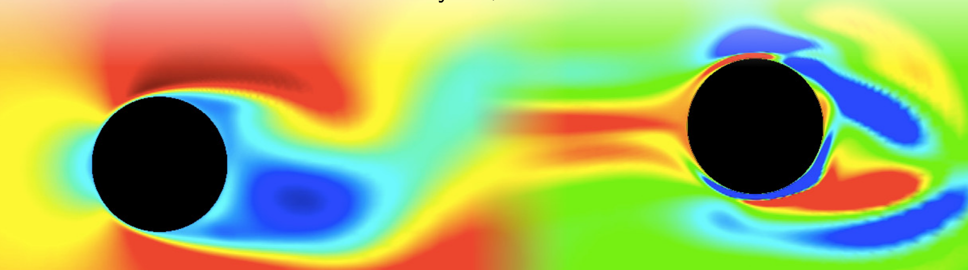
Unsteady discrete adjoint on unstructured meshes with source-transformed OpenMP 4.0

Jan Hückelheim*
Jens-Dominik Müller

Queen Mary, University of London
School of Engineering and Material Science

6th. European Conference on Computational Fluid Dynamics
July 23, 2014



Our goal

- **What we have:**
  - Unstructured, node based FV solver with geometric multigrid
  - Dual time stepping: BDF2 outer, JT-KIRK[1] implicit inner iteration
  - Adjoint generated with Tapenade[2] (and some tricks)
  - Snapshots stored at physical time steps, fixed-point loop (aka. Christianson's method) for pseudo steps
- **What we want:**
  - Parallelise it with OpenMP

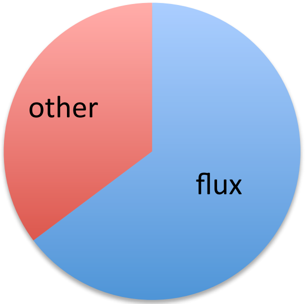
[1] S. Xu, D. Radford, M. Meyer, J-D. Müller: Stabilisation of discrete steady adjoint solvers, submitted to Journal of Computational Physics

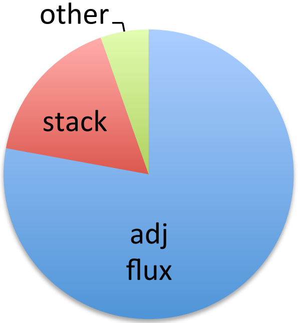
[2] L. Hascoët, V. Pascual: The Tapenade automatic differentiation tool: Principles, model, and specification, ACM Transactions on Mathematical Software Vol. 39 Issue 3, 2013

This is not as easy as it sounds

- We are not in full control of the source code
  - adjoint code made by AD
  - no hand-differentiation: automatic differentiation to ensure consistency
- High-level optimisations (replacing self-adjoint routines etc.)
- We accept preprocessing and complicated makefiles. Necessary evil, to manage differentiation procedure
- We will not hand-fix the adjoint code: It has to be automatic for consistency

We spend our time on flux calculations





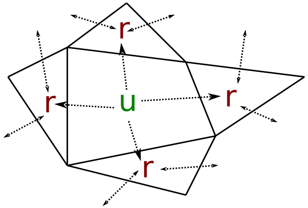
Primal

- 1st order flux for system matrix generated by Tapenade in forward mode from 2nd order fluxes

Adjoint

- 2nd order reverse flux generated by Tapenade in reverse mode
- Additional cost: taping

All fluxes have the same structure



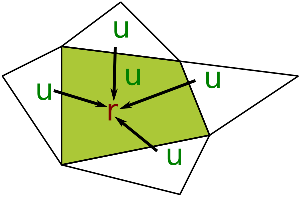
foreach *edge* **do**

$i, j \leftarrow connectivity(edge)$;
$coeff \leftarrow calc(state_{i,j})$;
$flux \leftarrow f(coeff, state_{i,j})$;
$res_{i,j} \leftarrow res_{i,j} + flux$;

end

- Iterate over all edges
- Assemble node residual, assemble system matrix
- Update node values (explicit or system solve)

