Source-transformation adjoints for an unstructured solver with OpenMP

Jan Hückelheim*

*Queen Mary University of London

August 20, 2015

ABOUTflow

AD and OpenMP - didn't we have this already?

- OpenMP in ADOL-C for operator-overloading AD[1]
- Parallelisation of vector mode AD, parallel computation of Hessians, manual parallelisation of AD code[2]
- Toward source-transformation OpenMP[3]
- New here:
  1. **Automatic** OpenMP-parallelisation of source-transformed adjoint (i.e. it happens in the Makefile after code preparation)
  2. Hopefully **efficient** (i.e. adjoint as scalable as primal)
  3. Exploiting properties of a (very common) special case

[1] Bischof, Gürtler, Kowarz, Walther (2008): Parallel Reverse Mode Automatic Differentiation for OpenMP Programs with ADOL-C

[2] Martin Bücker et. al. (2001, 2002, 2004, 2008)

[3] Förster, Naumann, Utke (2011): Toward Adjoint OpenMP

The primal code

- MGopt: Queen Mary University of London in-house Finite Volume flow solver
- Runtime is important: some cases run for a week and more
- Much time spent in linear solver (not brute-force AD'ed)
- Other expensive part: Residual computation
  - OpenMP used for shared-memory parallelisation of primal
  - Tapenade used to create adjoint residual
  - We need a parallel adjoint residual

Primal structure

- Edge-based residual:

```
do edge=1...nEdges
  i,j = nodes(edge)
  res(i), res(j) += flux(u(i), u(j))
end do
```

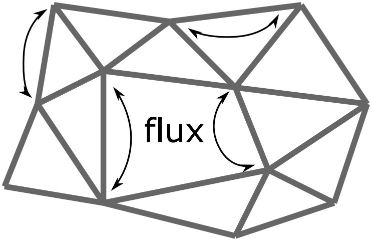
Primal structure

- Edge-based residual:

```

do edge=1...nEdges
  i,j = nodes(edge)
  res(i), res(j) += flux(u(i), u(j))
end do
```

- Equivalent to iterating over edges in the graph:



How not to parallelise this: Part 1

- Can we do this?

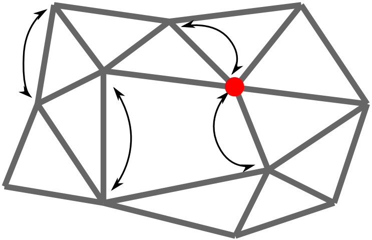
```
!$OMP PARALLEL DO PRIVATE(edge,i,j)
do edge=1...nEdges
  i,j = nodes(edge)
  res(i), res(j) += flux(u(i), u(j))
end do
```

How not to parallelise this: Part 1

- Can we do this?

```
!$OMP PARALLEL DO PRIVATE(edge,i,j)
do edge=1...nEdges
  i,j = nodes(edge)
  res(i), res(j) += flux(u(i), u(j))
end do
```

- No. There can be conflicting writes:



How not to parallelise this: Part 2

- Ok, but what about this?

```
!$OMP  PARALLEL DO PRIVATE(edge,i,j)
!$OMP& REDUCTION(+,res)
do edge=1,nEdges
  i,j = nodes(edge)
  res(i), res(j) += flux(u(i), u(j))
end do
```

How not to parallelise this: Part 2

- Ok, but what about this?

```
!$OMP  PARALLEL DO PRIVATE(edge,i,j)
!$OMP& REDUCTION(+,res)
do edge=1,nEdges
  i,j = nodes(edge)
  res(i), res(j) += flux(u(i), u(j))
end do
```

- Segfault (not enough memory):
    - local copy of `res` for each thread
    - perfect for scalars, not so if `size(res) = meshsize`.

How not to parallelise this: Part 2

- Ok, but what about this?

```
!$OMP  PARALLEL DO PRIVATE(edge,i,j)
!$OMP& REDUCTION(+,res)
do edge=1,nEdges
  i,j = nodes(edge)
  res(i), res(j) += flux(u(i), u(j))
end do
```

- Segfault (not enough memory):
  - local copy of `res` for each thread
  - perfect for scalars, not so if `size(res) = meshsize`.
- Slow:
  - every thread writes only few values, local `res` copies are sparse
  - all elements on each thread are initialised with neutral element, then everything (here mostly zeroes) is reduced

Primal parallelisation

- Solution: edge colouring

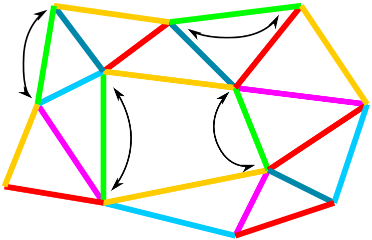
```
do colour=1,nColours
  !$OMP PARALLEL DO PRIVATE(edge,i,j)
  do edge=firstEdge(colour),lastEdge(colour)
    i,j = nodes(edge)
    res(i), res(j) += flux(u(i), u(j))
  end do
end do
```

Primal parallelisation

- Solution: edge colouring

```
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



How not to adjoint this: Part 1

- Tapenade used to treat OpenMP pragmas as regular comments
- Comments are dumped in the adjoint code at (roughly) the same location as in the primal
- Unpredictable results, or does not compile

How not to adjoint this: Part 2

- Taf can do OpenMP, Tapenade could be extended
- How good can a general purpose AD tool do here?

