c}_1 + 1, t)u_f + (l+1)u_b,$$

*where $t$ is the unique integer satisfying $\beta(\mathbf{c}_1, t) \leq l < \beta(\mathbf{c}_1, t+1)$.*

*2.4.2 $K = 2$, $\mathbf{w}_1 = \mathbf{r}_1 = 0$, $\mathbf{c}_2 = \infty$; Disk-Revolve.* This variant of the problem has received increasingly attention in the recent years with the introduction of a second level of storage of infinite capacity but with access (write and read) costs [Aupy and Herrmann 2017; Aupy et al. 2016; Pringle et al. 2016; Schanen et al. 2016; Stumm and Walther 2009]. Indeed, with the increase in the sizes of the problem, the memory was not sufficient anymore to solve the problems in a reasonnable time. Hence solutions have started considering the usage of disks to store some of the intermediary data.

We consider that there are two levels of storage ($K = 2$): a first level with limited space ($\mathbf{c}_1$ is finite) with negligible access costs ($\mathbf{w}_1 = \mathbf{r}_1 = 0$), and a second level with unlimited space ($\mathbf{c}_2 = \infty$) with a writing and reading cost that should be taken into account.

This model can be used to represent an architecture with a two-level storage system where the processor can write in a close cheap-to-access memory and can access a far storage, such as a disk, if needed. The problem consisting of minimizing the makespan of the Adjoint Computation problem on such an architecture was defined as DiskProb$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$ in our previous work [Aupy et al. 2016].

**Problem 3** (DiskProb$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$). We want to minimize the makespan of the AC problem with the following parameters:

| | | Initial state: |
|---:|---|---|
| AC graph: | size $l$ | |
| Steps: | $u_f, u_b$ | |
| Storage level 1: | $\mathbf{c}_1, \mathbf{w}_1 = \mathbf{r}_1 = 0$ | $\mathcal{M}_1 = \emptyset$ |
| Storage level 2: | $\mathbf{c}_2 = \infty, \mathbf{w}_2, \mathbf{r}_2$ | $\mathcal{M}_2 = \emptyset$ |
| Buffers: | $\mathcal{B}^\top, \mathcal{B}^\perp$ | $\mathcal{B}^\top = \{x_0\}, \mathcal{B}^\perp = \{\bar{x}_{l+1}\}$ |

**Definition 5** (TimeDiskRev$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$).
Given $l \in \mathbb{N}$, $\mathbf{c}_1 \in \mathbb{N}$, $\mathbf{w}_2 \in \mathbb{R}^+$ and $\mathbf{r}_2 \in \mathbb{R}^+$, TimeDiskRev$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$ is the execution time of an optimal solution to DiskProb$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$.

Several work have considered this problem. Stumm and Walther [Stumm and Walther 2009] have provided a first heuristic that uses the schedule provided by Revolve, where the checkpoints the least used are executed on disk (level 2 storage). Some implementations such as the one provided by Pringle et al. [Pringle et al. 2016] are based on a two level checkpointing strategy: the first pass (forward mode) of the adjoint graph checkpoints periodicaly to disk (level 2), then the second pass (reverse mode) reads those disk checkpoints one after the other and uses Revolve with only memory (level 1) checkpoints. The main parameter (period used for the forward checkpointing) can be chosen by the user. We discuss in greater details this algorithm in Section 3.2, in addition we prove a closed-form formula for the period for this algorithm that minimizes asymptotically the makespan based on system parameters. This implementation has the advantage of being fairly easy to implement with the current implementation of adjoint computations.

Aupy et al. [Aupy et al. 2016] designed an algorithm, denoted by Disk-Revolve, to solve DiskProb$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$ optimally. This algorithm uses a sub-routine, 1D-Revolve, that uses only on checkpoint slot in the second level of storage. Because details of this algorithm are unimportant for this paper and would complicate the reading process, we refer the interested reader to our previous work (Definition 3.13, [Aupy et al. 2016]) for more details.

In a subsequent work, Aupy and Herrmann [Aupy and Herrmann 2017] showed that the optimal solution returned by Disk-Revolve is weakly periodic, meaning that the number of forward computations performed between two consecutive checkpoints into the second level of storage is always the same except for a bounded number of them. This period (defined as $m_X$) only depends on $\mathbf{c}_1$, $\mathbf{w}_2$, and $\mathbf{r}_2$, and does not depend on the size of the AC graph to reverse. The authors also provided a conjecture for the close formula of $m_X$ (Conjecture 1). In addition, they showed that a solution that checkpoints periodically data to disks (level 2) is asymptotically optimal, both in the offline case (the number of steps is known before-hand), and in the online case (the number of steps is not known before-hand). These results support the search by Pringle et al. [Pringle et al. 2016] of a periodic algorithm.

**Conjecture 1.** *Consider a platform with two levels of storage (as defined in DiskProb$_\infty(l, \boldsymbol{c}_1, \boldsymbol{w}_2, \boldsymbol{r}_2)$). We define $f$ a function that takes three integers $x$, $y$ and $c$ as follow:*

$$f : (x, y, c) \mapsto \beta(c + 1, x + y - 1) - \beta(c + 1, y - 1).$$

*Let $i_r(\boldsymbol{r}_2, \boldsymbol{c}_1) = y$ or $y + 1$, where $y$ is the only integer that satisfies*

$$\beta(\boldsymbol{c}_1 + 1, y - 1) \leq \boldsymbol{r}_2 < \beta(\boldsymbol{c}_1 + 1, y).$$

*Let $i_w(\boldsymbol{w}_2, \boldsymbol{r}_2, \boldsymbol{c}_1) = x$, where $x$ is the only integer that satisfies*

$$\sum_{j=1}^{x-1} f(j, i_r(\boldsymbol{r}_2, \boldsymbol{c}_1), \boldsymbol{c}_1) < \boldsymbol{w}_2 \leq \sum_{j=1}^{x} f(j, i_r(\boldsymbol{r}_2, \boldsymbol{c}_1), \boldsymbol{c}_1).$$

*We conjecture that*

$$m_X = f(i_w(\boldsymbol{w}_2, \boldsymbol{r}_2, \boldsymbol{c}_1), i_r(\boldsymbol{r}_2, \boldsymbol{c}_1), \boldsymbol{c}_1).$$

### 2.5 Asynchronous Hierarchical AC problem

In this work we make the implicit asumption that data movement are synchronous. Specifically the application is either computing, or moving data but cannot do both at the same time

Recent studies [Kukreja et al. 2018a; Schanen et al. 2016] have also considered asynchronous data-movement. Schanen et al. [Schanen et al. 2016] study a variant of the two storage level problem where the bottom level (memory) is limited by size but with an infinite bandwidth, and the top level (disks or tape) is limited by latency and bandwidth. They can however write and read their checkpoints to disks asynchronously while still doing forward and backward steps. This means that in the schedule, writes and reads to external storage could be done at almost no-cost assuming they are not done too often.

Kukreja et al. [Kukreja et al. 2018a] consider the variant of the one storage level with a bounded number of checkpoints, where the write and read costs are non-zero. In this case, they show that it is not always optimal to use all available memory slots as they incur a cost, and that instead one may prefer to read and write asynchronously.

We do not consider this type of storage movement in our work.

## 3 MINIMIZING THE EXECUTION TIME FOR SYNCHRONOUS HIERARCHICAL STORAGE

In this section we present the general optimal strategy to execute an Adjoint Computation graph on a hierarchical storage system, H-Revolve (Section 3.1). We do not prove *per se* the optimality of the solution, but describe it and hint why one can expect it to be the optimal solution. Finally, we study the algorithm for $\textsc{DiskProb}_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$ by Pringle et al. [Pringle et al. 2016] used, and show how one can obtain the optimal size for the period between disk checkpoints in Section 3.2.

### 3.1 H-Revolve

To solve the adjoint problem, Grimm et al. [Grimm et al. 1996] developed a dynamic-programming based strategy for systems with one level of storage. Later, Aupy et al. [Aupy et al. 2016] improved the computation model so that this type of strategy can be extended to be optimal for two-levels of storage including one of unbounded size but with access cost (read/write).

*3.1.1 Intuition for the optimality of H-Revolve.* The optimality of these algorithms relies on two key properties:

- *Checkpoint persistence* is the idea that if the output of a forward operation is written to storage, then as long as the associated backward operation is not done, one will not read from any checkpoint written before this checkpoint. In addition, at that time, the checkpoint is discarded. Put differently, the checkpoints can be stacked and read in a LIFO order. The checkpoint at the top of the stack can be read multiple time, and/or moved between storage levels.
- *Checkpoint hierarchy* is the idea that on our stack of checkpointed data, the checkpoints are sorted per location. Specifically, the checkpoints that are the furthest away from the top of the stack are stored on the storage location the furthest away.

Intuitively those two properties stay true in a storage system with more than two level of storage when we have the property that reads and writes costs are increasing function of the level of storage. Specifically, for a platform with $K$ levels of storage, if we denote by vectors $\mathbf{r} \in \mathbb{R}^{+K}$ and $\mathbf{w} \in \mathbb{R}^{+K}$ the reading and writing costs for every level of storage, then we have the $\mathbf{w}_k \leq \mathbf{w}_{k+1}$ and $\mathbf{r}_k \leq \mathbf{r}_{k+1}$. It is based on these results that we can now develop our strategy for the general hierarchical AC problem as described in Definition 2.

In the following, let $K$ be the number of levels of storage, let $\mathbf{c} \in \mathbb{N}^K$ be their respecive size. Let $\mathbf{r} \in \mathbb{R}^{+K}$ and $\mathbf{w} \in \mathbb{R}^{+K}$ represent their respective reading and writing costs.

As defined in Section 2.3, we refer to the problem consisting of finding an optimal execution of a AC chain on such an architecture as $\textsc{HierProb}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$ (Problem 1). As described in Definition 6, we denote with $\text{HierTime}_K(\text{l}, \mathbf{c}, \mathbf{w}, \mathbf{r})$ the execution time of an optimal solution to $\textsc{HierProb}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$.

