Algorithmic differentiation in high-performance computing

Jan Hückelheim, Imperial College London

Also:
Paul Hovland, Argonne National Laboratory
Michelle Strout, University of Arizona
Jens-Dominik Müller, Queen Mary University of London

November 29, 2017

Overview

- Motivation:
  - Why Algorithmic Differentiation (AD)
  - Why reverse-mode
  - Why shared-memory parallel?
- What are the challenges?
- What are previous solutions?
- Some new approaches for AD of parallel codes, for
  - simple cases (SSMP),
  - slightly harder cases (TF-MAD),
- and some outlook: manual optimisation + verification

Algorithmic differentiation (AD)

- Given a program ("primal") that implements some function

$$J = F(\alpha),$$

- algorithmic differentiation generates a new program that implements its derivative.

Algorithmic differentiation (AD)

There are two fundamentally different modes:

Tangent mode

- Computes the Jacobian-vector product

$$\dot{J} = (\nabla F(x)) \cdot \dot{\alpha}.$$

- Derivatives are propagated along with the original computation.

Adjoint mode

- Computes the transpose Jacobian-vector product

$$\bar{\alpha} = (\nabla F(x))^T \cdot \bar{J}.$$

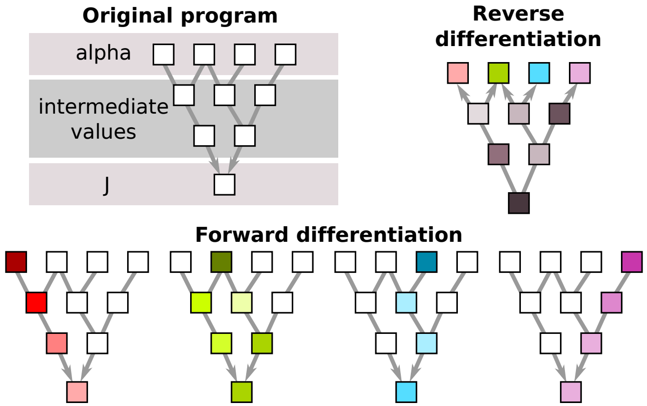
- Path through original computation is traced, derivatives are propagated in reverse order.

Why would we want AD?

- Example: A fluid dynamics code that computes pressure loss in a pipe, subject to pipe geometry.
- AD computes derivative of pressure loss wrt. design parameters.
- We could do shape optimisation, and design a pipe with less air resistance.
- There are many useful optimisation applications: reducing aircraft noise and fuel consumption, stiffness of bridges, heat transfer of cooling fins, arrangement of tidal turbines...
- Other applications: Machine learning, high-frequency trading...

Forward vs. reverse

- Tangent mode is simple to understand and implement, but: Need to re-run for every input.
- Adjoint mode is cheaper for many inputs and few outputs (run once, get all directional derivatives).



AD approaches

There are at least two ways of implementing AD:

Source-to-source transformation

- Creates code that computes partial derivative of each operation, and assembles them with chain-rule.
- Fast, efficient, but hard to get right. Mainly Fortran/C

Operator overloading

- Trace the computation at runtime, compute adjoints based on trace. Slow, huge memory footprint, easy to implement. Works for most high-level languages.

Optimist: Source transformation efficient, operator overloading easy to use.

Source transformation example

- Each instruction is augmented by its derivative instruction
- Variables are augmented by derivative variables to store accumulated chain-rule result

```
SUBROUTINE FOO(a, b)
  IMPLICIT NONE
  REAL :: a, b
  INTRINSIC SIN
  b = SIN(a)
  b = 2.0*b
END SUBROUTINE FOO
```

$\longrightarrow$

```
SUBROUTINE FOO_D(a, ad, b, bd)
  IMPLICIT NONE
  REAL :: a, b
  REAL :: ad, bd
  INTRINSIC SIN
  bd = ad*COS(a)
  b = SIN(a)
  bd = 2.0*bd
  b = 2.0*b
END SUBROUTINE FOO_D
```

- Operator overloading would instead trace the forward computation, and then apply an interpreter to that trace to compute derivatives.

Why do we need AD for parallel code?

- Processors are no longer becoming much faster at sequential tasks.

Parallelism is used to speed up large computations.

