Automatic Differentiation of Parallelised Convolutional Neural Networks - Lessons from Adjoint PDE Solvers

Jan Hückelheim, Imperial College London
Paul Hovland, Argonne National Laboratory

December 9, 2017

About me

- M.Sc. from RWTH Aachen, Germany, 2012
- PhD from Queen Mary University of London, 2017
- Research Associate at Imperial College London, present
    - Inria, work on Tapenade static analysis
    - Argonne National Laboratory, parallel AD
- AD and verification in Computational Fluid Dynamics, Seismic Imaging

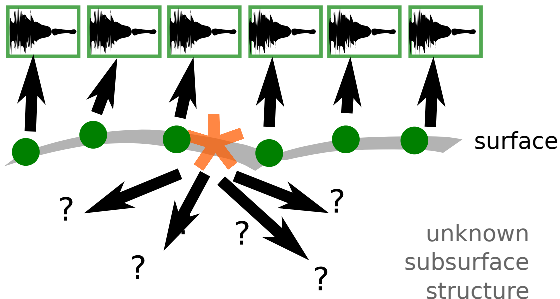
An example from PDE solvers: Seismic imaging

- Seismic imaging: Explore the subsurface geological structure
- In real life: Shots are being fired, and the reflections recorded

mic
mic
mic
shot
mic
mic
mic
surface
subsurface structure

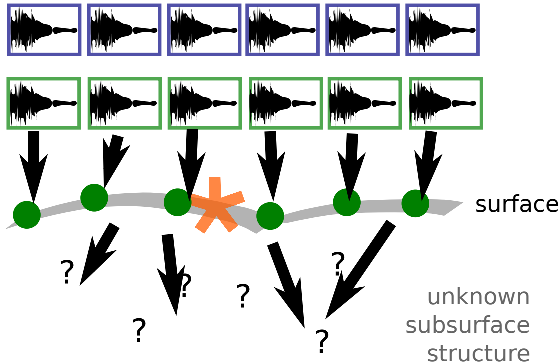
An example from PDE solvers: Seismic imaging

- In simulation, the same experiment is conducted
- Since we don't know the subsurface yet, we assume something



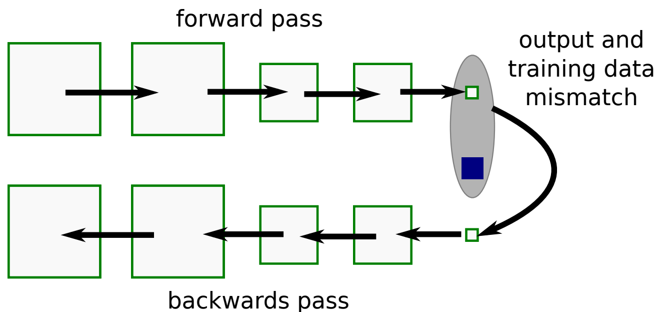
An example from PDE solvers: Seismic imaging

- Back-propagate the mismatch between simulation and measurement
- Minimise mismatch by updating assumed subsurface structure



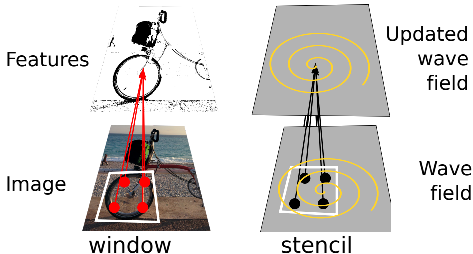
Back-propagation in CNNs

- Convolutional layers, subsampling layers, unknown weights everywhere
- Models are "trained" to minimise misclassifications



More similarities

- Stencil computations in PDE solvers look like convolutions



Note that there are also differences:

- CNNs have few layers, compared to many iterations in PDE solvers
- Loop bodies more complex in PDE solvers
- Boundary treatment is different

Let's see how much AD knowledge we can transfer.