**Definition 6** (HierTime$_K$(l, $\mathbf{c}$, $\mathbf{w}$, $\mathbf{r}$)).
Given $l \in \mathbb{N}$, $\mathbf{c} \in \mathbb{N}^K$, $\mathbf{r} \in \mathbb{R}^{+K}$, and $\mathbf{w} \in \mathbb{R}^{+K}$, HierTime$_K$(l, $\mathbf{c}$, $\mathbf{w}$, $\mathbf{r}$) is the execution time of an optimal solution to $\textsc{HierProb}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$.

Note that in the following of the section we use the term *optimal* eventhough the proof is not formally written. We leave it out for readability reason, but it would be essentially similar to the proof for two levels of storage [Aupy et al. 2016].

*3.1.2 H-Revolve algorithm.* In this section, we provide a dynamic program to compute the value of HierTime$_K$(l, $\mathbf{c}$, $\mathbf{w}$, $\mathbf{r}$). Using the same reasoning as in our previous work, we have to define the sub-problem $\overline{\textsc{HierProb}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})}$ where the data $x_0$ is initially stored in the top buffer and the $K$-th level of storage. We define $\overline{\text{HierTime}_K}$(l, $\mathbf{c}$, $\mathbf{w}$, $\mathbf{r}$) the execution time of an optimal solution to $\overline{\textsc{HierProb}}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$.

**Problem 4** ($\overline{\textsc{HierProb}}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$). We want to minimize the makespan of the AC problem with the following parameters:

| | | Initial state: |
|---:|---|---|
| AC chain: | size $l$ | |
| Steps: | $u_f, u_b$ | |
| Storage level $\forall k \in \{1, ..., K-1\}$: | $\mathbf{c}_k$, $\mathbf{w}_k$, $\mathbf{r}_k$ | $\mathcal{M}_k = \emptyset$ |
| Storage level K: | $\mathbf{c}_K$, $\mathbf{w}_K$, $\mathbf{r}_K$ | $\mathcal{M}_K = \{x_0\}$ |
| Buffers: | $\mathcal{B}^\top$, $\mathcal{B}^\perp$ | $\mathcal{B}^\top = \{x_0\}$, $\mathcal{B}^\perp = \{\bar{x}_{l+1}\}$ |

**Definition 7** ($\overline{\text{HierTime}}_K$(l, $\mathbf{c}$, $\mathbf{w}$, $\mathbf{r}$)).
Given $l \in \mathbb{N}$, $\mathbf{c} \in \mathbb{N}^K$, $\mathbf{r} \in \mathbb{R}^{+K}$, and $\mathbf{w} \in \mathbb{R}^{+K}$, $\overline{\text{HierTime}}_K$(l, $\mathbf{c}$, $\mathbf{w}$, $\mathbf{r}$) is the execution time of an optimal solution to $\overline{\textsc{HierProb}}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$.

Theorem 2 gives a dynamic program to compute the value of HierTime$_K$(l, $\mathbf{c}$, $\mathbf{w}$, $\mathbf{r}$). Based on this dynamic program, we can design a polynomial algorithm H-Revolve that, given the value $l$, the vectors $\mathbf{c}$, $\mathbf{w}$, and $\mathbf{r}$ returns H-Revolve$(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$ an optimal sequence for $\textsc{HierProb}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r})$. The proofs that the dynamic program and the polynomial algorithm are correct strongly resemble the ones for Disk-Revolve and TimeDiskRev$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$ from our previous work. Thus, we won't detail them here. In the rest of this section, for a given vector $\mathbf{c}$ of dimension $K$, we define $\mathbf{c}^-$, the vector of dimension $K-1$ such that $\forall k \in \{1, ..., K-1\}, \mathbf{c}_k = \mathbf{c}^-_k$. Let also define for every $K$, the vector $\mathbf{e}_K \in \mathbb{N}^K$, such that every element is equal to zero except the last one that is equal to 1.

**Theorem 2.** *(HierTime$_K$(l, $\mathbf{c}$, $\mathbf{w}$, $\mathbf{r}$)) Let $l \in \mathbb{N}$, $\mathbf{c} \in \mathbb{N}^K$, $\mathbf{r} \in \mathbb{R}^{+K}$, and $\mathbf{w} \in \mathbb{R}^{+K}$, such that $\mathbf{c}_K \neq 0$, and $\forall k \in \{1, ..., K-1\}$, $\mathbf{w}_k \leq \mathbf{w}_{k+1}$ and $\mathbf{r}_k \leq \mathbf{r}_{k+1}$.*
*Then, for $l = 0$:*

$$\overline{HierTime}_K(0,\ \mathbf{c},\ \mathbf{w},\ \mathbf{r}) = u_b$$
$$HierTime_K(0,\ \mathbf{c},\ \mathbf{w},\ \mathbf{r}) = u_b$$

*For $l > 0$, $K = 1$, and $\mathbf{c}_1 = 1$:*

$$\overline{HierTime}_1(l,\ (1),\ (\mathbf{w}_1),\ (\mathbf{r}_1)) = l\mathbf{r}_1 + \frac{l(l+1)}{2}u_f + (l+1)u_b$$
$$HierTime_1(l,\ (1),\ (\mathbf{w}_1),\ (\mathbf{r}_1)) = \mathbf{w}_1 + \overline{HierTime}_1(l,\ (1),\ (\mathbf{w}_1),\ (\mathbf{r}_1))$$

*For $l > 0$, $K = 1$, and $\mathbf{c}_1 > 1$:*

$$\overline{HierTime}_1(l, (\mathbf{c}_1), (\mathbf{w}_1), (\mathbf{r}_1)) = \min \begin{cases} \overline{HierTime}_1(l, (1), (\mathbf{w}_1), (\mathbf{r}_1)) \\ \min_{1 \le j \le l-1}\{ju_f + HierTime_1(l - j, (\mathbf{c}_1 - 1), (\mathbf{w}_1), (\mathbf{r}_1)) \\ \qquad\qquad + \mathbf{r}_1 + \overline{HierTime}_1(j - 1, (\mathbf{c}_1), (\mathbf{w}_1), (\mathbf{r}_1))\} \end{cases}$$

$$HierTime_1(l, (\mathbf{c}_1), (\mathbf{w}_1), (\mathbf{r}_1)) = \mathbf{w}_1 + \overline{HierTime}_1(l, (\mathbf{c}_1), (\mathbf{w}_1), (\mathbf{r}_1))$$

*For $l > 0$, $K > 1$:*

$$\overline{HierTime}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r}) = \min \begin{cases} HierTime_{K-1}(l, \mathbf{c}^-, \mathbf{w}^-, \mathbf{r}^-) \\ \min_{1 \le j \le l-1}\{ju_f + HierTime_K(l - j, \mathbf{c} - \mathbf{e}_K, \mathbf{w}, \mathbf{r}) \\ \qquad\qquad + \mathbf{r}_K + \overline{HierTime}_K(j - 1, \mathbf{c}, \mathbf{w}, \mathbf{r})\} \end{cases}$$

$$HierTime_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r}) = \min \begin{cases} HierTime_{K-1}(l, \mathbf{c}^-, \mathbf{w}^-, \mathbf{r}^-) \\ \mathbf{w}_K + \overline{HierTime}_K(l, \mathbf{c}, \mathbf{w}, \mathbf{r}) \end{cases}$$

**Algorithm 1** $\overline{\text{H-Revolve}}$

```
procedure $\overline{\text{H-Revolve}}(K, l, \mathbf{c}, \mathbf{w}, \mathbf{r})$
    $\mathcal{S} \leftarrow \emptyset$
    if $\overline{\text{HierTime}}_K(\text{l}, \mathbf{c}, \mathbf{w}, \mathbf{r}) = \text{HierTime}_{K-1}(\text{l}, \mathbf{c}^-, \mathbf{w}^-, \mathbf{r}^-)$ then
        $\mathcal{S} \leftarrow \text{H-Revolve}(K - 1, l, \mathbf{c}^-, \mathbf{w}^-, \mathbf{r}^-)$
    else
        Let $j$ such that
    $\overline{\text{HierTime}}_K(\text{l}, \mathbf{c}, \mathbf{w}, \mathbf{r}) = \min_{1 \le j \le l-1} \{ju_f + \text{HierTime}_K(l - j, \mathbf{c} - \mathbf{e}_K, \mathbf{w}, \mathbf{r}) + \mathbf{r}_K + \overline{\text{HierTime}}_K(j - 1, \mathbf{c}, \mathbf{w}, \mathbf{r})\}$
        $\mathcal{S} \leftarrow F_{0 \to (j-1)}$
        $\mathcal{S} \leftarrow \mathcal{S} \cdot \text{Shift}\ (\text{H-Revolve}(K, l - j, \mathbf{c} - \mathbf{e}_K, \mathbf{w}, \mathbf{r}), j)$
        $\mathcal{S} \leftarrow \mathcal{S} \cdot R_0^K \cdot \overline{\text{H-Revolve}}\ (K, j - 1, \mathbf{c}, \mathbf{w}, \mathbf{r})$
    end if
    return $\mathcal{S}$
end procedure
```

**Algorithm 2** H-Revolve

```
procedure H-Revolve$(K, l, \mathbf{c}, \mathbf{w}, \mathbf{r})$
    $\mathcal{S} \leftarrow \emptyset$
    if $\text{HierTime}_K(\text{l}, \mathbf{c}, \mathbf{w}, \mathbf{r}) = \text{HierTime}_{K-1}(\text{l}, \mathbf{c}^-, \mathbf{w}^-, \mathbf{r}^-)$ then
        $\mathcal{S} \leftarrow \text{H-Revolve}(K - 1, l, \mathbf{c}^-, \mathbf{w}^-, \mathbf{r}^-)$
    else
        $\mathcal{S} \leftarrow W_0^K \cdot \overline{\text{H-Revolve}}\ (K, l, \mathbf{c}, \mathbf{w}, \mathbf{r})$
    end if
    return $\mathcal{S}$
end procedure
```

### 3.2 Per-Rev-Revolve

For the specific case of two-level storage systems, we have seen that several theoretical solutions exist. Those can often be complicated to implement. There exist already some non-optimal implementation that rely on simple properties. Specifically in this section we discuss the *Divided Adjoint* feature developed in OpenAD by Pringle et al. [Pringle et al. 2016].

