RWTH AACHEN UNIVERSITY

The present work was submitted to the
Institute of Software and Tools for Computational Engineering

Parallel Taping in Adjoint Algorithmic Differentiation

Bachelor Thesis

Presented by Yannik Hesse

Matr.-Nr. 406800

Supervised by Univ.-Prof. Dr. rer. nat. Uwe Naumann

Univ.-Prof. Dr. rer. nat. Jürgen Giesl

Aachen, September 21, 2022

Contents

A Appendix

I Acknowledgements

I would like to express my sincere thanks to Uwe Naumann for the interesting topic and freedom given in approaching it.

I would also like to thank my family and friends who have accompanied me through my studies and these unusual years. Especially thanks to Marc Meyer, David Heye, Tom Hilgers, and Jana Hauer for reading this thesis and providing valuable feedback.

II Executive Summary

Algorithmic Differentiation tools are used for a wide range of different applications from simulation and modelling to gradient descent algorithms in machine learning. As such they play an important role in the scientific computing ecosystem. While Algorithmic Differentiation produces highly accurate derivatives, these come at a computational cost. In particular, the memory-intensive reverse mode (adjoint mode) of Algorithmic Differentiation presents a challenge. With the proliferation of multicore architectures in the computing world, most computing resources go unused unless parallelization is employed. This thesis adds to the already highly researched topic of parallelizing adjoint Algorithmic Differentiation.

Typical Algorithmic Differentiation tools, working with operator overloading, follow the basic serial program design of recording the tape, seeding the adjoints, using the chain rule for back-propagation (reversal) of the tape, and lastly harvesting the results. Recording the tape is numerous times more computationally and memory intensive than just computing the non-differentiated function output. In this work parallelization approaches are devised to hide this taping overhead. This can be done by splitting the differentiable input function (primal) into partial functions. These partial functions can then be concurrently recorded, reducing the overall taping time. Memory limitations are addressed and play a significant role in how this problem is approached. Employing checkpointing and rematerialization schemes as well as just in time recording of the tape.

A parallel implementation of a functional adjoint Algorithmic Differentiation tool, implementing the proposed theoretical results, is provided. Encompassing a testing suite, verification and profiling features, as well as a timing system for later analyses. Primal functions of different complexity are benchmarked in a serial and parallel environment. These benchmark results are in line with what was theorized. On a single machine, speedup factors of up to four were observed. These overall very promising results showcase the benefits and potential of parallel taping. However, real boundaries are imposed, as the serial fraction, given in the back-propagation, quickly becomes the dominating factor of program runtime. The next step will be to implement the here achieved results in state of the art automatic differentiation tools like `dco/c++`.

1 Introduction

Motivation

Derivatives in calculation as we know them today have been existing since the early 17th century. Before the advent of computers, the calculation of derivatives had to be done by hand. With the help of the chain rule of differential calculus, mathematical functions could be differentiated symbolically. Numerical approximations using finite differences are also possible, but can be inaccurate due to a bad conditioning of the original problem. Algorithmic (also Automatic) Differentiation (AD) closes the gap between these two approaches. It generates machine-accurate numerical derivatives with a similar computational effort as the finite difference approach. [13] Algorithmic differentiation is very useful when large computer generated and data dependent simulations are modeled. Use cases include[1] economics, weather simulations, fluid dynamics and a wide range of optimization problems, especially in the machine learning field. With the increase in compute power these numerical simulations are getting more detailed and vastly more expensive to compute. To beat this ever-growing challenge, Algorithmic Differentiation tools need to be heavily optimized and adapted. This may be done by leveraging the power of large compute clusters, a network of interconnected high-performance computers, with orders of magnitude more memory, storage capacity, and compute resources than in any single system, or by making use of specialized compute accelerators like GPUs and FPGAs [11]. Most modern processors have reached a limit in single core performance and have instead opted to scale performance by relying on multicore architectures [15]. This means that even on a single machine there are a lot of potentially unused resources that could be used to speed up the Algorithmic Differentiation process. However, Algorithmic Differentiation is a rather serial problem, as there are numerous data dependency challenges when trying to make use of additional system resources. [13] As part of this thesis one area of the adjoint Algorithmic Differentiation process is going to be analyzed for potential speedup. There are multiple challenges that will be addressed, like data and memory consistency, parallel verification and how to effectively deal with the increased memory overhead.

Goal and Structure

The goal of this thesis is to test and analyze parallel taping in adjoint Algorithmic Differentiation. With the goal of shortening the overall AD runtime, at a higher computational cost. Hereby the input functions are interpreted as a black box and a general approach to parallel taping is desired. In an effort to analyze and support proposed methods, a companion Algorithmic Differentiation tool was created. This companion tool includes a fully functional adjoint sandbox environment with rich features. With the overall intent to propose an adaptable, scalable, and easily portable solution to parallel taping.

The work is simply structured which makes these concepts easy to understand. At the beginning, basic knowledge and preliminary information are provided, including a brief introduction to AD. Then the current AD landscape is considered and research on this field is discussed. Theoretical approaches are devised and constructed. The resulting models and ideas are then implemented in the companion tool. Core program parts and concepts are highlighted and explained in more detail in the implementation section. Lastly, the tool is used to create benchmarks and tests that are then timed, analyzed, and discussed.

[1] https://www.autodiff.org/?module=Applications

2 Fundamentals

To understand the concepts introduced in later sections, it is important to understand the preliminaries. These will be explained and expanded upon in the following paragraphs. Starting with the most important background (Algorithmic Differentiation) followed by core concepts that will be employed to implement parallel taping.

2.1 Algorithmic Differentiation

The goal of Algorithmic Differentiation (AD) is to automatically calculate numerical accurate derivatives of multivariate vector functions. The algorithmically differentiable function, given as code, is called the **primal function**. A major advantage of primal functions given in code is that they can be written using complex code constructs like loops, branches, recursion, and even user-defined classes. Any primal function can be interpreted as a composition of other primal functions. For the lowest level of these primal functions the symbolic derivatives are known. These are called **elemental functions** and include for example: addition, subtraction, multiplication, or division. Using techniques, that will be discussed in more detail later, like source code transformation and operator overloading any arbitrary primal can be decomposed into elemental operations. This decomposition induces a directed acyclic graph (DAG) that illustrates derivative dependencies. Take for example the primal given in C++ code in Listing 1.

Listing 1: Example primal f

```
template <typename T>
void f(T& x1, T x2) {
  T tmp = sin(x2);
  for(int i = 0; i < 2; i++) {
    tmp = x1 + sin(tmp);
  }
  x1 = tmp*x2;
}
```

Here T could be any fitting datatype, for example a double. The mathematical function representing the primal f can be rewritten as

$$f : \mathbb{R}^2 \to \mathbb{R} : f(x_1, x_2) = (x_1 + \sin(x_1 + \sin^2(x_2)))x_2 \ . \tag{1}$$

This induces the following DAG = (V,E) shown in Figure 1, in which edges represent the dependencies.

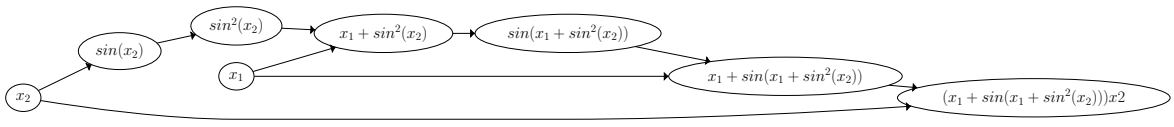


Figure 1: Decomposed DAG of function 1

Another representation of this (compositional) DAG would be the single assignment code (SAC). Each node in the DAG is given a variable v_i that represents the node as seen in Table 1. For each elemental composition a variable is introduced. The compositional dependencies of each v_i is given in the variables it uses for the calculation of its respective value. A data structure like the SAC, with which the compositional DAG of a primal can be derived, is called a **tape** structure.

$v_0 = x_1$		$v_4 = v_0 + v_3$	$= x_1 + sin^2(x_2)$
$v_1 = x_2$		$v_5 = sin(v_4)$	$= sin(x_1 + sin^2(x_2))$
$v_2 = sin(v_1)$	$= sin(x_2)$	$v_6 = v_0 + v_5$	$= x_1 + sin(x_1 + sin^2(x_2))$
$v_3 = sin(v_2)$	$= sin^2(x_2)$	$v_7 = v_6 * v_1$	$= (x_1 + sin(x_1 + sin^2(x_2)))x_2$

Table 1: Example SAC of DAG in Figure 1

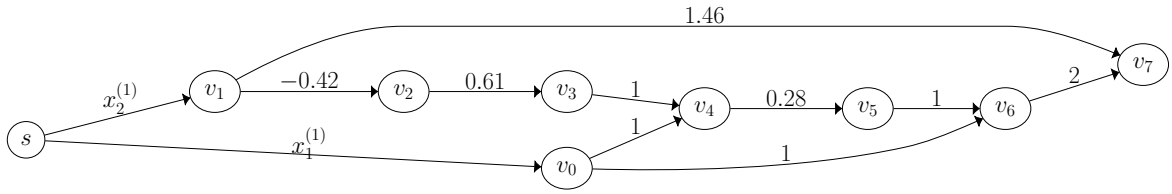
2.1.1 Tangent Algorithmic Differentiation

Tangent Algorithmic Differentiation (TAD) is the forward approach in Algorithmic Differentiation. Take the differentiable function $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^m$. Its Jacobian at position $x \in \mathbb{R}^n$ is $\nabla\mathcal{F}(x) \equiv \nabla\mathcal{F}$. Then the tangent linear model of $\mathcal{F}$ is defined as [13]

$$y^{(1)} = \mathcal{F}^{(1)}(x, x^{(1)}) \equiv \nabla\mathcal{F} \cdot x^{(1)} \ . \tag{2}$$

To compute $y^{(1)}$ with respect to the direction $x^{(1)}$, the derivative of each v_i (of the SAC) with respect to its arguments needs to be computed. This is best understood graphicly, take for example the above function 1 at $f(x_1 = 0.5, x_2 = 2)$. This induces the following $DAG = (V, E)$ in TAD, where the edge weights are defined by

$$w((v_i, v_j)) = \frac{\partial v_j}{\partial v_i} \tag{3}$$



Variables are introduced in $v_0^{(1)} = x_1^{(1)}$ and $v_1^{(1)} = x_2^{(1)}$. Setting these variables is called seeding. Seeding $x_1^{(1)} = 1$ and $x_2^{(1)} = 0$ and then parsing the DAG using the chain rule [13]

$$v_i^{(1)} = \sum_{(x,i)\in E} \frac{\partial v_i}{\partial v_x} \cdot v_x^{(1)} \tag{4}$$

will provide the derivative of f with regard to x_1

$$\frac{\partial f}{\partial x_1} = y^{(1)} = v_7^{(1)} = 2 \cdot v_6^{(1)} + 1.46 \cdot v_1^{(1)} = ... = 2.56 \ . \tag{5}$$

This can be computed during function evaluation. Resulting in a manageable overhead compared to running the primal. When the entire Jacobian is of interest using tangent mode would result in a runtime of $\mathcal{O}(n) \cdot cost(f)$, where $cost(f)$ denotes the compuational cost of running the primal [13].

2.1.2 Adjoint Algorithmic Differentiation

Adjoint Algorithmic Differentiation (AAD) works in many ways similar to TAD. However instead of parsing the DAG in forward direction it is traversed and reduced starting from the outputs. That is why AAD is also called the backwords approach to Algorithmic Differentiation. Take the differentiable function $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^m$. Its Jacobian at position $x \in \mathbb{R}^n$ is $\nabla\mathcal{F}(x) \equiv \nabla\mathcal{F}$. Then the adjoint model of $\mathcal{F}$ is defined as [13]

$$x_{(1)} = \mathcal{F}_{(1)}(x, y_{(1)}) \equiv \nabla\mathcal{F}^{\intercal} \cdot y_{(1)} \ . \tag{6}$$

To compute $x_{(1)}$ with respect to the direction $y_{(1)}$, the derivative of each v_i with respect to its *consumers* needs to be computed.

$$v_{i(1)} = \sum_{(i,x)\in E} \frac{\partial v_i}{\partial v_x} \cdot v_{x(1)} \tag{7}$$

That is why before doing a back-propagation (interpretation) computation the required DAG needs to be recorded prior in a forward (**taping**) run. Taping is recording the dependency and derivative structure of a primal.

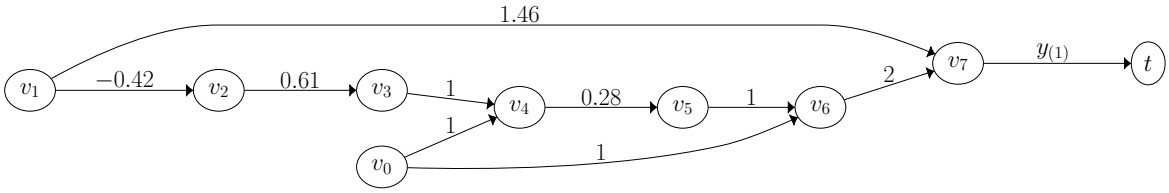


Figure 2: Computed adjoint tape of Listing 1

This is again best understood graphically. Take the $DAG = (V, E)$ in AAD (Figure 2). Here only one additional variable is introduced $y_{(1)}$. Seeding $y_{(1)} = v_{7(1)} = 1$ will provide the entire Jacobian when reduced using Equation (7).

$$\begin{aligned}
\nabla\mathcal{F} &= \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{pmatrix} = (2.56\ 1.32) \\
\frac{\partial f}{\partial x_1} &= v_{0(1)} = v_{4(1)} + v_{6(1)} = ... = 2.56 \\
\frac{\partial f}{\partial x_2} &= v_{1(1)} = cos(v_{1(1)})v_{2(1)} + v_{7(1)} = ... = 1.32
\end{aligned}$$

While structurally similar, AAD is much more memory expensive. To start the interpretation step in AAD the outputs need to be computed in order to be seeded. The full tape needs to be stored in memory, making the problem memory bound. Memory requirements are proportional to the runtime cost of the primal function. Even for rather small problems this results in memory needs far beyond of what typical compute environments offer [16]. AAD is very beneficial if $n \gg m$, as m adjoint computations would suffice to compute the entire Jacobian as opposed to n computations in TAD. This results in a runtime of $\mathcal{O}(m) \cdot cost(f)$,

where $cost(f)$ denotes the computational cost of running the primal [13]. Typical AAD tools follow the same four basic steps in their design. The tape needs to be recorded in the *taping* process. Then the output variables are *seeded*, based on the required derivatives. After that the tape *interpretation* (also called back-propagation and reversal) can be started and later the resulting derivatives are *harvested*.

2.1.3 Algorithmic Differentiation by Overloading

There are multiple ways of generating the tape (sometimes called gradient tape [12]) needed for AAD and TAD. One method is source code transformation. This approach uses a precompiler to inject the needed tape generation statements into the primal function. For every elemental operation the respective derivative is added to a tape data structure. This method allows for an efficient use of compiler optimization, as no custom floating-point datatypes are employed [8].

Another method is to use programming languages that allow operator overloading, like `C++`[14]. Operator overloading makes use of custom datatypes that implement most mathematical operations of standard primitive datatypes, mostly overloading a double type. This datatype can be used like any other by the programmer, while in the background, for each executed operation, additional dependency and derivative information is stored in the tape. This allows seamless integration of any arbitrary primal into any general-purpose AD tool. Using template functions, the overloaded datatype does not even need to be considered by the user. This is the approach that is going to be used in the companion tool and throughout the thesis. Listing 1 shows an example of a primal using a templated function.

2.2 Checkpointing

Checkpointing in the general scientific computing sphere is the idea to save a state of a program, in this case the primal function, and to be able to restore this state again. These saves are called **checkpoints**. Reasons for employing checkpointing could be to achieve fault tolerance or to be able to interrupt a program and continue execution at a later time. In the case of AD checkpoints are used to save and restore the state of the primal function. Long running primals can be segmented into connected functions, where the output of one is the input of another (see the example in Equation (10)). The output of these partial primals can then be used as a save-state. As primals get quite large and there is need for continuous backtracking, this is a trade-off between memory and compute power. This process is also called recalculation (rematerialization); and makes the adjoint AD mode feasible even for extremely long runnning simulations [6].

$$f(x) = (... \circ f_{i-1} \circ f_i \circ f_{i+1})(x) \tag{8}$$
$$check_i = (... \circ f_{i-1} \circ f_i)(x) \tag{9}$$
$$f_{i+1}(check_i) = f(x) \tag{10}$$

There are different approaches to checkpointing: The first option is to hide the use of checkpoints from the user. Reasons for doing so might be that the primal function itself is computer generated. However, this comes with obvious drawbacks: The primal could have different memory requirements throughout (by using temporary variables) and there may be

checkpoints set at suboptimal positions, leading to increased storage costs. Using compute intensive processes this can be optimized as well, but that would be outside the realm of this thesis [16]. The easier approach, and the one that will be used in the companion tool, is to provide explicit checkpointing statements that the user has to add inside the primal, as well as the necessary routines to restore from a provided checkpoint.