- More compute nodes (each node with its own memory and processor)
- More cores (each processor can do several unrelated things at once)
- Vectors (each core can apply the same operation to multiple values)

Each of these lends itself to different programming models:

- Message-passing (e.g. MPI)
- Shared-memory parallelism (Pthreads, OpenMP, OpenACC)
- SIMD/SIMT vectorisation (intel intrinsics, OpenMP, CUDA, OpenCL)

AD for MPI

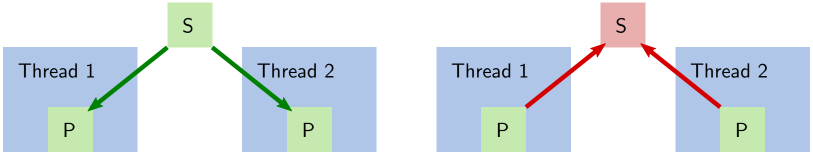
- In principle, everything is easy.
- If the original code sends, the adjoint code must receive.
- If the original code receives, the adjoint code must send.
- "Some minor problems" with non-blocking communication and other subtleties.
- Adjoint MPI: libraries are available, and used in practice.

AD for multi-core/many-core/SIMD

- Most processors today have multiple cores
- Examples:
    - Intel Core i5, between 2 and 6 cores
    - Intel Xeon Platinum, up to 28 cores
    - Intel XeonPhi, up to 68 cores
    - Raspberry Pi: 4 core ARM Cortex-A53
    - iPhone X: 6 cores (4+2 different cores)
- If we aren't using the cores, we are wasting resources.
- **If the original code is using all cores, the generated adjoint code should also use them!**

1-slide OpenMP course

- Multiple threads run in parallel (e.g. on multi-core CPU)
- Memory visible to all threads, no explicit communication
- Variables are shared between threads by default, but can be declared private
- Parallel read-access is fine, parallel write access is a problem



- Avoid parallel write access, or use atomic/critical sections (that can only be executed by one thread at a time)

Reverse AD and OpenMP - the challenge

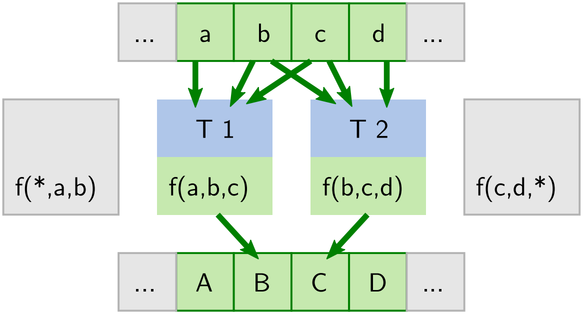
- Situation: primal code is parallelised with OpenMP.
- Source-transformation used to generate adjoint code.
- Problem 1: We don't know if and when communication between threads happens.
- Problem 2: Adjoint may have write conflicts even if the primal doesn't.

Example: parallel 1D heat/Burgers/... equation

- Finite-difference scheme:

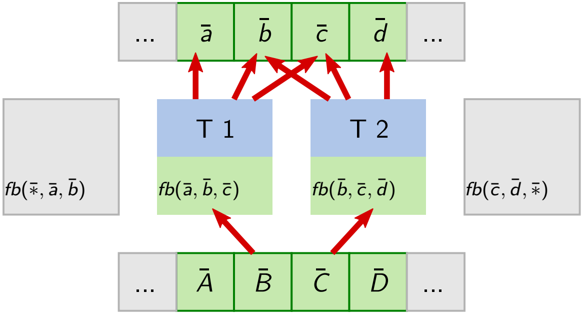
$$r_i = f(u_{i-1}, u_i, u_{i+1})$$

- r and u vectors shared
- Write access exclusive for indices in r, overlapping read access



Problem: adjoint doesn't parallelise!

- Overlapping write access to $\bar{u}$


...
$\bar{a}$
$\bar{b}$
$\bar{c}$
$\bar{d}$
...
T 1
T 2
$fb(\bar{*}, \bar{a}, \bar{b})$
$fb(\bar{a}, \bar{b}, \bar{c})$
$fb(\bar{b}, \bar{c}, \bar{d})$
$fb(\bar{c}, \bar{d}, \bar{*})$
...
$\bar{A}$
$\bar{B}$
$\bar{C}$
$\bar{D}$
...