This feature (called Per-Rev-Revolve in this work) is a two-level checkpointing approach similar to the periodic algorithms developed in Aupy and Herrmann [Aupy and Herrmann 2017]:

- During the forward phase, output data is checkpointed periodically (following a period $m$) to the outter level of storage (*Disks*).
- Then, as a second phase, within each of these blocks, the binomial strategy Revolve is used.

Pringle et al. [Pringle et al. 2016] leave the work of choosing the outter period size to the user or system administrator. Similarly to our previous work where we studied the periodic version of Disk-Revolve [Aupy and Herrmann 2017], one can study the value of the optimal period $m_X^{rr}$ that minimizes asymptotically (when the size of the adjoint graph is large) the execution time of Per-Rev-Revolve as a function of the system parameters, $c$, $w_d$ and $r_d$.

**Theorem 3.** *Denote $m_X^{rr}$ the period that minimizes asymptotically the execution time of Per-Rev-Revolve, $m_X^{rr}$ is the solution of the following equation:*

$$\min_m \frac{w_d + r_d + \textit{Time-Revolve}(m-1, c)}{m}. \tag{4}$$

The proof of this result is very similar to the one used for Disk-Revolve [Aupy and Herrmann 2017].

Finally, we can compute a closed-form formula for the value of $m_X^{rr}$:

**Theorem 4.** *The value $m_X^{rr}$ that minimizes Equation 4 is:*

$$m_X^{rr} = \beta(c, t_{w_d,r_d}), \tag{5}$$

*where $t_{w_d,r_d}$ is the only integer such that:*

$$\beta(c+1, t-1) \le w_d + r_d < \beta(c+1, t)$$

Proof. Using the value for Time-Revolve (Theorem 1) we can rewrite Eq. (4):

$$\begin{aligned}\min_m & \frac{w_d + r_d + \text{Time-Revolve}(m-1, c)}{m} \\ &= \min_t \min_{\beta(c,t) \le m-1 < \beta(c,t+1)} \frac{w_d + r_d + (t+1)m - \beta(c+1,t)}{m}.\end{aligned}$$

We first focus on the second minimization problem:

$$\min_{\beta(c,t) \le m-1 < \beta(c,t+1)} t + 1 + \frac{w_d + r_d - \beta(c+1,t)}{m}. \tag{6}$$

Denote by $t_{w_d,r_d}$, the only $t$ that satisfies: $\beta(c+1, t-1) \le w_d + r_d < \beta(c+1, t)$.

*Assume $t \le t_{w_d,r_d} - 1$.* In this case, $\beta(c+1,t) \le w_d + r_d$ and Eq. (6) is minimized when $m$ is maximal, that is $m = \beta(c, t+1)$.

Hence we can reduce the problem to

$$\min_t t + 1 + \frac{w_d + r_d - \beta(c+1,t)}{\beta(c,t+1)}.$$

In order to study this problem, let define $f : t \mapsto (t+1) + \frac{w_d + r_d - \beta(c+1,t)}{\beta(c,t+1)}$. We show that $f$ is decreasing for $t \leq t_{w_d,r_d} - 1$, by studying: $f(t) - f(t-1) \leq 0$.

$$\begin{aligned}
&t + 1 + \frac{w_d + r_d - \beta(c+1,t)}{\beta(c,t+1)} - t - \frac{w_d + r_d - \beta(c+1,t-1)}{\beta(c,t)} && \leq 0 \\
&\beta(c,t)\beta(c,t+1) + (w_d + r_d)(\beta(c,t) - \beta(c,t+1)) - \beta(c,t)\beta(c+1,t) + \beta(c+1,t-1)\beta(c,t+1) && \leq 0 \\
&-(w_d + r_d)\beta(c-1,t+1) + \beta(c,t)(\beta(c,t+1) - \beta(c+1,t)) + \beta(c+1,t-1)\beta(c,t+1) && \leq 0 \\
&-(w_d + r_d)\frac{(c+t)!}{(c-1)!(t+1)!} + \frac{(c+t)!}{c!t!}\left(\frac{(c+t+1)!}{c!(t+1)!} - \frac{(c+t+1)!}{(c+1)!t!}\right) + \frac{(c+t)!}{(c+1)!(t-1)!}\frac{(c+t+1)!}{c!(t+1)!} && \leq 0 \\
&-(w_d + r_d)c + \frac{(c+t+1)!}{(c+1)!t!}(c-t) + \frac{(c+t+1)!}{(c+1)!(t-1)!} && \leq 0 \\
&-(w_d + r_d) + \frac{(c+t+1)!}{(c+1)!t!} && \leq 0 \\
&\beta(c+1,t) - (w_d + r_d) && \leq 0
\end{aligned}$$

Hence,

$$f(t) \leq f(t-1) \iff \beta(c+1,t) \leq w_d + r_d.$$

Because $t \leq t_{w_d,r_d} - 1$, we have $\beta(c+1,t) \leq w_d + r_d$, and $f$ is decreasing and minimized for $t_1 = t_{w_d,r_d} - 1$.

Hence, when $t \leq t_{w_d,r_d} - 1$, Equation (6) is minimized for the values $t_1 = t_{w_d,r_d} - 1$ and $m_1 = \beta(c, t_{w_d,r_d})$. Thus,

$$C_1 = t_{w_d,r_d} + \frac{w_d + r_d - \beta(c+1, t_{w_d,r_d} - 1)}{\beta(c, t_{w_d,r_d})}$$

is a possible minimum for the equation.

*Assume* $t \geq t_{w_d,r_d}$. In this case, $\beta(c+1,t) > w_d + r_d$ and Eq. (6) is minimized when $m$ is minimal, that is $m = \beta(c,t) + 1$.

Hence we can reduce the problem to

$$\min_t t + 1 + \frac{w_d + r_d - \beta(c+1,t)}{\beta(c,t) + 1}.$$

In order to study this problem, let define $g : t \mapsto (t+1) + \frac{w_d + r_d - \beta(c+1,t)}{\beta(c,t)+1}$. We show that $g$ is non-decreasing for $t \geq t_{w_d,r_d}$, by studying: $g(t+1) - g(t) > 0$.

$$\begin{aligned}
&t + 2 + \frac{w_d + r_d - \beta(c+1,t+1)}{\beta(c,t+1) + 1} - t - 1 - \frac{w_d + r_d - \beta(c+1,t)}{\beta(c,t) + 1} && > 0 \\
&(\beta(c,t+1) + 1)(\beta(c,t) + 1) + (w_d + r_d)(\beta(c,t) - \beta(c,t+1)) - \beta(c+1,t+1)(\beta(c,t) + 1) \\
&\qquad + \beta(c+1,t)(\beta(c,t+1) + 1) && > 0 \\
&-(w_d + r_d)\beta(c-1,t+1) + \beta(c,t)(\beta(c,t+1) - \beta(c+1,t+1)) + \beta(c+1,t)\beta(c,t+1) \\
&\qquad + \beta(c,t+1) + \beta(c,t) + 1 - \beta(c+1,t+1) + \beta(c+1,t) && > 0 \\
&-(w_d + r_d)\beta(c-1,t+1) - \beta(c,t)\beta(c+1,t) + \beta(c+1,t)\beta(c,t+1) + \beta(c,t) + 1 && > 0 \\
&\beta(c-1,t+1)(\beta(c+1,t) - (w_d + r_d)) + \beta(c,t) + 1 && > 0
\end{aligned}$$

Hence,

$$\beta(c+1,t) \geq (w_d + r_d) \implies g(t+1) > g(t)$$

Because we study it for $t \geq t_{w_d,r_d}$, we have $\beta(c+1,t) \geq w_d + r_d$, and $g$ is non-decreasing and minimized for $t_2 = t_{w_d,r_d}$.

Hence, when $t \geq t_{w_d,r_d}$, Equation (6) is minimized for the values $t_2 = t_{w_d,r_d}$ and $m_2 = \beta(c, t_{w_d,r_d})+1$. Thus

$$C_2 = t_{w_d,r_d} + 1 + \frac{w_d + r_d - \beta(c+1, t_{w_d,r_d})}{\beta(c, t_{w_d,r_d}) + 1}$$

is a possible minimum for the equation.

