PARAD: A Work-Efficient Parallel Algorithm for Reverse-Mode Automatic Differentiation *

Tim Kaler [†‡] Tao B. Schardl [†‡] Brian Xie [†‡] Charles E. Leiserson [†‡] Jie Chen [§¶]
Aldo Pareja [§¶] Georgios Kollias [§¶]

Abstract

Automatic differentiation (AD) is a technique for computing the derivative of function $F : \mathbf{R}^n \to \mathbf{R}^m$ defined by a computer program. Modern applications of AD, such as machine learning, typically use AD to facilitate gradient-based optimization of an objective function for which $m \ll n$ (often $m=1$). As a result, these applications typically use reverse (or adjoint) mode AD to compute the gradient of F efficiently, in time $\Theta(m \cdot T_1(F))$, where T_1 is the work (serial running time) of F. Although the serial running time of reverse-mode AD has a well known relationship to the total work of F, general-purpose reverse-mode AD has proven challenging to parallelize in a work-efficient and scalable fashion, as simple approaches tend to result in poor performance or scalability.

This paper introduces PARAD, a work-efficient parallel algorithm for reverse-mode AD of determinacy-race-free recursive fork-join programs. We analyze the performance of PARAD using work/span analysis. Given a program F with work $T_1(F)$ and span (critical-path length) $T_\infty(F)$, PARAD performs reverse-mode AD of F in $O(m \cdot T_1(F))$ work and $O(\log m + \log(T_1(F))T_\infty(F))$ span. To the best of our knowledge, PARAD is the first parallel algorithm for performing reverse-mode AD that is both provably work-efficient and has span within a polylogarithmic factor of the original program F.

We implemented PARAD as an extension of Adept, a C++ library for performing reverse-mode AD for serial programs that is known for its efficiency. Our implementation supports the use of Cilk fork-join parallelism and requires no programmer annotations of parallel control flow. Instead, it uses compiler instrumentation to dynamically trace a program's series-parallel structure, which is used to automatically parallelize the gradient computation via reverse-mode AD. On eight machine-learning benchmarks, our implementation of PARAD achieves 1.5× geometric-mean multiplicative work overhead relative to the serial Adept tool, and 8.9× geometric-mean self-relative speedup on 18 cores.

*Support is gratefully acknowledged from the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000; the National Science Foundation (NSF) under grants IIS-1447786, CNS-1017058, CCF-1162148, CCF-1314547, CCF-1563880, CCF-1533644; and, the MIT-IBM Watson AI Lab under grant 027397-00059. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

[†]MIT CSAIL, MIT-IBM Watson AI Lab

[‡]{tfk, neboat, brianxie, cel}@mit.edu

[§]MIT-IBM Watson AI Lab, IBM Research

[¶]{chenjie, aldo.pareja, gkollias}@us.ibm.com

1 Introduction

Automatic differentiation[1] [30, 45, 57, 58], or ***AD*** for short, aims to numerically compute the derivative of a function $F : \mathbf{R}^n \to \mathbf{R}^m$ defined by a computer program. Although AD has a long history of exploration and development in applications such as computational fluid dynamics, molecular dynamic simulations, engineering design optimization, sensitivity analysis, and uncertainty quantification, AD is commonly used today as a fundamental computational step in training neural networks for machine learning. In that context, the program is a neural network that defines a function F mapping a set of weights $W \in \mathbf{R}^n$ to a loss $L \in \mathbf{R}^m$, for which $m \ll n$ (often $m=1$). AD is used when training the neural network to facilitate gradient-based optimization of L [13]. In contrast to symbolic [33] or numerical differentiation [17], AD provides an efficient way to compute partial derivatives for functions of many input variables, which makes AD appealing for training neural networks [5]. More broadly, efficient general-purpose AD sees a diversity of uses today, from general applications of AD in machine learning, such as in differentiable programming systems (e.g., [14, 42, 57]), to various applications in computational science.

For a function F computed serially in time $T_1(F)$, the gradient of F can be computed using either ***forward-mode*** AD [68], in time $\Theta(n \cdot T_1(F))$, or using ***reverse-mode*** (or ***adjoint-mode***) AD [52, 65], in time $\Theta(m \cdot T_1(F))$. Reverse-mode AD is therefore well suited for machine learning and has therefore grown in popularity.

This paper addresses the problem of automatically parallelizing general-purpose reverse-mode AD for programs implemented using ***(recursive) fork-join parallelism***, as supported by parallel programming languages including dialects of Cilk [27, 43, 50], Fortress [1], Kokkos [24], Ha-

[1]Also known as algorithmic differentiation.

banero [4], Habanero-Java [18], Hood [12], HotSLAW [54], Java Fork/Join Framework [47], OpenMP [3, 56], Task Parallel Library [49], Threading Building Blocks (TBB) [60], and X10 [19]. Fork-join parallelism allows subroutines to be spawned recursively in parallel and iterations of parallel loops to execute concurrently. Fork-join programs expose fine-grained tasks that are allowed to execute in parallel, but are not required to. The execution and synchronization of fine-grained tasks is managed "under the covers" by a runtime system, which typically implements a randomized work-stealing scheduler [2,9,11,27] to schedule and load-balance the computation among parallel ***worker threads***. Constructs such as **parallel for** can be implemented as syntactic sugar on top of the fork-join model. As long as a fork-join program contains no ***determinacy races*** [25] (also called ***general races*** [55]) — no cases where two logically parallel operations access the same memory location, and at least one access writes to the location — then it is ***deterministic***, meaning that every execution of the program on a given input performs the same operations, regardless of scheduling. Fork-join parallelism has emerged as a popular parallel-programming model that allows many programs to be implemented efficiently as deterministic parallel programs [8, 44, 64].

This paper explores the following problem: given a determinacy-race-free function $F : \mathbf{R}^n \to \mathbf{R}^m$ defined by a recursive fork-join parallel program, automatically parallelize the reverse-mode AD computation of F in a work-efficient and scalable manner that is efficient in practice. We have observed that, in practice, many applications of reverse-mode AD, including machine-learning applications, implement functions that satisfy these constraints.

General-purpose reverse-mode AD has long posed a challenge to parallelize efficiently [7], despite its substantial history of research and development (for a survey of previous work, see [5]). Reverse-mode AD can be performed in parallel for the m dimensions of the output of F, but this approach yields minimal parallelism when m is small. Specialized algorithms have been developed to perform parallel reverse-mode AD for specific computations [38–41], but these specialized approaches do not apply to recursive fork-join computations in general. Previously developed solutions [6, 61] to parallel reverse-mode AD either are not ***work efficient*** — the total computation involved is $\omega(m \cdot T_1(F))$ — or they suffer in parallel performance and scalability, for example, due to lock contention.

The challenges of parallelizing reverse-mode AD

To see the challenges in automatically parallelizing reverse-mode AD, let us first examine serial reverse-mode AD on a function F.

Intuitively, AD views the computation of F as a sequence of primitive arithmetic operations and primitive functions — such as addition, multiplication, sine, and cosine — and performs a nonstandard interpretation of F to calculate derivatives. Reverse-mode AD computes the derivative of F by applying the chain rule from differential calculus, starting from the outermost function, which is the last operation or function in the sequence. More precisely, let $L \in \mathbf{R}^m$ be the dependent variable computed by F, and let n be the length of the operation sequence to compute F. After at each step i, the reverse-mode AD algorithm has evaluated some suffix S_{n-i} of the computation of F, and it stores a set of ***gradients*** that encode the adjoint $\partial L/\partial W_{n-i}$, where W_{n-i} is the set of inputs to S_{n-i}. Step $i+1$ of the algorithm grows the evaluated suffix S_{n-i-1} from S_{n-i} by updating the set of gradients to store $\partial L/\partial W_{n-i-1} = (\partial L/\partial W_{n-i})(\partial W_{n-i}/\partial W_{n-i-1})$.

Reverse-mode AD is most commonly accomplished through maintenance of an auxiliary ***tape*** data-structure.[2] Conceptually, reverse-mode AD first performs an augmented execution of the function F, known as the ***forward pass***, to record data about F's computation onto the tape. After the forward pass completes, reverse-mode AD performs a ***reverse pass***, in which it applies the chain rule to the operations on the tape in reverse order. Section 2 describes an efficient serial reverse-mode AD algorithm in detail.

At a high level, the tape and the set of gradients pose two key challenges to parallelizing reverse-mode AD.

Parallelizing the tape. Parallel reverse-mode AD must accommodate the parallelism in the computation of F. For a fork-join parallel program F, the spawning and synchronization of logically parallel tasks imposes a directed acyclic graph (DAG) of dependencies operations, rather than a sequence. Dependencies between primitive operations must be recorded efficiently in parallel during the forward-pass execution of F. In addition, the DAG structure of dependencies implies logical parallelism between operations in the reverse pass, which should be exploited to achieve performance and scalability.