2.3 Parallel Programming Model

Typical modern CPUs follow a multi-core architecture. That means that there are several independent logical CPU cores that can compute concurrently. Each logical core can also have multiple virtual cores (hyperthreads), but these are considered to be the same for the sake of this thesis. When talking about the available parallelism of a system the number of real concurrently running cores is meant. A process running on an operating system can have multiple threads, these are autonomously executed on different cores. Using more threads than there are cores available on the CPU will not increase parallelism. In the context of this thesis parallelism and concurrency are used interchangeably.

2.3.1 OpenMP

OpenMP [3] is a declarative programming standard, defined for `C++`, C and Fortran. It is used to easily and simply allow serial code to make use of parallelisation. This is done by adding compile instructions inside the code. Afterwords these instructions are compiled and transformed to call system threads. Inside so called *parallel sections* [3] OpenMP manages a group of threads, called a thread pool. These threads are then assigned tasks, for example executing individual loop iterations.

Listing 2: Parallelising a loop in OpenMP

```
1  for(int i = 0; i < 7; i++) {
2    doSomething(i);
3  }
4
5  #pragma omp parallel for
6  for(int i = 0; i < 7; i++) {
7    doSomething(i);
8  }
```

The most common use case is to parallelize independent loop iterations. The ease of use is demonstrated in Listing 2. The first loop will be executed sequentially while the second can make use of the CPU's multicore architecture to parallelize the loop. In this case the programmer has to make sure that *doSomething(i)* is a thread-safe routine or else undefined behavior can occur.

OpenMP loops can contain various statements on how loops are scheduled. Depending on caching and used data structures this may have a significant impact on performance and behavior. To give programmers control over task scheduling, there are different options available. One is the ordered clause. This clause states that before loop iteration i can be assigned to any thread in the thread pool, iteration $i-1$ needs to have been assigned. This is useful when there is dependency between loop iterations. Using the statement `#pragma omp ordered` the loop can be put back into a sequential state inside this control block. Loop iteration i may only enter once $i-1$ has left the ordered clause. This can be used to create various reduction loops that are based on non-associative algebra, but it also limits the

parallelism if used incorrectly. The code presented in Listing 3 shows this exact design pattern.

Listing 3: Parallelising and then Sequentialising in OpenMP

```
#pragma omp parallel for ordered
for(int i = 0; i < 7; i++) {
  doSomethingParallel(i);
#pragma omp ordered
{
  doSomethingSequential(i);
}
}
```

Defining a schedule and a size, divides the iterations among the available threads in the thread pool. In combination with work intensive tasks and an `#pragma omp ordered` statement a loop schedule using blocks larger than one will cause a lot of unproductive time in the threads waiting to process the ordered block. Examples of two very different OpenMP loop schedules are shown in Figure 3 and Figure 4 respectively.

Figure 3: Random Loop Iteration distribution by OpenMP. Colors refer to Threads

Figure 4: User Defined Scheduling / Ordered / Block Size of One. Colors refer to Threads

3 Related Work

Parallelization has become quintessential in scientific computing, with the rise of computational power and large distributed systems. Especially with the increase of logical cores and the stagnation of single core performance [15]. As such, a lot of research that discusses the use of parallel software architectures with regards to differentiability has been created.

Many recent works, actively developed tools, and use cases can be found on the autodiff[2] website. Most of the fundamental knowledge and mathematical derivations behind the AD by overloading approaches presented in Section 2 are taken from Uwe Naumann's book "The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation" [13]. This thesis builds, in particular, upon major research in the area of checkpointing in the reverse mode of AD. One of the first fundamental papers regarding this topic was "Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation" by Andreas Griewank [4], which proposes a bisection of primal functions to decouple the linear growth of program runtime and memory requirements. Essentially making the adjoint mode of AD a viable option for the limited hardware available at the time. A Fortran implementation of recursive checkpointing [10] based on the prior research was devised. Although this work used forks to realize these checkpoints, the approach to recursive checkpointing is very similar to the one presented later in this thesis. What must be noted at this point, is that these theoretical and practical results were devised at a time where parallelization, as known today, was more of a theoretical concept. As such the focus was likely on making optimal use of the limited hardware available, as opposed to writing for hardware that has not existed during that point of time. This changed with the introduction of multicore architectures and parallel programming models, like OpenMP [3] and the Message Passing Interface (MPI) [7], that are still widely used today. These tools provide a basis for how parallelization is talked about and discussed. In a major collection and summary of prior works, Andreas Griewank and Andrea Walther include detailed discussions about different checkpointing schemes [6]. Illustrating some theoretical optimal binomial checkpointing and uniform checkpointing schemes. As well as stating "The promise and urgency to exploit this added parallelism grow as multicore processing become more and more prevalent." [6] about AD methods. Some early and general parallel methods are discussed, some of which are similar to those presented in this thesis, yet the authors stated themselves that these methods have not been implemented as of writing [6]. ADOL-C [5] and `dco/c++`[14] represent state-of-the-art AD tools. As far as the author is aware, parallel taping approaches have not been implemented in these tools yet.

Nonetheless, significant research about more scenario dependent parallel AD methods has been devised and implemented. As OpenMP provides a clean structure in which parallel functions can be written, these can also be used by AD tools that then make use of primal concurrency themselves [1]. MPI is used to parallelize on a much larger cluster level scale, resulting in a highly more complex program structure. Here researchers also found methods that could differentiate primals written in MPI [18]. Other approaches take advantage of parallelism by reasoning about certain geometric structures within certain primary functions, which can then be used to partition the problem[17].

Solutions discussing the parallelization of taping in adjoint AD are limited. One approache that is similar to the one presented in this thesis, is the bachelor thesis by Jannick Kremer about "Parallel adjoint taping approaches with OpenFOAM" [9], which was developed concurrently to this one. In his work he implements parallel taping for the adjoint version of the OpenFOAM[3]

[2] https://www.autodiff.org/
[3] https://www.openfoam.com/

solver, with the employed method being expandable to more general AD problems. Some important conceptual differences must be highlighted. For one his work[9] builds parallelization on top of MPI and `dco/c++`, as opposed to OpenMP being used in the companion AAD tool of this thesis. Extensive discussions of AD's memory limitations were not included, nor was the topic of checkpointing, which is an essential part of this work's approach. These works can be considered complementary as they they reach a similar conclusion regarding the potential speedup, parallel efficiency, and overall approach to the parallel taping process, although they take different paths to reach this conclusion.

4 Theoretical Approaches

In this section theoretical approaches are discussed step by step. To begin the general idea of cutting the tape and reconnecting it is explained. After that the idea behind parallel taping is expanded upon. This will then be used to create a model that is implementable in a less constrained form of AAD. This model is then adapted to work in a memory bound environment.

4.1 Tape Cutting

It was already established that the first step in the AAD process is taping. Taping is recording the dependency and derivative structure of the primal. In most cases this tape is not actually stored as a DAG, but it is a useful tool for visualization. Here however, it is better to think of the tape as a large vector. This vector contains all necessary information on how to differentiate the primal. An example of how to encode such a DAG in a vector can be seen in Vector 11, which encodes the DAG of Figure 2. This vector is interpreted starting from the back and is read like this: Node 7 has 2 predecessors, node 6 (with partial derivative $\frac{\partial v_7}{\partial v_6} = 2.0$) and node 1 (with partial derivative $\frac{\partial v_7}{\partial v_1} = 1.46$).

$$t = (..., 1.0, 3, 1.0, 0, 2, 4, 0.28, 4, 1, 5, 1.0, 0, 1.0, 5, 2, 6, 1.46, 1, 2.0, 6, 2, 7) \in \mathbb{R}^{30} \tag{11}$$

As computing this vector does not require any data from the vector itself, taping can be done independently of one another. Different chunks of the tape can be computed individually. A chunk consists of a partial tape of a defined range. An example of splitting the tape into three chunks can be seen below.

$$\begin{aligned} t &= t_1 +\!\!+\, t_2 +\!\!+\, t_3 \\ t_1 &= (-0.42, 1, 1, 2, 0.61, 2, 1, 3, 1.0, 3) \in \mathbb{R}^{10} \\ t_2 &= (1.0, 0, 2, 4, 0.28, 4, 1, 5, 1.0, 0) \in \mathbb{R}^{10} \\ t_3 &= (1.0, 5, 2, 6, 1.46, 1, 2.0, 6, 2, 7) \in \mathbb{R}^{10} \end{aligned}$$

4.2 Parallel Taping in non-memory bound Environments

The overloaded primal runtime ($t_{overload}$) divided by the primal runtime (t_{primal}) is defined as the overload factor ($\rho := \frac{t_{overload}}{t_{primal}}$). Running the primal function using a compiled version in `C++` is numerous times faster than running the overloaded primal with a custom datatype that is recording the tape simultaneously. This is not at all surprising as each operation is overloaded, which provides a constant overhead per step. Realistic measurements that were obtained throughout this thesis were in the range of $\rho \in (3, 103)$, but the actual value of ρ is heavily dependent on the primal function and the compiler optimization employed.

In this first approach memory allocation time and size limitations are neglected but will be addressed in the next section. It is assumed that the complete tape fits into main memory. The tape can be cut into different chunks. These chunks are partial tapes that can then be

reconnected to the complete tape. Splitting the tape into chunks allows parallel tape recording. With this, the overload-factor can be divided by the amount of cores present in the system $\frac{\rho}{thread}$. However, this would assume that each thread could start at the same time. That is not the case. As a partial function evaluation upto the start of a chunk is needed to start taping that chunk. This provides a natural upper bound in the non-overloaded primal function or a best-case overload factor of one.

$$\lim_{thread \to \infty} 1 + \frac{\rho}{thread} = 1 \tag{12}$$

To reach this overload factor however an arbitrarly large number of threads is needed.

Taking into consideration that the limitation is indeed the concurrency the host system allows, a more realistic model can be derived by running the primal function once and starting equidistant chunk taping whenever a chunk starting point (checkpoint) is reached. A model of this can be seen in Figure 5.

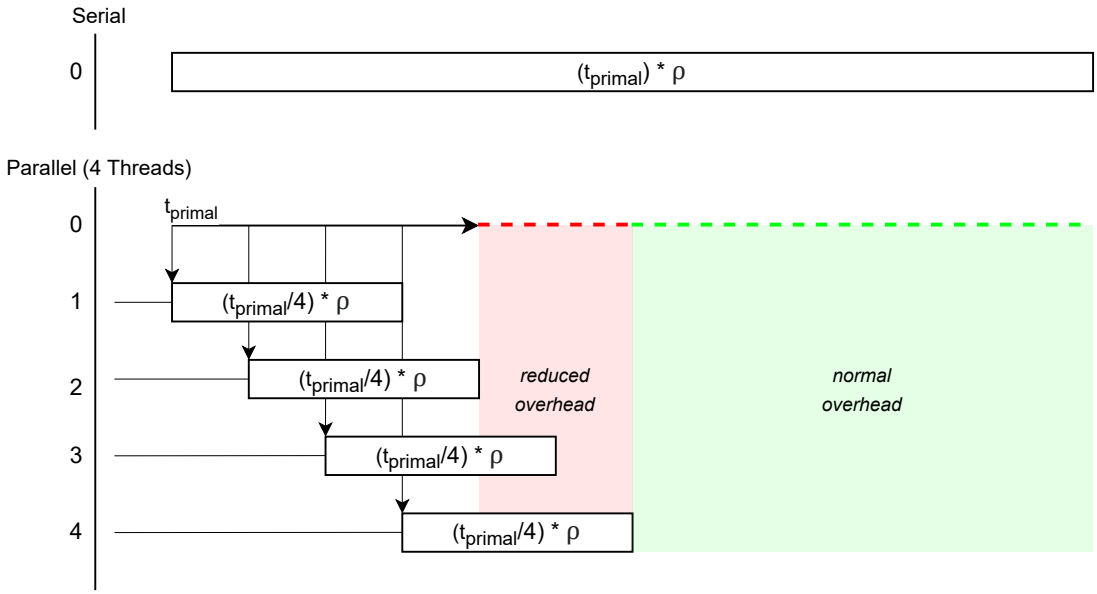


Figure 5: Illustration of realistic reduction of ρ using four parallel threads

This leads to an improved overloaded primal runtime ($t_{overload+}$) of

$$t_{overload+} = t_{primal} * (\frac{\rho - 1}{threads} + 1) \tag{13}$$

This means for a 16 threaded system, a primal runtime of 10ms, and an overloading factor of 31; an improvement of 300ms to 28.75ms, or a speedup of 10, in taping time.

This approach only looks at the taping process, the reversal step is still a sequential one. As this thesis will only take a look at parallelizing the taping process discussions around ways to optimize or parallelize the reversal will be omitted. This is not to say that there are no relevant techniques that could be employed, and further research should definitely look at how adjoint back-propagation could be parallelized.

$$t_{total} = t_{overload+} + t_{reversal} \tag{14}$$

Equation 14 provides the achievable time employing parallel taping, using the above method.

4.3 Parallel Taping in memory bound Environments

When talking about realistic applications for adjoint solvers, the relevant problem is always memory bound. The techniques applied to the non-memory bound environment are built upon and expanded. However due to this factor, the approach presented in Figure 5 becomes infeasible. If taping were to be done from the first chunk, there would not be any space left for the last chunk, which is needed to begin the seeding and reversal step. There are a lot of ways of tackling this problem, like writing the tape to an external drive and then only reading parts of it into memory. However, the simplest and most efficient way is to blur the lines between the two steps of first recording and then reversing.

As established, the serial fraction of this problem is bound to be the reversal runtime ($t_{reversal}$). That means a just in time recording of the tape would suffice. The parallel recording process must be fast enough to continuously supply tape to the reversal process. For this approach the complete tape will be split into chunks. Each of these chunks can be seen as an independent task. This task must first retrieve the checkpoint data to know from which point to start tape recording. Then it records the tape and waits for the adjoint vector, since that is where the information for the reversal is stored. Lastly it passes the adjoint vector to the next task and frees the used memory. An illustration of this process can be seen in Figure 6. As primals that are given to the adjoint program can vary in size and memory use, all the checkpoints can most likely not fit into memory, however this is omitted in this discussion for now. The adjoint program will do a checkpointing run, storing each checkpoint in a list. The last few checkpoints, depending on the level of parallelism the system allows, are then recorded concurrently in a parallel window. As soon as a task is finished, the parallel window moves to the next task and the process starts over again.

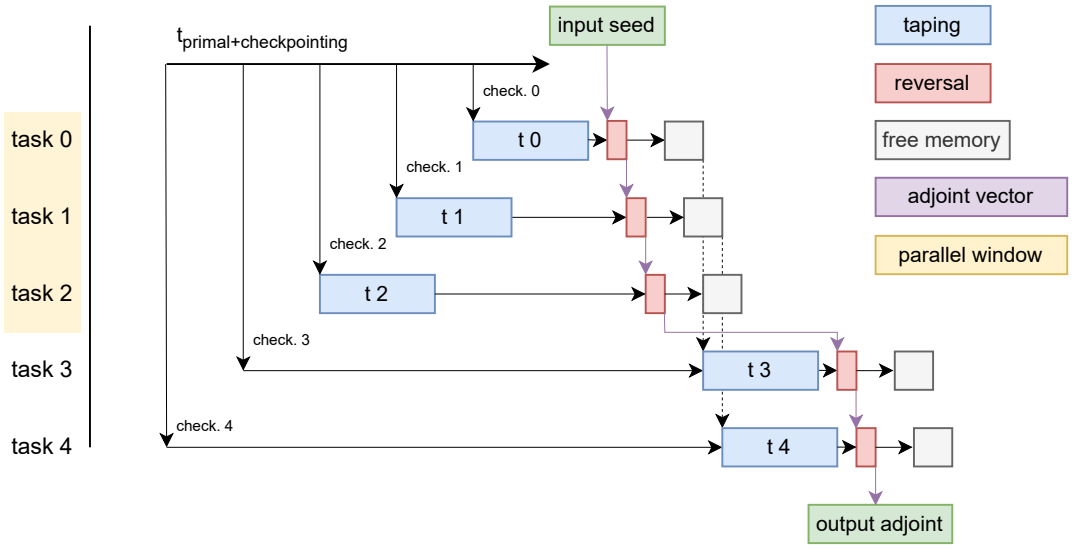


Figure 6: Illustration of the theoretical tasking process. Demonstrated here with 5 tasks and a parallel tasking window of 3. On task completion, system memory and computing resources become available again so that the next task may be started.

Checkpointing time consists of running the primal once and storing the checkpoints. The very last chunk (first tape) must be recorded in the serial and parallel case. After that the reversal

starts, while the sequential version must interrupt the reversal and record the tape every time, the parallel version could already have a tape ready, reducing the runtime significantly. A illustration, looking at this process from the broader perspective, on how this approach reduces overall runtime can be seen in Figure 7.