To conclude, we have seen that the minimum is reach either for $t_1 = t_{w_d,r_d} - 1$ and $m_1 = \beta(c, t_{w_d,r_d})$ or for $t_2 = t_{w_d,r_d}$ and $m_2 = \beta(c, t_{w_d,r_d}) + 1$. We can study Eq. (4) for both these values. We can show that the minimum is reached for $t_1$ and $m_1$, by studying $C_1 - C_2 < 0$.

$$\begin{aligned}
&t_{w_d,r_d} + \frac{w_d + r_d - \beta(c+1, t_{w_d,r_d} - 1)}{\beta(c, t_{w_d,r_d})} - (t_{w_d,r_d} + 1) - \frac{w_d + r_d - \beta(c+1, t_{w_d,r_d})}{\beta(c, t_{w_d,r_d}) + 1} && < 0 \\
&t_{w_d,r_d}\beta(c, t_{w_d,r_d})(\beta(c, t_{w_d,r_d}) + 1) + (\beta(c, t_{w_d,r_d}) + 1)(w_d + r_d - \beta(c+1, t_{w_d,r_d} - 1)) && \\
&\qquad - (t_{w_d,r_d} + 1)\beta(c, t_{w_d,r_d})(\beta(c, t_{w_d,r_d}) - \beta(c, t_{w_d,r_d})(w_d + r_d - \beta(c+1, t_{w_d,r_d})) && < 0 \\
&-\beta(c, t_{w_d,r_d})(\beta(c, t_{w_d,r_d}) + 1) + w_d + r_d - (\beta(c, t_{w_d,r_d}) + 1)\beta(c+1, t_{w_d,r_d} - 1) + \beta(c,t)\beta(c+1,t) && < 0 \\
&w_d + r_d + \beta(c, t_{w_d,r_d})(-1 - \beta(c, t_{w_d,r_d}) - \beta(c+1, t_{w_d,r_d} - 1) + \beta(c+1,t)) - \beta(c+1, t-1) && < 0 \\
&w_d + r_d - \beta(c, t_{w_d,r_d}) - \beta(c+1, t_{w_d,r_d} - 1) && < 0 \\
&w_d + r_d - \beta(c+1, t_{w_d,r_d}) && < 0
\end{aligned}$$

This is true by definition of $t_{w_d,r_d}$, so the minimum is met when $t = t_1$ and $m = m_1$, hence the result. □

We design an algorithm implementing this periodic disk checkpoint strategy, Per-Rev-Revolve (Algorithm 3). In Per-Rev-Revolve, the optimal period $m_X^{rr}$ is computed at line 3 using the close formula of Theorem 4. The complexity of Per-Rev-Revolve is reduced compared to the one of Rev-Revolve. Indeed, we just have to compute the sequences returned by Revolve$(l, \mathbf{c}_1)$ for every $l \leq m_X^{rr}$, and then, the final sequence can be build in linear time.

## 4 UTILIZATION OF THE H-Revolve LIBRARY

This section presents implementation details of the library as well as the different algorithms implemented. We present how we expect it to be used as an algorithm library in an Adjoint Computation software such Devito [Louboutin et al. 2018]. The implementation is in Python and available at https://gitlab.inria.fr/adjoint-computation/H-Revolve.

In this section, we first describe the general framework of the code, the inputs and outputs and how to read them (Section 4.1). We describe the main algorithm H-Revolve in Section 4.2. For the specific problem DiskProb$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$[3] which has been well studied in the literature and in our previous work, we provide an implementation of the optimal solution, as well as several heuristics computing the schedule more efficiently (both compute-wise and memory-wise). These algorithms are presented in Section 4.3.

[3] Two storage location, one with latency and bandwidth limitation, *disks*, the other with space limitation, *memory*

**Algorithm 3** Per-Rev-Revolve

```
procedure Per-Rev-Revolve($l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2$)
    $\mathcal{S} \leftarrow \emptyset$
    $m_X^{rr} \leftarrow$ ComputeMXrr($\mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2$)
    $idx \leftarrow 0$
    while $idx + m_X^{rr} < l$ do
        $\mathcal{S} \leftarrow \mathcal{S} \cdot W_{idx}^2 \cdot F_{idx \rightarrow (idx + m_X^{rr} - 1)}$
        $idx \leftarrow idx + m_X^{rr}$
    end while
    $\mathcal{S} \leftarrow \mathcal{S} \cdot$ Shift (Revolve($l - idx, \mathbf{c}_1$), $idx$)
    while $idx > 0$ do
        $idx \leftarrow idx - m_X^{rr}$
        $\mathcal{S} \leftarrow \mathcal{S} \cdot R_{idx}^2 \cdot$ Shift (Revolve($m_X^{rr}, \mathbf{c}_1$), $idx$)
    end while
    return $\mathcal{S}$
end procedure
```

## 4.1 Framework

To this date (March 2019), the code provides five main libraries that can be used:

- *HRevolve.py* provides an implementation of the H-Revolve algorithm (Section 3.1) to reverse an AC graph on a platform with an arbitrary number of storage levels.
- *Disk-Revolve.py* provides the implementation of the Disk-Revolve algorithm [Aupy et al. 2016] (Section 4.3.1).
- *Periodic-Disk-Revolve.py* provides the periodic variant of Disk-Revolve: the Per-Disk-Revolve algorithm [Aupy and Herrmann 2017]. When specifying the option "--one_read_disk", this file provides the implementation of Per-Rev-Revolve (Section 3.2).
- *1D-Revolve.py* provides the implementation of a subroutine 1D-Revolve used by Disk-Revolve and Per-Disk-Revolve (Aupy and Herrmann [Aupy and Herrmann 2017]). This code can be used when considering the online version of DiskProb$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$ (as briefly discussed in Section 3.2) or when considering compact versions of the schedules.
- *Revolve.py* provides an implementation of the Revolve algorithm [Griewank and Walther 2000].

*4.1.1 Inputs.* All the algorithms described above need as arguments, $l$ the size of the AC graph to reverse, and a description of the parameters of the computational platform. The options used to described the platform depend on the number of storage levels ($K = 1$, $K = 2$, or an arbitrary $K$) and are described below. All the algorithms below have two optional arguments: "--uf" to specify the cost of a forward computation; and "--ub" to specify the cost of a backward computation. A "--concat" option can be used to modify the output format of the returned sequence (more details about this can be found in the README of the code). All these arguments are summarized in Table 2.

*4.1.2 Outputs.* When running the algorithm, the following outputs are provided:

- *Sequence*: the list of operations in the sequence. The meaning of every operation can be found in Table 3 for H-Revolve and in Table 4 for Disk-Revolve, Per-Disk-Revolve, Per-Rev-Revolve, and 1D-Revolve. The "--concat" option can be used to simplify the output (see Section 4.3.1).

| Argument | Description | Default value |
|---|---|---|
| $l$ (int) | Size of the AC graph to be differentiated | *Mandatory Argument* |
| --uf float | Cost of a forward computation: $u_f$ | 1.0 |
| --ub float | Cost of a backward computation: $u_b$ | 1.0 |
| --concat int | Output format for the returned sequence | 0 |

Table 2. Common arguments for every algorithms below

| | Operation |
|---|---|
| F_i | Executes one forward step $F_i$ (for $i \in \{0, \ldots, l-1\}$). |
| F_i->j | Denotes the sequence of operations $F_i; F_{i+1}; \ldots; F_{i'}$. |
| B_i | Executes the backward step $\bar{F}_i$ ($i \in \{0, \ldots, l\}$). |
| W∧k_i | Writes the value $x_i$ of the top buffer into the level $k$ of storage. |
| R∧k_i | Reads the value $x_i$ from the level $k$ of storage, and puts it into the top buffer. |
| D∧k_i | Discard the value $x_i$ from the level $k$ of storage. |

Table 3. Operations performed by a schedule as returned in the output of H-Revolve.

| | Operation |
|---|---|
| F_i | Executes one forward step $F_i$ (for $i \in \{0, \ldots, l-1\}$). |
| F_i->j | Denotes the sequence of operations $F_i; F_{i+1}; \ldots; F_{i'}$. |
| B_i | Executes the backward step $\bar{F}_i$ ($i \in \{0, \ldots, l\}$). |
| WM_i | Writes the value $x_i$ of the top buffer into memory (level 1 of storage). |
| WD_i | Writes the value $x_i$ of the top buffer into disk (level 2 of storage). |
| RM_i | Reads the value $x_i$ from the memory (level 1 of storage), and puts it into the top buffer. |
| RD_i | Reads the value $x_i$ from the disk (level 2 of storage), and puts it into the top buffer. |
| DM_i | Discard the value $x_i$ from the memory (level 1 of storage). |

Table 4. Operations performed by a schedule as returned in the output of Disk-Revolve, Per-Disk-Revolve, Per-Rev-Revolve, and 1D-Revolve.

- *Memory* (only for *Revolve.py*, *Disk-Revolve.py* and *Periodic-Disk-Revolve.py*): the list of all the outputs of forward operations that were stored into the first level of storage, in the order that they were saved to memory.
- *Disk* (only for *Disk-Revolve.py* and *Periodic-Disk-Revolve.py*): the list of all the outputs of forward operations that were stored into the second level of storage, in the order that they were saved to memory.
- *Storage level k* (only for *HRevolve.py*): the list of all the outputs of forward operations that were stored into the $k$-th level of storage, in the order that they were saved.
- *Makespan*: the total makespan of the returned schedule.
- *Compute Time*: the effective computation time (in milliseconds) to run the algorithm.

## 4.2 Implementation of H-Revolve

In this section we consider the general problem with $K$ limited levels of storage, where writing and reading data have (potentially) a cost (see Problem 1). To run the code, one has to specify the size $l$ of the AC graph and a description of the platform. The platform description should be provided in a file with the following format:

- The first line is an integer $K$ representing the number of storage levels.

- The $K$-th next lines are triplets $(\mathbf{c}_i, \mathbf{w}_i, \mathbf{r}_i)$ representing the size, the writing cost and the reading cost of level $i$ of the storage system.