Parallel maintenance of gradients. As Section 2 describes, one well-known feature of reverse-mode AD is that a variable-read operation in the given function F corresponds to a write of a gradient value during the reverse pass of the reverse-mode AD computation of F, and vice versa [7, 61]. As a result, even if F is determinacy-race free, logically parallel read operations during the forward pass become logically parallel write operations during the reverse pass. Intuitive approaches to managing gradients can lead to poor performance. For example, coordinating updates to gradients using locks can result in high contention that inhibits scalability. Alternatively, one might imagine maintaining gradients using P thread-local tables, where P is the number of processors executing the reverse pass. But a simple use of thread-local gradient tables is not work efficient, because reads of gradient values can occur asynchronously during a parallel execution of the reverse pass, and $O(P)$ work is needed to read a gradient out of the thread-local tables. Synchronizing these reads can reduce the total work, but at the cost of parallel scalability.

Previous approaches. Previous work [6] has explored parallel reverse-mode AD for OpenMP programs using

[2] Also called a Wengert list [68].

ADOL-C [67], a library for reverse-mode AD in C++ programs. To accommodate OpenMP threads, a separate instance of ADOL-C is created for each thread, such that each thread operates on its own tape and set of gradients. Separate user code is invoked to combine gradient information from these parallel tapes at serial points in the computation. This thread-local AD approach can work efficiently for programs with simple parallel control flow, such as a sequence of parallel loops. But it is unclear how to generalize the approach to handle arbitrary recursive fork-join parallel programs while maintaining work efficiency and scalability. Nested parallel control flow in particular presents a significant challenge, because work efficiency can be precluded by the total work performed over the entire reverse pass to combine gradient information at nested synchronization points.

Previous work has also explored similar node-local approaches to parallelize reverse-mode AD for MPI programs [61], by assigning each MPI node to operate on a separate tape and replacing MPI communications in the forward pass with appropriate reversed communications to communicate gradient information in the reverse pass. These approaches are not work efficient [36], due to the cost of communicating and combining parallel gradient information.

PARAD*: Work-efficient and scalable reverse-mode AD*

This paper introduces **PARAD** a provably efficient parallel algorithm for reverse-mode AD for determinacy-race-free recursive fork-join programs. Given such a program F, PARAD computes the gradient of F using reverse-mode AD work-efficiently and with parallel scalability comparable to that of F itself. In particular, PARAD exploits the logical parallel control flow of the input of F to parallelize the reverse-mode AD computation of F.

In particular, Section 4 analyzes the parallel performance of PARAD using work/span analysis [21, Ch. 27]. The ***work*** of a computation is the total number of instructions executed, and the ***span*** is the length of a longest path of dependencies in the program. Section 4 shows that, given an input function $F : \mathbf{R}^n \to \mathbf{R}^m$ which takes work $T_1(F)$ and span $T_\infty(F)$ to compute, PARAD computes reverse-mode AD of F in work $\Theta(m \cdot T_1(F))$ and span $O(\log m + \log(T_1(F))T_\infty(F))$. To the best of our knowledge, PARAD is the first parallel algorithm for performing reverse-mode AD that both is provably work-efficient and has span within a polylogarithmic factor of $T_\infty(F)$.

To efficiently parallelize reverse-mode AD, PARAD implements an ***SP-Tape*** data structure, which records a tape for F efficiently in parallel, and a novel parallel algorithm for maintaining gradients.

A work-efficient, scalable, deterministic parallel tape. Section 3 describes the SP-Tape data structure for recording a tape of the forward-pass execution of F, including all series-parallel relationships between primitive operations and functions in F. The SP-Tape data structure ensures that, if F is determinacy-race free, then the SP-Tape recorded for the forward-pass execution of F is the same, regardless of how the forward-pass is scheduled at runtime. Section 3 justifies that the SP-Tape records the tape in a work-efficient and scalable manner with bounded contention. The maintenance of an SP-Tape resembles techniques in previous work for recording series-parallel dependencies in recursive fork-join programs [59], but with substantial extensions to implement reverse-mode AD.

Algorithm	T_s/T_1	T_s/T_{18}	T_1/T_{18}
PARAD	0.57	5.12	9.02
PARAD+S	0.66	5.88	8.89
Locks	0.45	2.77	6.14
Worker-Local	0.94	4.96	5.27

Table 1: Performance comparison of PARAD, PARAD+S, Locks, and Worker-Local over the 8 application benchmarks discussed in Section 6. Average (geometric mean) work-efficiencies T_s/T_1 and 18-core speedups T_s/T_{18} are provided relative to the serial runtime T_s of Adept.

Work-efficient parallel maintenance of gradients. To achieve a work efficient and scalable reverse pass, PARAD uses a novel algorithm to maintain gradients that overcomes the problems of approaches based on locks or thread-local gradient tables. In contrast to lock-based approaches, the algorithm avoids the overheads of locks and bounds the contention involved in updating gradients. In contrast to approaches that use thread-local gradient tables, the algorithm allows gradients to be read in an asynchronous manner, while maintaining both work efficiency and span comparable to the input function F. Section 4 describes this algorithm for maintaining sets of gradients in parallel.

The LibPARAD *parallel reverse-mode AD library*

We implemented PARAD in a library, called LibPARAD, to evaluate the empirical performance of PARAD. LibPARAD extends the Adept [35] library for reverse-mode AD computation of serial C++ programs, which is known to exhibit low overheads in practice. LibPARAD supports the use of the Cilk programming language [27, 43, 50] to encode recursive fork-join parallelism. LibPARAD captures the series-parallel relationships between operations using compiler-inserted program instrumentation, based on a version of the CSI framework [62] that instruments the Tapir compiler intermediate representation of recursive fork-join parallelism [63]. This approach has the extra benefit that programmers need not annotate the logical parallelism in the program. Instead, LibPARAD captures this logical parallelism based on the linguistic constructs in the original program. Section 5 describes the implementation of LibPARAD as well as several practical optimizations that LibPARAD implements on top of the PARAD algorithm.

We evaluated LibPARAD's serial and parallel performance in practice on a variety of machine-learning benchmarks, and we compare the performance of LibPARAD to Adept applied to the serial projection [27, 63] of each benchmark, as well as to implementations that use fine-grained locks (Locks) and thread-local gradient tables

(Worker-Local). Table 1 summarizes our empirical evaluation of LibPARAD in which we compared PARAD and PARAD+S, which incorporates additional optimizations described in Section 5, with algorithms using locks and worker-local tables. On average, the PARAD and PARAD+S algorithms have better scalability than Locks and Worker-Local both in terms of self-relative speedup and relative to the serial Adept code. Section 6 dives into the empirical evaluation of LibPARAD and examines how the scalability of each algorithms varies across benchmarks.

Contributions

This paper makes the following contributions.

1. We introduce PARAD, an algorithm that performs reverse-mode AD for determinacy-race-free recursive fork-join parallel programs with substantially fewer overheads than existing systems. Given a determinacy-race-free recursive fork-join program $F : \mathbf{R}^n \to \mathbf{R}^m$, PARAD automatically parallelizes the reverse-mode AD computation of F.
2. Using work/span analysis [21, Ch. 27], we show that PARAD performs reverse-mode AD on a given function F in work $O(m \cdot T_1(F))$ and $O(\log m + \log(T_1(F))T_\infty(F))$. To the best of our knowledge, PARAD is the first parallel algorithm for performing reverse mode AD that is both provably work-efficient and has span within a polylogarithmic factor of $T_\infty(F)$.
3. We introduce LibPARAD, an implementation of PARAD for performing automatically parallel reverse-mode AD for determinacy-race-free recursive fork-join programs. LibPARAD extends the Adept [35] C++ library for serial reverse-mode AD, which is known to outperform other C++ AD libraries [66].
4. We study the empirical performance of LibPARAD on eight machine-learning benchmarks and compare the the PARAD, PARAD+S, Locks, and Worker-Local algorithms for reverse-mode AD.

Organization

The remainder of the paper is organized as follows. Section 2 provides background on AD and fork-join parallelism. Section 3 describes the SP-Tape data structure and analyzes it in terms of work and span. Section 4 presents and analyzes PARAD's work-efficient and scalable reverse-mode algorithm. Section 5 describes the implementation of LibPARAD, a C++ library implementation of PARAD, based on the Adept C++ library for serial reverse-mode AD. Section 6 compares the empirical performance of LibPARAD against Adept and a lock-based implementation of reverse-mode AD. Section 7 discusses related work in parallel AD. Section 8 provides concluding remarks.

2 Preliminaries

This section provides background information on the serial algorithm for reverse-mode AD, recursive fork-join parallelism, the dag model of multithreading, and work/span analysis. To simplify the description of reverse-mode AD, we shall consider input programs $F : \mathbf{R}^n \to \mathbf{R}^m$ for which $m = 1$. It is straightforward to extend this description for programs where $m > 1$.

A serial algorithm for reverse-mode AD

In general, a serial reverse-mode AD computation, as implemented by tools including Adept [35], ADOL-C [67], and Tapenade [32], operates in two passes. Given a function $F : \mathbf{R}^n \to \mathbf{R}^m$, the forward pass first executes F and records the primitive operations of F onto a tape data structure. The reverse pass maintains a table of gradient values and processes the tape data structure step by step in reverse, updating the gradient values at each step.