Exclusive read access

- Overlapping write access to $\bar{u}$ happens if there was overlapping read access from u in primal.
- We can only easily parallelise adjoint if primal had *exclusive read access**
- How can we detect this?
- What can we do otherwise?

* Förster (2014): Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP

Exclusive read access examples

- Do these loops have exclusive read access?

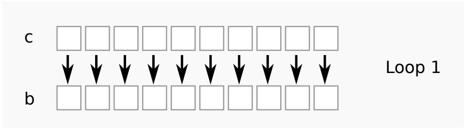
```
! Example loop 1

real, dimension(10) :: b,c

!$omp parallel do
do i=1,10
  b(i) = sin(c(i))
end do
```

Exclusive read access examples

- Do these loops have exclusive read access?

```
! Example loop 1

real, dimension(10) :: b,c

!$omp parallel do
do i=1,10
  b(i) = sin(c(i))
end do
```

- Answer: Yes



Exclusive read access examples

- Do these loops have exclusive read access?

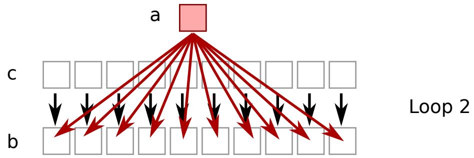
```
! Example loop 2:

real :: a
real, dimension(10) :: b,c

!$omp parallel do
do i=1,10
  b(i) = a+c(i)
end do
```

Exclusive read access examples

- Do these loops have exclusive read access?

```
! Example loop 2:

real :: a
real, dimension(10) :: b,c

!$omp parallel do
do i=1,10
  b(i) = a+c(i)
end do
```

- Answer: No



Exclusive read access examples

- Do these loops have exclusive read access?

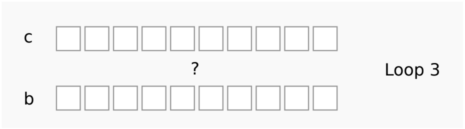
```
! Example loop 3:

real, dimension(10) :: b,c
integer, dimension(10) :: neigh
call read_from_file(neigh)

!$omp parallel do
do i=1,10
  b(i) = sin(c(neigh(i)))
end do
```

Exclusive read access examples

- Do these loops have exclusive read access?

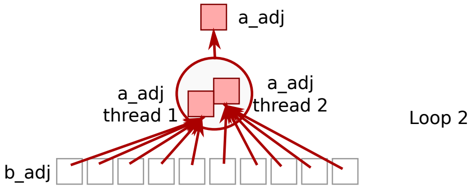
```
! Example loop 3:

real, dimension(10) :: b,c
integer, dimension(10) :: neigh
call read_from_file(neigh)

!$omp parallel do
do i=1,10
  b(i) = sin(c(neigh(i)))
end do
```

- Answer: Depends on file contents



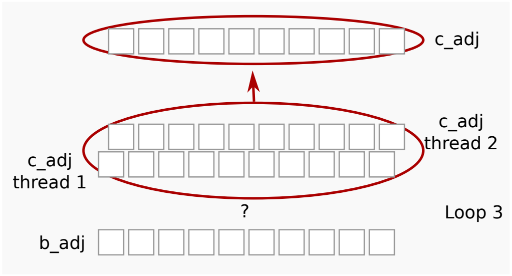
What if there's no exclusive read?

- Or: what if we are not sure?
- Use atomic operations or critical sections (potentially slow)
- Use OpenMP reduction



Reduction memory footprint

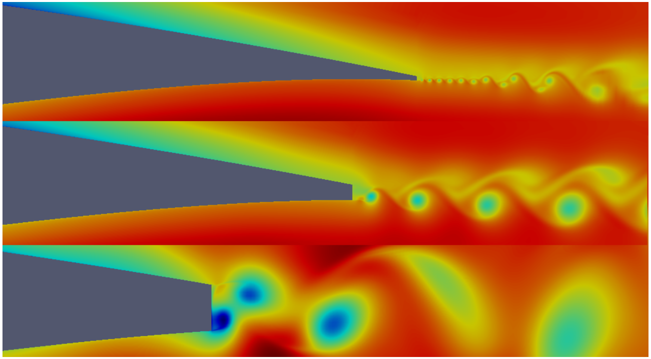
- Depending on OpenMP implementation, reduction may require temporary private copy on every thread
- What if the array is large, and we have dozens/hundreds of threads?