Algorithmic differentiation (AD)

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

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

AD generates a program that implements the derivative

Tangent mode

- Computes the Jacobian-vector product

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

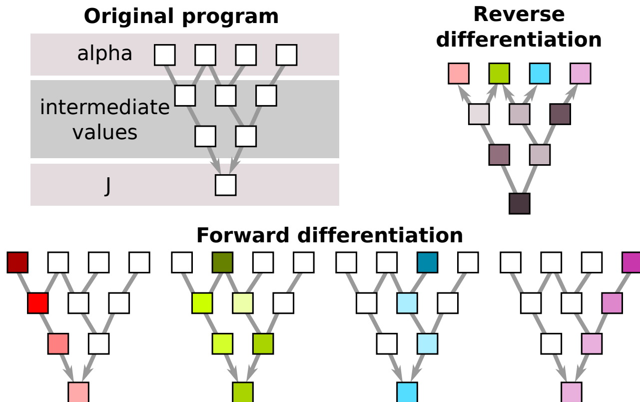
Adjoint mode

- Computes the transpose Jacobian-vector product

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

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.

Source transformation can lead to more efficient derivative codes, Operator overloading is often easier to use, better language support.

Source transformation example

- Each instruction is augmented by its derivative instruction
- Variables are augmented by derivative variables
- Data flow reversal: r receives from a and b, rb sends to ab and bb.

```
float f_d(float a, float ad, float b, float bd, float *f) {
    *f = a*b;
    return ad*b + a*bd;
}
```

forward mode

```
float f(float a, float b) {
    return a*b;
}
```

reverse mode

```
void f_b(float a, float *ab, float b, float *bb, float fb) {
    float f;
    *ab = *ab + b*fb;
    *bb = *bb + a*fb;
}
```

Why do we need AD for parallel code?

- We can't wait for faster processors.

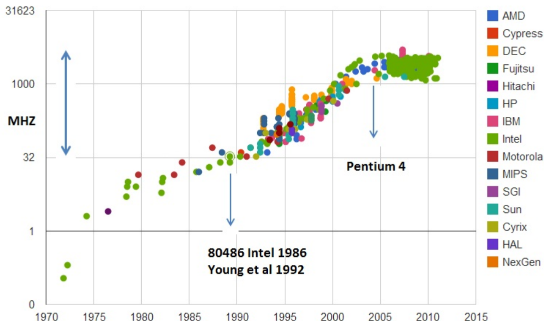


Image from https://en.wikipedia.org/wiki/File:Clock_CPU_Scaling.jpg

See also: Andrew Danowitz et.al., Recording Microprocessor History, Communications of the ACM, Vol. 55 No. 4, Pages 55-63 10.1145/2133806.2133822

Parallelism has many dimensions

- 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)

There are also performance portability frameworks.

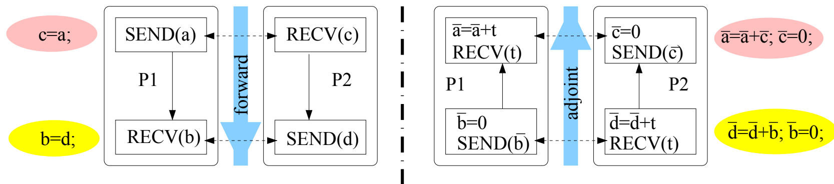
What can AD do?

- Best case: AD always generates efficient parallel codes (unrealistic)
- Second-best case: AD generates efficient parallel codes if the input was well parallelised (realistic?)

AD for MPI

- If the original code sends, the adjoint code must receive
- If the original code receives, the adjoint code must send
- Remaining problems with non-blocking communication and other subtleties
- Adjoint MPI: libraries are available, and used in practice

easy adjoints for blocking calls



Graphic: J. Utke, Adjoints of MPI programs, ECCO2 meeting slides, Argonne National Laboratory, 2008