Listing 1 represents a file describing a platform with 3 levels of storage. The first one can store up to one checkpoint at the same time and has negligible writing and reading costs. The second level has 2 slots of storage, while writing and reading a checkpoint from it have a cost 2. The last level has 10 slots of storage, while writing and reading a checkpoint from it have a cost 3.

```
#Content of platform_example.txt
3
1 0 0
2 2 2
10 3 3
```

Listing 1. Description of an architecture with 3 levels of storage

The file *HRevolve.py* provides an implementation of the H-Revolve algorithm (Section 3.1). It can be run with the following command:

./HRevolve.py l file_describing_platform [options in Table 2]

Listing 2 provides an example with $l = 20$ on the platform described in Listing 1. The meaning of every operation in the output sequence can be found in Table 3. We can see that the third level of storage is used only once, to store the data $x_0$ at the beginning. $x_0$ is read twice, and the second time it is immediately transferred to the first level of storage.

```
$> ./HRevolve.py 20 platform_example.txt
Sequence:  [W^2_0, F_0->6, W^1_7, F_7->11, W^1_12, F_12->16,
W^0_17, F_17->19, B_20, R^0_17, F_17->18, B_19, R^0_17, F_17,
B_18, R^0_17, B_17, D^0_17, R^1_12, F_12->13, W^0_14,
F_14->15, B_16, R^0_14, F_14, B_15, R^0_14, B_14, D^0_14,
R^1_12, W^0_12, F_12, B_13, R^0_12, B_12, D^0_12, R^1_7,
F_7->8, W^0_9, F_9->10, B_11, R^0_9, F_9, B_10, R^0_9, B_9,
D^0_9, R^1_7, W^0_7, F_7, B_8, R^0_7, B_7, D^0_7, R^2_0,
F_0->2, W^0_3, F_3->5, B_6, R^0_3, F_3->4, B_5, R^0_3, F_3,
B_4, R^0_3, B_3, D^0_3, R^2_0, W^0_0, F_0->1, B_2, R^0_0,
F_0, B_1, R^0_0, B_0, D^0_0]
Storage level 1 :  [17, 14, 12, 9, 7, 3, 0]
Storage level 2 :  [7, 12]
Storage level 3 :  [0]
Makespan:  89
Compute time: 1.417 ms
```

Listing 2. The output of H-Revolve

In conclusion, H-Revolve returns the optimal sequence to reverse an AC graph on an arbitrary hierarchical platform. It takes into account data transfers from one level to another when needed. The output of the code is easy to understand and we strongly believe that it can be integrated in many automatic differentiation applications.

### 4.3 Algorithms for DiskProb$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$

In this section, the algorithms reverse an AC graph on a platform with two levels of storage ($K = 2$): a first level with limited space ($\mathbf{c}_1$ is finite) with negligible access costs ($\mathbf{w}_1 = \mathbf{r}_1 = 0$), and a second level with unlimited space ($\mathbf{c}_2 = \infty$), also called *disk*, with a writing and reading cost that should be taken into account (see Problem 3). To run the code, one has to specify the size $l$ of the AC graph,

| Argument | Description | Default value |
|---|---|---|
| $\mathbf{c}_1$ (int) | Number of slots in the first level of storage | *Mandatory Argument* |
| --wd float | Cost to write a checkpoint in the disk: $\mathbf{w}_2$ | 5.0 |
| --rd float | Cost to read a checkpoint in the disk: $\mathbf{r}_2$ | 5.0 |

Table 5. Common arguments for every algorithms dealing with a two-levels storage platform (Disk-Revolve, Per-Disk-Revolve, Rev-Revolve, and Per-Rev-Revolve)

the number of available storage slots $\mathbf{c}_1$ in the first level of storage, and the writing and reading cost, $\mathbf{w}_2$ and $\mathbf{r}_2$, for the disk. Table 5 displays the arguments that can be given to the algorithms presented in this section (on top of the ones displayed in Table 2).

*4.3.1 Disk-Revolve.* The file *Disk-Revolve.py* provides an implementation of the Disk-Revolve algorithm (Section 2.4). It can be run with the following command:

./Disk-Revolve.py l $\mathbf{c}_1$ [options in Table 2 or 5]

Listing 3 provides an example with $l = 10$, $\mathbf{c}_1 = 2$, $\mathbf{w}_2 = 2$, and $\mathbf{r}_2 = 1$. This means that in this example, it is twice as costly to write a checkpoint into the disk than to perform a forward computation, while reading back the checkpoint will be as costly as a forward computation. The meaning of every operation in the output sequence can be found in Table 4. In this example, we can see that the disk (level 2 of storage) is used only once: to store the data $x_0$ at the beginning. The next time $x_0$ is read from the disk, it is immediately written in the memory (level 1 of storage) and read from here when needed later.

```
$> ./Disk-Revolve.py 10 2 --wd 2 --rd 1 --ub 0
Sequence: [WD_0, F_0->4, WM_5, F_5->7, WM_8, F_8->9, B_10,
RM_8, F_8, B_9, RM_8, B_8, DM_8, RM_5, F_5, WM_6, F_6, B_7,
RM_6, B_6, DM_6, RM_5, B_5, DM_5, RD_0, WM_0, F_0->2, WM_3,
F_3, B_4, RM_3, B_3, DM_3, RM_0, F_0, WM_1, F_1, B_2, RM_1,
B_1, DM_1, RM_0, B_0, DM_0]
Memory: [5, 8, 6, 0, 3, 1]
Disk: [0]
Makespan: 22
Compute time: 0.298 ms
```

Listing 3. The output of Disk-Revolve(10,2,2,1)

*Extensions: compact schedules, "--concat [int]".* For the algorithms around $\text{DiskProb}_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2)$, we propose an option to provide compact schedules: the "--concat [int]" option.

In practice this option factorizes subsets of the output sequence with respect to the Revolve algorithm ("--concat 1"), or as a function of the 1D-Revolve algorithm ("--concat 2"). To obtain the uncompacted version of the subschedules, one can then call *Revolve.py* (Revolve) or *1D-Revolve.py* (1D-Revolve). Note that to return the right schedule, *1D-Revolve.py* only needs the number of memory checkpoints and the cost to read from disk. It accepts the "concat" option. We present an example in Listing 4.

```
$> ./Disk-Revolve.py 100 2 --wd 10 --rd 2 --ub 0 --concat 2
Sequence: [WD_0, F_0->15, WD_16, F_16->31, WD_32, F_32->47, WD_48,
F_48->63, WD_64, F_64->79, WD_80, F_80->90, Revolve(9, 2), RD_80,
1D-Revolve(10, 2), RD_64, 1D-Revolve(15, 2), RD_48, 1D-Revolve(15, 2),
RD_32, 1D-Revolve(15, 2), RD_16, 1D-Revolve(15, 2), RD_0,
1D−Revolve(15, 2)]
(...)

$> ./1D−Revolve.py 15 2 −−rd 2 −−concat 1
Sequence:  [F_0−>5, Revolve(9, 2), RD_0, Revolve(5, 2)])
(...)
```

Listing 4. Output of Disk-Revolve(100,2,10,2) with the option "--concat 2" and 1D-Revolve(100,2,10,2) with the option "--concat 1"

Finally the last option ("--concat 3") is more subtle and relies on theoretical notions of the schedules ([Aupy and Herrmann 2017, Algorithm 2]). In practice, a schedule provided by Disk-Revolve is always of the form: *Forward sweep; Turn; Backward Sweep.* Specifically,

- The *Forward Sweep* is a sequence of Disk writes followed by forward operations.
  E.g. for `./Disk-Revolve.py 100 2 -wd 10 -rd 2`: *"WD_0, F_0->15, WD_16, F_16->31, WD_32, F_32->47, WD_48, F_48->63, WD_64, F_64->79, WD_80, F_80->90"* is the forward sweep
- The *Turn* is a Revolve function.
  E.g. for `./Disk-Revolve.py 100 2 -wd 10 -rd 2`: *"Revolve(9, 2)"* is the turn
- The *Backward Sweep* is a sequence of read disks followed by 1D-Revolve operations.
  E.g. for `./Disk-Revolve.py 100 2 -wd 10 -rd 2`: *"RD_80, 1D-Revolve(10, 2), RD_64, 1D-Revolve(15, 2), RD_48, 1D-Revolve(15, 2), RD_32, 1D-Revolve(15, 2), RD_16, 1D-Revolve(15, 2), RD_0, 1D-Revolve(15, 2)"* is the backward sweep.

Hence to obtain a schedule, one can simply focus on giving the number of steps between consecutive writes to disks in the forward sweep. This is what is provided by the third concat option (see Listing 5).

```
$> ./Disk−Revolve.py 100 2 −−wd 10 −−rd 2 −−ub 0 −−concat 3
Sequence:  (16, 16, 16, 16, 16, 11; 10)
(...)
```

Listing 5. The output of Disk-Revolve(100,2,10,2) with the option "--concat 3"

We call this sequence a period-based solution, where the algorithm only returns the *Periods* of the Disk-Revolve algorithm.

### 4.3.2 Periodic Algorithms.

One can also choose to enforce all the periods to be identical. Then the algorithms are called *periodic*. We have seen in this work and in our previous work that in general there exists an *optimal* period (meaning that it is asymptotically optimal for the class of periodic algorithms).

*Periodic-Disk-Revolve.py* provides an implementation of the two periodic algorithms [Aupy and Herrmann 2017; Pringle et al. 2016] using their respective optimal period. Note that to obtain the implementation of Pringle et al. [Pringle et al. 2016] discussed in Section 3.2, one must use the "--one_read_disk" flag.