Case study: Unstructured CFD solver

- The Open-Source Flow Solver "STAMPS", written in F90 with OpenMP, differentiated with Tapenade*
- Unstructured compressible DNS, LES, DES, RANS, multigrid...
- Inspired by Rolls Royce's Hydra CFD code
- Mesh is loaded from file (just like "loop 3" example)

* http://www-sop.inria.fr/tropics/tapenade.html

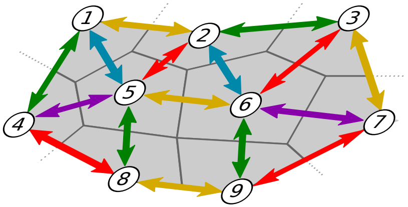


Primal parallelisation

- Solver loops over edges in graph, edge colouring to avoid conflicts.

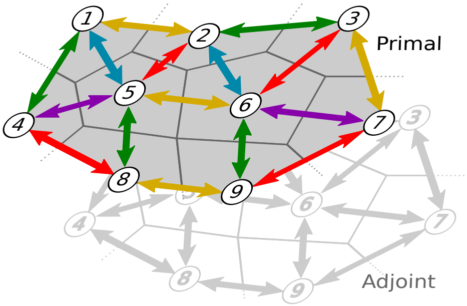
```
do colour=1,nColours
  !OMP PARALLEL DO PRIVATE(edge,i,j)
  do edge=firstEdge(colour),lastEdge(colour)
    i,j = nodes(edge)
    res(i), res(j) += flux(u(i), u(j))
  end do
end do
```

- All edges of one colour can be done in parallel



Adjoint parallelisation

- Hand-coded adjoints would simply re-use the same colouring.
- We imitate this with SSMP - *Symmetric shared-memory parallelisation*
- J. Hückelheim, P. Hovland, M. Strout, J-D. Müller (2017): Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver. International Journal of High Performance Computing Applications



How does it work?

- Static analysis can not guarantee exclusive read here (edge-node mapping only known at runtime, mesh dependent)

```
do colour=1,nColours
  !OMP PARALLEL DO PRIVATE(edge,i,j)
  do edge=firstEdge(colour),lastEdge(colour)
    i,j = nodes(edge)
    res(i), res(j) += flux(u(i), u(j))
  end do
end do
```

- But: input and output indices are the same (often detectable).
- If read access is not exclusive, then the write access isn't, either, and the code has race conditions.
- Assume that the primal code is correct, and conclude that read access is exclusive ("Garbage in, garbage out approach")

Test results

- SSMP allows more efficient parallelisation, compared with conservative approach using "atomic" pragmas.

CPU	SSMP	Atomic	Difference
BEND	1.86M	1.61M	13.4%
FEV	2.09M	1.84M	12.0%
RR	3.40M	2.58M	24.1%
MIC MIC	SSMP	Atomic	Difference
BEND	1.99M	1.43M	28.1%
FEV	1.75M	1.34M	23.4%

Number of edge residual updates per second (computational speed) for FEV, RR, and BEND with 600k nodes.

The "slightly harder" case

- Structured-mesh CFD solver, using stencil approach:
- Gather data from neighbours, update local cell. Trivial to parallelise.

Stencil codes and AD

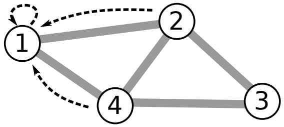
- Non-exclusive read access, causing data races in conventional adjoint.
- Non-conventional adjoint code is needed.
- TF-MAD: *transposed forward-mode algorithmic differentiation*, combines forward and reverse mode to compute adjoints using the original communication pattern.
- Idea: Split code up into segments where each segment writes to only one index. The redistribute these segments so that everything that writes to the same index is collected in the same iteration.

Parallelising the adjoint, step 1: Look at the primal

- A stencil code that pulls data from neighbours to update some value.
- Outer loop: parallel loop over all nodes i.
- Inner loop: sequential loop, reading from all neighbours of i and updating i (write/increment denoted by $\uparrow$).
- On the right: Small example mesh, we'll come back to this.