Adjoint MPI: Some references

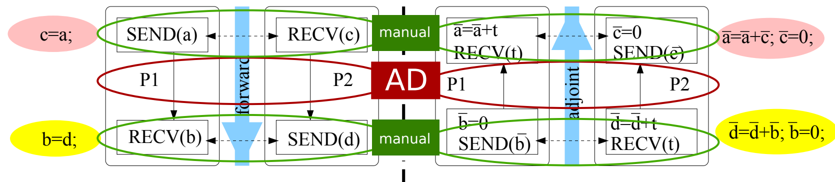
- P. Hovland, *Automatic differentiation of parallel programs*, PhD thesis, 1997
- J. Utke et al, *Toward adjoinable MPI*, IPDPS, 2009
- AdjointMPI, AMPI, with more references: `https://www.stce.rwth-aachen.de/research/software/ampi`
- AdjoinableMPI, also with more references: `https://trac.mcs.anl.gov/projects/AdjoinableMPI`

What can AD do?

- AD can generally handle this well enough for practical use.

The brutal way to adjoint MPI

- In practice, AD tool support is often not necessary
- Hand-differentiate the MPI layer, and apply AD only to some kernel



- Just make sure that `P1` and `P2` don't contain communication calls ("`grep -ri MPI`" is your friend)

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!**

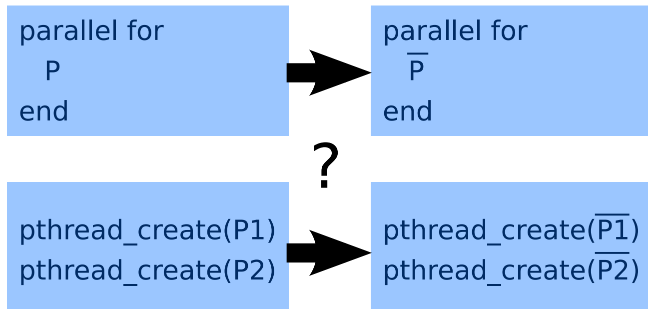
Shared-memory parallelism

- Multiple threads run in parallel (e.g. on multi-core CPU)
- Memory visible to all threads, no explicit communication
- Parallel read-access is fine, parallel write access is a problem

- Avoid parallel write access
  (if necessary, use atomic updates, critical sections or barriers)

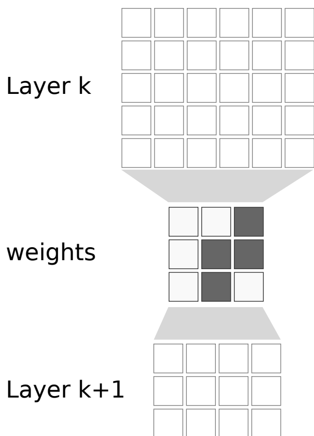
Reverse AD and OpenMP - the challenge

- Situation: primal code is parallelised with OpenMP.
- Source-transformation used to generate adjoint code.
- AD support for OpenMP, Pthreads, CUDA, OpenCL etc is poor.
- Can we use the brutal method that worked with MPI?