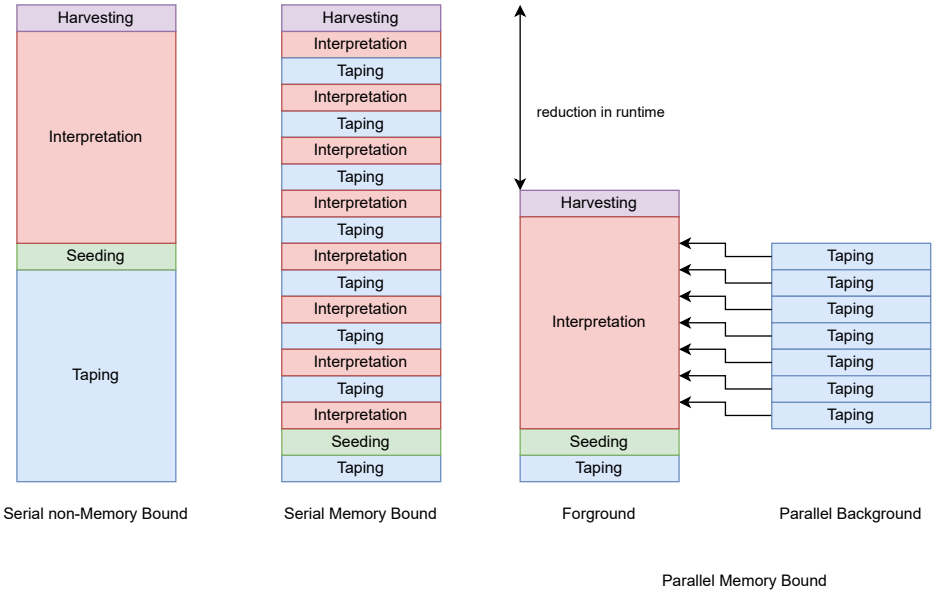


Figure 7: Summary and overview of the parallel approach presented in this section.

It was already mentioned that checkpoints could not feasibly be stored in memory with very long running primal functions. That is why the following checkpointing schemes were implemented.

4.3.1 Reactive Checkpointing

The most straightforward method to obtain the checkpoints is to do a forward run starting from the beginning to the exact checkpoint the task needs to begin taping, as illustrated in Figure 8.

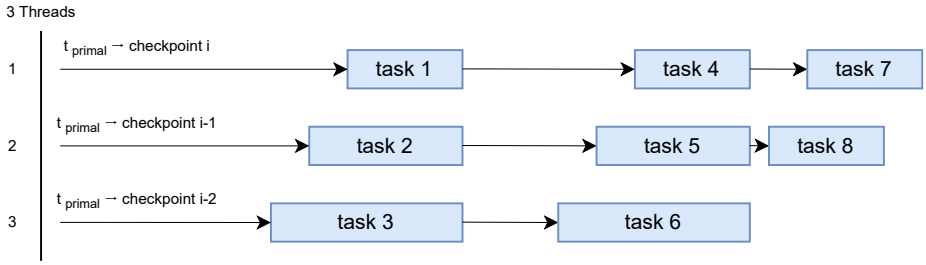


Figure 8: Illustration of the no reactive checkpointing process

This method comes with obvious drawbacks compared to the other proposed methods. However, it does not require synchronization between tasks and is as such stupidly parallel and can be

useful in certain environments. If n is the amount of tasks (and also checkpoints) that need to be computed it takes on average

$$\frac{\sum_{i=0}^{n-1} \frac{t_{primal} * i}{n}}{n} = \frac{(n-1) * t_{primal}}{2n} \tag{15}$$

$$\lim_{n \to \infty} \frac{(n-1) * t_{primal}}{2n} = \frac{t_{primal}}{2} \tag{16}$$

This is not at all surprising, but one can quickly see that this method must perform a quadratic amount of recalculations. If it takes $t_{check} \cdot n = t_{primal}$, then the total time for complete checkpointing is

$$n \cdot (\frac{(n^2 - n) \cdot t_{check}}{2n}) = \frac{(n^2 - n)}{2} \cdot t_{check} \in \mathcal{O}(n^2) \tag{17}$$

It is included as a baseline, to showcase the benfits of the other proposed methods that follow below.

4.3.2 Proactive Uniform Disk Offloading

When recording the checkpoints this method stores the checkpoints in files that are then later read. In modern computers storage space is usually not a problem and can be increased on demand. However, writing to the disk does not provide a magical solution to the problem, as writing and reading from the disk can cause bottlenecks. With modern hardware like NVME-SSDs or distributed parallel I/O the time spent waiting on write and read calls can also be minimized.

Theoretical analysis of this approach is non-trivial. While the access times to obtain a checkpoint are constant and independent of t_{primal}, disk speeds may vary for many reasons. As such a theoretical analysis is omitted for now but will be discussed in more detail in the benchmarking section. A rough estimate for the average checkpoint retrieval time can be given in

$$\frac{t_{primal} * \text{checkpoint writing overhead}}{n} + t_{\text{checkpoint reading time}} \tag{18}$$

Where values like the "checkpoint writing overhead" or $t_{\text{checkpoint reading time}}$ heavily depend on the hardware and implementation. The total time for complete checkpointing has a linear dependency on the number of checkpoints

$$\approx t_{\text{checkpoint reading time}} \cdot n \in \mathcal{O}(n) \tag{19}$$

4.3.3 Reactive Binary Bisection

Even though not all checkpoints fit into the main memory, there is space for a small finite number of checkpoints. These can be filled with a subset of the total checkpoints. While the total number of checkpoints, needed for complete differentiation, remains the same, in this model, not all checkpoints are stored in memory. The first checkpoint is stored, then the second to be stored would be the one in the middle between the first and the last and so on, as shown in Figure 9. These checkpoints are stored in a stack.

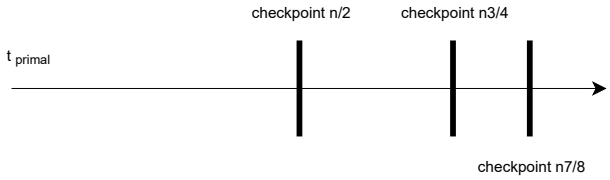


Figure 9: Illustration of the recursive checkpointing process

Instead of computing from the very start upto the checkpoint needed, each task starts from the top of the stack, while also recording onto the stack using this adaptive halving method. A drawback of this approach is that multiple parts of the primal are computed numerous times, just as with the reactive checkpointing approach. But as read and write access on the stack is required, this entire process is sequential with no easy possibility for parallelization. Only a vector size of $\log_2(n)$ is needed to store these n checkpoints efficiently. A theoretical and mathematical analysis of the average checkpointing time turned out to be more complicated than anticipated. This is due to the time taken per checkpoint being highly dependent on the current state of the stack and the total number of checkpoints. As such similar research was taken into account [4, 10] to approximate the average checkpoint retrieval time to being

$$\approx \frac{t_{primal} \log_2(n)}{2n} \, . \tag{20}$$

The recursive checkpointing model was simulated and tested against this approximation using the code provided in Appendix A.2.

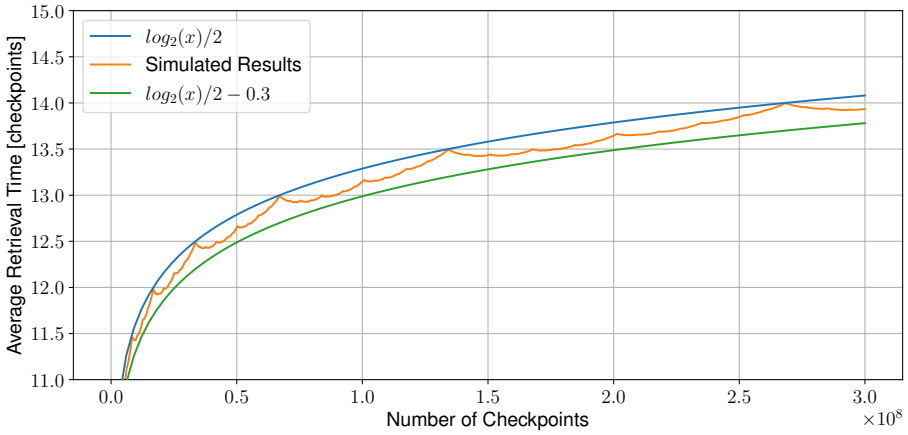


Figure 10: Simulation of the average retrieval time of checkpoints using the recursive halving model. The x-axis shows the number of checkpoints. The y-axis shows the average number of checkpoints that had to be passed in order to obtain the checkpoint.

The results, as seen in Figure 10, underline this approximation for $n < 300 \cdot 10^6$.

While, given the $\lim_{n\to\infty}$, the average retrieval time approaches 0, the total time to obtain all checkpoints still grows with

$$t_{check} \cdot n \cdot \log_4(n) \in \mathcal{O}(n \log n) \, . \tag{21}$$

This adaptive halving process is not optimal, nor new, and has been researched and implemented already [10]. While not optimal, it is "good enough" to compare to the other two checkpointing schemes.

5 Implementation

The following sections explain how the derived theoretical parallelization approach was implemented in the AAD companion program. To be able to use and understand the source code better, an overview of the project structure is provided. Thereafter, the tape structure is elaborated. It should be noted that the parallel algorithm could be ported to any tape structure with some effort. Here the selected tape structure allowed for easy and seamless implementation of adjoint merging and tape cutting. The parallel algorithm is then explained in detail, as well as the chosen approach to checkpointing in the primal. Lastly quality of life features like the profiler, a timing system, verification, and unit-testing are expanded upon. What this section will not cover in detail is a program user guide (Appendix A.1.2) or a developer guide (Appendix A.1.3). Those can be found in the Appendix, as well as a general readme file (Appendix A.1.1). Additionally the full source code is available in the git[4] repository.

5.1 Project Structure

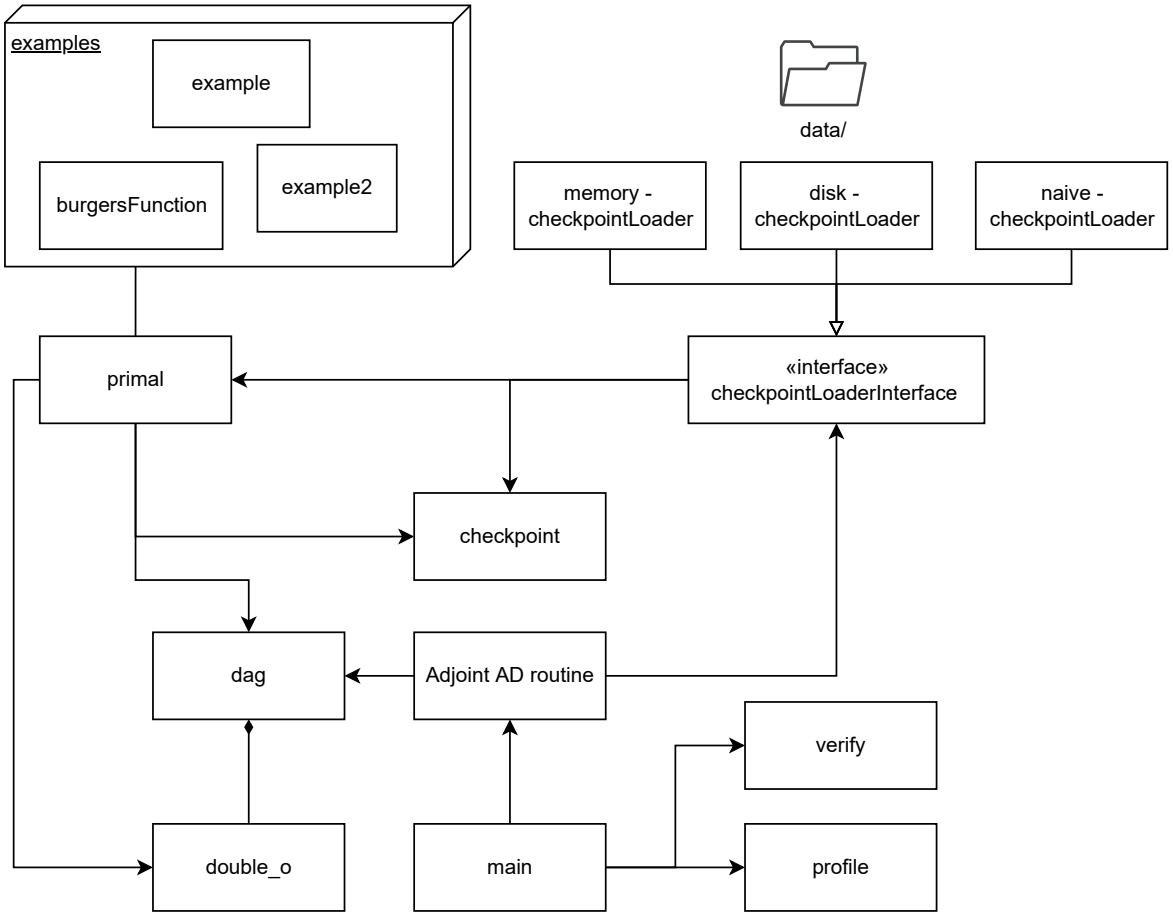


Figure 11: Map of the project structure

Figure 11 provides a simplified overview of the major components of the companion AD tool and their interactions with each other. It is a good introduction into the code base and adds

[4] https://gitlab.stce.rwth-aachen.de/hesse/parallel-taping-adjoint-ad

to the orientation inside this chapter. Most of components are implemented in source files that corresponds to their name, so for example `dag` can be found in `src/dag.hpp` and `double_o` in `src/double_o.hpp`. The entrance to the program is provided in the main routine. This routine can be replaced with entirely different programs that want to make use of the AD tools interface in the background. An example main file is provided in Appendix A.1.2. The heart of the program is the Adjoint AD routine (*aad*), in this all of the other components are jointly used to differentiate the program input (the primal). A detailed breakdown of the implementation of this routine can be found in below in Section 5.3. Three different `checkpointLoaders` implementing the `checkpointLoaderInterface` are implemented. In the source folder `checkpointLoaders` are given an entire folder and namespace. More implementation specific details can be seen below in Section 5.4. The program input is the primal function, as well as the input parameters and primal parameters. The input vector can be provided externally and in this case is given in the main. The other important parameters that define the primal are given in `src/primal/primal.cpp`. Here important configurable options include:

`size`:
: amount of loop iterations to do (scaling of the primal)

`windowSize`:
: amount of loop iterations per partial primal (defines the memory limit the tool is allowed to use)

`cores`:
: parallelism with which to execute the program (defaults to the maximal available threads)

As well as global program parameters given in `src/primal/primal.hpp`:

selected checkpointLoader:
: currently either memory, disk, or naive

DEBUG flag:
: either Timing, Verbose, or Minimal

selected primal:
: the namespace in which the primal input is implemented with *namespace::primal*

Different pre-implemented primals can be found in the examples/ folder and include the following:

example/burgers:
: Explained in more detail in Section 6.1.2

example/example:
: Detailed instructions on how to write primals can be found in Appendix A.1.2, where this primal is implemented

example/example2:
: The primal given in Listing 1

These primals are used to illustrate how the tool works and are later used for benchmarking and analyzing. Any differentiable multivariate function, given as templated `C++` code, can be used in this program, but some adaptions have to be made. See the user-guide for more details (Appendix A.1.2). The `dag` class represents the implemented tape structure and will be

discussed in more detail in the next sub-section. The `double_o` class represents the overloaded double datatype. When initialized with a `dag` struct it will begin populating and recording derivative and heritage information onto the tape. The following elemental operations are currently supported by the program:

- $sin(x)$
- $cos(x)$
- $pow(x, n)$
- $x + y$
- $x - y$
- $x \cdot y$
- $\frac{x}{y}$

Where $x, y \in \mathbb{R}$ and $n \in \mathbb{Z}$. This list of elemental operations is easily expandable as long as the symbolic derivative of the function is known. How this is done in detail is explained in the developer guide (Appendix A.1.3).

5.2 Tape Structure

The data structure used to implement the tape was heavily inspired by Uwe Naumanns work [12]. As already demonstrated in Section 4.1, the tape is implemented as a vector. But instead of one large vector, this vector is split in two, in oder to minimize memory requirements by not storing every int as a double. The first vector is used to store partial derivative values and, interpreted as a DAG, can be seen as the edge weights. The second vector consists of graph dependencies (E). To be able to interpret and reverse this DAG, node values representing the $v_{i(1)}$ need to be stored in a vector as well. This is done in a third adjoint vector. While the first two vectors are accessed sequentially, similar to the presented approach in Section 4.1. The third vector is accessed randomly. The concept behind the data structure is to minimize the memory requirements of the adjoint vector (random access memory). This is done by only storing needed values and replacing ones for which all their predecessors have been computed already. When recording onto this tape structure, variables need to be identified as either persistent or temporary variables. Persistent variables are the ones that are left of any assignment (lvalues). Analogously, temporary variables are the ones that are never stored and only used in an expression on the right hand side (rvalues). With this the memory requirements of the random access memory (adjoint vector) will not exceed the memory requirements of the original primal. This is important as splitting the adjoint vector is not easily possible. As variables are reused when reassigned, the graph can by interpreted as a directed cyclic graph.