In particular, the Per-Disk-Revolve algorithm has been shown by Aupy and Herrmann [Aupy and Herrmann 2017] to be asymptotically optimal both in the offline and online cases for DiskProb$_\infty(l, \mathbf{c}_1, \mathbf{w}_2, \mathbf{r}_2$ and may be a lot easier to implement in real-life framework. To compute the schedule returned by Per-Disk-Revolve, one can run the following command:

./Periodic-Disk-Revolve.py l $\mathbf{c}_1$ [options in Table 2 or 5 or 6]

Listings 6 and 7 provide examples with $l = 10$, $\mathbf{c}_1 = 2$, $\mathbf{w}_2 = 2$, and $\mathbf{r}_2 = 1$ with and without the flag "--one_read_disk" (note that in this specific case, the algorithms return the same outputs).

| Argument | Description | Default value |
|---|---|---|
| --fast | To use the formula of $m_X$ in Conjecture 1 | deactivated |
| --mx int | Personalized value of $m_X$ | the optimal one |

Table 6. Common arguments for every periodic algorithms dealing with a two-levels storage platform (Per-Disk-Revolve and Per-Rev-Revolve)

```
$> ./Periodic-Disk-Revolve.py 10 2 --wd 2 --rd 1 --ub 0
We use periods of size 3
Sequence: [WD_0, F_0->2, WD_3, F_3->5, WD_6, F_6->8, WM_9,
F_9, B_10, RM_9, B_9, DM_9, RD_6, WM_6, F_6, WM_7, F_7, B_8,
RM_7, B_7, DM_7, RM_6, B_6, DM_6, RD_3, WM_3, F_3, WM_4, F_4,
B_5, RM_4, B_4, DM_4, RM_3, B_3, DM_3, RD_0, WM_0, F_0, WM_1,
F_1, B_2, RM_1, B_1, DM_1, RM_0, B_0, DM_0]
Memory: [9, 6, 7, 3, 4, 0, 1]
Disk: [0, 3, 6]
Makespan: 25
Compute time: 0.282 ms
```

Listing 6. The output of Per-Disk-Revolve(10,2,2,1)

```
$> ./Periodic-Disk-Revolve.py 10 2 --wd 2 --rd 1 --ub 0
                                        --one_read_disk
We use periods of size 3
Sequence: [WD_0, F_0->2, WD_3, F_3->5, WD_6, F_6->8, WM_9,
F_9, B_10, RM_9, B_9, DM_9, RD_6, WM_6, F_6, WM_7, F_7, B_8,
RM_7, B_7, DM_7, RM_6, B_6, DM_6, RD_3, WM_3, F_3, WM_4, F_4,
B_5, RM_4, B_4, DM_4, RM_3, B_3, DM_3, RD_0, WM_0, F_0, WM_1,
F_1, B_2, RM_1, B_1, DM_1, RM_0, B_0, DM_0]
Memory: [9, 6, 7, 3, 4, 0, 1]
Disk: [0, 3, 6]
Makespan: 25
Compute time: 0.275 ms
```

Listing 7. The output of Per-Rev-Revolve(10,2,2,1)

Note that for both these examples we have an additional output: the sentence "`We use periods of size` *n*". This output returns the optimal period for both algorithms with respect to the parameters used.

*Extensions: "--fast" and "--mx".* For the periodic case we have two additional options: "--fast" and "--mx" (Table 6).

As stated in Section 2.4.2, the optimal period $m_X$ for Per-Disk-Revolve can be computed in two different ways. We can either use Algorithm 4 in [Aupy and Herrmann 2017] to compute the value of $m_X$ in a certified way (that the option by default in *Periodic-Disk-Revolve.py*), or we can use the close formula in Conjecture 1. This option can be implemented with the flag "--fast".

In addition, we have also added a flag "--mx" to enforce manually a period chosen by the user.

## 5 SIMULATION RESULTS

In this section, we compare the performance of the algorithms presented above on different computational platforms, using their Python implementation publicly available here: https://gitlab.inria.fr/adjoint-computation/H-Revolve.

The experiments presented in this section were carried out using the PLAFRIM experimental testbed. Since we are interested in the number of forward re-computation, we use in the rest of this section, $u_f = 1$ and $u_b = 0$.

In this section we study three different contexts: we start by showing the gain of using H-Revolve on a platform with a single level of storage compared to Revolve when the access costs are non-zero in Section 5.1. Then we study the case with two level of storage in Section 5.2. In this section we are interested by the loss due to the utilization of the suboptimal algorithm Per-Rev-Revolve in comparaison to H-Revolve. Finally in Section 5.3, we study two different architectures with multiple levels of storage and show how H-Revolve performs compared to different tweaks of existing algorithms.

### 5.1 Platforms with one level of storage and non negligeable costs

The state-of-the-art algorithm Revolve (Section 2.4.1) minimizes the number of forward recomputations when reversing an AC graph on a platform with one level of storage and negligible writing and reading costs. In real life, writing and reading into the memory is not instantaneous and, for some applications, it can actually become significant. In most of real-life applications, the access costs to the memory are not taken into account and Revolve is used as a sub-optimal heuristic. When running a sismic inversion application on Argonne's JLSE system's Skylake, the cost of writing and reading a checkpoint into the memory can take up to 3 times the cost of a forward computation [Kukreja et al. 2018b]. Hence, using Revolve to determine which data to checkpoint as if the access to the memory was instantaneous can lead to significant overheads.

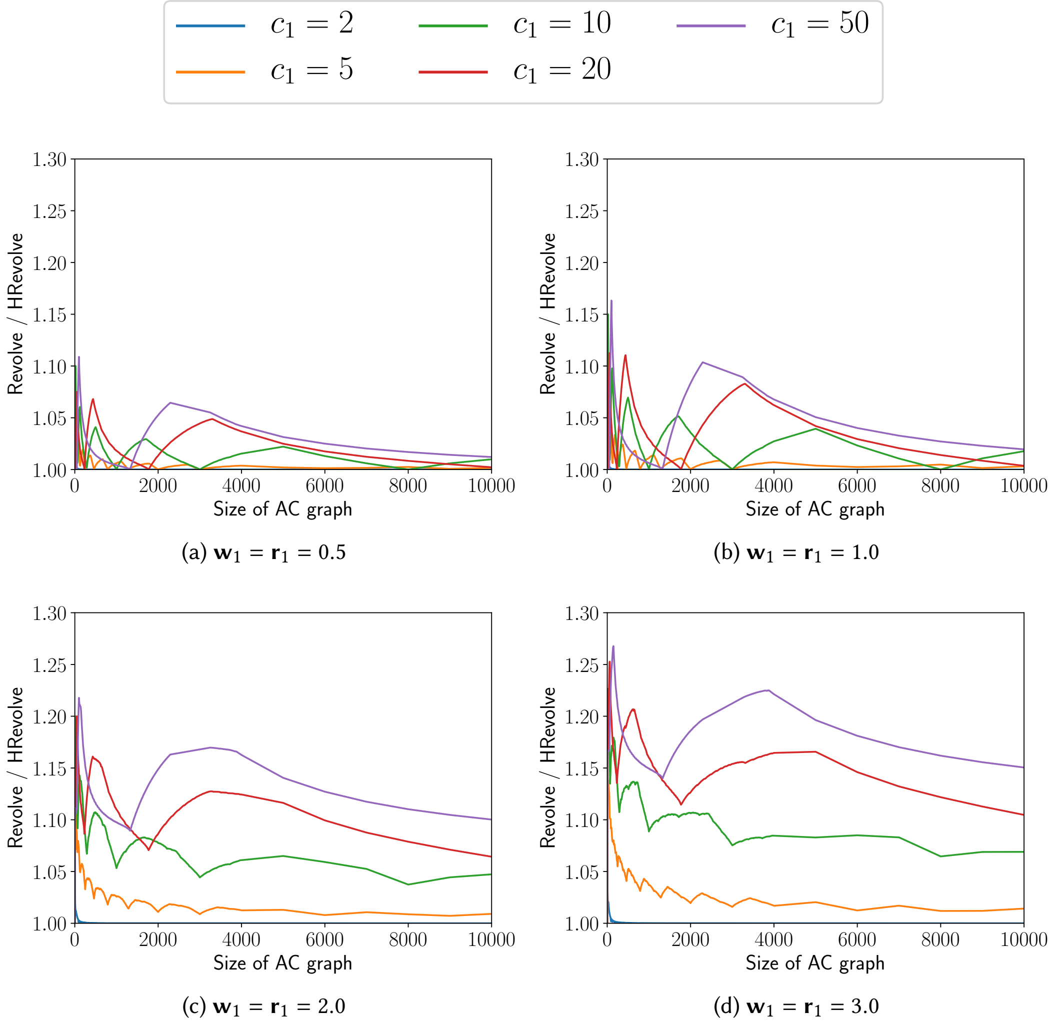
H-Revolve (Section 3.1) minimizes the number of forward recomputations when reversing an AC graph on a platform with an arbitrary number of storage levels and arbitrary access costs. Therefore, it can be used on a platform with one level of storage and non-negligeable costs.

In this section, we evaluate the performance of Revolve (that returns the optimal solution if the access costs were null) compared to H-Revolve (that returns the optimal solution taking into account the non-negligeable costs) on a platform with one level of storage and non negligeable costs. Figure 2 displays the relative performance of Revolve compared to H-Revolve for various memory sizes $\mathbf{c}_1$ and when the writing and reading costs into the memory are $\mathbf{w}_1 = \mathbf{r}_1 = 0.5$ (Figure 2a), $\mathbf{w}_1 = \mathbf{r}_1 = 1.0$ (Figure 2b), $\mathbf{w}_1 = \mathbf{r}_1 = 2.0$ (Figure 2c), and $\mathbf{w}_1 = \mathbf{r}_1 = 3.0$ (Figure 2d)

We can see that, in general, high writing and reading costs into the memory and high number of memory slots, lead to more significant overhead for Revolve compared to H-Revolve. This comes from the fact that with high numbers of memory slots, Revolve uses them as much as possible since it considers that they are *cost-free*, while H-Revolve is more cautious depending on their actual cost. We can note that, on all tested cases, the overhead is less than 28%. While it seems from these figures that the overhead tends to be really small when the graph size is really large, it actually *bounces* back again to a value close to their historical maximum (similarly to what the plots for $\mathbf{c}_1 = 10$ are doing in Figures 2a and 2b) when one continues to increase the graph size.