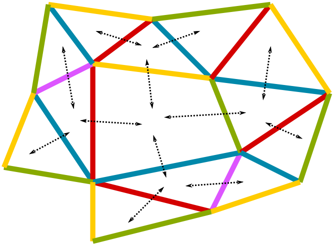
We can't simply do this in parallel



!$OMP PARALLEL
foreach *edge* **do**
  $i, j \leftarrow connectivity(edge)$;
  $coeff \leftarrow calc(state_{i,j})$;
  $flux \leftarrow f(coeff, state_{i,j})$;
  $res_{i,j} \leftarrow res_{i,j} + flux$;
end

- Conflicting writes have to be avoided

Avoiding write conflicts with colouring

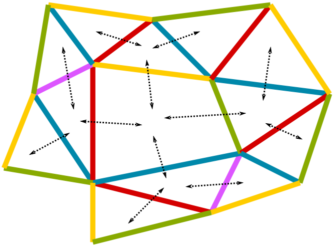


foreach *colour* **do**
  !$OMP PARALLEL
  foreach *edge in colour* **do**
    $i, j \leftarrow connectivity(edge)$;
    $coeff \leftarrow calc(state_{i,j})$;
    $flux \leftarrow f(coeff, state_{i,j})$;
    $res_{i,j} \leftarrow res_{i,j} + flux$;
  end
end

- Solution: group the edges (colour) so that we can run parallel within each colour[3].

[3] Vectorizing Unstructured Mesh Computations for Many-core Architectures I. Z. Reguly, E. Laszlo, G. R. Mudalige, M. B. Giles, Proceedings of Programming Models and Applications on Multicores and Manycores, 2014

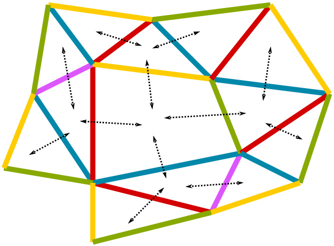
This also works in reverse



foreach *colour* **do**
  !$OMP PARALLEL
  foreach *edge in colour* **do**
    $i, j \leftarrow connectivity(edge)$;
    $fluxb \leftarrow resb_{i,j}$;
    $cvb_{i,j}, coeff_b \leftarrow f_b(fluxb)$
    $cvb_{i,j} + \leftarrow calc_b(coeff, coeff_b)$;
  end
end

- We can use the same colouring scheme for the reverse flux

This also works in reverse



foreach *colour* **do**
    !$OMP PARALLEL
    foreach *edge in colour* **do**
        $i, j \leftarrow connectivity(edge)$;
        $fluxb \leftarrow resb_{i,j}$;
        $cvb_{i,j}, coeff_b \leftarrow f_b(fluxb)$
        $cvb_{i,j} + \leftarrow calc_b(coeff, coeff_b)$;
    end
end

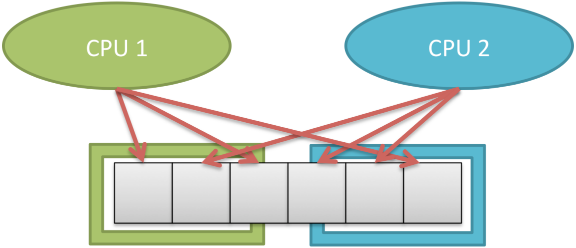
- We can use the same colouring scheme for the reverse flux
- Note the nonlinear term calc_b. We need *coeff*, so we need to store this in primal and restore it in reverse

There are some technical problems

1. Tapenade does not know OpenMP **Solution:** Hide pragmas
2. AD Colour loop is less efficient **Solution:** Hide colour loop
3. Push/pop: Storage mechanism not thread-safe. **Solution:** Implement thread-safe stack, reroute push/pop calls