Example. The DAG in Figure 2 recorded onto this data structure as a directed cyclic graph (DCG) can be seen in vectors d (derivatives) and v (vertices and edge information). Persistent variables are given negative identifiers, while temporary variables are given unique non-negative identifiers ($\in \mathbb{N}_0$).

$$d = (-0.416147, 1, 0.6143, 1, 1, 1, 0.278012, 1, 1, 1, 2, 1.46058, 1) \in \mathbb{R}^{13}$$

$$v = (0, -1, 0, -2, 0, -3, -2, 1, 0, ..., -3, 1, 3, -1, 3, 2, 4, 4, 1, -3, -3, -2, 2, 5, 5, 1, -1) \in \mathbb{R}^{39}$$

Vertice vector v is interpreted from right to left as (-1) has 1 predecessor and that is (5), (5) has 2 predecessors and those are (-2) and (-3) and so on. As persistent variables are used more than once, a DCG is induced, as shown in Figure 12.

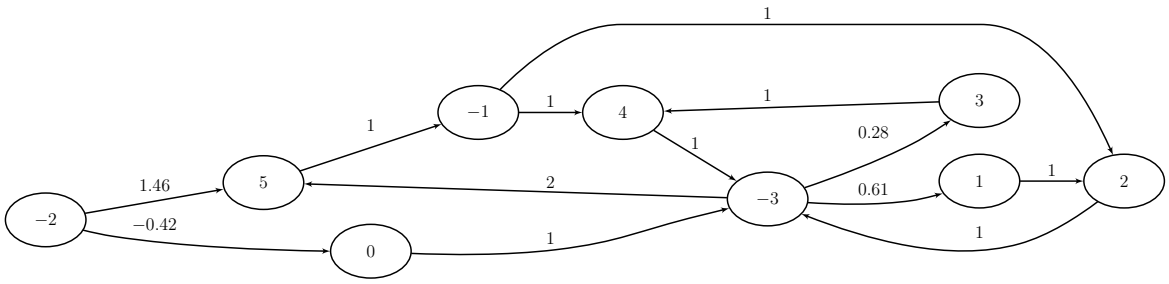


Figure 12: DCG of the DAG in Figure 2 and vertices vector v and derivative vector d.

In this case (-1) represents the variable x_1 in code, (-2) x_2, and (-3) *tmp*. As *tmp* is assigned multiple times it is the vertex with the most arrows going out and in. This DCG is still interpreted like a normal DAG, this can be seen in Figure 30 which is an illustration of this same DCG as a DAG.

5.2.1 Interpretation (Reversal)

The vertices vector beginning at the end, provides step by step instructions to differentiate the primal. As such the vector can easily be cut into chunks or streamed. The `interpret` routine, using the chain rule of differentiation (Equation (7)) as a reduction function, is then called on the tape.

Example. Due to the given primal only ever using three persistent variables (x1, x2, tmp) and one temporary variable for the expression buffer, the adjoint vector can minimized to a length of four. The seeding process of setting $y_{(1)} = 1$ has to be done in the adjoint vector by setting the very first value to one. This is due to how the output was implemented as x_1. The following node id to adjoint id mapping is used

$$map(x) = \begin{cases} a_{(-x-1)}, & \text{if } x < 0. \\ a_3 & \text{otherwise.} \end{cases} \tag{22}$$

$$\begin{aligned}
a &= (1,0,0,0) \in \mathbb{R}^4 \\
a &= (0,0,0,1) && [a_3 += d_{12} a_0] \\
&= (0,1.46,2,0) && [a_1 += d_{11} a_3; a_2 += d_{10} a_3] \\
&= (0,1.46,0,2) && [a_3 += d_9 a_2] \\
&= (2,1.46,0,2) && [a_0 += d_8 a_3; a_3 += d_7 a_3] \\
&= (2,1.46,0.56,0) && [a_2 += d_6 a_3] \\
&= (2,1.46,0,0.56) && [a_3 += d_5 a_2] \\
&= (2.56,1.46,0,0.56) && [a_0 += d_4 a_3; a_3 += d_3 a_3] \\
&= (2.56,1.46,0.34,0) && [a_2 += d_2 a_3] \\
&= (2.56,1.46,0,0.34) && [a_3 += d_1 a_2] \\
&= (2.56,1.32,0,0) && [a_1 += d_0 a_3]
\end{aligned}$$

After this process the updated adjoint vector can be passed onto the next chunk for it to use as seed or, as in this case, the adjoints $x_{1(1)} = 2.56$ $x_{2(1)} = 1.32$ could be harvested from the adjoint vector.

5.2.2 Approaches to Tape Cutting

A first approach to cutting the tape was based on the idea that was developed in the theory Section 4.1. Cutting the tape like this could be achieved due to the data structure being split into sequential access and random access vectors. The initial idea was to handle splitting of the tape in the *dag* class itself. As this class is responsible for recording the *dag*, it tracks all important metadata, like what ids are currently being used, information about the bandwidth, and persistent variables. An additional count variable could be introduced to count all elemental operations that needed recording onto the respective vector. These recording operations are then only ever parsed correctly, when the count variable is in a certain predefined range. This way the tape is always in a consistent state and different partial tapes could easily be reconnected to form the complete tape. The primal is then taped in parallel with different ranges. This approach however has caveats, mainly each primal needs to be started from the initial input to keep the overall metadata, like ids, consistent throughout. This approach does not work well with checkpoints, as taping would require metadata information that are only ever available after computing, resulting in dependencies between tapes. An additional drawback is that the *dag* struct is passively recording the tape and has no way of interrupting the execution after the counter variable leaves the recording range, leading to the primal being executed unnecessarily to completion.

A second approach is directly built on-top of the concept of checkpointing. Each checkpoint splits the primal into a partial primal, that then can be recorded to form a partial tape. These partial tapes are independently recorded from other partial tapes. This however poses the challenge of ids and tape metadata not being consistent. Merging these partial tapes would cause issues like duplicate ids, or having multiple ids for the same persistent variable. The solution here is to stop looking at the global tape and to only ever consider the local tapes. This comes with its own set of problems, as there needs to be a mapping between variables of one partial tape and the next, otherwise the adjoint vector might drift into an inconsistent state. Here the simple solution is to always use the same mapping that is consistent with the store-load order of checkpoints. Checkpoints store all persistent variables, these have to be stored and loaded in the exact same order. A custom store-load system could be adapted and implemented, but was omitted as it would derail the focus of this thesis. An illustration of how the tape ecosystem might look in practice is presented in Figure 13.

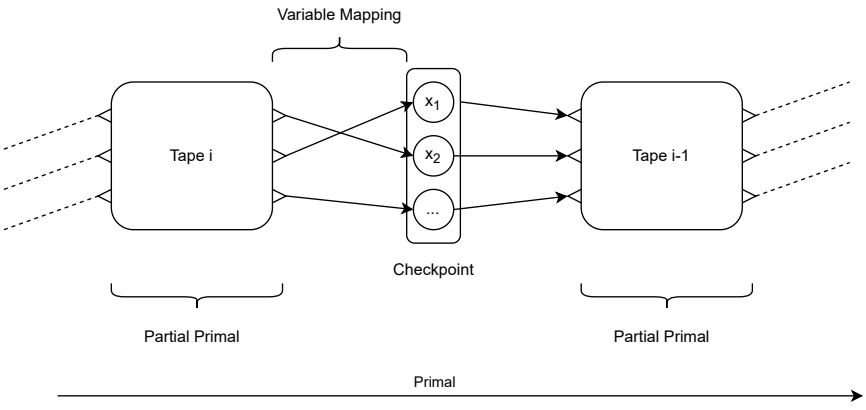


Figure 13: Illustration of the tape ecosystem

Every lvalue needs to be recorded onto the tape, even if it is not needed to compute the final derivatives. As each partial primal only ever has a local view on the entire system it is unaware

of possible dependencies and has to assume the worst case. If necessary checkpoints could be expanded to track this additional information, but this was left out in this thesis. Using the same store-load order the computed adjoint vector is always in a consistent state after each `dag->interpret()` routine. Every persistent variable has a clearly defined id in this adjoint vector, allowing it be used as an input by the next chunk without any additional mapping. Since the method for writing checkpoints relies on explicit checkpointing instructions, the adjoint vector's temporary variable space is empty.

5.2.3 Further Measures for Reduction of Tape Size

As already stated in Section 5.2 the tape structure consists of three distinct vectors, the vertices and dependency information vector v, the derivative vector d and the adjoint vector. The interpretation of the tape works with all three of these vectors but only consumes v and d while computing on the adjoint vector. The struct is fundamentally designed to reduce the size of the adjoint vector, but this comes at a slight size increase in the v and d vectors [12]. Most of the available memory used by the tape is for storing v and d. Depending on the size of the partial primal these vectors can exceed gigabytes in memory. The datatypes for v and d are *int* and *double* respectively. Reducing the derivative vector to only single precision floats would directly impact the precision of the AD tool and is as such not an option. The vertices vector consists of storing the persistent and temporary variable ids as well as the number of predecessors a node has. The tape bandwidth (b) is defined as the "longest" edge between two temporary variables. Mathematically expressed as[12]

$$b = max\{j - i | (v_i, v_j) \in E \wedge v_i, v_j \in V_{temp}\} \tag{23}$$

where $V_{temp} = V \setminus V_{persistent}$.

These bandwidths can also be seen visually on the following two examples where Listing 4 has a bandwidth of 1 and Listing 5 a bandwidth of 2. Additional derivation and graphs can be seen in Figure 28 and Figure 29 in Appendix A.5.

Listing 4: Example 1

```
void f(vector<T> x) {
  x[0] = x[1] + (x[2] + x[3]);
}
```

Listing 5: Example 2

```
void f(vector<T> x) {
  x[0] = cos(x[1]) + cos(x[2]);
}
```

Together the sum of persistent variables and tape bandwidth result in the size needs of the adjoint vector. The mapping of node id to position in the adjoint vector is taken from [12] and is as follows:

$$adj_{id}(x) = \begin{cases} -x - 1, & \text{if } x < 0. \\ |V_{persistent}| + (x \ mod \ b), & \text{otherwise.} \end{cases} \tag{24}$$

As temporary variables are created for each expression, this can quickly become a problem. If the number of temporary variables $|V_{temp}| > 2^{31}$ there are no more unique ids available. Looping around to -2^{31} is not an option as the negative space is reserved exclusively for persistent variables. Looping back to 0 is possible. However, if implemented carelessly this can also cause an issue. Since the internal mapping to the adjoint vector is based on bandwidth $(x \ mod \ 2^{31}) \ mod \ b \neq x \ mod \ b$. But if $x \ mod \ b = 0$, then doing a loop around with the incremental counter and setting it to 0 is okay. This only works if the tape bandwidth $\ll 2^{31}$,

as that would imply that already assigned temporary variable ids can be reassigned if there is at least a gap of size bandwidth. But the bandwidth is initially unknown. The bandwidth value is computed as the tape is recorded and the final value is only ever "final" at the end of tape recording. As a loop around is necessary to even be able to tape large primals, an arbitrary threshold value has to be chosen. It is assumed that after this amount of rvalue computations the bandwidth has reached a stable value. Depending on the primal this may not be the case and could cause errors and wrong derivatives, which the verification system (Section 5.6) should be able to catch and the primal can then be changed accordingly. As such the following loop around is implemented:

```
if (counter > threshold && counter % bandwidth == 0) counter = 0;
```

Looking back on the values that are stored in the vertices vector, it mostly consists of very small values. As such it is ambitious, but worth it, to reduce the vector datatype from int (32 bit) to a short (16 bit). This alone reduces the tape size by about $\sim 50\%$, and reduces the interpretation time as well as the taping time significantly. But this also imposes new limits on the primal. Restricting it to have at most 2^{16} persistent variables. With each single computation being at most dependent on 2^{16} other variables and the threshold variable being set at $\ll 2^{16}$. This is a non-neglectable cost and depending on the primal might not be possible, for example when dealing with large matrix operations. That is why in the `dag.hpp` a `typedef id` is provided. This allows changing of the vertices vector datatype and if necessary even allows for extending the number of persistent variables allowed, by setting the datatype of *id* to *int*64_*t* for example. The threshold for the loop around of temporary ids can also be individually defined in `dag.hpp`.

5.3 Adjoint AD Routine

As discussed in Section 4.2, the tape is split up into chunks and each chunk is part of a so-called *task*. These tasks include finding the starting point of the chunk, recording the tape, performing a local reversal, and finally passing the resulting adjoint vector to the next task. Realizing these tasks is the main idea behind the parallel loop. These are then equally, and in-order, divided onto the available system threads using OpenMP parallelization. How this is implemented in the `aad` routine will now be explained using the provided excerpt in Listing 6.

Listing 6: Excerpt of the Adjoint AD (aad) routine

```
#pragma omp parallel for ordered schedule(dynamic, 1)
    for (int64_t i = chunks; i >= 1; --i) {
        checkpoint check;
        /*
         * Checkpoint for chunk i
         */
        c.getCheckpoint(i, check);

        /*
         * Overloading run
         */
        dag* g = new dag();
        primal(check, g);
#pragma omp ordered
        {
            /*
             * Ordered reversal run
             */
            g->interpret(adjoints);
        };
        delete g;
    }
```

First a checkpoint is initialized (line 3), this checkpoint is then obtained depending on the selected checkpointing method used (line 7), which is explained in more detail in the next section. The checkpoint also contains information about the length of the tape. A `dag` is initialized, this is the *DAG* structure the tape is recorded onto. The tape recording is then started, when running the primal with the `dag` and the checkpoint (line 12-13). This is the most expensive part of the computation of each task and is ensured to be done concurrently. Using the ordered paradigm, the loop is turned sequential, and the threads are put to sleep to await their turn to interpret the tape. The tape interpretation requires the shared vector of adjoints (line 19). After the interpretation is complete the task can free its memory and the thread may now go onto the next task (line 21).

Listing 7: Excerpt of the adapted aadMulti routine

```
1  [...] // left out for brevity
2
3    /*
4    * Ordered reversal run
5    */
6  #pragma omp parallel for default (none) shared(adjoints, g)
7  for (int j = 0; j < adjoints.size(); ++j) {
8      g->interpret(adjoints[j]);
9  }
10
11 [...] // left out for brevity
```

A small adaption to the `aad` routine is the `aadMulti` routine. When the entire Jacobian is of interest, or there is need to differentiate the same function with multiple different seeds, this extension is useful. It allows to input a set of adjoint vectors, that will all then be computed concurrently. This is numerous times faster and more efficient then if `aad` where to be called multiple times. Making use of efficient parallelism, as the tape only needs to be recorded once for multiple different adjoint vectors. This is possible, due to how the data structure is constructed, storing all active reversal information only in these adjoint vectors. As such it is possible to parallelize the independent interpretations, an adopted code can be seen in Listing 7. For example: $g : \mathbb{R}^n \to \mathbb{R}^3$ A simple naive approach would need five threads per `aad` routine to be as fast as possible in differentiating g, as there are three outputs that would amount to 15 threads needed. `aadMulti` can achieve the same result with just five base threads for taping and two threads for additional parallelism in the reversal.

5.4 Checkpointing Implementation

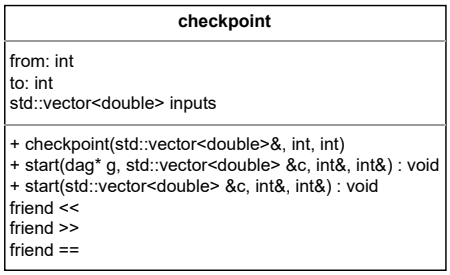


Figure 14: Checkpoint Class

The checkpoint class, outlined in Figure 14, is responsible for storing and loading the state of the primal. When a new checkpoint object is created, the variables *from* and *to*, as well as, the current persistent variables, given in a vector, are stored. Restoring is done in a rudimentary way, by a range (*from*, *to*) of the primal. These values are then read out by the primal when `start()` is called, to know in what range to execute. This sort of structure has limitations and does not apply to all use cases; requiring knowledge of loop iterations at compile time and as such may not always be applicable, for example when doing newton iterations up to a certain accuracy. In this chunking

approach, the division of labor depends on dividing-and-conquering these ranges. It is best to look at how the provided examples are adopted in the source code, or in Appendix A.1.1 at the user and developer guide for more information on how to write primals. When `start()` is additionally called with a pointer to a tape object (*dag*) the stored vector of persistent variables is recorded onto the tape.