```
do colour=nColours,1
 !$OMP  PARALLEL DO PRIVATE(edge,i,j)
 !$OMP& REDUCTION(+,ub)
 do edge=lastEdge(colour),firstEdge(colour)
  i,j = nodes(edge)
  ub(i),ub(j) += flux_b(u(i), u(j), res(i), &
                      & res(j), resb(i), resb(j))
 end do
end do
```

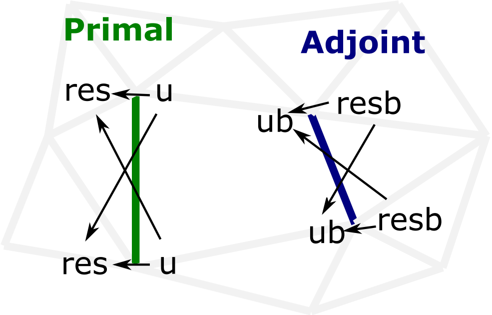
- No tool will understand our colouring on its own
- Tool has to be conservative, either use reduction (slow, memory) or atomic/critical sections (slow)

Adjoint parallelisation in principle

- Actually it is all easy: communication is symmetric!

Adjoint parallelisation in principle

- Actually it is all easy: communication is symmetric!
- Primal and adjoint read and write in the same way



Adjoint parallelisation with colouring

- The edges are still separated by colour after brute-force AD
- We can use the same OpenMP pragma as before

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

Adjoint parallelisation in practice

- How to we make this automatic? We know that:
  - if variable `foo` is private, `foob` is private
  - if variable `bar` is shared, `barb` is shared
  - **what happens to new variables that are created in adjoint code, e.g. temp1, temp2, arg1, arg2...?**
- Need some post-processing that will
  - Wipe all misplaced OpenMP statements from Tapenade output
  - Place new pragmas with correct scoping
  - **How to do this correctly without using a full Fortran parser?**

"Outlining" of parallel regions

- OpenMP compiler trick: place parallel region into new subroutine to take care of scoping[4]
- shared variables are arguments, passed call-by-reference
- private variables are local variables inside subroutine
- **idea: let's do this before passing code to AD tool**

[4] Liao et.al. (2007): OpenUH: An optimizing, portable OpenMP compiler

Source-transformed outlined code

- OpenMP defaults: everything shared by default, except loop counter (edge) and local subroutine variables (i,j)
- this is exactly what we need for the adjoint as well

```
do colour=1,nColours !forward
  !$OMP PARALLEL DO
  do edge=firstEdge(colour),lastEdge(colour)
    call flux_loopbody(edge,res,u)
  end do
end do
do colour=nColours,1 !reverse
 !$OMP PARALLEL DO
 do edge=lastEdge(colour),firstEdge(colour)
    call flux_loopbody_b(edge,res,resb,u,ub)
 end do
end do
```

Another small technicality

- Forward sweep pushes intermediate results to stack
- Reverse sweep pops them from stack
- Need to make sure that
  - each thread has its own stack
  - parallel regions are used in forward and reverse code consistently

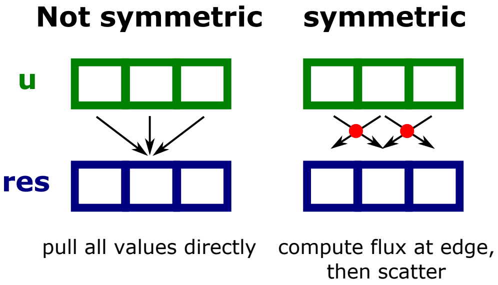
How it works, finally

- All bodies of parallel loops are placed in subroutine with special suffix (`_loopbody`)
- Python script removes all OpenMP pragmas from Tapenade output
- Before every loop that contains call to `*_loopbody` or `*_loopbody_b`, insert

  `!$OMP PARALLEL DO DEFAULT(SHARED)`
- Replace all calls to `PUSHINTEGER(n)` by `THREADPUSHINTEGER(n,numThread)`, likewise for all other push/pop routines.
- Link thread-safe stack instead of Tapenade stack

Conclusion: What works, what doesn't?

- if communication pattern is symmetric, we can automatically generate parallel AD code
- sometimes, code can be reorganised to become symmetric[5]



- some things are not symmetric. Boundaries are a problem.

[5] ask me for an example code if interested

Acknowledgments

This work has been conducted within the **About Flow** project on "Adjoint-based optimisation of industrial and unsteady flows".

`http://aboutflow.sems.qmul.ac.uk`

About Flow has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under Grant Agreement No. 317006.

This research utilised Queen Mary University of London's MidPlus computational facilities, supported by QMUL Research-IT and funded by EPSRC grant EP/K000128/1.