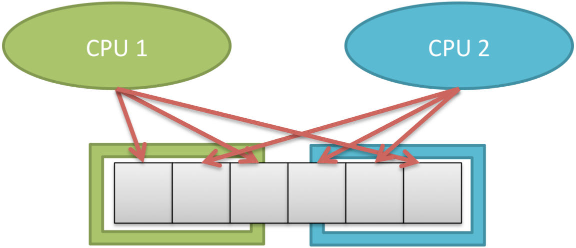
There are some technical problems

1. Tapenade does not know OpenMP **Solution:** Hide pragmas
2. AD Colour loop is less efficient **Solution:** Hide colour loop
3. Push/pop: Storage mechanism not thread-safe. **Solution:** Implement thread-safe stack, reroute push/pop calls
4. False sharing: Writes to shared variables are slow if within the cache line of another thread. Introduce local variables to reduce global writes.



There are some technical problems

1. Tapenade does not know OpenMP **Solution:** Hide pragmas
2. AD Colour loop is less efficient **Solution:** Hide colour loop
3. Push/pop: Storage mechanism not thread-safe. **Solution:** Implement thread-safe stack, reroute push/pop calls
4. False sharing: Writes to shared variables are slow if within the cache line of another thread. Introduce local variables to reduce global writes. **Advantage of source transformation: We can spot (and fix) problems like this!**



There are harder problems

1. Even if OpenMP supported (TAF): Tool can not know colouring, must assume write conflicts. Correct, but slow
2. Additional temporary variables in adjoint (`temp1`, `temp2`...) private or shared? User must understand Tapenade output
3. Danger if Tapenade changes naming of temporary variables? It will break our code (or introduce data races)
4. Poor scalability due to Taping $\Rightarrow$ more details on the following slides

A way to understand performance limits and bottlenecks

- Roofline model[4]: Show performance bottlenecks for a given code on a given platform

[4] W. Samuel: Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures, Lawrence Berkeley National Laboratory, 2009

[5] J. D. McCalpin: Memory Bandwidth and Machine Balance in Current High Performance Computers, IEEE Computer Society, 1995

A way to understand performance limits and bottlenecks

- Roofline model[4]: Show performance bottlenecks for a given code on a given platform
- **Peak performance:** Data sheet of processor/GPU/XeonPhi
- **Memory bandwidth:** customised STREAM[5] benchmark

[4]W. Samuel: Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures, Lawrence Berkeley National Laboratory, 2009

[5]J. D. McCalpin: Memory Bandwidth and Machine Balance in Current High Performance Computers, IEEE Computer Society, 1995

A way to understand performance limits and bottlenecks

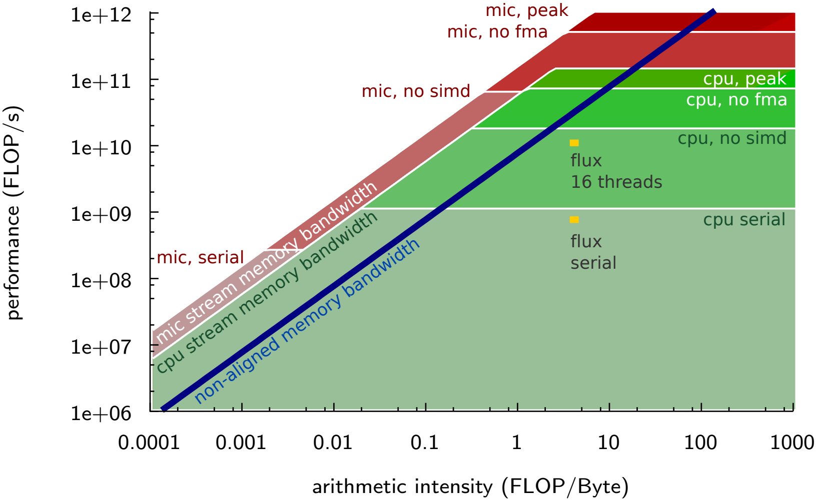
- Roofline model[4]: Show performance bottlenecks for a given code on a given platform
- **Peak performance:** Data sheet of processor/GPU/XeonPhi
- **Memory bandwidth:** customised STREAM[5] benchmark
- **Arithmetic intensity:**
  - **FLOP:** Soft-float + profiler, operator overloading with counter, **count operations in source code** → over-estimate arithmetic intensity
  - **Byte:** Assume that every iteration stays inside cache → over-estimate arithmetic intensity