A checkpointing interface (Figure 15) was created to be able to test and experiment with different checkpointing methods easily. Modeling the performance of some methods, like the uniform disk offloading, accurately can be difficult; as well as measuring overhead introduced by creating thread safe routines. Any `checkpointingLoader` provides the same basic functionality. Firstly, initiating the `checkpointing` object and performing any pre-`aad` calculations. Secondly a `getCheckpoint` method that is thread safe. One can pass a checkpoint using call-by-reference and a checkpoint id. The checkpoint will then be populated with the values belonging to checkpoint id. When selecting the checkpointing method, it needs to be provided in a namespace with `namespace::checkpointingLoader` addressing a implementation of `checkpointLoaderInterface`. Two important models that were implemented are the uniform disk offloading of Section 4.3.2, from now on referenced as Disk, and the binary bisection of Section 4.3.3, from now on called Memory.

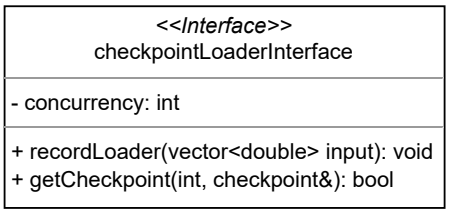


Figure 15: Interface

The Disk method works by creating files that store all the checkpoints. This way later populating the checkpoint can be done in $\mathcal{O}(1)$. For this the output and input operators («, ») where added to the checkpoint class. Using the available threads, the recording is done concurrently using task local I/O. Each thread gets assigned an equally sized consecutive portion of checkpoints. To prevent overwriting issues that arise when running multiple AAD instances simultaneously, every newly initialized Disk-`checkpointLoader` creates a random identifier (runID) to prepend to the file name of all its generated files. Each thread also gets a unique number that represents the start of their respective block of checkpoints (blockStart). Files are then created in a data folder. An example file name could look like this: `run-998575-data0.ch` (`run-{runID}-data{blockStart}.ch`). This number combined with the generated identifier allows for later retrieval of these checkpoints. The `getCheckpoint` method given in pseudocode can be seen in Listing 8.

Listing 8: Pseudocode of the Disk getCheckpoint implementation

```
Barrier: Wait till all checkpoints with a higher ID have been requested
if (currentFile open)
        checkpoint = currentFile.getNextCheckpoint()
else
  if (currentFile == lastFile)
    return false
  else
    currentFile = nextFile
    checkpoint = currentFile.getNextCheckpoint()
return true
```

The Memory method does not require any precomputations and does the required legwork during the main loops execution. It manages a stack that contains the stored checkpoints. The first checkpoint is always the starting checkpoint. The maximal size of this stack can

be hard coded, but for most instances it is just left unbounded. In most scenarios this stack will not exceed 30 (total of 2^{30} checkpoints) checkpoints, for which most systems can spare the memory. Whenever a new checkpoint is requested Listing 9 `getCheckpoint` method is executed.

Listing 9: Pseudocode of the Memory getCheckpoint implementation

```
Barrier: Wait till all checkpoints with a higher ID have been requested
if (myID < stack.peak().id or stack.isEmpty())
  return false
if (stack.peak().id == myID)
  checkpoint = stack.pop()
else
  checkpoint = record_to_stack_and_provide_myid(from stack.peak() to myCheckpoint)
  if (stack.peak().id == myID)
    stack.pop()
return true
```

Where the routine `record_to_stack_and_provide_myid` starts from the last checkpoint in the stack and records all checkpoints that are half way between the last checkpoint and the requested checkpoint. Once a new checkpoint gets added the last checkpoint also gets updated and the cycle continues recursively.

5.5 Timing and Profiling

To create a reliable and comparable timing environment, it is first necessary to clarify what is to be timed. In the case of this AAD tool the actual `aad` routine provides the core of the program and will be timed. To accurately measure timings the `C++` Chrono library[5] is used internally, with millisecond-accuracy. Statements to begin timing and end timing are injected into the `aad` routine directly. There are many elements that could be individually timed, but to keep focus only four sections are timed. Those include the main `aad` routine from start to finish (t_{total}). The initialization time the selected `checkpointLoader` needs, this is relevant for loaders like the Disk model that require processing prior to the main adjoint loop (t_{pre}). Another important time is the reversal or interpretation time (t_{reversal}). Once the first checkpoint is successfully taped, seeding starts. The t_{reversal} time counts from the seeding point towards the end of the `aad` runtime. The t_{reversal} also includes another important time, the reversal idle time (t_{idle}). The t_{idle} is defined as the time where no active reversal is processing, this may be because the next tape is not available yet. The goal of parallelization is to minimize t_{idle} as far as possible by always having tape ready when the reversal of a prior tape finishes. As such it is an important time measurement when trying to determine potential time savings. Lastly the time it takes to acquire the first checkpoint and to record its tape is also measured (t_{first}). With all these measured times the following equation should always hold.

$$t_{\text{total}} = t_{\text{pre}} + t_{\text{first}} + t_{\text{reversal}} \tag{25}$$

And it also allows estimating the, with this method, best achievable time (t_{best}), that is when $t_{\text{idle}} = 0$.

$$t_{\text{best}} = t_{\text{pre}} + t_{\text{first}} + (t_{\text{reversal}} - t_{\text{idle}}) \tag{26}$$

[5] https://en.cppreference.com/w/cpp/chrono

This time may not be achievable, as context switching, and thread management overhead will lead to $t_{\text{idle}} > 0$. Also, multiple runs may produce slightly different timing results, even in the exact same environment, as such when benchmarking, scenarios are run multiple times. To be able to acquire the timing output of an AAD run, the *DEBUG* flag needs to be set to either *Verbose* or *Timing*. After enabling the timing, outputs will be flushed to cout. An example output can be seen below.

```
Checkpoint Generation Time: 0
Chunk One Overloading Time: 777
Total Reversal Time: 12835
Reversal Idle Time: 2217
Total Execution Time: 13614
```

The optional profiler provides useful debug data. Part of this data is the System Environment, like what operating system it is currently being run, but also data about the available memory or the useable CPU threads. Next to the environment it also provides rudimentary benchmarks and timing information about the primal. For example, the t_{primal}, $t_{overload}$, and ρ (overload factor) as well as reversal time of a single partial tape. Volume information like the size of a single checkpoint or the entire chunk window are also being estimated. The total data volume needed for differentiation is also provided, this value is likely many times higher than the available memory, as the tape is only loaded and unloaded consecutively. All this information can be useful to adjust the parameters for differentiation. The profiler also provides rudimentary optimization suggestions. One of those is the ratio by which the *windowSize* can be in/decreased to exactly fit the available system memory. However, this does not mean that these optimizations are useful, since they were created before the benchmarks. In the case of *windowSize* it turned out it mostly does not matter if it is fully utilized or not, as checkpointing schemes using rematerialization can account for this quite well. Another more relevant datapoint is the efficiency factor. It is the ratio between how fast tapes are made available versus how long the reversal needs for a singular tape. This is quite important as a value above one will increase the t_{idle} significantly, while a value below one suggests that the given level of parallelization is more than enough for just-in-time taping. An example profiler output can be seen below.

```
OS: Windows
Cores: 8
Mem: 17GB
Running Primal took: 1078ms
Checkpoint Size: 2464 Byte
CheckpointTapeTime: 209ms
Overload Factor: 46.53x
Used Memory: ~1043 MB(14.77x)
Total Tape Size: ~31 GB
CheckpointReversalTime: 74ms
EfficiencyFactor: 0.40 (Optimal: 4 cores)
Estimated Execution Time: 18.838ms
```

5.6 Verification and Testing

Verification is important in any application that produces and processes sensitive data. As AD programs are heavily built and relied upon in industrial applications, the outputs may have a significant impact on decision making. For example, used in climate modeling and atmospheric chemistry or when modulating radiation therapy[6]. As such it is of enormous interest and importance to know if the produced output is correct. For simple programs this may be done symbolically, and the outputs verified with the AD program. But this

[6] https://www.autodiff.org/?module=Applications&category=all

approach quickly becomes infeasible for large primals, as the reason for using the AD program is to omit the expensive human work of differentiating by hand. Another approach is formal verification. This approach works well for small programs and algorithms but as soon as a complexity is reached like multithreading, vast object orientation or recursion it also becomes infeasible. Since AD is a mathematical process, there are ways to check the input and output and determine if the results are likely to be correct. Not only can such a method be used on any arbitrary primal but it can also provides a useful debugging tool.

Using the software the adjoints $x_{(1)}$ can be calculated. From Equation (6) it follows

$$x_{(1)}^{\intercal} = y_{(1)}^{\intercal} \nabla \mathcal{F} \tag{27}$$

and the tangents follow from Equation (2)

$$x^{(1)} = \nabla \mathcal{F}^{-1} y^{(1)} \tag{28}$$

the following equality holds

$$x_{(1)}^{\intercal} x^{(1)} = y_{(1)}^{\intercal} \nabla \mathcal{F} \nabla \mathcal{F}^{-1} y^{(1)} = y_{(1)}^{\intercal} y^{(1)} \tag{29}$$

Using a randomized tangent $x^{(1)}$ to seed the finite difference approximation, allows harvesting of $y^{(1)}$ [13].

$$\frac{f(x + \Delta x \cdot x^{(1)}) - f(x)}{\Delta x} \approx \nabla \mathcal{F} x^{(1)} = y^{(1)} \tag{30}$$

where

$$\Delta x \coloneqq \|x\| \cdot 2^{-32} \tag{31}$$

Thus all the variables are available and the equality in Equation (29) should hold if the adjoint program produces the correct output of $x_{(1)}$. Using this technique the output can be verified. To account for rounding and approximation errors, when dealing with floating point operatations, the following equility ist tested in the adjoint program. Where ϵ is the desired numerical accuracy.

$$|x_{(1)}^{\intercal} x^{(1)} - y_{(1)}^{\intercal} y^{(1)}| < \epsilon \tag{32}$$

This however is rather a necessary condition and not a sufficient condition to prove correctness. To give more weight to the result of this calculation, it is performed dozens of times with random seeds. This is implemented in *verify.hpp*.

There are also other aspects of the adjoint program that required testing and verification. These are done by rudimentary unit tests that cover most of the important aspects of program operation. These are implemented on-top of the GoogleTest[7] framework. These tests include, but are not limited to, writing and reading to checkpoints, writing and reading from disk, testing if elemental operations and derivatives of the overloaded double datatype are performed correctly, and if the program produces the same results as a symbolic derivative done by hand.

[7] https://github.com/google/googletest

6 Results

In this section the adjoint program is tested and benchmarked. Two hardware environments are introduced, as well as two distinct primal functions. For each environment and each primal a benchmark is created to test the performance and speedup that the implemented parallel taping algorithm provides. These results are then discussed and summarized.

6.1 Environment

The environment in which results are obtained is crucial to give them any significant weight. It allows for replicability and may show weaknesses in the used methodology. A distinction is made between the external and internal environment. The external environment is the hardware on which the software and benchmarks are being executed, and the internal environment is the primal of which the first order adjoint is to be computed as well as its parameters.

6.1.1 External Environment

A distinction between two runtime environments is made; detailed specifications can be seen in Table 2. The RWTH Compute Cluster[8] (abrv. HPC) is comparable to an industrial high-performance machine and is likely above what most users have available. On the other hand, the consumer grade personal computer (abrv. PC) is, as the name suggests, of average achievable performance. For the benchmarks and tests on the HPC environment the SLURM system was used, more information can be found in Appendix A.3.

	RWTH Compute Cluster (HPC)	Personal Computer (PC)
Operating System	Linux (CentOS 7.9)	Windows (Windows 10)
C++ Compiler	icpc version 19.0.1.144	gcc 7.2.0
CPU	Intel Skylake Platinum 8160	Intel(R) Core(TM) i7-4770
System Threads	48	8
Useable Memory	144 GB	16 GB
Disk	Lustre (High Performance)	SATA HDD

Table 2: Overview and specification of the different environments

While "Lustre" is not a hard disk, but rather a distributed file system, it is more than the sum of its components, as it works similarly to RAID 0 on an even larger parallel scale.

6.1.2 Internal Environment

The first order adjoint of the numerical solution to the Burgers' equation [2] is considered as the primal input for the AAD software. The reason for choosing this is that it is a suboptimal problem for AAD with rather complex internal logic, leading to a tape that tends to get quite wide and large. It also provides two parameters that can be used to arbitrarily scale the difficulty of the problem. Firstly the input vector length to scale the width of the tape

[8] https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/

and secondly the newton iterations (size of the main loop) to scale the length of the tape. The original code is taken from the "Aachener Informatik-Berichte" of `dco/c++`[19], where a detailed problem analysis is provided. The adapted code can be found in the git[9] under *examples/burgers* and will be referenced as "burgersFunction".

The second primal input is Listing 1, the code presented in the beginning of this thesis. From now on referenced as "example2". The outer loop can be increased to inflate the difficulty of the problem, leading to a tape that is slim and large in length. As this is just an example, generated to showcase AAD application, it does not provide any real meaning or useful results. Nonetheless it was included as it contrasts the burgersFunction well.

Many parameters influence the AAD programs performance on any given primal. The AAD input vector and the *size* parameter have a direct effect on the output and define the AD problem. Internal parameters that do not influence the results, but impact the runtime nonetheless are: The $windowSize < size$, a user-set variable that represents the amount of loop iterations that, when recorded onto the tape, still fit into main memory (defines the memory use). Together the *size*, *windowSize* and the respective amount of CPU threads (*cores*), are used to calculate the tasks (chunks $= \frac{\text{size} \cdot \text{cores}}{\text{windowSize}}$). Latstly, the checkpointing model that is used to record and derive checkpoints.

6.1.3 Preliminary Benchmarks

Compiler	-O0	-O3
gcc 11.2.0	157,235	8,461
clang/12	111,628	8,971
intel/19.0	221,110	8,429

Table 3: [HPC/10] C++ compiler optimization impact on execution time [ms]

The used compiler can play a significant role in achievable execution time. Different compilers may make use of the underlying hardware to a different extent. For example, the intel `C++` compiler[10] may make better use of the intrinsic CPU vector operations (SIMD) than the general purpose gcc `C++` compiler[11]. Also, the implementation of the OpenMP specification may vary in quality. A detailed analysis of this can be seen in Table 3, which was obtained by compiling the same problem with different compiler and optimization settings and averaging the runtime results. The unoptimized version is about a factor ~ 26 slower than the optimized version. The differences between the compilers in the optimized version are negligible. For the following tests the compiler optimization flag (-O3) is set. The reason for testing in two very different environments (HPC - PC) is to show the overall speedup trend that is coherent throughout.

[9] https://gitlab.stce.rwth-aachen.de/hesse/parallel-taping-adjoint-ad
[10] https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html
[11] https://gcc.gnu.org/

	burgersFunction		example2	
	HPC	PC	HPC	PC
ρ (overload factor)	75-87	41-61	3.8-4.7	2.5-3.6
checkpoint size	2464 B		64 B	
total windowSize	880 MB		2 GB	
total tape size	26 GB		16 GB	
optimal parallelism	5-7	4-6	4-6	5-7

Table 4: Profiler output

An important tool that was created alongside the AAD routine, is the profiler (Section 5.5). The profiler can provide interesting insight about what to expect and provide additional information. In this case some of the more useful results were collected in Table 4. Both problems seem to be memory intensive, especially the burgersFunction which has to cycle through 26 GB of tape. The ρ factor is striking, as it is quickly noticeable that there are massive differences between the burgersFunction and example2. The provided burgersFunction is quite computational expensive and has complex internal logic. In contrast example2 is simple, as such it has a way smaller ρ. The impact of compiler optimization can also be seen, even though overall runtime is similar the overload factor is different from HPC to PC.

	Load		Store	
n	HPC	PC	HPC	PC
3	0.204ms	0.845ms	0.003ms	0.012ms
33	1.72ms	7.25ms	0.03ms	0.066ms
303	16.3ms	69.6ms	0.306ms	0.9ms
3003	159ms	670ms	3.52ms	6.83ms

Table 5: Uniform disk offloading, store and load speeds per checkpoint [ms]

Additionally Table 5 shows timed loading and storing of checkpoints from non-volatile storage. n denotes the length of the vector that is stored in these checkpoints. For the burgersFunction checkpoints of length 303 are used while example2 only uses checkpoints of length 3. What can be taken from these results is the suboptimal loading time in both PC and HPC. While storing data can be handled efficiently, loading data requires reading a file backwards, which is not the fastest. For $n = 3003$ a single checkpoint can take up to 670ms in a routine that is not parallelized. How this effects the AAD program when using the Disk model will be shown in the next section.

6.2 Benchmarks

Benchmarks performed by the environment y and averaged over x runs will be denoted by [y/x]. So for example in Table 3 the results were obtained using the HPC cluster and averaged over ten runs. The tupel (*size*, input length, *windowSize*) represents the configured *size*, input vector length and *windowSize*.