### 5.2 Algorithms for two-levels storage platforms

In this section, we focus on a platform with two levels of storage ($K = 2$): a first level with limited space ($\mathbf{c}_1$ is finite) with negligible access costs ($\mathbf{w}_1 = \mathbf{r}_1 = 0$), and a second level with unlimited space ($\mathbf{c}_2 = \infty$), also called *disk*, with a writing and reading cost that should be taken into account



(a) $\mathbf{w}_1 = \mathbf{r}_1 = 0.5$

(b) $\mathbf{w}_1 = \mathbf{r}_1 = 1.0$

(c) $\mathbf{w}_1 = \mathbf{r}_1 = 2.0$

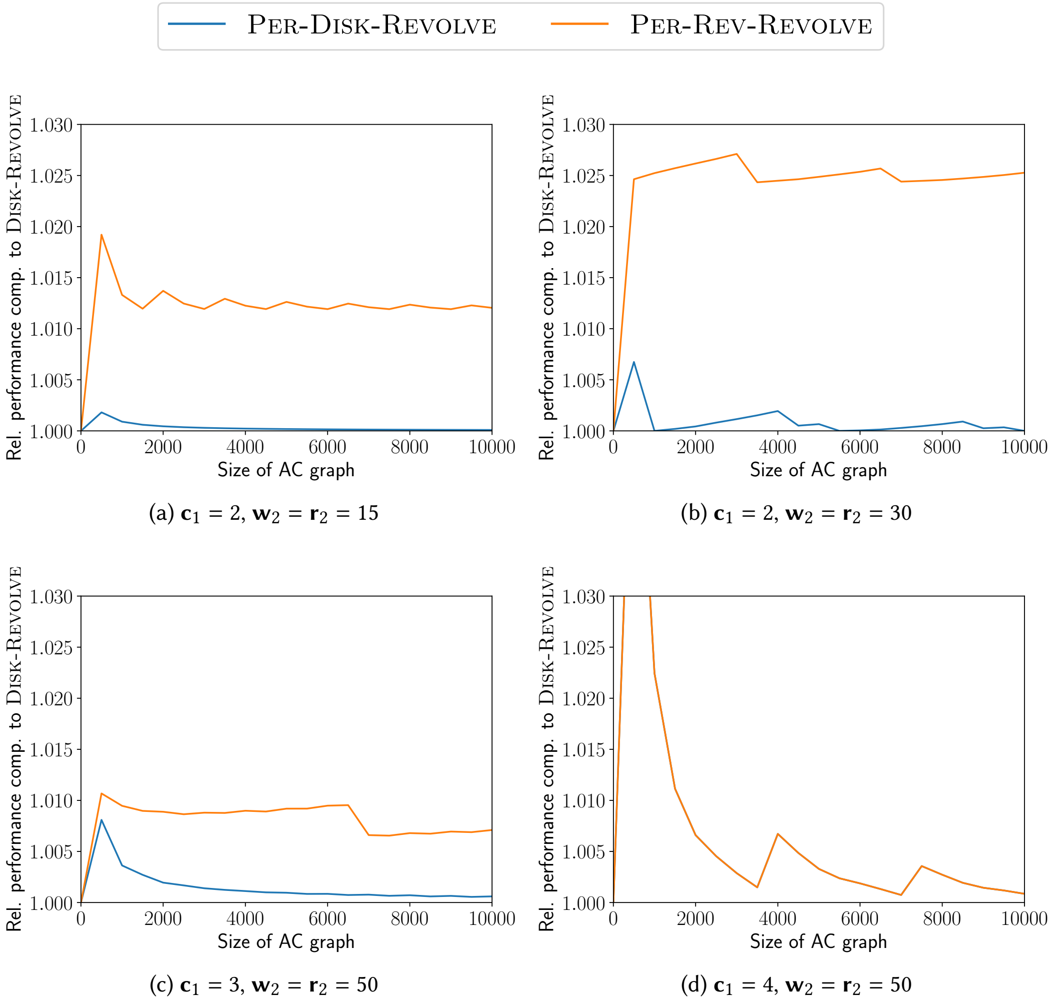
(d) $\mathbf{w}_1 = \mathbf{r}_1 = 3.0$

Fig. 2. Relative performance of Revolve compared to the optimal solution when $\mathbf{w}_1$ and $\mathbf{r}_1$ are not negligible.

(see Problem 3). Algorithms Disk-Revolve (Section 2.4.2), Per-Disk-Revolve (Section 4.3.2), and Per-Rev-Revolve (Section 3.2) can be used to reverse an AC graph on such a platform.

Disk-Revolve provides the optimal solution, while the three other heuristics provide satisfactory easy-to-implement alternatives. Figure 3 displays their performance compared to the optimal solution on four different platforms $\mathbf{c}_1 = 2$, $\mathbf{w}_2 = \mathbf{r}_2 = 15$ (Figure 3a), $\mathbf{c}_1 = 2$, $\mathbf{w}_2 = \mathbf{r}_2 = 30$ (Figure 3b), $\mathbf{c}_1 = 3$, $\mathbf{w}_2 = \mathbf{r}_2 = 50$ (Figure 3c), and $\mathbf{c}_1 = 4$, $\mathbf{w}_2 = \mathbf{r}_2 = 50$ (Figure 3d).

We can see that Per-Disk-Revolve and Per-Rev-Revolve have an overhead of less than 3% compared to Disk-Revolve on the tested cases. The makespan for reversing an AC graph with Per-Disk-Revolve converges to the optimal one when the size of the graph increases, which corroborates the fact that Per-Disk-Revolve is asymptotically optimal (see our previous work [Aupy and Herrmann 2017]). Note that, in Figure 3d, the plots for Per-Disk-Revolve and Per-Rev-Revolve are overlapping. Indeed, when $\mathbf{c}_1$, $\mathbf{w}_2$ and $\mathbf{r}_2$ are large enough, the data stored onto the disk in



(a) $\mathbf{c}_1 = 2, \mathbf{w}_2 = \mathbf{r}_2 = 15$

(b) $\mathbf{c}_1 = 2, \mathbf{w}_2 = \mathbf{r}_2 = 30$

(c) $\mathbf{c}_1 = 3, \mathbf{w}_2 = \mathbf{r}_2 = 50$

(d) $\mathbf{c}_1 = 4, \mathbf{w}_2 = \mathbf{r}_2 = 50$

Fig. 3. Relative performance of the two storage levels heuristics compared to Disk-Revolve.

1D-Revolve for the optimal period is read only once. Thus, Per-Disk-Revolve and Per-Rev-Revolve will return the exact same solution.

### 5.3 Heuristics performances for hierarchical storage

In this section, we focus on the general hierarchical AC problem on a platform with an arbitrary number of storage levels (Definition 2). We have seen that the current storage architecture is often deeper than two-levels. In this section, we assess the gain one can obtain using H-Revolve to reverse an AC graph on an arbitrary hierarchical storage platform with the optimal checkpointing strategy.

*Architectures.* We ran our experiments for two different architectures. The first one (*arch_1.txt* described in Listing 8) has four levels of storage for a total of 24 storage slots. The second one (*arch_2.txt* described in Listing 9) has three levels of storage for a total of 42 storage slots.

```
#Content of arch_1.txt
4
1 1 1
1 5 5
2 10 10
20 20 20
```

Listing 8. Description of arch_1 with 4 levels of storage

```
#Content of arch_2.txt
3
2 0.5 0.5
20 10 10
20 20 20
```

Listing 9. Description of arch_2 with 3 levels of storage

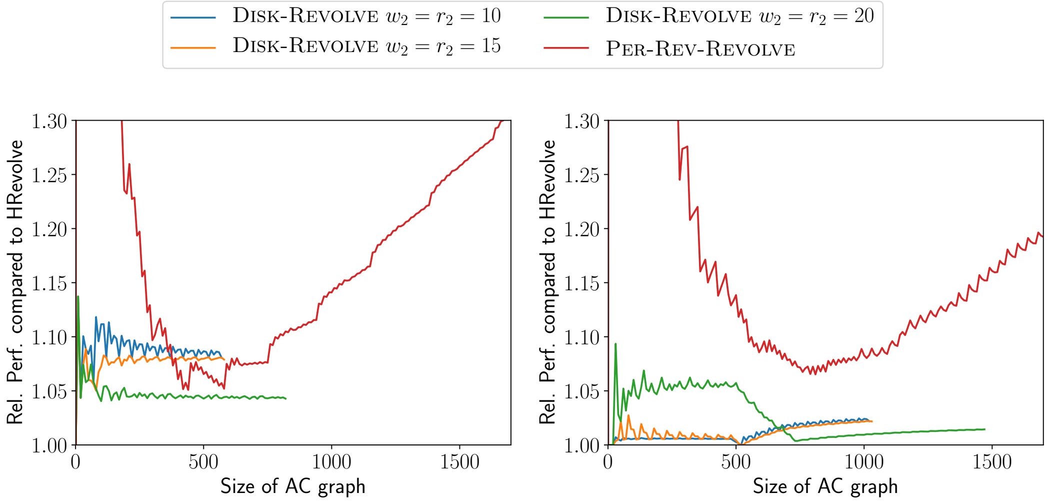
*Algorithms evaluated.* There is not much literature on algorithms for platforms with more than 2 levels of storage. In order to study the performance of H-Revolve, we split the different levels into two levels and compare H-Revolve to algorithms that reverse an AC graph on platforms with two levels of storage (such as Disk-Revolve and Per-Rev-Revolve). One of the issue in this implementation is to choose the values for $w_d$ and $r_d$ for the second level. In addition, Disk-Revolve considers that the architecture has two levels of storage: one limited level that is free, and another one unlimited but costly. Hence in our case, Disk-Revolve might use more storage slots from the second level of storage than what is actually available, and, thus, we can expect that Disk-Revolve will fail to execute graphs that are beyond a certain size. In those case, we choose an implementation of Per-Rev-Revolve where the size of the periods (see Section 4.3.2) is the size of the graph divided by the number of slots available in the second level of storage.

Finally, we also compare to an implementation of Revolve where the checkpoints the least used are stored on the storage the furthest away (similarly to the approach by Stumm and Walther [Stumm and Walther 2009]).