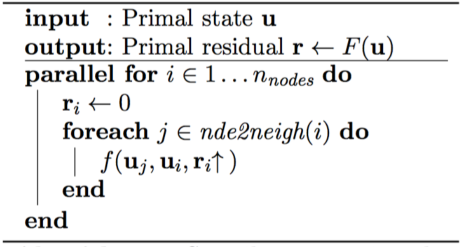
```
input  : Primal state $\mathbf{u}$
output: Primal residual $\mathbf{r} \leftarrow F(\mathbf{u})$
parallel for $i \in 1 \ldots n_{nodes}$ do
    $\mathbf{r}_i \leftarrow 0$
    foreach $j \in nde2neigh(i)$ do
        $f(\mathbf{u}_j, \mathbf{u}_i, \mathbf{r}_i\uparrow)$
    end
end
```

Primal code



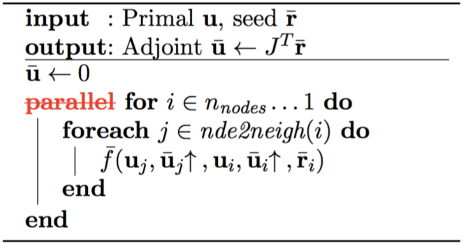
Parallelising the adjoint, step 2: Sequential adjoint

- Primal outer loop is parallel, adjoint outer loop is not.
- Reason: Every inner iteration writes to $\bar{u}_j$ (some neighbour), and maybe some other thread is writing to this at the same time.



```
input  : Primal state $\mathbf{u}$
output: Primal residual $\mathbf{r} \leftarrow F(\mathbf{u})$
parallel for $i \in 1 \ldots n_{nodes}$ do
    $\mathbf{r}_i \leftarrow 0$
    foreach $j \in nde2neigh(i)$ do
        $f(\mathbf{u}_j, \mathbf{u}_i, \mathbf{r}_i \uparrow)$
    end
end
```

Primal code



```
input  : Primal $\mathbf{u}$, seed $\bar{\mathbf{r}}$
output: Adjoint $\bar{\mathbf{u}} \leftarrow J^T \bar{\mathbf{r}}$
$\bar{\mathbf{u}} \leftarrow 0$
~~parallel~~ for $i \in n_{nodes} \ldots 1$ do
    foreach $j \in nde2neigh(i)$ do
        $\bar{f}(\mathbf{u}_j, \bar{\mathbf{u}}_j \uparrow, \mathbf{u}_i, \bar{\mathbf{u}}_i \uparrow, \bar{\mathbf{r}}_i)$
    end
end
```

Sequential Adjoint code

Parallelising the adjoint, step 3: Segmented adjoint

- Loop body is split into two segments, each writes to only one index.

```
input  : Primal $\mathbf{u}$, seed $\bar{\mathbf{r}}$
output: Adjoint $\bar{\mathbf{u}} \leftarrow J^T \bar{\mathbf{r}}$
$\bar{\mathbf{u}} \leftarrow 0$
~~parallel~~ for $i \in n_{nodes} \ldots 1$ do
    foreach $j \in nde2neigh(i)$ do
        $\bar{f}(\mathbf{u}_j, \bar{\mathbf{u}}_j\uparrow, \mathbf{u}_i, \bar{\mathbf{u}}_i\uparrow, \bar{\mathbf{r}}_i)$
    end
end
```

Sequential Adjoint code

```
input  : Primal $\mathbf{u}$, seed $\bar{\mathbf{r}}$
output: Adjoint $\bar{\mathbf{u}} \leftarrow (J)^T \bar{\mathbf{r}}$
$\bar{\mathbf{u}} \leftarrow 0$
~~parallel~~ for $i \in n_{nodes} \ldots 1$ do
    foreach $j \in nde2neigh(i)$ do
        $\bar{f}_o(\mathbf{u}_j, \bar{\mathbf{u}}_j\uparrow, \mathbf{u}_i, \bar{\mathbf{r}}_i)$
        $\bar{f}_d(\mathbf{u}_j, \mathbf{u}_i, \bar{\mathbf{u}}_i\uparrow, \bar{\mathbf{r}}_i)$
    end
end
```

Segmented Adjoint code

Parallelising the adjoint, step 3: Redistributed adjoint

- “Transpose the off-diagonal term”
- Why does this work? See next slides.