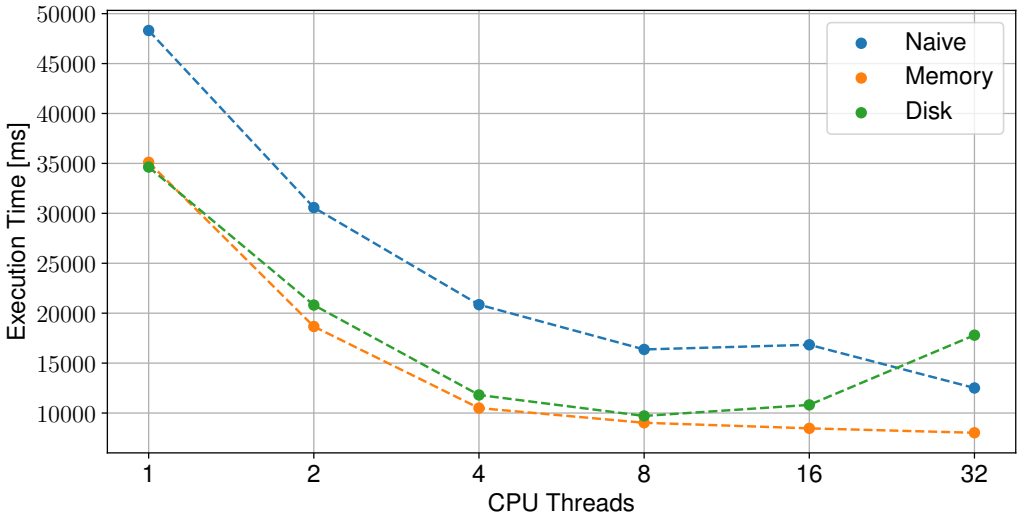


Figure 16: [HPC/20] (30k,303,1k) burgersFunction

The graph seen in Figure 16 has been computed on the HPC environment running the burgersFunction with the following input values *size*=30k, input vector length=303, and *windowSize*=1k. The total execution time is plotted on the y-axis and the system parallelism is plotted on the x-axis. Three different checkpoint models were tested. The Naive method presented in Section 4.3.1 the Memory method of Section 4.3.3 and the Disk method illustrated in Section 4.3.2. The results were averaged over ten runs and then plotted. There is a clear performance increase when adding more CPU threads. A major runtime decrease can be seen within the first eight threads. The best performing checkpointing model is Memory, closely followed by Disk. This is not at all surprising as the Naive model is, as the name implies, rudimentary. Nonetheless every model greatly benefits from added concurrency. What is unanticipated however is the increase in execution time of the Disk model from 8 threads to 32. This can be explained with the large input vector and as such the rather large checkpoints that need to be stored and loaded from non-volatile storage. With an increase in threads the overall checkpoints increase, while their respective execution range decreases. This causes rapid loading of checkpoints to be a noticeable bottleneck. As such the Disk model seems to be suboptimal, for the burgersFunction with these input parameters.

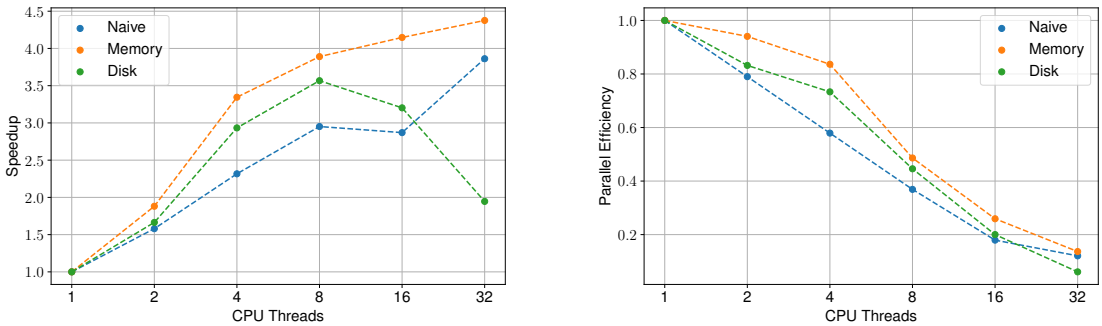


Figure 17: Speedup and Parallel Efficiency of Figure 16

Figure 17 shows the speedup ($\frac{T(1)}{T(n)}$) and the parallel efficiency ($\frac{T(1)}{n*T(n)}$) of Figure 16. A speedup of > 4 was achieved using the Memory model. With the trendline indication that even more threads would result in a higher speedup. Looking at the parallel efficiency however, one can see the rapid and static decline of return on investment.

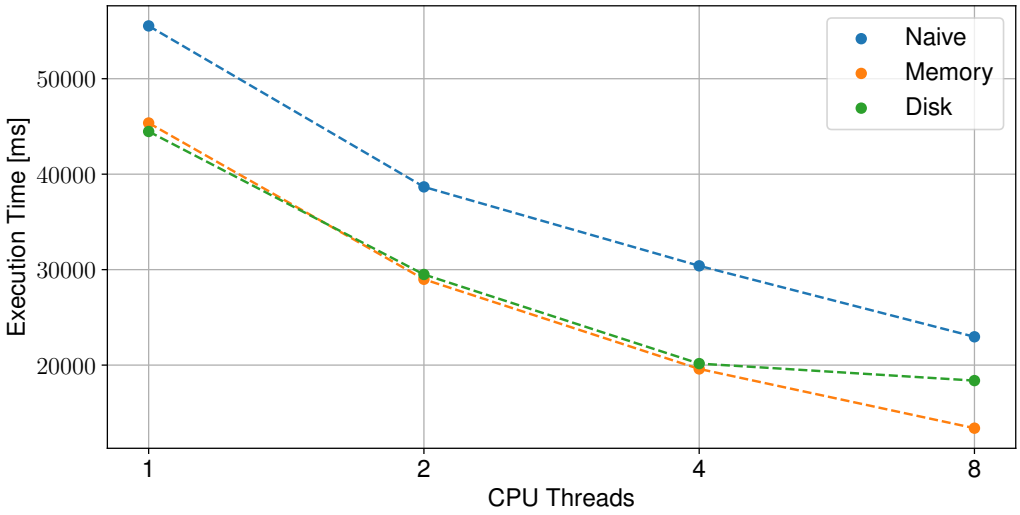


Figure 18: [PC/10] (30k,303,1k) burgersFunction

Executing the same tests on the PC environment produces similar looking results (Figure 18). Overall, the execution times are worse than in Figure 16, due to the hardware being less powerful. Though trends in speedup and efficiency are comparable. Except for the Disk model, which already seems to be falling off at 8 cores, compared to the Memory model. This is likely due to the significant differences in non-volatile data handling in the two environments. Comparing the relative results of Figure 18 to Figure 16: In the serial case using the Memory model, the PC executed about 29% slower then the HPC. In the parallel case using the 8 threaded Memory model, this was increased to 58%. The speedup factor of the 8 threaded Memory model is 3.38 which is about the same as 3.89 in Figure 17.

Further tests were made with the example2 function taken from Equation (1), here the outer loop is scaled up to produce timeable results. The graph in Figure 19 shows the execution time per core per checkpointing model. Compared to previous tests the x-axis showing the threads is filled with all options ranging from one to eight cores. The first noticeable difference is that the graph flatlines very fast and that the Disk method outperforms the Memory method by quite a margin. This can be explained by the small number of double precision floating point operations per loop iteration. As such computing the loop variable is a significant part of computation, which can be massively reduced using the Disk method, as many parts of this computation needed to be only done once. Overall, the trend seems to be in line with that of Figure 16. One additional thing to notice are the small hiccups, at for example Memory(4) or Naive(5). This can be explained with the an uneven range of the last checkpoint compared to the other ones. If the $size/(windowSize/cores)$ is not a integer value, the last checkpoint is reduced in execution range. This can either be beneficial or not (depending on the checkpointing method used), as the reversal can start quicker.

To illustrate the inner workings of the AAD program, a stacked chart was created (Figure 20). The different sections are derived from the timings of the main adjoint loop, explained in more

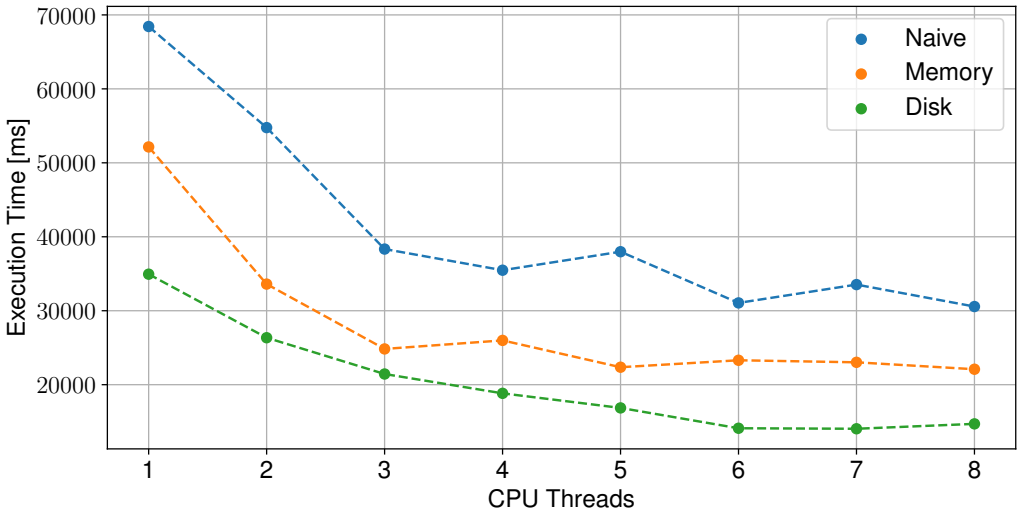


Figure 19: [HPC/10] (300mio,3,40mio) example2

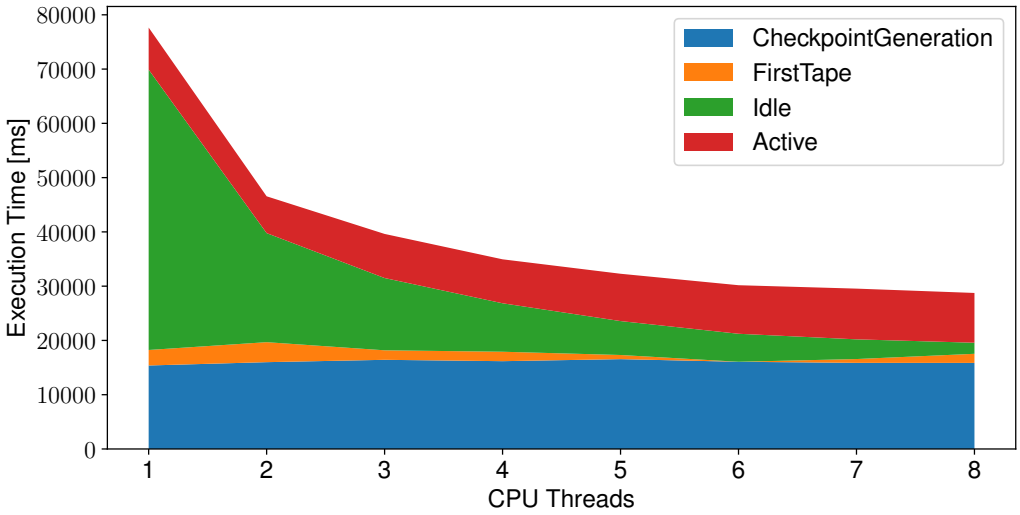


Figure 20: [PC/5] (300mio,3,40mio) example2

detail in Section 5.5. The AAD input was the Disk model on the PC environment with the parameters as in Figure 19. The checkpoint generation time (t_{pre}) refers to the time it takes to create all the checkpoints and write them to non-volatile storage. This data, supported by the theory, suggests that this is roughly a constant time and independent of thread count. The active time ($t_{\text{active}} = t_{\text{reversal}} - t_{\text{idle}}$) represents the time the reversal needs to fully interpret the whole tape. This should, in theory, also be independent of the number of threads. The t_{idle} is the accumulated time where the reversal needed to be interrupted, as data was missing to continue interpretation. t_{idle} is massively reducible using parallelization. As seen in the graph Figure 20, t_{idle} is reduced and as such the whole execution time goes down. This also provides a limit to the scalability, as t_{idle} approaches zero the benefits of adding more cores goes down. Lastly, the first tape value reflects the time needed to fully tape the last checkpoint. Only after first tape and t_{pre} can the reversal start. The first tape value is not stable and changes

minimally depending on the number of threads. This is because, the last checkpoint can differ in range, as mentioned before. These minimal fluctuations are probably in part responsible for the hiccups noticed in Figure 19.

6.3 Discussion

These benchmarks underline the theoretical results (Section 4). Overall, all methods benefitted greatly from added parallelism. Achieving speedup factors of as high as > 4. Even, with just two threads, the limited PC hardware outperformed the serial version running in the powerful HPC environment. Different scenarios showcased the strengths and weaknesses of chosen checkpointing models. While the Disk model fell off in certain scenarios, likely due to the bad checkpoint load times as tested in Table 5, it could overall compete with the Memory model quite well. The same cannot be said about the Naive approach, which as the name suggests, did perform quite underwhelming compared to Disk and Memory. However, scenarios could be constructed to play to its strengths of being lock and synchronization free.

Additional benchmarks (Appendix A.4) were created to showcase different aspects of the companion tool. In the above scenarios the main memory was capped at 1 GB (example2) and 880 MB (burgersFunction). Different checkpointing models are designed to make use of memory limited environments by employing rematerialization concepts. Limiting the *windowSize* does show a significant performance impact, most notably in the Naive approach. But also, the Disk model falls off when the time taken to retrieve a single checkpoint is longer than the taping and reversal time combined, as seen in Figure 22. Figure 24 shows the loglinear relationship between number of checkpoints (n) and runtime in the Memory model, the quadratic relationship for the Naive loader, contrasting the linear dependency on n in the Disk model.

It must be stated that these results, while conducted to the best of the authors ability, do contain some bias. The companion AAD tool, with which these benchmarks were obtained, was specifically designed to showcase the benefits of parallel taping. From the chosen tape structure to the main AD loop. As such comparing the parallel to the serial version might not be fair. The limited number of tested environments and primals adds to this. It was not possible to test the here achieved results and timings against state-of-the-art tools like `dco/c++`, as these are optimized in other ways that are out of the scope of this thesis. However, the results that were achieved, are in line with theoretical considerations as well as similar research [6, 9]. To make it as easy as possible to do further research and tests the companion tool will be fully open sourced and is available publicly in the RWTH git[12] and GitHub[13]. A detailed user-guide can be found in Appendix A.1.2.

[12] https://gitlab.stce.rwth-aachen.de/hesse/parallel-taping-adjoint-ad
[13] https://github.com/y-hesse/parallel-taping-adjoint-ad

7 Conclusion

Summary

The overall goal of this thesis was to devise, test, and analyse approaches for parallel taping in adjoint Algorithmic Differentiation. In a first theoretical approach this was achieved by making use of the fact that recording a primal is several times slower the running the primal directly. This led to splitting the primal into partial primals, which then could be taped concurrently. A promising first avenue, already achieving substantial speedups. However, the initially prototyped method quickly became unusable as it was introduced to a memory-bound environment. Using checkpointing schemes and just in time taping, this challenge was overcome. A parallel adjoint solver, written in `C++`, was implemented and the proposed theoretical algorithms adapted. Next to the main design concepts, also quality of life features were explained. Finally, the AAD tool was extensively timed and tested in different environments with different primal inputs. Speedup factors of > 4 were achieved. Even with comparably little added concurrency, these results are very promising.

Outlook

This thesis already shows significant improvements to the traditional serial approach of adjoint AD. Here it was not only shown that it can work theoretical but also practical implementations and benchmarks were performed. With even just one additional thread for taping the improvement was already significant. This section will try and provide an outlook as well as ask research questions that were left unanswered.

Using MPI would allow the AD program to run on more than just one machine and enables harnessing the power of an entire compute cluster. When using the minimized tape structure, only the adjoint vector would need to be passed through the network. As such it may look like a promsing next step, but as the benchmarks and test have shown the speedup already approaches a limit, even on just a single machine. It can be doubted that massively increasing the concurrency is a worthwhile investment.

State of the art tools like `dco/c++` are used in many industrial applications[14]. Combining the here researched results and algorithms directly into `dco/c++` could be promising. This would ideally allow users to add a single line of code, which enables parallelization and speeds up their runtime significantly.

The parallel method implemented in this tool can accumulate a lot of "unproductive" time, when the available cores are idly waiting for their turn to start the reversal process. This time could be used to perform tape reduction techniques that keep the mathematical structure intact, while reducing the work of the reversal, by accumulating edges and removing vertices. A simple and non-optimized vertex elimination method has been implemented in a similar parallel taping approach[9], but their research shows that this method has a negligible effect on reducing overall runtime.