To examine the algorithm more closely, let us first examine the tape data structure and the forward pass. The serial tape data structure consists of two stacks: a ***statement stack***, corresponding to writes to variables in the program F, and an ***operation stack***, that records the values read to compute statements. Each differentiable variable v in the program, corresponding to an executed statement in F, is assigned a unique integer identifier $\mathit{index}(v)$, called the ***gradient index***. A statement-stack entry contains the index associated with the left-hand-side variable v, and the length of the operation stack when the statement was inserted. Each operation-stack entry contains the index of the variable u being read, as well as the partial derivative of the statement's left-hand-side variable v with respect to u. Figure 1 illustrates the statement and operation stacks recorded for a simple program with 3 statements.

To see how the forward pass populates the statement and operation stacks, consider the forward-pass executing of TwoByTwoMatVecSqLoss in Figure 1 line by line. Line 1 computes the statement $g = a \cdot e + b \cdot f$. The expression on the right-hand side of the statement is evaluated first and the operations O_0, O_1, O_2, O_3 are pushed onto the operation stack. After the right-hand expression is evaluated, an entry S_0 is pushed onto the statement stack recording the gradient index $\mathit{index}(g)$ of the left-hand-side variable g, and the current length 4 of the operation stack. As a result, after line 1 is executed, statement S_0 on the statement stack identifies operations O_0, O_1, O_2, and O_3 as storing partial derivatives $\partial g/\partial a$, $\partial g/\partial e$, $\partial g/\partial b$, and $\partial g/\partial f$, respectively. Line 2 is handled similarly by pushing operations O_4,O_5,O_6,O_7 to the operation stack and then pushing statement S_1 to the statement stack. Lastly, to handle line 3, operations O_8, O_9 are pushed onto the operation stack and statement S_2 onto the statement stack.

After executing the function, the reverse pass creates a ***gradient table*** with one entry for each gradient index created during the execution of the forward pass. Initially, all entries of the gradient table are set to 0, except for (one or more) variables whose derivatives are already known. In the example in Figure 1, there is a single loss variable L whose gradient index is initialized to 1.

The gradient-table state after each step of the reverse-mode

Forward pass program

TwoByTwoMatVecSqLoss(a,b,c,d,e,f):

1 $g = a \cdot e + b \cdot f$
2 $h = c \cdot e + d \cdot f$
3 $L = g^2 + h^2$
4 **return** L

Statement Stack

	index	end index
S_0	$index(g)$	4
S_1	$index(h)$	8
S_2	$index(L)$	10

Operation Stack

	index	mul
O_0	$index(a)$	$\partial g/\partial a = e$
O_1	$index(e)$	$\partial g/\partial e = a$
O_2	$index(b)$	$\partial g/\partial b = f$
O_3	$index(f)$	$\partial g/\partial f = b$
O_4	$index(c)$	$\partial h/\partial c = e$
O_5	$index(e)$	$\partial h/\partial e = c$
O_6	$index(d)$	$\partial h/\partial d = f$
O_7	$index(f)$	$\partial h/\partial f = d$
O_8	$index(g)$	$\partial L/\partial g = 2g$
O_9	$index(h)$	$\partial L/\partial h = 2h$

Gradient table state after each step of reverse-mode AD

Step	∂a	∂b	∂c	∂d	∂e	∂f	∂g	∂h	∂L
0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	$2g$	$2h$	0
2	0	0	$2h \cdot e$	$2h \cdot f$	$2h \cdot c$	$2h \cdot d$	$2g$	0	0
3	$2g \cdot e$	$2g \cdot f$	$2h \cdot e$	$2h \cdot f$	$2h \cdot c + 2g \cdot a$	$2h \cdot d + 2g \cdot b$	0	0	0

Figure 1: Illustration of a serial reverse-mode AD computation on simple program, TwoByTwoMatVecSqLoss. The tables in the top right show the statement and operation stacks, recorded during the execution of the forward pass of the program. The bottom table shows the state of the gradient table after processing each statement (in reverse order) during the reverse pass.

SerialReversePass(S,O,G):

1 $i = |O| - 1$
2 **for** $k = |S| - 1, |S| - 2, ..., 0$
3 $\quad \alpha = G[S_k.index]$
4 $\quad G[S_k.index] = 0$
5 $\quad$ **while** $i > S_{k-1}.endIndex$
6 $\quad\quad G[O_i.index] \mathrel{+}= \alpha \cdot O_i.mul$
7 $\quad\quad i \mathrel{-}= 1$

Figure 2: Pseudocode for the serial algorithm for computing the reverse pass over the statement stack S and operation stack O with input gradients in the table G.

AD algorithm is presented in Figure 1. In the figure, each row presents the state of the gradient table after a step of the reverse pass. Each row therefore encodes the coefficients of a differential expression obtained via differentiation of an implicit function. Row 0, for example, encodes the initial (trivial) differential expression $\partial L = 1 \cdot \partial L$. After processing statement S_2, this expression is transformed to $\partial L = (\partial L/\partial g)\partial g + (\partial L/\partial h)\partial h = 2g\ \partial g + 2h\ \partial h$. The gradient table on Row 1 encodes this transformation, in which the coefficient of ∂L is 0, and the coefficients of ∂g and ∂h are $2g$ and $2h$ respectively. To process statement S_1 the reverse pass reads the relationship between ∂h and the operations used to compute h encoded on the operation stack: $\partial h = c\ \partial e + e\ \partial c + d\ \partial f + f\ \partial d$. The reverse pass then uses this relationship to update the differential expression encoded in the gradient table. The new differential expression, encoded on Row 2, is $\partial L = 2g\ \partial g + 2h(c\ \partial e + e\ \partial c + d\ \partial f + f\ \partial d)$. The last step processes statement S_0 as we processed S_1, and results in the final gradient table (Row 3) that contains the partial derivatives of L with respect to each variable in the program.

Figure 2 gives psuedocode for the SerialReversePass procedure that performs the reverse pass, specifically, the updates to the gradient table using a previously recorded statement stack S, operation stack O, and input gradients G. The total work to maintain the two stacks and execute SerialReversePass is $\Theta(|O|) + \Theta(|S|) = \Theta(T_1(F))$ for an input program F with work $T_1(F)$.

Observations Conceptually, PARAD makes use of two observations of this serial reverse-mode AD algorithm to parallelize it. First, during the forward-execution of a parallel program, the serial tape data structure behaves like a list-monoid with an append operator. Second, the read and write sets of the forward-execution are swapped in the reverse-pass over the tape, i.e., every read becomes a write, and every write becomes a read. PARAD leverages these observations in its design of the SP-tape for recording the forward pass, and in its algorithm for automatically parallelizing the reverse pass to compute derivatives.

Recursive fork-join parallelism

Recursive fork-join parallelism allows logical parallelism in a program to be exposed using the keywords [21, Ch. 27] **spawn**, **sync**, and **parallel for**. When preceding a function call F, the **spawn** keyword ***spawns*** F, allowing F to execute in parallel with its ***continuation*** — the statement immediately after the spawn of F. The **sync** keyword complements the **spawn** keyword and acts as a local barrier that joins together, or ***syncs***, the parallelism specified by **spawn**. When a function F reaches a **sync**, control is not allowed to pass that **sync** until all functions spawned previously in F return. These keywords can be used to implement other parallel control constructs, such as the **parallel for** loop, which allows all of its iterations to operate logically in parallel.

A recursive fork-join program has a ***serial projection*** [27, 63], which intuitively is the serial program derived by removing parallel keywords from the fork-join program. If the fork-join program contains no determinacy races, then every execution of the program matches that of its serial projection.

The computation dag model

An execution of a fork-join program can be modeled as a computation dag $G=(V,E)$. Each directed edge represents a ***strand***, that is, a sequence of executed instructions with no **spawn** or **sync** statements. The execution of a **spawn** statement results in a ***spawn vertex***, which contains two successor strands. The execution of a **sync** statement results in a ***sync vertex***, which contains multiple incoming edges.

The dag G is a ***series-parallel dag*** [25], which means that G has two distinguished vertices — a ***source*** vertex, from which one can reach every other vertex in G, and a ***sink*** vertex, which is reachable from every other vertex in G — and can be constructed by recursively combining pairs of series-parallel dags using series and parallel combinations. A ***series combination*** combines two dags G_1 and G_2 by identifying the sink vertex of G_1 with the source vertex of G_2. A ***parallel combination*** combines two dags G_1 and G_2 by identifying their source vertices with each other and their sink vertices with each other.

The recursive construction of a series-parallel dag can be represented as a binary tree, called the ***SP tree*** [25], as follows. Each leaf in the SP tree represents a strand in the computation dag, and each internal node is either an S-node or a P-node. A subtree of the SP tree represents a series-parallel subdag of the computation dag. An S-node represents a series composition of the two subdags represented by its children. A P-node represents a parallel composition of the two subdags represented by its children.

Work/span analysis

Given a fork-join program whose execution is modeled as a DAG A, we can bound the P-processor running time $T_P(A)$ of the program using ***work/span analysis*** [21, Ch. 27]. The work $T_1(A)$ is the number of instructions in A, and the span $T_\infty(A)$ is the length of a longest path in A. Greedy schedulers [15, 23, 28] can execute a deterministic program with work T_1 and span T_∞ on P processors in time T_P satisfying $\max\{T_1/P, T_\infty\} \le T_p \le T_1/P + T_\infty$. A similar bound can be achieved by more practical work-stealing schedulers [10, 11]. The ***speedup*** of an algorithm on P processors is T_1/T_P, which the inequality shows to be at most P in theory. The ***parallelism*** T_1/T_∞ is the greatest theoretical speedup possible for any number of processors.