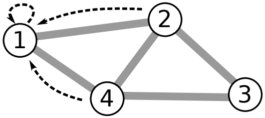
```
input  : Primal $\mathbf{u}$, seed $\bar{\mathbf{r}}$
output: Adjoint $\bar{\mathbf{u}} \leftarrow (J)^T \bar{\mathbf{r}}$
$\bar{\mathbf{u}} \leftarrow 0$
~~parallel~~ for $i \in n_{nodes} \dots 1$ do
    foreach $j \in nde2neigh(i)$ do
        $\bar{f}_o(\mathbf{u}_j, \bar{\mathbf{u}}_j\uparrow, \mathbf{u}_i, \quad \bar{\mathbf{r}}_i)$
        $\bar{f}_d(\mathbf{u}_j, \quad \mathbf{u}_i, \bar{\mathbf{u}}_i\uparrow, \bar{\mathbf{r}}_i)$
    end
end
```

Segmented Adjoint code

```
input  : Primal $\mathbf{u}$, seed $\bar{\mathbf{r}}$
output: Adjoint $\bar{\mathbf{u}} \leftarrow (J)^T \bar{\mathbf{r}}$
parallel for $i \in n_{nodes} \dots 1$ do
    $\bar{\mathbf{u}}_i \leftarrow 0$
    foreach $j \in nde2neigh(i)$ do
        $\bar{f}_o(\mathbf{u}_i, \bar{\mathbf{u}}_i\uparrow, \mathbf{u}_j, \quad \bar{\mathbf{r}}_j)$
        $\bar{f}_d(\mathbf{u}_j, \quad \mathbf{u}_i, \bar{\mathbf{u}}_i\uparrow, \bar{\mathbf{r}}_i)$
    end
end
```

Redistributed Parallel Adjoint

Why does this work? Back to the example mesh.



- The maths for this mesh:

$$\vec{r} = \underbrace{\begin{bmatrix} f(\vec{u}_2,\vec{u}_1)+f(\vec{u}_4,\vec{u}_1) \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{\text{(let's look at this term in detail)}} + \begin{bmatrix} 0 \\ f(\vec{u}_1,\vec{u}_2)+f(\vec{u}_3,\vec{u}_2)+f(\vec{u}_4,\vec{u}_2) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(\vec{u}_2,\vec{u}_3)+f(\vec{u}_4,\vec{u}_3) \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ f(\vec{u}_1,\vec{u}_4)+f(\vec{u}_2,\vec{u}_4)+f(\vec{u}_3,\vec{u}_4) \end{bmatrix}. \quad (1)$$

The first inner iteration in detail

- The primal code computes $\vec{r}_1 = f(\vec{u}_2, \vec{u}_1) + f(\vec{u}_4, \vec{u}_1)$
- For any i, j, k we define $\frac{\partial f(u_i, u_j)}{\partial u_k} = \partial f_k^{i,j}$
- Then, the tangent-linear code computes

$$\dot{\vec{r}}_1 = \begin{bmatrix} (\partial f_1^{2,1} + \partial f_1^{4,1}) & \partial f_2^{2,1} & 0 & \partial f_4^{4,1} \end{bmatrix} \begin{bmatrix} \dot{\vec{u}}_1 \\ \dot{\vec{u}}_2 \\ \dot{\vec{u}}_3 \\ \dot{\vec{u}}_4 \end{bmatrix} .$$

- The adjoint code computes

$$\begin{bmatrix} \bar{\vec{u}}_1 \\ \bar{\vec{u}}_2 \\ \bar{\vec{u}}_3 \\ \bar{\vec{u}}_4 \end{bmatrix} += \begin{bmatrix} (\partial f_1^{2,1} + \partial f_1^{4,1}) \\ \partial f_2^{2,1} \\ 0 \\ \partial f_4^{4,1} \end{bmatrix} \bar{\vec{r}}_1 ,$$