The, in this thesis, proposed and implemented checkpoint schemes have been shown to be suboptimal[6]. In the conducted benchmarks different scenarios showed weaknesses in all the proposed methods. Exploring hybrid checkpointing techniques, that are effectively parallelizable, could be promising. Overall parallel taping has very real limitations regarding

[14] https://www.nag.com/content/algorithmic-differentiation-software

achievable speed up. As such, combinations with other parallel AD approaches should be explored. This thesis omitted looking into parallel back-propagation. With the way typical tapes are structured, making use of repeating patterns and independent primal sub routines could be a promising avenue for parallel reversal approaches.

Research regarding the parallel taping in adjoint AD is just in its infancy and this thesis and others like it show its very real potential.

III References

[1] Christian Bischof et al. "Parallel Reverse Mode Automatic Differentiation for OpenMP Programs With ADOL-C". In: *Advances in Automatic Differentiation*. Springer, 2008, pp. 163–173.

[2] Johannes Martinus Burgers. "A mathematical model illustrating the theory of turbulence". In: *Advances in applied mechanics* 1 (1948), pp. 171–199.

[3] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry Standard API for Shared-Memory Programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[4] Andreas Griewank. "Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation". In: *Optimization Methods and software* 1.1 (1992), pp. 35–54.

[5] Andreas Griewank, David Juedes, and Jean Utke. "Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++". In: *ACM Transactions on Mathematical Software (TOMS)* 22.2 (1996), pp. 131–167.

[6] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008, pp. 261–298.

[7] William Gropp et al. *Using MPI: Portable Parallel Programming With the Message-Passing Interface*. Vol. 1. MIT press, 1999.

[8] Laurent Hascoet and Valérie Pascual. "The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification". In: *ACM Transactions on Mathematical Software (TOMS)* 39.3 (2013), pp. 1–43.

[9] Jannick Kremer. "Parallel Adjoint Taping Approaches With OpenFOAM". Bachelor's Thesis. RWTH Aachen University, 2022.

[10] Koichi Kubota. "A Fortran77 Preprocessor for Reverse Mode Automatic Differentiation With Recursive Checkpointing". In: *Optimization Methods and Software* 10.2 (1998), pp. 319–335.

[11] William S Moses et al. "Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–16.

[12] Uwe Naumann. "Reduction of the Random Access Memory Size in Adjoint Algorithmic Differentiation by Overloading". In: *arXiv preprint arXiv:2207.07018* (2022).

[13] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2012. ISBN: 161197206X, 9781611972061.

[14] Uwe Naumann, Klaus Leppkes, and Johannes Lotz. *dco/C++ User Guide*. RWTH Aachen, Department of Computer Science, 2014.

[15] Karl Rupp et al. *42 Years of Microprocessor Trend Data*. 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data (visited on 09/05/2022).

[16] Jeffrey Mark Siskind and Barak A Pearlmutter. "Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation". In: *Optimization Methods and Software* 33.4-6 (2018), pp. 1288–1330.

[17] Markus Towara, Michel Schanen, and Uwe Naumann. "MPI-Parallel Discrete Adjoint OpenFOAM". In: *Procedia Computer Science* 51 (2015), pp. 19–28.

[18] Jean Utke et al. "Toward Adjoinable MPI". In: *IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–8.

[19] Johannes Lotz Uwe Naumann Klaus Leppkes. *dco/C++: Derivative Code by Overloading in C++C++*. Aug. 2016. URL: http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2016/2016-08.pdf (visited on 07/20/2022).

A Appendix

A.1 ReadMe

A.1.1 README.md

Parallel Taping in Adjoint Algorithmic Differentiation

Description

This software was created as part of a Bachelor Thesis at the RWTH Aachen University[15].

The goal is to provide a reference implementation for testing and analysis of proposed parallel adjoint taping methods.

This tool is a fully functional Adjoint Algorithmic Differentiation (AAD) tool. That said, it is not designed to be the fastest AAD tool, as such be careful when using and building on top of it.

Installation

Clone the project and use cmake

```
cd {Project Root}
cmake .
make adjoint
./adjoint
```

Getting Started

See the User Guide(Appendix A.1.2) for documentation and a quick start.

If you want to contribute or embed this program in your own we recommend looking into the Developer Guide(Appendix A.1.3).

Features

- Adjoint calculation
- Tools for analyzing performance
- Two methods of parallel checkpoint generation
- Mathematical verification of results
- Profiler
- Checkpointing
- A set of predefined double operators

Documentation

The Software uses Doxygen[16]. Docs can be generated by running the following commands

[15] https://www.stce.rwth-aachen.de/
[16] https://doxygen.nl/

```
cd doc
doxygen ./Doxyfile.in
```

This generates a detailed overview of the classes and descriptions of methods, but it is also helpful to look into the source code. Especially the primal/primal.hpp file, as a lot of control parameters can be defined there.

Testing

The Software uses the Google Test framework[17]. To run the tests build the adjoint_test executable:

```
make adjoint_test
./adjoint_test

or

ctest adjoint_test
```

There may be problems compiling the tests when using icpc. We have found no issues using clang or gcc.

Note: not all functionality is currently tested.

Happy differentiating!

A.1.2 USER_GUIDE.md

User Guide

This software's intended use is to calculate adjoints of algorithmically differentiable C++ routines

This is done by overloading the algorithmic operators such as {+, -, *, /, sin, cos}. This operator list can be easily extended (check the developer guide(Appendix A.1.3)).

Any C++ routine provided needs to be adjusted and created in a separate namespace. Your routine needs to be rewritten to support three different function signatures.

```
template<typename T>
void primal(std::vector<T> &inout);
void primal(checkpoint c, dag* D);
void primal(checkpoint c, std::function<void(checkpoint)> addCheckpoint);
```

The Checkpoint class:

The Checkpoint class provides an API to segment your function. These checkpoints are then used to overload different "areas" of your function concurrently. To make use of this efficiently you need to have a rough estimate of the runtime and loop iterations your function will need.

[17] https://github.com/google/googletest

You also need to be aware of the memory or assisting variables your function needs. These need to be identified and in the worst cased stored in the checkpoints. More details on this step will follow.

The double_o class:

This class represents the overloaded double datatype it provides *most* functionality a double provides and should be used as a drop in replacement for any double's in your code.

Example (original):

```
template<typename T>
void primal(std::vector<T> &input_output) {
    T u;
    for (int i = 1; i < 100; ++i) {
        T u;
        u = sin(x[i-1]);
        x[i] = u*u + x[0];
    }
}
```

Example (transformed):

```
namespace myPrimal {
    template<typename T>
    void primal(std::vector<T> &input_output) {
        for (int i = 1; i < 100; ++i) {
            T u;
            u = sin(input_output[i-1]);
            input_output[i] = u*u + input_output[0];
        }
    }

    void primal(checkpoint c, dag* D) {
        std::vector<double_o> input_output;
        uint64_t from; to;

        // Aquire all the values from the checkpoint
        // and register the inputs on the Graph
        c.start(D, input_output, from, to);

        // It is important to reuse temporary variables by
        // pulling them out of the loop instead of reinitializing them
        // in each iteration.
        T u;
        for (uint64_t i = from; i < to; ++i) {
            u = sin(input_output[i-1]);
            input_output[i] = u*u + input_output[0];
        }
    }
```

```
    void primal(checkpoint c, std::function<void(checkpoint)> addCheckpoint) {
        std::vector<double_o> input_output;
        uint64_t from; to;

        // Aquire all the values from the checkpoint
        c.start(input_output, from, to);

        // It is important to reuse temporary variables by
        // pulling them out of the loop instead of reinitializing them
        // in each iteration.
        T u;
        for (uint64_t i = from; i < to; ++i) {
            for (i % windowThreadedSize == 0) {
                checkpoint tmp_check(input_output, from, from+windowThreadedSize);
                addCheckpoint(tmp_check);
            }
            u = sin(input_output[i-1]);
            input_output[i] = u*u + input_output[0];
        }
    }
}
```

Now to embed the newly transformed primal function into the code we need to add it to be compiled in the cmake file.

And to be used by the software we need to edit primal.hpp and set the size and define our PRIMAL namespace

```
#define PRIMAL myPrimal
...
static uint64_t size = 100;
```

Global external parameters that can be useful when deciding on where to make a cut in your function are:

- size
- windowThreadedSize

Size represents the total loop iterations your function will do

WindowThreadedSize is the maximal size of consecutive loop iterations that fit into your local machines memory when overloading the function.

To then calculate the adjoints simply create a main.cpp file that looks like this

```
#include <aad.hpp>
#include <profiler.hpp>
int main() {
    /*
```

```
     * Initialize the Input Vector
     */
    std::vector<double> in(100, 1);

    /*
     * Initialize the output vector
     * If the Vector is to small it will be autofilled with 0's
     */
    std::vector<double> adjoints = {0,1};

    /*
     * Calculate the Adjoints
     */
    aad(in, adjoints);
    return 0;
}
```

A.1.3 DEVELOPER_GUIDE.md

Developer Guide

Overview

- Adjoint calculation (aad.hpp)
- Optimization / Understanding the profiler (profiler.hpp)
- The DAG / DCG struct (dag.hpp)
- Supporting custom checkpointLoaders
- Supporting more basic Operators (double_o)

Adjoint Calculation

When calculating adjoints via overloading cpp routines there are three basic steps each adjoint software needs to do.

Step1: Record the Tape, this is done by running the primal function with an overloaded double datatype (double_o) this allows the dag to record dependencies between variables and to store a dag that contains the derivatives accordingly.

Step2: Tape reversal, this was not the focus of this project so the reversal function is very primitive. The adjoint vector is seeded with the user's input of the adjoint vector, after that the graph is traversed in reverse order and the derivatives are joined using the chain rule of differentiation.

Step3: Harvesting the adjoints, the tape reversal provides a vector of adjoints.

Aligned with these three basic steps the aad.hpp provides a routine called aad. AAD takes an input vector for the primal and an adjoint vector that is later used to seed the tape reversal.

The focus of this project was to speed up tape recording by utilizing the parallel cpu architecture. That is why the aad method can be run with different tape recording functions. More details about that can be found in the "Supporting custom checkpointLoaders" subsection.

The recording is split in two parts, recording the checkpoints and recording the dag. As this problem is memory bound, there is no way to fit a complete dag of a compute intensive problem into memory. That is why the primal is segmented via checkpoints.

Using OpenMP multithreading Step1 and Step2 are combined. This results in tape recording and tape reversal being done simultaneously on different segments of the primal.

Using this we try to hide the overhead created by recording the tape compared to running the primal.

Be careful when doing changes to this routine as it is the heart of the programm that connects all the separate parts.

Optimization & Understanding the profiler

Running the software in profiler mode can be very useful to get a rough estimate of the runtime the main aad routine will need. It is also useful when looking at parameters like the *windowSize* as they can be increased depending on the primal and device you are running.

An example output of the profiler may look like:

```
OS: Windows 32-bit
Cores: 8
Mem: 17GB
Running primal took: 2544.8ms
Checkpoint size: 8064Byte
Overloading checkpoint took: 998ms
Overload-Factor: 31.3738x
Memory-Factor: 6.11289x
Best-Case time to finish: 86928.8ms
```

Next to the system information like the OS, the usable CPU cores / memory other parameters like the overloading factor are also of interest.

The overloading factor shows how much slower recording the tape is compared to running the primal.

The memory factor shows the factor by which the *windowSize* can be multiplied to optimally use the available memory.

The best case ttf shows the theoretical best achievable time.

One other aspect that should be considered when wanting to improve performance significantly is looking at the compiler and compiler optimization used. More information about this can be found in the thesis performance analysis.

The DAG / DCG struct

The tape data structure can be interpreted as a Directed Acyclic Graph (DAG) and is implemented as a Directed Cyclic Graph (DCG).

The design is heavily inspired by Prof. Dr. Uwe Naumann's research paper on "Reduction of the Random Access Memory Size in Adjoint Algorithmic Differentiation by Overloading" [12]. The DCG design is coverd in detail in the thesis.

Supporting custom checkpointLoaders

As part of the thesis two distinct methods of generating and storing checkpoints where added

- Equidistant Disk Checkpointing
- In Memory Adaptive Checkpointing

How these two methods hold up is part of the thesis analysis.

When adding other custom loaders you should implement the abstract class checkpointLoader-Interface and also orient yourself by looking at already implemented checkpointLoaders for example the memory::checkpointLoader.

```
class checkpointLoader {
    /**
     * @param concurrent defines the CPU threads the checkpointLoader
     * is allowed to make use of.
     *
     * Useful for debugging
     */
    checkpointLoader(int concurrent);

    /**
     * Called at the start of the programm, for initialization or complete recording
     */
    void recordLoader(std::vector<double> &input);

    /**
     * Each thread will request a chunk by providing the chunk id.
     */
    bool getCheckpoint(int64_t i, checkpoint &c);
}
```

Supporting more basic Operators

The double_o datatype is a drop in replacement for the cpp double. However not all operators and other functions are supported. The default supported operators include: $\{+, -, \cdot, \div, sin, cos, pow\}$

When adding your own adjust the double_o class in double_o.hpp.

Example: adding f = pow(a, b)

We need to know the derivative of our function in the directions of all inputs, as pow only has one input this is simply f' = b*pow(a,b-1)

```
double_o pow(double_o in, int exponent);
```

this would be the function signature.

In the double_o.hpp file add

```
friend double_o pow(const double_o &d1, const int exponent) {
    /**
     * The newly generated output
```

```
     */
    double_o res;

    /**
     * Test if the provided input is currently being recorded on a dag.
     */
    if (d1.g != nullptr) {
        // the output should also be recorded in that case.
        res.g = d1.g;

        // the derivative in direction of d1
        d1.record_arg(exponent * pow(d1.value, exponent-1));
        res.record_res(1);

    }
    // real operation
    res.value = pow(d1.value, exponent);
    return res;
}
```

It could also be helpful to look at how other operators are implemented to get a good reference.

A.2 Reactive Binary Bisection: Checkpoint Retrieval Time

Listing 10: Simulation of checkpoint retrievel time, written in JavaScript

```
function test(x) {
  let checkpoints = x;
  let totalLoad = 0;
  let checks = [0];
  let checkLength = 1;
  for (let i = checkpoints; i >= 1; i--) {
      let lastCheck = checks[checks.length - 1];
      if (lastCheck == i) {
          if (checkLength < checks.length) checkLength = checks.length;
          checks.pop();
          continue;
      }
      for (let x = lastCheck; x < i;) {
          let x_old = x;
          x = Math.ceil((x + i) / 2);
          if (x != i) checks.push(x);
          totalLoad += x - x_old;
      }
  }
  return totalLoad;
}

function compareAndTest() {
    let n = 300000000;

    for (let i = Math.ceil(Math.random() * n / 1000); i < n; i += n / 1000) {
        let u = test(i) / i;
        console.log(i, u);
    }
}

compareAndTest();
```

A.3 Batch Script for HPC Cluster

```
1  #!/usr/local_rwth/bin/zsh
2
3  # ask for eight cores
4  #SBATCH --cpus-per-task=8
5  #SBATCH --mem-per-cpu=1G
6
7  #SBATCH --job-name=8
8
9  #SBATCH --output=output.%J.txt
10
11 ./adjoint
```

An example bash script for running benchmarks in the HPC environment. This has to be used in order to schedule "jobs" on the RWTH SLURM[18] system. Based on demand cores and memory can be requested (line 4–5). The output is stored in an output file, that is generated based on the job name (line 9). This file is then parsed by hand, as the adjoint program provides all the necessery values to the console. While this process could have been automated, the amount of benchmarks created was manageable and the required information quite varied.

A.4 Additional Benchmarks

In this section several additional benchmarks are listed. An overview of all the performed benchmarks as well as their respective input parameters is provided in Table 6. Window size scaling's, in essence providing more memory to the program, are performed in Appendix A.4.1. Additionally scaling the input parameters to make the AD problem harder and larger are presented in Appendix A.4.2.

Figure	Env.	Primal	Input Size	windowSize	size	cores	tests
Tab. 3	HPC	burgersFunction	303	1k	30k	8	10
16	HPC	burgersFunction	303	1k	30k	N/A	20
18	PC	burgersFunction	303	1k	30k	N/A	10
19	HPC	example2	3	40mio	300mio	N/A	10
20	PC	example2	3	40mio	300mio	N/A	5
21	HPC	burgersFunction	303	N/A	300k	8	5
22	HPC	burgersFunction	303	N/A	300k	16	5
23	HPC	example2	3	N/A	300mio	4	5
24	HPC	example2	3	N/A	300mio	32	5
25	HPC	burgersFunction	3003	1k	30k	N/A	5
26	HPC	burgersFunction	303	1k	300k	N/A	5
27	HPC	burgersFunction	3003	1k	300k	N/A	5

Table 6: Overview of all the benchmarks performed and their respective parameters. The number of tests denotes how many times the benchmark was run and averaged over.