3 The SPTape Data Structure

This section describes the SPTape data structure that PARAD uses to record, in parallel, the statement and operation stacks for reverse-mode AD, and the series-parallel dependencies in the program. After recording, the SPTape supports parallel traversals with work and span proportional to that of the original recorded program. For later use, we describe and analyze generic parallel traversal algorithms over the SPTape and provide a set of rules governing memory access during the traversal that ensure the absence of determinacy races.

For didactic simplicity, we describe the SPTape data structure for *binary* recursive fork-join programs, in which each node in the computation dag has at most two incoming edges.

Basic structure of an SPTape

The SPTape data structure for a recursive fork-join program stores an SP tree [25] of the program augmented with ***data nodes*** that store "subtapes." A ***subtape*** contains a statement and operation stack that are used to record derivative dependencies within a strand. The subtapes in the SPTape represent an ordered partitioning of the statement and operation stack data structures employed by the serial AD tool discussed in Section 2.

Recording an SPTape ***serially***

An SPTape is constructed incrementally during the execution of the forward pass. We shall first see a serial algorithm for constructing the SP-Tape, and then we shall see how to parallelize this algorithm.

To record an SPTape, a single processor maintains a ***shadow stack*** that is updated based on the parallel control flow of the forward-pass execution. Each entry on the shadow stack stores a local SPTape T, which is initialized with a single root node. When an entry is popped from the shadow stack, the local SPTape T for the popped entry is appended to the children of the SPTape node of the new top entry on the stack.

Figure 3 presents pseudocode for creating an SPTape. At a high level, this pseudocode creates S and P nodes in the SP-Tape based on **spawn** and **sync** statements in the program, and it records derivative dependencies in the subtape stored in the data node at the top of the shadow stack. Figure 4 illustrates an example of how the operations on the shadow stack execute in a simple example fork-join program. As the example shows, a P-node is pushed onto the stack when a program executes a **spawn**, reflecting the fact that both the spawned function and its continuation can execute in parallel. S-nodes are pushed onto the stack when the program enters a function or a continuation of a **spawn**. Nodes are popped off the stack at the ends of functions and around **sync** operations.

The following lemma shows that, when the serial execution of a series-parallel computation dag G completes, the local SPTape at the top of the shadow stack records the gradient information and series-parallel dependencies for G.

Lemma 1. *When a serial execution completes a computation dag G, the top of the shadow stack stores a local* SPTape *that records the execution of G.*

Proof. The proof follows by induction on the structure of G. In the base case, G is a single strand u, and the top of the shadow stack stores an SPTape with a single S-node containing a single child data node that records the derivative depen-

```
PushShadow(K,type)
1  Push(K)
2  Top(K).T.type = type

PopShadow(K)
1  τ = Top(K).T
2  Pop(K)
3  Append(Top(K).T.children,τ)

On entering a function:
1  PushShadow(K,Series)

On executing a spawn:
1  PushShadow(K,Parallel)

Before executing arithmetic:
1  if Top(K).T.type ≠ Data
2      PushShadow(K,Data)

On executing the continuation of a spawn:
1  PushShadow(K,Series)

On returning from a function:
1  if Top(K).T.type == Data
2      PopShadow(K)
3  PopShadow(K)

Immediately before executing a sync:
1  if Top(K).T.type == Data
2      PopShadow(K)
3  PopShadow(K)

Immediately after executing a sync:
1  PopShadow(K)

Combine(K,τ)
1  Append(Top(K).T.children,τ)
```

Figure 3: Pseudocode for the maintenance of the SPTape data structure. The variable K denotes the shadow stack, each entry of which contains a local SP-Tape T. The field $T.type$ identifies the type of the root node of T. The field $T.children$ is a list of children of the root node of T. The Combine method is used to incorporate an SPTape τ recorded in parallel.

```
TwoByTwoMatVecSqLoss(a,b,c,d,e,f,K):
1  PushShadow(K,Series)
2  PushShadow(K,Parallel)
3  spawn λ{
4      PushShadow(K,Series)
5      PushShadow(K,Data) // (D1)
6      g = a·e + b·f
7      PopShadow(K) // (D1)
8      PopShadow(K) // Series
9  }
10 PushShadow(K,Series)
11 PushShadow(K,Data) // (D2)
12 h = c·e + d·f
13 PopShadow(K) // (D2)
14 PopShadow(K) // Series
15 sync
16 PopShadow(K) // Parallel
17 PushShadow(K,Data) // (D3)
18 L = g^2 + h^2
19 PopShadow(K) // (D3)
20 PopShadow(K) // Series
21 return L
```

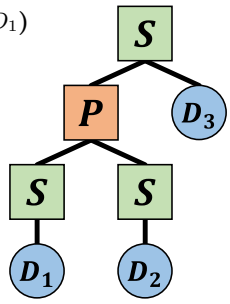

Figure 4: An example of the SPTape recorded for a parallel implementation of TwoByTwoMatVecSqLoss.

dencies in u. Otherwise G is the result of a series or parallel composition of subdags G_1 and G_2. In either case, the pseudocode in Figure 3 ensures that an entry for G is pushed onto the shadow stack at the beginning of G, and separate pushes and pops occur at the beginning and end, respectively, of each subdag. When popping the shadow stack at the end of each subdag, PopShadow in Figure 3 shows that the new top of the shadow stack is either a P-node, if a spawn executed, or an S-node otherwise. In addition, Figure 3 shows that the local SPTape formerly at the top of the stack is appended to the list of children for the new top of the stack. Hence the top of the shadow stack stores a node of the correct type and correct child SPTape structures for the execution of G_1 and G_2. □

Parallelizing SPTape *construction*

To construct an SPTape in parallel, the parallel execution of a computation dag G is augmented to maintain separate shadow stacks. Intuitively, consider two series-parallel subdags G_1 and G_2 of G that are composed in parallel and are scheduled to execute in parallel. The execution uses distinct shadow stacks to separately record the SPTape structures of G_1 and G_2 and then combines those SPTape structures when execution reaches the common sink vertex of G_1 and G_2. In modern dialects of Cilk, this behavior can be accomplished using reducer hyperobjects [26]. We describe this behavior generically for series-parallel computation dags.

We model the scheduling of a computation dag G as a partitioning of the strands into ***scheduling components***, such that logically parallel strands execute in parallel if the scheduler places the strands into distinct scheduling components. We assume that the scheduler maintains the following properties of scheduling components:

- A new scheduling component can only begin at a successor strand u of a spawn vertex in G.
- Consider a spawn vertex s, which is the source vertex of a series-parallel subdag G produced via parallel composition. A new component C that begins at a successor strand u of s terminates at a predecessor strand v of the sink vertex of G such that v is reachable from u.

These properties create a correspondence between scheduling components and series-parallel subdags of G. In particular, the same scheduling component contains both the first strand in a series-parallel subdag and the last strand in that subdag. We shall also assume that the execution of strands

```
WalkSPTape(node,G,VisitSubTape,dir):
 1  if node.type=D:
 2      VisitSubTape(node.S,node.O,G,dir)
 3  C=node.children
 4  If dir is right first reverse C.
 5  if node.type=S:
 6      for c∈C
 7          RightFirstTraversal(c,G,dir)
 8  if node.type=P:
 9      parallel for c∈C:
10          RightFirstTraversal(c,G,dir)
```

Figure 5: Pseudocode for the parallel right-first traversal of SPTape.

within the same component follows a depth-first traversal consistent with the execution of the serial projection of that computation. Practical work-stealing schedulers [10, 11] typically satisfy these assumptions.

The parallel execution of the computation dag is augmented to maintain separate SPTape structures for distinct scheduling components. At the first strand s of each scheduling component C, the scheduler creates a new local SPTape for C. Subsequent operations within C operate on C's local SPTape as described in Figure 3. At the end of C, the local SPTape τ for C is combined into the shadow stack K' of the scheduling component C' that contains the predecessor w of s. In particular, τ is appended to the list of children of the root note of the SPTape at the top of K', using the Combine method in Figure 3. The following lemma extends Lemma 1 to incorporate combining local SPTape structures.

Lemma 2. *When execution completes a computation dag G, the top of the shadow stack stores a local* SPTape *that records the execution of G.*

Proof. The proof follows by extending the induction in Lemma 1 to handle computation dags G resulting from a composition involving a subdag G' whose strands belong to a different scheduling component. Suppose the lemma holds for such a subdag G'. Because the initial strand of G' must be a successor of a spawn vertex, G must be the result of a parallel composition. Hence, the pseudocode in Figure 3 shows that, at the end of executing G, the top of the shadow stack contains an SPTape structure T rooted at a P-node. The Combine routine appends the SPTape τ for G' to the children of this P-node, yielding an SPTape T that correctly represents G as a parallel composition involving G'. □

Race-free traversals of SPTape The PARAD algorithm makes use of parallel traversals over the SPTape. Here we provide conditions on which these traversals are race-free[3], and analyze their work and span.