All iterations together: Standard adjoint

- Every outer iteration writes almost everywhere

$$\begin{bmatrix}\bar{\vec{u}}_1\\ \bar{\vec{u}}_2\\ \bar{\vec{u}}_3\\ \bar{\vec{u}}_4\end{bmatrix} = \begin{bmatrix}(\partial f_1^{2,1} + \partial f_1^{4,1})\bar{\vec{r}}_1\\ \partial f_2^{2,1}\bar{\vec{r}}_1\\ 0\\ \partial f_4^{4,1}\bar{\vec{r}}_1\end{bmatrix} + \begin{bmatrix}\partial f_1^{1,2}\bar{\vec{r}}_2\\ (\partial f_2^{1,2} + \partial f_2^{3,2} + \partial f_2^{4,2})\bar{\vec{r}}_2\\ \partial f_3^{3,2}\bar{\vec{r}}_2\\ \partial f_4^{4,2}\bar{\vec{r}}_2\end{bmatrix}$$

$$+ \begin{bmatrix}0\\ \partial f_2^{2,3}\bar{\vec{r}}_3\\ (\partial f_3^{2,3} + \partial f_3^{4,3})\bar{\vec{r}}_3\\ \partial f_4^{4,3}\bar{\vec{r}}_3\end{bmatrix} + \begin{bmatrix}\partial f_1^{1,4}\bar{\vec{r}}_4\\ \partial f_2^{2,4}\bar{\vec{r}}_4\\ \partial f_3^{3,4}\bar{\vec{r}}_4\\ (\partial f_4^{1,4} + \partial f_4^{2,4} + \partial f_4^{3,4})\bar{\vec{r}}_4\end{bmatrix}$$

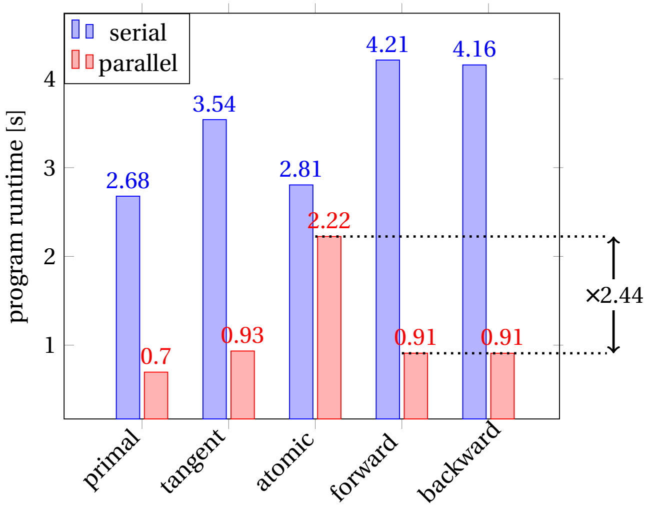
All iterations together: Reorganised adjoint

- Every outer iteration writes only to one index

$$
\begin{aligned}
[\bar{\vec{u}}_1] &= \left[(\partial f_1^{2,1} + \partial f_1^{4,1})\bar{\vec{r}}_1 + \partial f_1^{1,2}\bar{\vec{r}}_2 + \partial f_1^{1,4}\bar{\vec{r}}_4\right] \\
[\bar{\vec{u}}_2] &= \left[\partial f_2^{2,1}\bar{\vec{r}}_1 + (\partial f_2^{1,2} + \partial f_2^{3,2} + \partial f_2^{4,2})\bar{\vec{r}}_2 + \partial f_2^{2,3}\bar{\vec{r}}_3 + \partial f_2^{2,4}\bar{\vec{r}}_4\right] \\
[\bar{\vec{u}}_3] &= \left[\partial f_3^{3,2}\bar{\vec{r}}_2 + (\partial f_3^{2,3} + \partial f_3^{4,3})\bar{\vec{r}}_3 + \partial f_3^{3,4}\bar{\vec{r}}_4\right] \\
[\bar{\vec{u}}_4] &= \left[\partial f_4^{4,1}\bar{\vec{r}}_1 + \partial f_4^{4,2}\bar{\vec{r}}_2 + \partial f_4^{4,3}\bar{\vec{r}}_3 + (\partial f_4^{1,4} + \partial f_4^{2,4} + \partial f_4^{3,4})\bar{\vec{r}}_4\right]
\end{aligned}
$$