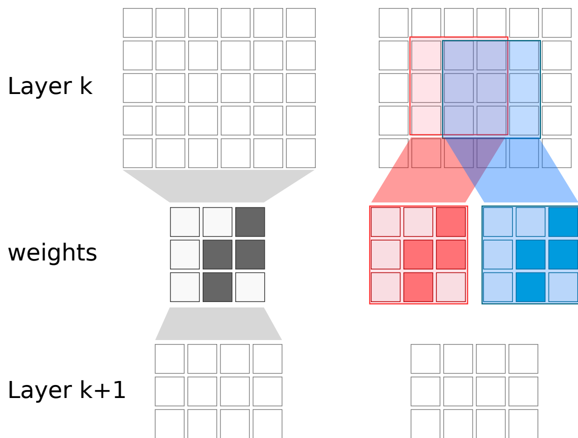
Example: a convolution

- Let's apply a filter to layer k, resulting in layer $k+1$



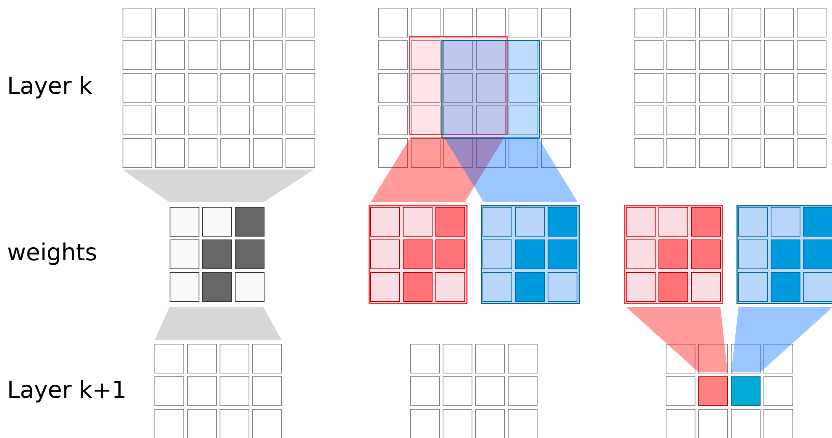
Example: a convolution

- We could do this in parallel, with two threads



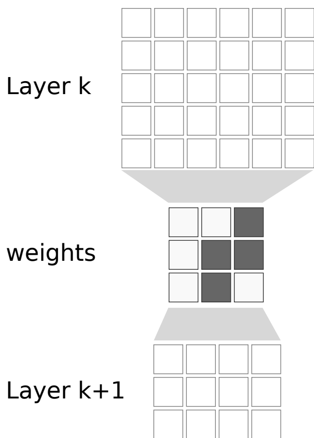
Example: a convolution

- Each thread writes to its own output index, no problem



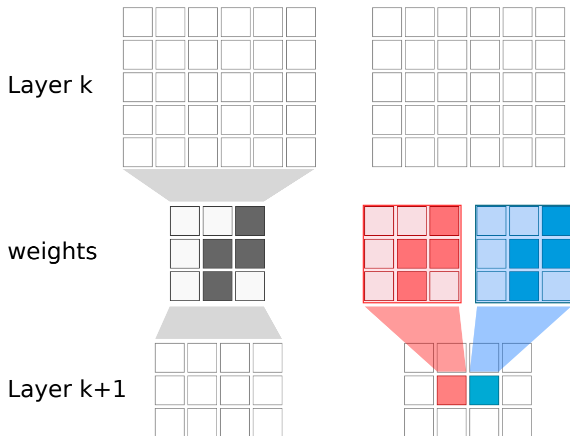
Example: a convolution

- What about the back-propagation?



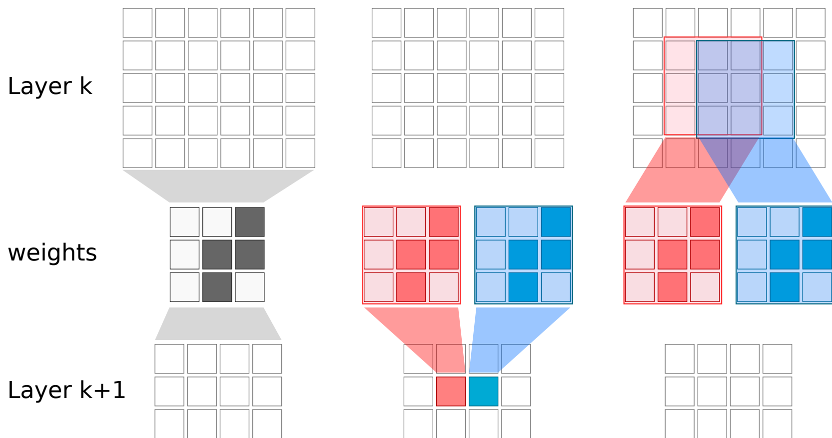
Example: a convolution

- Each thread reads from its own index...



Example: a convolution

- And scatters the result to overlapping memory regions. Conflict!



Why did this happen?