[3]These conditions assume that the recorded program was, itself, free of determinacy races.

For didactic purposes, consider a table G_x mapping gradient indices to distinct memory locations. Consider a call to VisitSubTape(S,O,G,*dir*,G_x), and let I_S,I_O be the sets of gradient indices appearing in the statement stack S and operation stack O respectively. We say a parallel traversal of an SP-Tape obeys the ***safe adjoint access property*** if VisitSubTape performs a read of $G_x[gid]$ only if $gid \in I_S \cup I_O$, and only performs a write to $G_x[gid]$ if $gid \in I_S$.

As Lemma 3 shows below, a parallel traversal of an SP-Tape produced by a determinacy-race-free program is free of data-races on entries of G_x if the traversal has the safe adjoint access property. Due to space limitations, we omit the proof of Lemma 3.

Lemma 3. *Let T be an* SPTape *produced by a determinacy-race-free fork-join program P. Consider a parallel traversal of T that obeys the safe adjoint access property. Then, the parallel traversal of T has no data races on accesses to tables G_x indexed by gradient indices appearing in statements and operations on the tape.*

Work/span analysis of SPTape **traversal** Subsequent algorithms will perform parallel traversals over the SPTape data structure. Lemma 4 provides bounds on the work and span of a traversal whose VisitSubTape evaluates a function $f(\cdot)$ on each operation and statement stack entry which performs $O(\sigma)$ amortized work (amortized over the whole traversal), and has worst-case span $O(\upsilon)$. Due to space limitations, we omit the proof of Lemma 4.

Lemma 4. *Consider a parallel traversal by* WalkSPTape *of a* SPTape *T produced by a parallel program P with work $T_1(P)$ and span $T_\infty(P)$. If* VisitSubTape *processes each statement and operation with a function $f(\cdot)$ that performs $O(\sigma)$ amortized work (amortized over the whole traversal) and has $O(\upsilon)$ worst-case span, then* WalkSPTape *performs $T_1 = O(\sigma T_1(P))$ work and has $T_\infty = O(\upsilon T_\infty(P))$ span.*

4 The PARAD algorithm

This section presents PARAD, a work-efficient algorithm for performing parallel reverse-mode AD. For a determinacy-race-free fork-join program F with work $T_1(F)$ and span $T_\infty(F)$, the program R that performs reverse-mode AD on F has work $T_1(R) = \Theta(m \cdot T_1(F))$ and span $T_\infty(R) = \Theta(\log m + T_\infty(F)\log(T_1(F)))$.

Design of the PARAD *algorithm*

The PARAD algorithm is designed to parallelize the serial reverse-mode AD algorithm based upon the series-parallel structure of the recorded program. The serial reverse-mode AD algorithm can be implemented by executing the SerialReversePass function on each subtape in the order of a right-first traversal of the SPTape. The problem with parallelizing this traversal, however, is the presence of write-write races on the gradient table, which occur when the original program read the same memory location in-parallel. The key challenge solved by PARAD is the resolution of

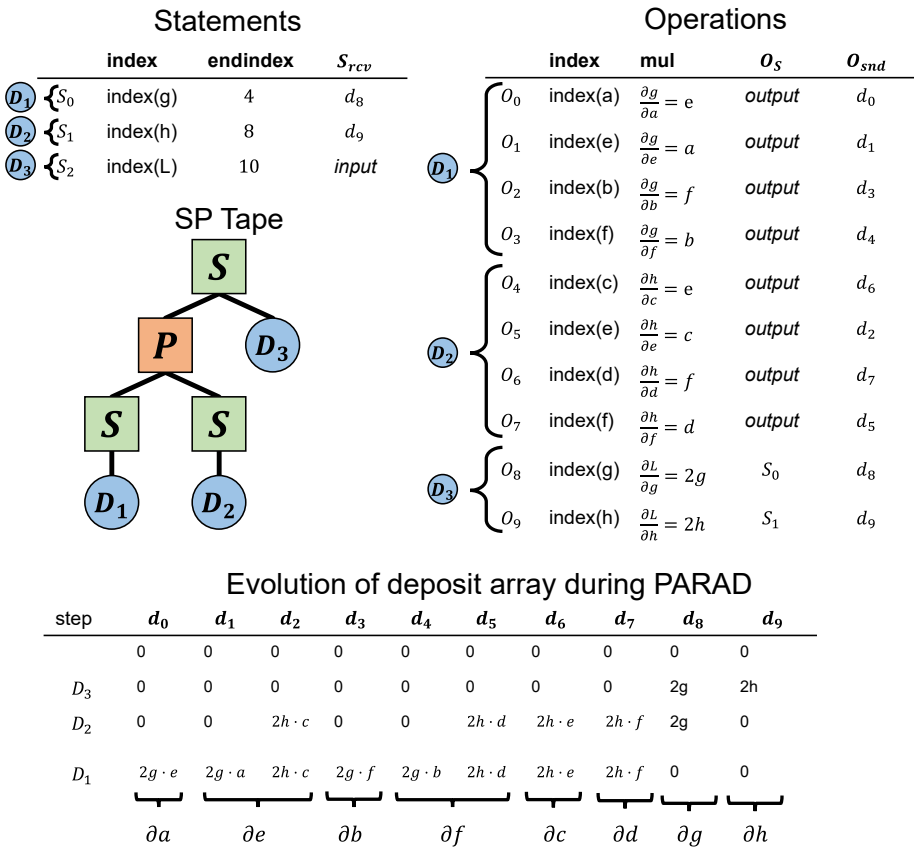

Figure 6: Illustration of PARAD performing reverse-mode AD on a parallel implementation of TwoByTwoMatVecSqLoss. This example continues the running example used in Figure 1 from Section 2 and Figure 4 from Section 3. The statement and operation stacks are presented in the style of the serial AD example in Figure 1, and the subarrays in these stacks that correspond to the subtapes D_1, D_2, and D_3 are indicated. Additional columns have been added to the statement and operation stacks to illustrate the S_{rcv}, O_S, and O_{snd} functions that map statements and operations to locations in the deposit array. For clarity, entries that would be undefined by these maps are marked as *input* or *output*. The deposit array state after processing each subtape during PARAD's right-first traversal over the tape is provided. The final gradients ∂a,∂b,...,∂h are obtained by summing the indicated subarrays of the deposit array.

these races in a work-efficient and scalable manner.

To resolve write-write races on the gradient table, PARAD employs the strategy to precompute a unique memory location where each operation can safely deposit its gradient contribution. These memory locations are assigned from a ***deposit array*** that provides one slot for each recorded operation. The gradient contributions are aggregated when a statement extracts its gradient value. In the serial AD algorithm, a statement can extract its gradient value by reading from the global gradient table. In PARAD, however, a statement must potentially collect and sum multiple gradient contributions. To allow this aggregation to be performed efficiently, the deposit array is organized so that all the gradient contributions a statement must collect are contiguous in the deposit array.

Let us discuss the structure of the PARAD algorithm whose pseudocode is provided in Algorithm 1.

The organization of the deposit array in PARAD is accomplished as follows. Step 1 of PARAD computes a table O_S that associate each operation with the statement that will consume its gradient contribution. Next, in Step 2 PARAD collects and semisorts all operations o by their associated statement $O_S[o]$[4]. The semisorted array of operations O^* is used to assign each operation a unique index in the deposit array D based upon its location in O^*. The assignment of operations to deposit array locations is recorded in a table O_{snd}. Since all operations associated with the same statement are contiguous in O^*, the gradient contributions for a statement are in a contiguous range of the deposit array D. A table S_{rcv} records, for each statement s, a reference to the subarray $D[m..m+k]$ of k gradient contributions to s.

The deposit array is now used by PARAD to avoid

[4]For didactic simplicity, the pseudocode of PARAD additionally semisorts by each operation's gradient index to make it simpler to export them to a global gradient table at the end of the computation.

Algorithm 1 PARAD Algorithm

Inputs: Gradient table G, and SP-Tape T.
Outputs: Updated gradient table G.

1. Construct O_S mapping operations to statements.
   - Perform a parallel left-first traversal of the SP-Tree. At statement s, set $G_S[s.gid]=s$. At operation o, set $O_S[o]=G_S[o.gid]$.
2. Construct O_{snd} and S_{rcv}.
   - Traverse the SP-Tape T and pack all operations into contiguous array O^*.
   - Semisort O^* using the key function $k_1(o)=o.gid$.
   - For each subarray of O^* with operations of equal $k_1(o)$, semisort using the key $k_2(o)=O_S[o]$.
   - For each subarray $O^*[m..m+k]$ of operations o with equal $k_2(o)$ do the following. Set $S_{rcv}[k_2(o)]=(m,m+k)$. For $l=m,m+1,...,m+k$ set $O_{snd}[O^*[l]]=l$.
3. Perform reverse-mode AD.
   - Allocate deposit array D of size $|O^*|$.
   - Perform a right-first traversal.
   - At statement s, let $(m,m+k)=S_{rcv}[s]$ and compute $\alpha=G[s.gid]+sum(D[m..m+k])$. Set $G[s.gid]=0$, and $D[m..m+k]=0$.
   - At operation o, compute $\beta=\alpha\cdot o.mul$ and set $D[O_{snd}[o]]=\beta$.
4. Export gradients in D to gradient table G.
   - Find subarrays $O^*[n..n+r]$ in O^* of operations o with equal gradient index (i.e. $k_1(o)=o.gid$).
   - For each sub subarray, set $G[O^*[n].gid]=sum(D[n..n+r])$.

write-write races during its right-first traversal of the SPTape. Step 3 of PARAD performs reverse-mode AD via a right-first traversal of the SPTape. The final gradients are then exported in Step 4 to a global gradient table for application-specific use (e.g., for a gradient descent step).