Specifically we study the following algorithms with respect to the performance of H-Revolve:

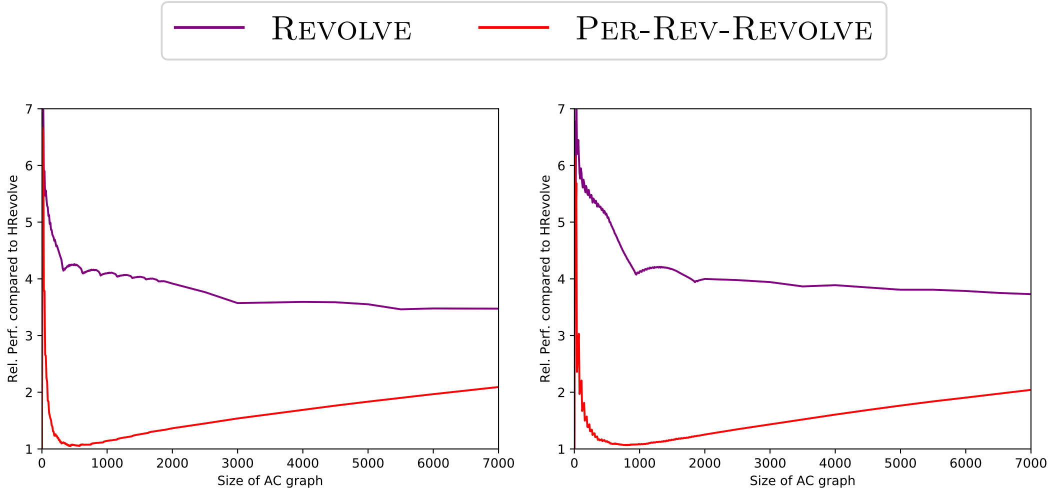
- Disk-Revolve where $c_m$ is equal to the first two levels of *arch_1.txt* (resp. the first level of *arch_2.txt*); and the second level of storage use the last two levels available. To run these algorithms, one needs to decide a cost for the disk access, we try:
  - the minimum costs of the last two levels ($\mathbf{w}_2 = \mathbf{r}_2 = 10.0$),
  - the maximum ($\mathbf{w}_2 = \mathbf{r}_2 = 20.0$),
  - the average ($\mathbf{w}_2 = \mathbf{r}_2 = 15.0$).
- Per-Rev-Revolve where (i) $c_m$ is equal to the first two levels of *arch_1.txt* (resp. the first level of *arch_2.txt*); the period $m_X^{rr}$ is equal to the length of the graph divided by the number of slots available in the last two levels of storage (equally partitionned). This is similar to what is implemented by Pringle et al. [Pringle et al. 2016].
- Revolve where the checkpoints the least used are stored on the most expensive location.

*Results.* Figure 4 displays the relative performance of the Disk-Revolve and Per-Rev-Revolve based heuristics compared to H-Revolve on small AC chains. As can be expected, the heuristics using Disk-Revolve perform relatively well (in general less than 5% overhead compared to the optimal solution), but are quickly limited when the size of the graph increases. Indeed, in the implementation of Disk-Revolve, the number of disk storage is considered unlimited which is not the case here. Hence, beyond a certain size of graph, the implementation is not usable. As can



(a) *arch_1.txt* (b) *arch_2.txt*

Fig. 4. Relative performance of the heuristics compared to the optimal solution on hierarchical platforms for small graph sizes.



(a) *arch_1.txt* (b) *arch_2.txt*

Fig. 5. Relative performance of the heuristics compared to the optimal solution on hierarchical platforms for large graph sizes.

be expected, the smallest the access costs ($\mathbf{w}_2 = \mathbf{r}_2 = 10$ vs $\mathbf{w}_2 = \mathbf{r}_2 = 20$), the less time there is between consecutive checkpoints to disks when there is an unlimited number of disk slots. Hence those algorithm fail for smallest graphs. Per-Rev-Revolve on the contrary does not perform well on small graphs (which can be expected since it does not use its optimal period), but allows to execute a larger set of graphs with a compromise on the performance.

Table 7. Execution time to compute the schedules for the different algorithms and different values of $l$. For the periodic algorithms, we used the option `-fast`.

| Algorithm | $l = 100$ | $l = 1\,000$ | $l = 10\,000$ |
|---:|:---:|:---:|:---:|
| Revolve($l$, 20) | 40ms | 4s | 6min |
| Disk-Revolve($l$, 20, 10, 2) | 40ms | 4s | 8min |
| Per-Disk-Revolve($l$, 20, 10, 2) | 0.3s | 0.5s | 1s |
| Per-Rev-Revolve($l$, 20, 10, 2) | 10ms | 60ms | 0.6s |
| H-Revolve($l$, *arch_1.txt*) | 30ms | 2.6s | 6min |

It is interesting to look at the results on even larger graphs (Figure 5). In this case we only study Revolve and Per-Rev-Revolve which are the only scalable algorithms when the total number of storage slots are bounded. The loss in performance incurred by Per-Rev-Revolve increases at a slow rate but linearly. Intuitively one can expect that there is no upper-bound on the performance ratio between Per-Rev-Revolve and H-Revolve. On the contrary, Revolve performs poorly (a performance ratio of 4 with H-Revolve), but seems to stay constant with the increase in the size of the graph.

To sum up, with multiple levels and when the number of checkpoints is limited it is definitely interesting to use H-Revolve. Per-Rev-Revolve may be an interesting direction depending on the ratio between the total number of available storage slots and the total size of the graph. Otherwise, with a very large graph, Revolve remains the acceptable solution with a performance degradation which seems to remain bounded.

### 5.4 Execution time to compute the schedule

In this final section, we present the time to compute the schedule by the different algorithms. In our implementation we chose to compute Revolve using the dynamic program approach which is significantly slower than the closed-form approach proposed by Griewank and Walther [Griewank and Walther 2000]. This allows us to compute in one go all values to generate all the figures by simply considering the maximum desirable values for the length and number of checkpointss.

The results are provided in Table 7. Note that the time to compute the optimal period in the case of Per-Disk-Revolve and Per-Rev-Revolve does not depend on the size $l$ of the graph. The difference in execution time is the time to print the schedule.

There is a significant difference between algorithms that go through the dynamic programming and the periodic heuristics that use the theoretical results: for a graph of length 10 000, the periodic approaches take 1s while they dynamic-program approaches take 8min. This is also what makes the period-based approaches particularly interesting, specifically when there are only two levels of storage and when the loss is negligible compared to the optimal strategy.

## 6 CONCLUSION

With the advent of data-base computations, the storage hierarchy of computing platform has seen an increase in dimensions. Data-intensive computations such as Adjoint Computations need to take this into their implementation for better performances.

In this work, we have presented an optimal algorithm H-Revolve to reverse AC graphs on such platforms. In addition we have provided an implementation along with a formalization of a reverse schedule which we believe can be easily integrated into existing frameworks implementing a gradient descent whether they are in automatic differentiation or in machine learning. In addition, we have provided several options to reduce the size of a solution or to compute efficient approximate solutions with minimal overhead.

Finally, we have also discussed recent work for platforms with two levels of storage and showed how one can compute an optimal schedule in this case. We show that this implementation is indeed a good strategy when considering two levels of storage, one of which being unbounded but costly.

## ACKNOWLEDGMENTS

We thank Paul Hovland, Navjot Kukreja and Krishna Narayanan for useful discussions.

This work was supported in part by the French National Research Agency (ANR) in the frame of DASH (ANR-17-CE25- 0004) and in part by the Project Région Nouvelle Aquitaine 2018-1R50119 “HPC scalable ecosystem”. The Plafrim platform is being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d’Aquitaine, University of Bordeaux and CNRS and ANR in accordance to the Programme d’Investissements d’Avenir: https://www.plafrim.fr/.

## REFERENCES

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, et al. 2008. MITgcm user manual.

Guillaume Aupy and Julien Herrmann. 2017. Periodicity in optimal hierarchical checkpointing schemes for adjoint computations. *Optimization Methods and Software* 32, 3 (2017), 594–624.

Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. 2016. Optimal Multistage Algorithm for Adjoint Computation. *SIAM Journal on Scientific Computing* 38, 3 (2016), 232–255.

Phil Brubaker. 2016. *Engineering Design Optimization using Calculus Level Methods.*

Andreas Griewank. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and software* 1, 1 (1992), 35–54.

Andreas Griewank and Andrea Walther. 2000. Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)* 26, 1 (2000), 19–45.

José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. 1996. Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs. In *Computational Differentiation: Techniques, Applications, and Tools*, Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank (Eds.). SIAM, Philadelphia, PA, 95–106.

Navjot Kukreja, Jan Hückelheim, and Gerard J Gorman. 2018a. Backpropagation for long sequences: beyond memory constraints with constant overheads. *arXiv preprint arXiv:1806.01117* (2018).

Navjot Kukreja, Jan Hückelheim, Mathias Louboutin, Kaiyuan Hou, Fabio Luporini, Paul Hovland, and G Gorman. 2018b. Combining checkpointing and data compression for large scale seismic inversion. (10 2018).

Mathias Louboutin, Michael Lange, Fabio Luporini, Navjot Kukreja, Philipp A. Witte, Felix J. Herrmann, Paulius Velesko, and Gerard J. Gorman. 2018. Devito: an embedded domain-specific language for finite differences and geophysical exploration. *CoRR* abs/1808.01995 (2018). arXiv:1808.01995 http://arxiv.org/abs/1808.01995

ORNL. [n.d.]. Summit Specifications and Features. Accessed: 2019-03.

Gavin J. Pringle, Daniel Jones C., Sudipta Goswami, Sri Hari Krishna Narayanan, and Daniel Goldberg. 2016. Providing the ARCHER community with adjoint modelling tools for high-performance oceanographic and cryospheric computation. (2016).

Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. 2016. Asynchronous Two-level Checkpointing Scheme for Large-scale Adjoints in the Spectral-element Solver Nek5000. *Procedia Computer Science* 80 (2016), 1147–1158.

Philipp Stumm and Andrea Walther. 2009. Multistage approaches for optimal offline checkpointing. *SIAM Journal on Scientific Computing* 31, 3 (2009), 1946–1967.

Andrea Walther and Sri Hari Krishna Narayanan. 2016. *Extending the Binomial Checkpointing Technique for Resilience.* Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).