- 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*
- Reference for this: M Förster, *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*, PhD thesis, 2014

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

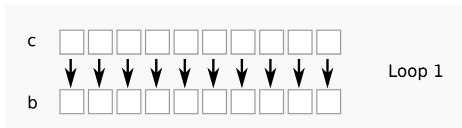
- Do these loops have exclusive read access?

```
! 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

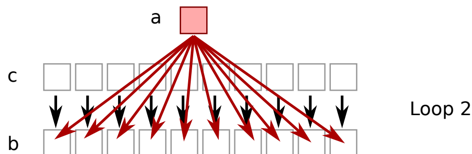
- Do these loops have exclusive read access?

```
! 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:
size = read_from_command_line(1)

!$omp parallel do
do i=1+size,10-size
  b(i) = 0
  do j=i-size,i+size
    b(i) = b(i) + c(j)
  end do
end do
```

Exclusive read access examples

- Do these loops have exclusive read access?

```
! Example loop 3:
size = read_from_command_line(1)

!$omp parallel do
do i=1+size,10-size
  b(i) = 0
  do j=i-size,i+size
    b(i) = b(i) + c(j)
  end do
end do
```

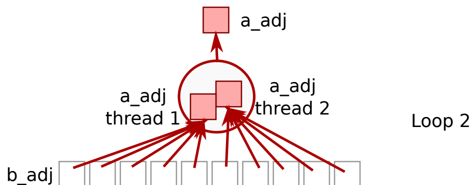
- Answer: Depends on `size`, unknown at compile time

The problem with exclusive read

- Any use of a global memory can become a problem
- Exclusive read is undecidable in general.
- Can't just use `grep` to find it.
- Are there heuristics? Maybe. (One example shown later, but mostly unexplored question)
- Can we rely on users giving pragmas?
- Can we generate several versions (efficient version, safe fallback) and decide at runtime

What if there's no exclusive read?

- Or: what if we are not sure?
- Use atomic updates (potentially slow)
- Atomic updates are acceptable if the computation is otherwise expensive enough to hide the overhead of few atomic updates
- Use OpenMP reduction (Taf does this)



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?

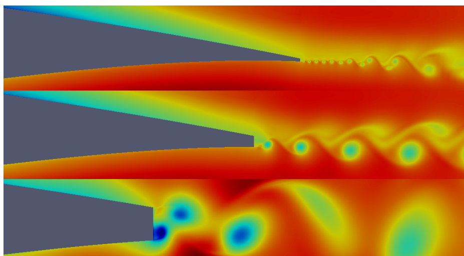
c_adj
c_adj thread 2
c_adj thread 1
?
Loop 3
b_adj

Summary so far:

- Primal parallelism does not imply adjoint parallelism (*)
- "Exclusive read" is a sufficient condition for (*) to hold.
- Exclusive read is impossible to detect in general.
- Can we detect it in practice?
- What if it doesn't hold?

Detection of exclusive read

- Static control flow, indices affine functions of loop counter? Maybe.
- Indirections, non-affine indexing, pointer arithmetic, dependence on user input? Maybe not.
- Special case where it works for complicated indexing with runtime-dependent indirections: set of read indices identical to set of write indices. In this case, exclusive read property follows from correct parallelisation of the primal (see our paper, *Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver*, 2017)

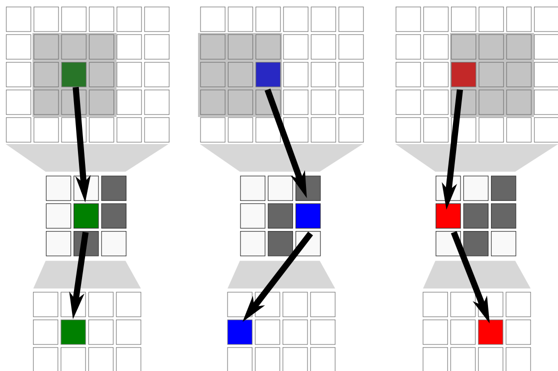


What if exclusive read doesn't hold?

- Traditional adjoint is not parallel.
- Can we do something non-traditional?
- 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.

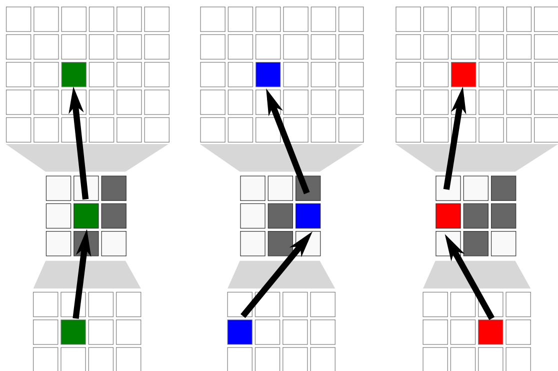
Forward stencil

- Each point in the k layer influences 9 points in the $k+1$ layer
- Right weight goes to left neighbour, left weight goes to right neighbour



Backward stencil

- Each point in k layer receives from 9 points in $k + 1$ layer
- Again, right neighbour goes through left weight, and vice versa



TF-MAD

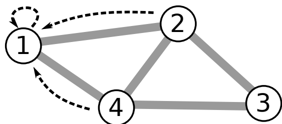
- Implement this more efficiently:
- Flip filter horizontally and vertically, switch from push to pull

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 u
output: Primal residual r ← F(u)
parallel for i ∈ 1 ... n_nodes do
    r_i ← 0
    foreach j ∈ nde2neigh(i) do
        f(u_j, u_i, r_i↑)
    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 u
output: Primal residual r ← F(u)
parallel for i ∈ 1...n_nodes do
    r_i ← 0
    foreach j ∈ nde2neigh(i) do
        f(u_j, u_i, r_i↑)
    end
end
```