Figure 6 provides an illustration of the deposit array structure for the parallelized TwoByTwoMatVecSqLoss function. The tables O_S, S_{rcv}, and O_{snd} are shown in Figure 6 as columns of the tables illustrating the statement and operation stacks. The deposit array structure is provided after processing each of the subtapes D_3, D_2, and D_1.

As stated in Theorem 5 below, PARAD and SerialReversePass are equivalent (i.e., they compute the same gradients) when ran on the same recorded program P. Due to space limitations, we omit the proof of Theorem 5.

Theorem 5. PARAD *and* SerialReversePass *compute the same gradients for a recorded program P that is free of determinacy races.*

Analysis of PARAD

We now analyze the work and span of Algorithm 1.

Theorem 6. *Given a program $F:\mathbf{R}^n\to\mathbf{R}^m$ with total work $T_1(F)$ and span $T_\infty(F)$, the program R performing reverse-mode AD on F performs work $T_1(R)=\Theta(m\cdot T_1(F))$ with span $T_\infty(R)=\Theta(\log m+T_\infty(F)\log T_1(F))$.*

Proof. We analyze Algorithm 1 for $m=1$. The bounds for larger m follow from spawning m instances of PARAD to process the m dimensions in parallel.

Step 1 performs a left-first traversal of the SPTape where $O(1)$ work is performed for each recorded statement and operation in the program F. We apply Lemma 4 with $\sigma=O(1),\upsilon=O(1)$ to conclude that the total work and span of this step is $O(T_1(F))$ and $O(T_\infty(F))$ respectively.

Step 2 performs a parallel compaction and a parallel scan over operations in F, both of which perform $O(T_1(F))$ work and have $O(\log(T_1(F)))$ span. The semisort performed in this step can be performed in $O(T_1(F))$ work and $O(\log(T_1(F)))$ span by using the logarithmic-depth semisort from [31], which semisorts N elements in $\Theta(N)$ work and $\Theta(\log N)$ span.

Step 3 performs a right-first traversal with constant work per-operation. The work performed for each statement may be $\Omega(1)$, but each unit of work is associated with a unique operation. The amortized work for each operation and statement is, therefore, $O(1)$. Accumulating the subarray $D[m..m+k]$ requires $\Theta(\log k)$ span, and the worst-case span is $O(\log T_1(F))$. Therefore, we apply Lemma 4 with $\sigma=O(1)$ and $\upsilon=O(\log T_1(F))$ to conclude that this step performs $O(T_1(F))$ work and has $O(T_\infty(F)\log T_1(F))$ span.

Step 4 performs a parallel scan and multiple in-parallel reductions with total work $O(T_1(F))$ and span $O(\log T_1(F))$.

Thus, the work and span is bounded by the time required to perform the right-first traversal of the SPTape, resulting in total work $O(T_1(F))$ and span $(T_\infty(F)\log T_1(F))$. □

5 Implementation of LibPARAD

This section describes LibPARAD which is an extension of the serial AD library Adept [35] that implements four different parallel algorithms for performing reverse-mode automatic differentiation, all of which employ the SPTape. Adept is a C++ library that implements reverse-mode AD via operator overloading. Other AD libraries using a similar approach include ADOL-C [67], Autograd [53], and PyTorch [57]. We chose Adept to base our implementation since its clever use of C++ expression templates and particularly concise representation for its tape data structure allows it to outperform other C++ AD libraries [66].

Implementation of the SPTape

LibPARAD uses a version of CSI that instruments the Tapir compiler representation of recursive fork-join parallelism [63] to insert operations in Figure 3 around **spawn** and **sync** statements, at the locations in the program code described in the figure.

LibPARAD modifies the parts of Adept that access its statement and operation stacks to instead use the SPTape data structure described in Section 3. We implemented the SPTape using a reducer hyperobject [26] and worker-local storage that is accessed by using the worker identifier returned by Cilk's GetWorkerNumber runtime call. Storage in data nodes for subtapes is allocated out of worker-local storage

to improve efficiency in practice. The operations described in Figure 3 for constructing an SPTape are inserted into the program automatically at compile-time through the use of the CSI framework [62]. Specifically, LibPARAD uses a version of CSI that instruments the Tapir compiler representation of recursive fork-join parallelism to insert operations around **spawn** and **sync** statements. The LibPARAD library is designed to operate on any existing code using Adept, and does not require any additional annotations from the user aside from the normal use of Cilk keywords to express parallelism.

Our implementation deviates from the description in Figure 3 only slightly to handle nonbinary spawns. First, immediately after a **sync**, multiple PopShadow calls are performed to pop the S- and P-nodes pushed onto the shadow stack for each continuation of an executed **spawn** statement that the **sync** statement syncs. Second, the SPTape reducer hyperobject optimizes the handling chains of continuations, rather than simply enqueue calls to Combine to operate only on SPTape structures that capture the complete execution of a series-parallel subdag. Neither of these changes impact the theoretical work/span of the computation or the structure of the SPTape.

Implementation of PARAD

The implementation of PARAD includes optimizations related to the construction of the deposit array. In our discussions of Algorithm 1 in Section 4 we explained how to resolve write-write races by assigning operations unique memory locations from the deposit array to deposit their gradient contributions. Our implementation of PARAD avoids performing this work for operations that cannot possibly participate in a write-write race. We identify operations whose gradient index never appears in a statement and for such operations we accumulate gradients using worker-local sparse arrays. Additionally, we identify operations in subtapes whose contribution to the gradient is extracted by a statement in the same subtape. For such operations, we set its deposit location in D to be the global gradient table G. This does not introduce data races because it obeys the safe adjoint access property discussed in Section 3. The remaining operations are filtered and packed in-parallel and are processed as described in Algorithm 1.

Implementation of PARAD+S

A commonly used strategy to resolve races in reverse-mode AD is to employ worker-local (or thread-local) gradient tables. The problem with this approach, however, is that it is not work-efficient — extracting the gradient value for a statement requires work proportional to the number of processors, which may be greater than the number of operations that provided gradient contributions.

In PARAD+S, a sampling-based algorithm is used to identify "heavy" statements that accumulate a large number (greater than P) of operations. Operations contributing to "heavy" statements can use worker-local gradient tables instead of the deposit array without compromising the work-efficiency of the algorithm. As such, work can be avoided in Step 2 of Algorithm 1 by filtering these operations during the traversal of the SPTape to pack operations into O^*.

The additional computation performed by PARAD+S introduces a small constant overhead relative to PARAD on some benchmarks, but does not compromise the theoretical work-efficiency or scalability of PARAD.

6 Performance Evaluation

This section evaluates the performance of PARAD and PARAD+S that were implemented in LibPARAD. All experiments were run on an 18-core (hyperthreading disabled) Intel Xeon CPU (E5-2666 v3, 2.9GHz) with 64GB RAM available as a 4th-generation compute-optimized machine from Amazon web services. LibPARAD uses Cilk Plus to express parallelism and compiles the codes using the Tapir [63] based on LLVM 6.0.

Locks and Worker-Local implementations. We implemented two additional reverse-mode AD algorithms Locks and Worker-Local that employ fine-grain locking and worker-local storage, respectively, to resolve data races on the gradient table. Both of these implementations use the SPTape to record series-parallel dependencies and automatically parallelize the reverse pass over the tape, but they differ from LibPARAD in that do not need to perform additional work to organize a deposit array. This, however, comes at the expense of a loss of scalability and work-efficiency.

Application benchmarks. We evaluated the performance of LibPARAD across 8 benchmarks with different performance characteristics relevant to reverse-mode AD. For each benchmark, we measured the time required to perform AD while training the weights of the network via gradient descent. Table 2 provides a breakdown of the performance results obtained across the eight benchmarks and four implemented AD algorithms. A summary (geometric mean over all benchmarks) of each implementation's performance is provided in Table 1 in Section 1.

Note on serial performance of Adept. Although, in general, Adept is highly efficient, we have observed on certain benchmarks (*mlp*, *gcn*, *lstm*) that the serial T_1 runtime of some of our algorithms outperforms the serial runtime T_s of Adept. This difference comes *entirely* from the forward pass of the computation, and we believe it relates to differences in how gradient identifiers are allocated in an SPTape and in Adept's serial data structures. This phenomenon also affects the algorithms using the SPTape, but negatively, on the *cnn* benchmarks, where it causes added overheads in the forward pass.