[18] https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/13ace46cfbb84e92a64c1361e0e4c104/

A.4.1 windowSize scaling

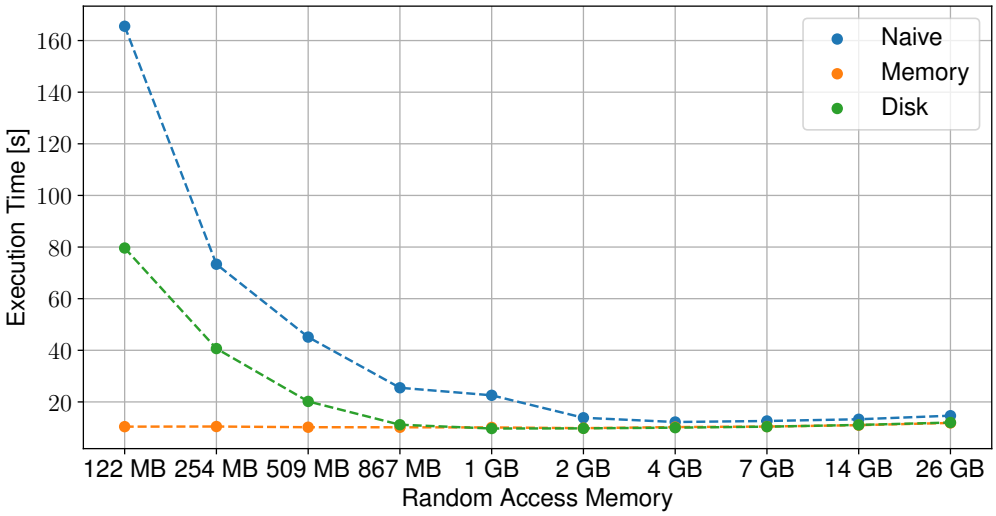


Figure 21: [HPC/5] (300k,303,8 Threads) burgersFunction. The x-axis shows the memory given to the program. The given memory has a direct impact on the number of checkpoints and with that the number tasks that need to be computed. One can see that when using less than 1 GB the entire programs speedup is limited by the checkpointing time rather than the actual reversal.

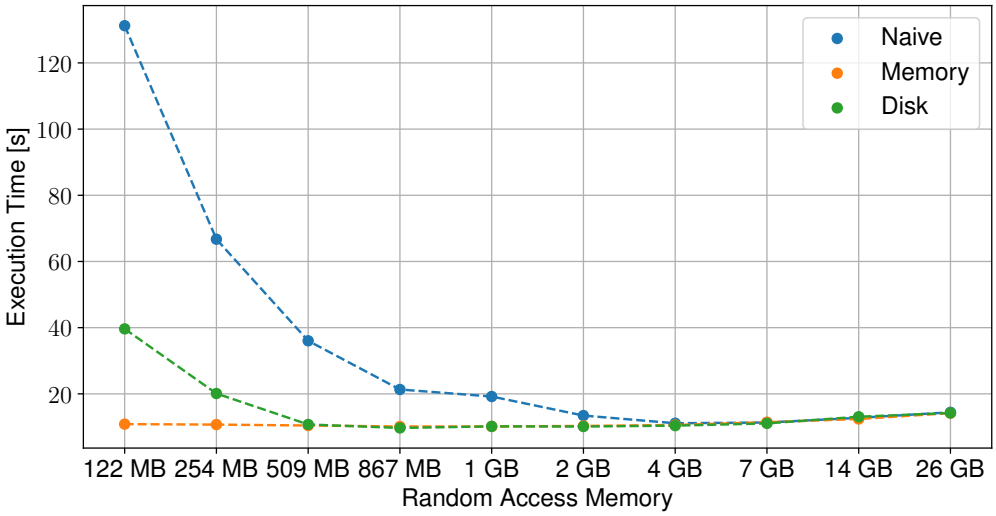


Figure 22: [HPC/5] (300k,303,16 Threads) burgersFunction. Similar results as when doing the same benchmark with 8 Threads (Figure 21). The main difference one can note is that the overall time does go down and that the Disk method does perform overall better.

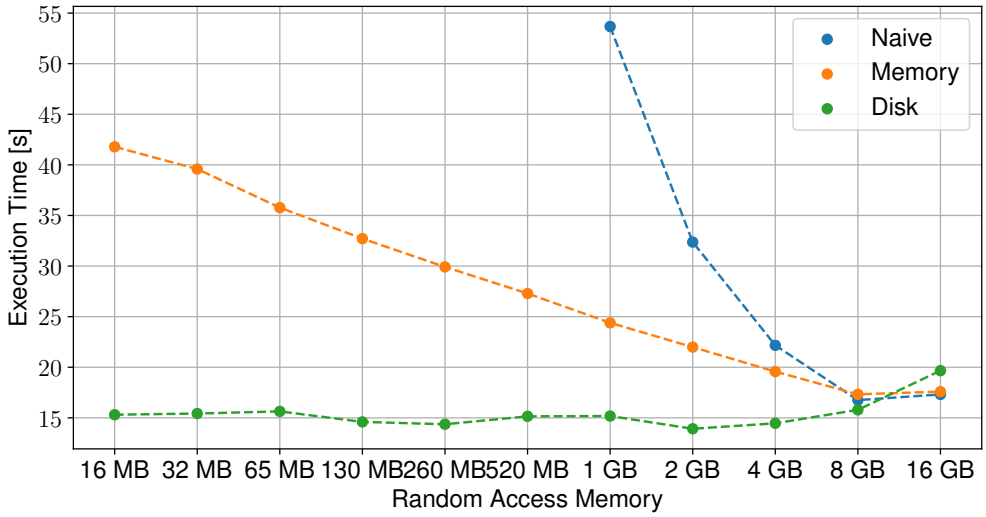


Figure 23: [HPC/5] (300mio,3,4 Threads) example2. With the reversal and overload factor of the example2 primal beeing so low, this shows in much more detail the impact of checkpointing time. Overall one can see that Disk performs quite well, while Naive has a quadratic dependency on the number of checkpoints and Memory a loglinear one.

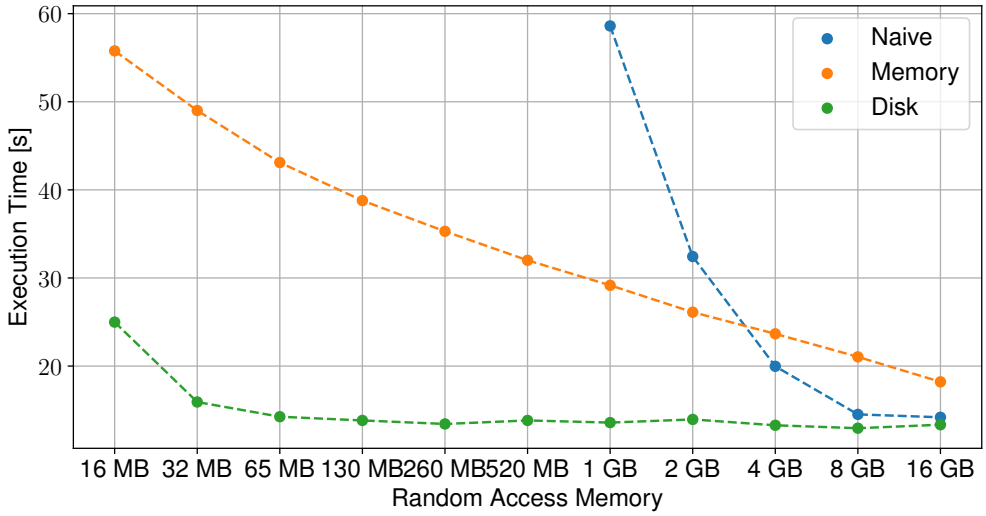


Figure 24: [HPC/5] (300mio,3,32 Threads) example2

A.4.2 burgersFunction scaling

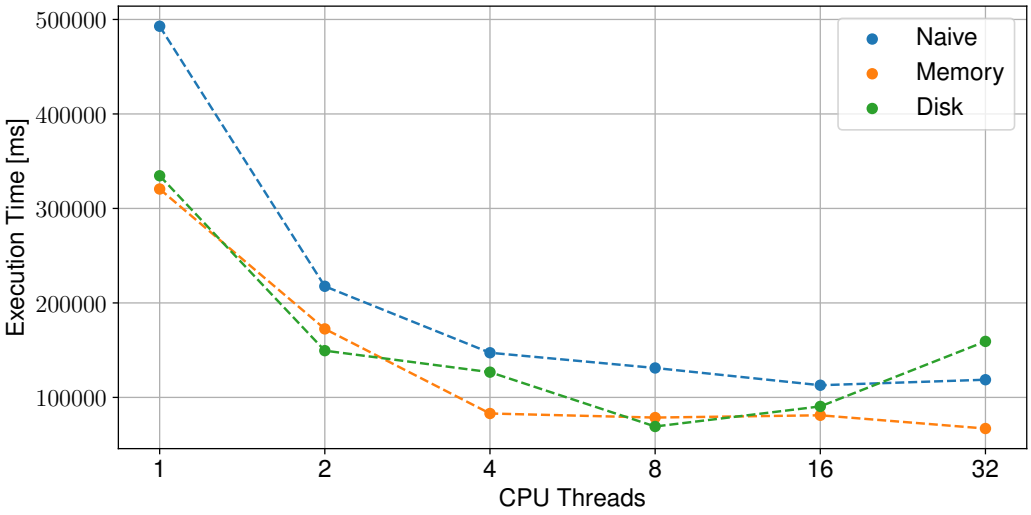


Figure 25: [HPC/5] (30k,3003,1k) burgersFunction. When scaling the input vector length, the problem overall is way more compute and memory intensive.

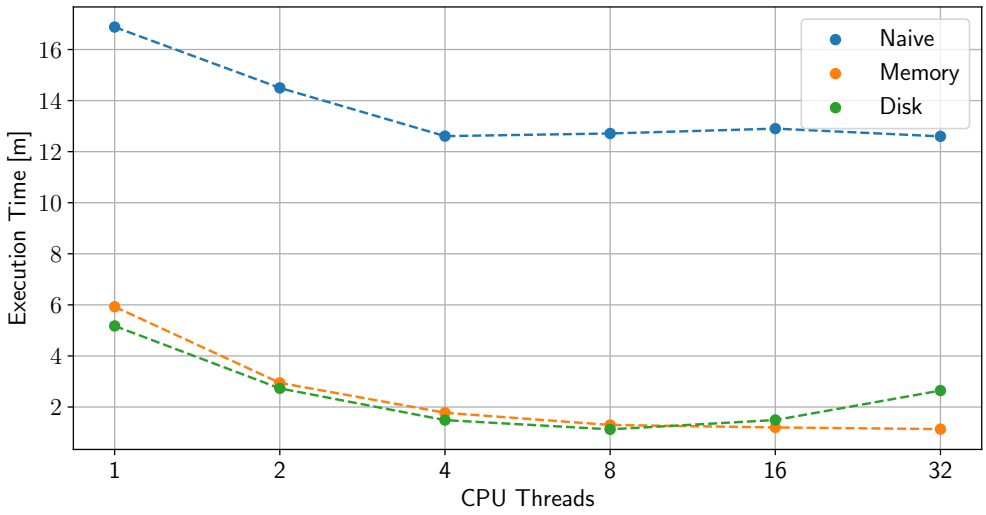


Figure 26: [HPC/5] (300k,303,1k) burgersFunction. Scaling the size by a factor of 10 has a proportionally large impact on the Naive method.

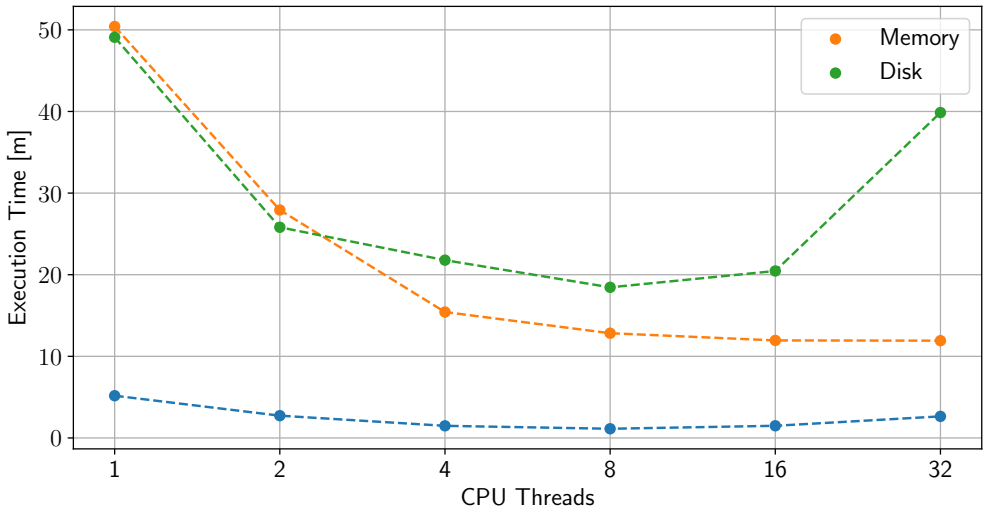


Figure 27: [HPC/2] (300k,3003,1k) burgersFunction. The Naive method was ommited as it took more then one hour to compute a single run, even with 32 cores.

A.5 Additional Graph Examples

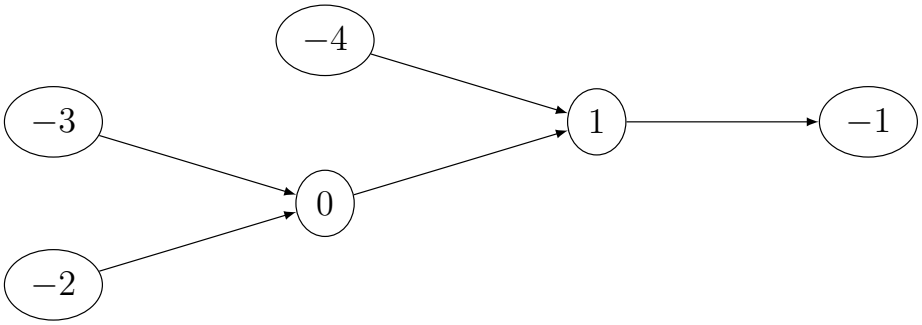


Figure 28: DCG of Listing 4. As temporary nodes are introduced, in 0 and 1, the bandwidth needed is $1 - 0 = 1$.

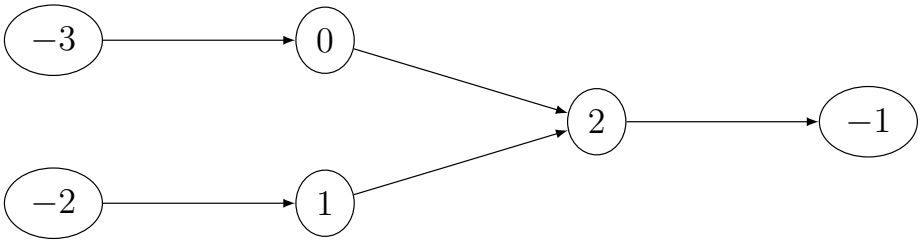


Figure 29: DCG of Listing 5. As temporary nodes are introduced, in 0, 1, and 2, the bandwidth needed is $2 - 0 = 2$.

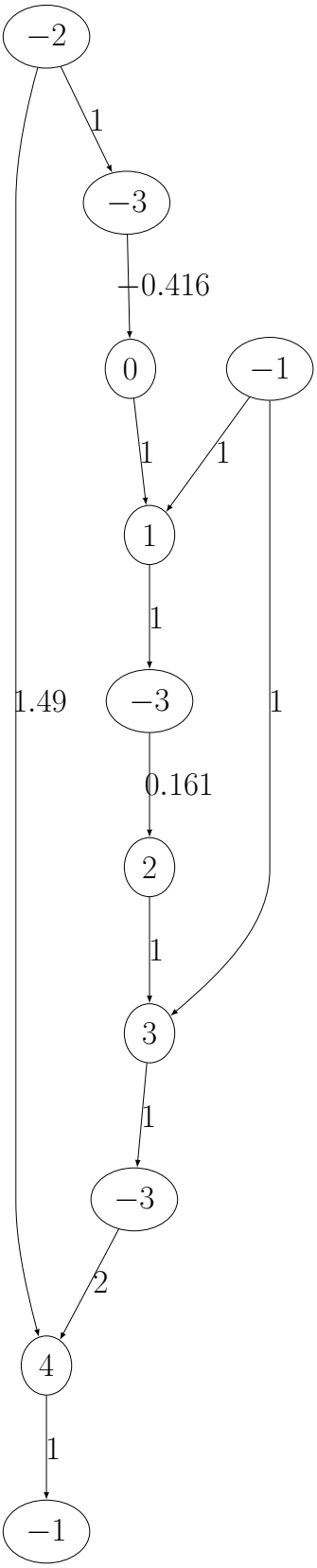


Figure 30: DAG of DCG 12. While semantically the same as DAG 2 it does include additional nodes, due to how the tape structure saves temporary and persistent variables.