Primal code

```
input : Primal u, seed r̄
output: Adjoint ū ← J^T r̄
ū ← 0
~~parallel~~ for i ∈ n_nodes...1 do
    foreach j ∈ nde2neigh(i) do
        f̄(u_j, ū_j↑, u_i, ū_i↑, r̄_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.
- Relis on multi-activity differentiation, see our paper, *Algorithmic Differentiation of Code with Multiple Context-Specific Activities*, ACM TOMS, 2017

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**
    $\bar{f}_o(\mathbf{u}_j, \bar{\mathbf{u}}_j\uparrow, \mathbf{u}_i, \quad \bar{\mathbf{r}}_i)$
    $\bar{f}_d(\mathbf{u}_j, \quad \mathbf{u}_i, \bar{\mathbf{u}}_i\uparrow, \bar{\mathbf{r}}_i)$
  end
end

Segmented Adjoint code

Parallelising the adjoint, step 3: Redistributed adjoint

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

```
input  : Primal u, seed r̄
output: Adjoint ū ← (J)^T r̄
ū ← 0
parallel for i ∈ n_nodes ... 1 do      (parallel struck out)
    foreach j ∈ nde2neigh(i) do
        f̄_o(u_j, ū_j↑, u_i,       r̄_i)
        f̄_d(u_j,       u_i, ū_i↑, r̄_i)
    end
end
```

Segmented Adjoint code