Multilayer perceptron benchmarks

The *mlp* benchmarks are feed-forward multilayer perceptron networks where *mlp1* has a single hidden layer of 800 nodes, and *mlp2* has two hidden layers with 400 and 100 nodes respectively. Both networks are trained on the MNIST data set [22]. Performance results are shown in Table 2.

There is little performance variation among the different

Benchmark	Algorithm	T_s	T_1	T_{18}	T_s/T_1	T_s/T_{18}	T_1/T_{18}
mlp1	PARAD	149.50	156.94	14.44	0.95	10.35	10.87
mlp1	PARAD+S	149.50	165.18	15.17	0.91	9.85	10.89
mlp1	Locks	149.50	219.33	16.64	0.68	8.98	**13.18**
mlp1	Worker-Local	149.50	**129.87**	**12.91**	**1.15**	**11.58**	10.06
mlp2	PARAD	83.66	92.18	8.22	0.91	10.18	11.22
mlp2	PARAD+S	83.66	95.93	8.77	0.87	9.54	10.94
mlp2	Locks	83.66	95.75	8.31	0.87	10.07	**11.52**
mlp2	Worker-Local	83.66	**71.05**	**7.23**	**1.18**	**11.58**	9.83
gcn1	PARAD	94.30	113.00	13.30	0.83	7.09	8.50
gcn1	PARAD+S	94.30	111.00	**13.00**	0.85	**7.25**	**8.54**
gcn1	Locks	94.30	216.00	77.10	0.44	1.22	2.80
gcn1	Worker-Local	94.30	**84.90**	14.20	**1.11**	6.64	5.98
gcn2	PARAD	19.20	26.90	**4.14**	0.71	**4.64**	**6.50**
gcn2	PARAD+S	19.20	28.20	4.76	0.68	4.03	5.92
gcn2	Locks	19.20	47.40	9.38	0.41	2.05	5.05
gcn2	Worker-Local	19.20	**17.60**	5.63	**1.09**	3.41	3.13
cnn1	PARAD	126.00	307.00	**27.00**	0.41	**4.67**	**11.37**
cnn1	PARAD+S	126.00	311.00	28.60	0.41	4.41	10.87
cnn1	Locks	126.00	314.00	42.30	0.40	2.98	7.42
cnn1	Worker-Local	126.00	**246.00**	80.50	**0.51**	1.57	3.06
cnn2	PARAD	159.00	381.00	**35.30**	0.42	**4.50**	**10.79**
cnn2	PARAD+S	159.00	380.00	36.40	0.42	4.37	10.44
cnn2	Locks	159.00	566.00	197.00	0.28	0.81	2.87
cnn2	Worker-Local	159.00	**284.00**	94.10	**0.56**	1.69	3.02
lstm1	PARAD	168.00	249.00	46.60	0.67	3.61	5.34
lstm1	PARAD+S	168.00	248.00	44.80	0.68	3.75	5.54
lstm1	Locks	168.00	441.00	62.60	0.38	2.68	**7.04**
lstm1	Worker-Local	168.00	**147.00**	**34.20**	**1.14**	**4.91**	4.30
lstm2	PARAD	168.00	933.00	93.80	0.18	1.79	9.95
lstm2	PARAD+S	168.00	241.00	23.40	0.70	7.18	**10.30**
lstm2	Locks	168.00	442.00	71.00	0.38	2.37	6.23
lstm2	Worker-Local	168.00	**147.00**	**18.00**	**1.14**	**9.33**	8.17

Table 2: Table of benchmark results for the eight application benchmarks and four algorithm implementations. The best runtime/speedup on each benchmark is in bold font.

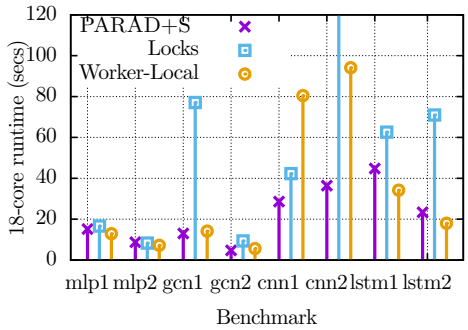

Figure 7: Comparison of the 18-core runtime of PARAD+S, Locks, and Worker-Local implementations over eight benchmarks.

AD algorithms on the multilayer perception benchmarks. On these benchmarks, most of the work is performed by library calls (using OpenBLAS) to perform matrix-vector multiplication, and all implementations include an optimization of the Adept library for concisely recording the derivative dependencies resulting from a matrix-vector multiplication. Furthermore, the network is very simple and regular.

Both the Locks and the Worker-Local implementations achieve relatively good scalability on the multilayer perceptron benchmarks. On these benchmarks, the shared weight matrices are large and used only once-per-element of a training batch in the forward pass. The average number of operations-per-statement is large (> 100) which causes Worker-Local gradient tables to be relatively efficient on 18-cores. Since these operations' contributions are well distributed and are performed on large weight matrices, however, there is little lock contention and Locks also achieves good scalability on 18-cores.

Similarly, PARAD and PARAD+S perform similarly and achieve reasonably good scalability. The optimizations outlined in Section 5 for PARAD eliminate almost all the overheads related to the organization of the deposit array. As such, the 18-core runtime is slower, but only slightly, than Worker-Local, and is still better than Locks.

Convolutional neural networks

The *cnn1* and *cnn2* benchmarks are convolutional neural networks (CNNs) based on the *lenet-5* architectures. The *cnn1* benchmark implements a modernized version of *lenet-5* using maxpooling layers and linear rectifiers as activation functions. The *cnn2* benchmark implements the *lenet-5* architecture as it was originally described in [22,48] with average pooling and the use of the *tanh* activation function. For both networks, we verified that we achieve the expected accuracy for these well known networks after training for sufficiently many epochs. The performance results are shown in Table 2.

The Worker-Local and Locks implementations scale poorly on the *cnn* benchmarks. The average number of operations-per-statement is approximately 2 on *cnn1* and 8 on *cnn2*. As such, Worker-Local performs substantially more work when executing on 18-cores than it does when it executes serially. As such, Worker-Local achieves just 3x self-relative speedup on the *cnn* benchmarks on 18-cores. Locks performs similarly poorly on the *cnn2* benchmark, but performs better on *cnn1* — achieving 7.4x self-relative speedup. Both of the *cnn* benchmarks have many parallel updates to small shared weight matrices which causes scalability issues for Locks. The *cnn1* benchmark, however, has a lot of dynamic sparsity (many gradients are zero) which substantially reduces lock contention.

The PARAD and PARAD+S algorithms scale well on the *cnn* benchmarks each achieving 10–11x self-relative speedup. Differences between the SPTape and Adept stack data structures results in a performance hit on these benchmarks that causes somewhat worse overheads. As a result, the speedup relative to Adept on *cnn* is about 4.5x on 18-cores for PARAD and PARAD+S.

Graph convolutional networks

The graph convolution network (GCN) benchmark *gcn1* performs community detection on the *pubmed* network [46]. The input embeddings for the vertices are sparse bags of words vectors and the network uses a single graph convolutional layer that learns an embedding of dimension 32 for each vertex in the graph. We follow the training method described in [20] and match their accuracy when training for the same number of epochs. The *gcn2* benchmark operates on the *email-Eu-core* network dataset [51,69] with random feature vectors of dimension 1024 and learns an embedding of dimen-

sion 64. The performance results are provided in Table 2.

The scalability of Locks and Worker-Local is mixed on the *gcn* benchmarks. Like *cnn* there are many parallel updates to small weight matrices, but the degree of contention varies since it depends on the structure of the graph. As such, the self-relative speedup of Locks and Worker-Local is mixed. The speedup relative to the serial Adept implementation, however, is uniformly poor (1.2–2x) for Locks on 18-cores. Worker-Local achieves 6.6x speedup relative to Adept on *gcn1* and 3.4x speedup on *gcn2*.

The scalability of PARAD and PARAD+S is better than Locks and Worker-Local on both *gcn* benchmarks. On *gcn1* they achieve 8.5x self-relative speedup, and on *gcn2* achieve 6–6.5x self-relative speedup. Relative to the serial adept code, they are about 7x faster on *gcn1* and 4–4.6x faster on *gcn2* on 18-cores. The PARAD+S algorithm is the best performing algorithm on *gcn1*, by a small margin. The more complex parallel structure in the *gcn* benchmarks causes the more sophisticated optimizations in PARAD+S to be, subtly, visible.

Long short-term memory networks

The long short-term memory network (LSTM) [34] benchmarks *lstm1* and *lstm2* implement a recurrent neural network to generate text given examples. We used a subset of the Paul Graham dataset consisting of 500 100-character data points, using a one-hot encoding of each character. The primary difference between *lstm1* and *lstm2* is that *lstm1* has very little parallelism. The *lstm2* benchmark expresses additional fine-grained parallelism and allows some algorithms to achieve improved performance.

On the *lstm* benchmarks, the Worker-Local implementation performs very well and Locks performs poorly. The locking overheads are significant on the *lstm* networks and so the scalability of Locks relative to Adept is poor 2x, even though its self-relative speedup is fairly good. The Worker-Local implementation scales relatively well achieving 4x and 8x self-relative speedup on *lstm1* and *lstm2* respectively. Worker-Local is pretty efficient relative to Adept on the *lstm* benchmarks as well since there are hundreds of operations, on average, per statement. As such, Worker-Local achieves 5x and 9x speedup relative to Adept on *lstm1* and *lstm2*.