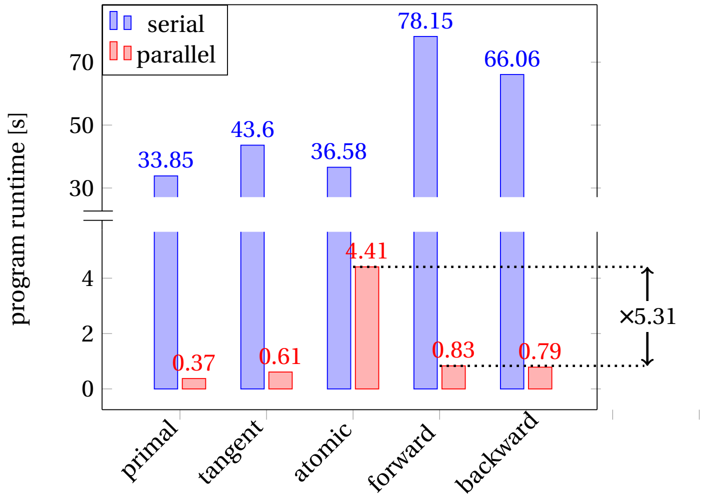
Speed of reorganised adjoint code (16 CPU threads)

- Reorganisation slows down serial code, but scales better
- Note: Serial times need recompilation without OpenMP



Speed of reorganised adjoint code (240 MIC threads)

- Overhead of atomics larger on many-core machine. Method pays off in this example.



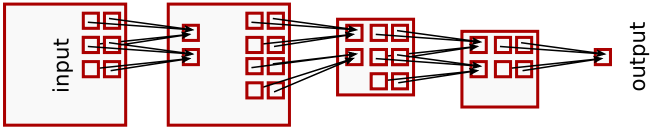
Caveats

- Both approaches shown here require certain symmetry conditions (see our papers below)
- That rules out some ways of handling boundaries (ghost-cells)
- Advantage of reorganisation will be smaller for larger loop bodies (where code duplication has larger impact)
- There are many other optimisations (polyhedral, cache-blocking, etc.) that are not taken into account.

Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver, International Journal of High Performance Computing Applications, 2017
Parallelisable adjoint stencil computations using transposed forward-mode algorithmic differentiation, in review

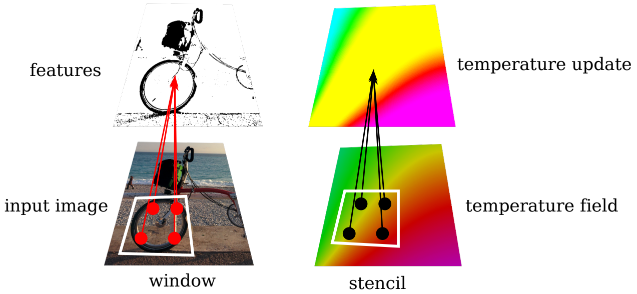
Outlook: Recent applications of AD

- Machine learning, neural networks:
- Convolutional layers, subsampling layers
- Models are "trained" to minimise misclassifications
- Training: Back-propagating to find sensitivity of overall error wrt. weights in each layer



Outlook: Recent applications of AD

- Back-propagation and reverse-mode AD is essentially the same
- CNN layers are similar to stencil computations
- CNN layers are performance-critical and often parallelised for GPU and Multi-Core (this runs on HPC systems, phones, ...)
- Can we recycle our knowledge about shared-memory parallel adjoints?



Challenges for modern applications

- Consider this trivial python function:

```
def foo(a,b):
  return a+b
```

- If this was Fortran, differentiation would be easy. But in Python:
- `a` and `b` could be integers, thus `foo` non-differentiable
- `a` and `b` could be pointers to the same memory location, requiring special treatment
- `a` and `b` could be objects with overloaded operators
- those operators can have side-effects that affect the differentiable part of the program
- All these problems apply also to C++ code with objects, templates, preprocessor or typedef
- Additionally, in scripting languages, all this can change at runtime
- AI researchers heavily use libraries from within e.g. R, Python, Julia, that are coded in other languages, e.g. TensorFlow, NumPy

Future work needed

- AD tool support for more complicated parallelisation
- AD support of vectorised code
- AD support of scripting language, mixed-language codes

Thank you

Questions?