```
input  : Primal u, seed r̄
output: Adjoint ū ← (J)^T r̄
parallel for i ∈ n_nodes ... 1 do
    ū_i ← 0
    foreach j ∈ nde2neigh(i) do
        f̄_o(u_i, ū_i↑, u_j,       r̄_j)
        f̄_d(u_j,       u_i, ū_i↑, r̄_i)
    end
end
```

Redistributed Parallel Adjoint

Illustration: Standard adjoint

- Every outer iteration writes almost everywhere

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

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

Illustration: Reorganised adjoint

- Every outer iteration writes only to one index

$$\left[\bar{\vec{u}}_1\right] = \left[(\partial f_1^{2,1} + \partial f_1^{4,1})\bar{\vec{r}}_1 + \partial f_1^{1,2}\bar{\vec{r}}_2 + \partial f_1^{1,4}\bar{\vec{r}}_4\right]$$

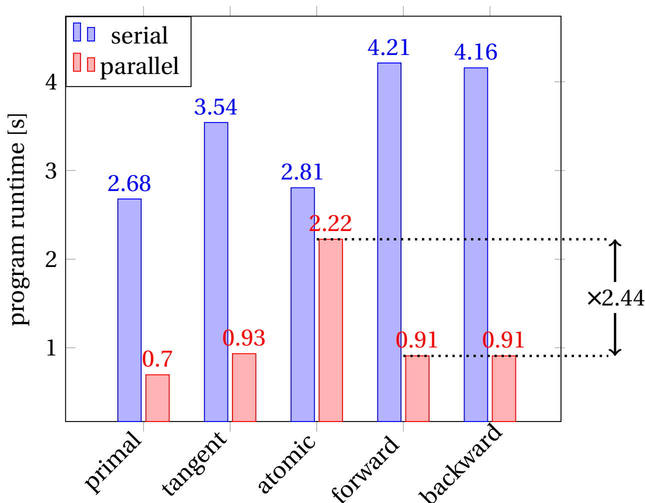
$$\left[\bar{\vec{u}}_2\right] = \left[\partial f_2^{2,1}\bar{\vec{r}}_1 + (\partial f_2^{1,2} + \partial f_2^{3,2} + \partial f_2^{4,2})\bar{\vec{r}}_2 + \partial f_2^{2,3}\bar{\vec{r}}_3 + \partial f_2^{2,4}\bar{\vec{r}}_4\right]$$

$$\left[\bar{\vec{u}}_3\right] = \left[\partial f_3^{3,2}\bar{\vec{r}}_2 + (\partial f_3^{2,3} + \partial f_3^{4,3})\bar{\vec{r}}_3 + \partial f_3^{3,4}\bar{\vec{r}}_4\right]$$

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

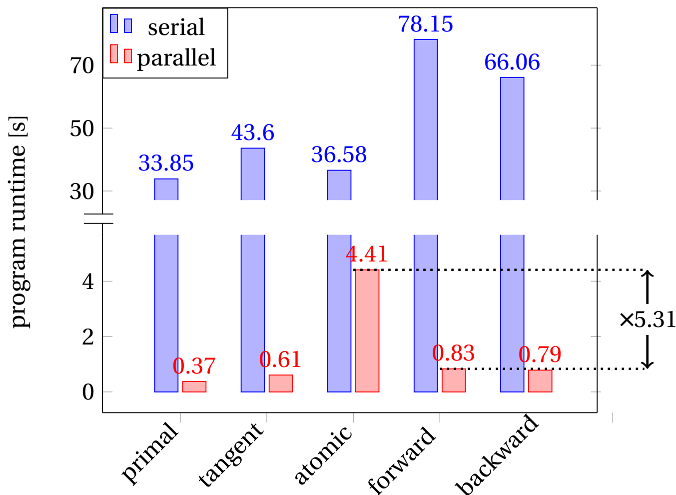
Speed of reorganised adjoint code (16 CPU threads)

- 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.



The good, the bad, the ugly

- The adjoint of a stencil computation can be a stencil computation. The adjoint of GEMM can be GEMM (and look like one to the compiler). This allows many compiler optimisations: blocking, polyhedral compilation, auto-vectorisation, ...
- Example: Polly (LLVM) can detect code that looks like GEMM, achieve speedups of up to 9X
- Both approaches shown here require certain symmetry conditions. That rules out some ways of handling boundaries. Boundaries need to be e.g. factored out into separate code part.
- Both approaches require high-level, user-given knowledge (e.g. through pragmas)
- None of this is available yet in an AD tool.

Our paper on this: *Parallelisable adjoint stencil computations using transposed forward-mode algorithmic differentiation*, in review

Future work needed

- New programming models emerge faster than AD can catch up
- Adjoint MPI: Two decades of research.
- Adjoint OpenMP: Two PhD theses so far.
- Needed: Discussion with users to get priorities right. There are more hard problems than we can solve.

Thank you

Questions?