The differences between PARAD and PARAD+S are very apparent in the *lstm* benchmarks. Although they perform similarly to one another on *lstm1*, the expression of additional fine-grained parallelism in *lstm2* causes an increase in PARAD's constant overheads. The additional optimizations in PARAD+S, however, enable it to perform consistently across both *lstm1* and *lstm2*. PARAD+S achieves 7x and 10.3x self-relative speedup on *lstm1* and *lstm2* respectively, and achieves 3.7x and 7x speedup relative to the serial Adept code. Although not as good as Worker-Local on these benchmarks, PARAD+S is not too far behind in terms of performance and scalability.

Overall performance of PARAD+S

Figure 7 illustrates the 18-core runtime of PARAD+S, Locks, and Worker-Local. The performance of PARAD+S is fairly robust across the benchmarks. On the two *lstm* benchmarks, PARAD+S is slower than Worker-Local by about 30% on average. On the remaining six benchmarks, PARAD+S is either the best or similar to the best performing implementation.

7 Related work

This section discusses related work on parallel AD.

Some previous work has examined parallelization of forward-mode AD. Forward-mode differentiation can be accomplished using ***dual numbers*** that tie to each parameter x the infinitesimal ϵ_x and perform computations on $\tilde{x}=(x+\epsilon_x)$. Hovland *et al.* have explored approaches that augment MPI communications to transmit dual numbers between nodes, in order to parallelizing forward-mode AD for MPI programs [36,37]. Forward-mode AD is less efficient than reverse-mode AD, however, for many applications, including machine-learning applications, for which the function $F: \mathbf{R}^n \rightarrow \mathbf{R}^m$ being differentiated has a low-dimensional output, that is, $m \ll n$. Bücker *et al.* [16] examine parallel forward and reverse-mode AD for OpenMP programs when $\min m,n$ is large. In contrast, PARAD's parallelization of reverse-mode AD is work-efficient and scalable, even when $m=1$.

Prior work has developed specialized parallel reverse-mode AD algorithms for specific computations. Gremse *et al.* developed optimized reverse-mode AD computations on GPUs of four input functions [29]. Hückelheim *et al.* developed a parallel reverse-mode AD algorithm for compressible flow solvers for unstructured meshes [38]. Other parallel reverse-mode AD algorithms have been devised for stencil computations [40,41], and convolutional neural networks [39]. The code transformations employed for these parallel reverse-mode AD algorithms do not generalize to handle arbitrary recursive fork-join programs. PARAD, meanwhile, handles arbitrary recursive fork-join parallel programs.

Prior work has developed several systems that support parallel reverse-mode AD. In the context of a parallel plasma simulation code, Bischof *et al.* explored parallel reverse-mode AD for OpenMP programs, using OpenMP-thread-local instances of ADOL-C [6]. Substantial prior work has explored parallel reverse-mode AD in message-passing programs [36, 37]. Schanen *et al.* [61] have developed an adjoint MPI library, which provides appropriate MPI communications to parallelize reverse-mode AD, based on communications in a given MPI program. These systems are not guaranteed to be work-efficient, because of the overheads they incur to combine parallel gradients. In contrast, the PARAD parallel AD algorithm targets shared-memory multicore systems and is provably work-efficient and scalable.

8 Conclusion

This paper presents PARAD, a work-efficient and scalable reverse-mode AD algorithm for determinacy-race-free recursive fork-join programs. PARAD performs reverse-mode AD on a given program with scalability similar to the input program and bounded contention. We have observed that PARAD works well in practice, achieving

good work efficiency and self-relative speedups on eight machine-learning benchmarks.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification Version 1.0*. Sun Microsystems, Inc., Mar. 2008.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[3] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, Mar. 2009.

[4] R. Barik, Z. Budimlić, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 735–736, Orlando, Florida, USA, 2009. ACM.

[5] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):55955637, Jan. 2017.

[6] C. Bischof, N. Guertler, A. Kowarz, and A. Walther. Parallel reverse mode automatic differentiation for openmp programs with adol-c. In *Advances in Automatic Differentiation*, pages 163–173, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] C. H. Bischof. Issues in parallel automatic differentiation. In *Proceedings of the 1991 International Conference on Supercomputing*, pages 146–153. ACM Press, 1991.

[8] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of Principles and Practice of Parallel Programming*, pages 181–192, 2012.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[10] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 1998.

[11] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[12] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, Oct. 1998. Technical Report, University of Texas at Austin.

[13] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[14] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. JAX: Composable transformations of Python+NumPy programs, 2018.

[15] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 1974.

[16] H. M. Bücker, B. Lang, D. an Mey, and C. H. Bischof. Bringing together automatic differentiation and openmp. In *Proceedings of the 15th International Conference on Supercomputing*, ICS 01, page 246251, New York, NY, USA, 2001. Association for Computing Machinery.

[17] R. Burden, J. Faires, and A. Burden. *Numerical Analysis*. Cengage Learning, 2015.

[18] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, 2011.

[19] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[20] J. Chen, T. Ma, and C. Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[22] L. Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[23] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 1989.

[24] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *JPDC*, 74(12):3202–3216, 2014.

[25] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[26] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, 2009.

[27] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[28] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 1966.

[29] F. Gremse, A. Höfter, L. Razik, F. Kiessling, and U. Naumann. Gpu-accelerated adjoint algorithmic differentiation. *Computer Physics Communications*, 200:300–311, 2016.

[30] A. Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[31] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 24–34, 2015.

[32] L. Hascoet and V. Pascual. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3), May 2013.

[33] M. Hitz, J. Grabmeier, E. Kaltofen, and V. Weispfenning. *Computer Algebra Handbook: Foundations·Applications·Systems*. Springer Berlin Heidelberg, 2012.

[34] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[35] R. J. Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):26, 2014.

[36] P. Hovland and C. Bischof. Automatic differentiation for message-passing parallel programs. In *IPPS*, pages 98–104, March 1998.

[37] P. D. Hovland, C. H. Bischof, and L. Roh. Automatic differentiation of a parallel molecular dynamics application. In *PPSC*. SIAM, 1997.

[38] J. Hückelheim, P. Hovland, M. M. Strout, and J.-D. Mller. Reverse-mode algorithmic differentiation of an openmp-parallel compressible flow solver. *The International Journal of High Performance Computing Applications*, 33(1):140–154, 2019.

[39] J. Hückelheim and P. D. Hovland. Automatic differentiation of parallelised convolutional neural networks - lessons from adjoint pde solvers. In *NIPS*, 2017.

[40] J. Hückelheim, N. Kukreja, S. H. K. Narayanan, F. Luporini, G. Gorman, and P. Hovland. Automatic differentiation for adjoint stencil loops. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[41] J. C. Hückelheim, P. D. Hovland, M. M. Strout, and J.-D. Müller. Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation. *Optimization Methods and Software*, 33:672–693, 2018.

[42] M. Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3(25):602, 2018.

[43] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[44] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. *Transactions on Parallel Computing*, 3(1):2:1–2:31, 2016.

[45] G. Kedem. Automatic differentiation of computer programs. Technical report, WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER, 1976.

[46] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[47] D. Lea. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*, pages 36–43, 2000.

[48] Y. LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20, 2015.

[49] D. Leijen and J. Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. Available from `http://msdn.microsoft.com/magazine/`.

[50] C. E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[51] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–es, 2007.

[52] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, Jun 1976.

[53] D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[54] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS '11)*, Oct. 2011.

[55] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[56] OpenMP application program interface, version 3.0. Available from http://www.openmp.org/mp-documents/spec30.pdf, May 2008.

[57] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[58] L. B. Rall and G. F. Corliss. An introduction to automatic differentiation. *Computational Differentiation: Techniques, Applications, and Tools*, 89, 1996.

[59] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, 2012.

[60] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[61] M. Schanen, U. Naumann, L. Hascoët, and J. Utke. Interpretative adjoints for numerical simulation codes using mpi. *Procedia Computer Science*, 1(1):1825 – 1833, 2010. ICCS 2010.

[62] T. B. Schardl, T. Denniston, D. Doucet, B. C. Kuszmaul, I.-T. A. Lee, and C. E. Leiserson. The CSI framework for compiler-inserted program instrumentation. *POMACS*, 1(2):43:1–43:25, Dec. 2017.

[63] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding recursive fork-join parallelism into llvms intermediate representation. *ACM Trans. Parallel Comput.*, 6(4), Dec. 2019.

[64] J. Shun. *Shared-Memory Parallelism Can be Simple, Fast, and Scalable*. Morgan & Claypool, 2017.

[65] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

[66] F. Srajer, Z. Kukelova, and A. Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*, 33(4-6):889–906, 2018.

[67] A. Walther, A. Griewank, and O. Vogel. Adol-c: Automatic differentiation using operator overloading in c++. In *PAMM: Proceedings in Applied Mathematics and Mechanics*, volume 2, pages 41–44. Wiley Online Library, 2003.

[68] R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8):463464, Aug. 1964.

[69] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 555–564, 2017.