---

[4] W. Samuel: Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures, Lawrence Berkeley National Laboratory, 2009

[5] J. D. McCalpin: Memory Bandwidth and Machine Balance in Current High Performance Computers, IEEE Computer Society, 1995

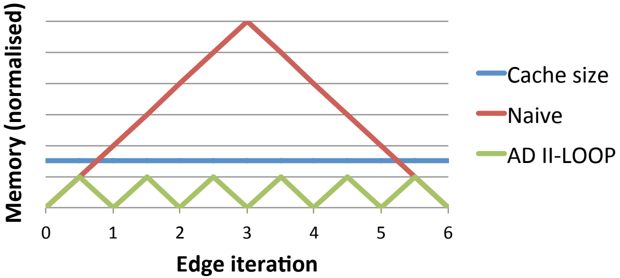
Roofline model: flux (primal)



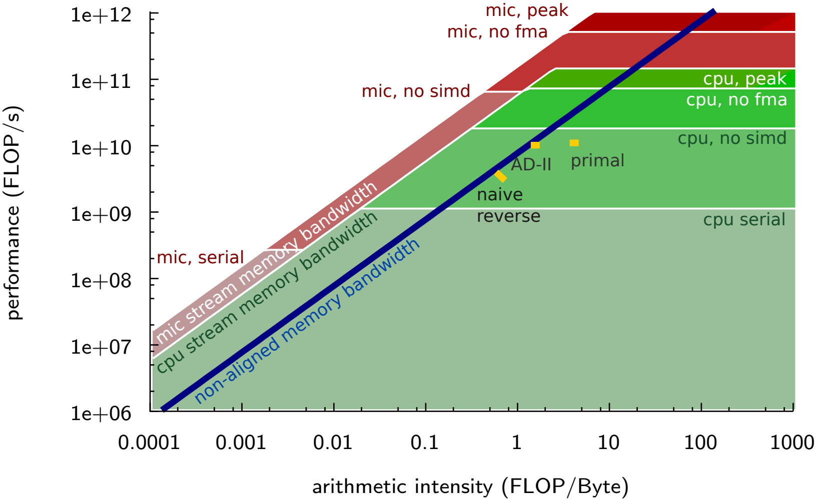
Two ways to compute reverse

foreach *edge* **do**
  $coeff \leftarrow calc(state)$;
  $store(coeff)$;
end
foreach *edge* **do**
  $restore(coeff)$;
  $state_b + \leftarrow calc_b(coeff, coeff_b)$;
end

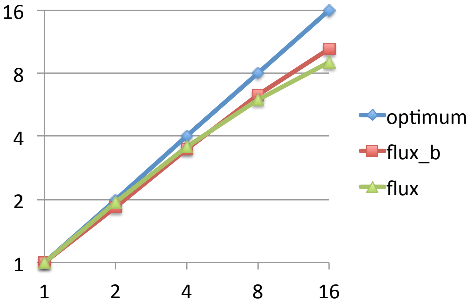
foreach *edge* **do**
  $coeff \leftarrow calc(state)$;
  $store(coeff)$;
  $restore(coeff)$;
  $state_b + \leftarrow calc_b(coeff, coeff_b)$;
end



Roofline model: flux (primal, naive adjoint, AD-II adjoint)

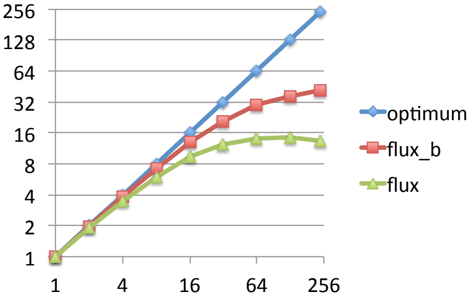


It scales well (on 2 x Xeon E5-2660)



- Scaling on the CPU is OK (considering the non-cached access)

Scaling is worse on XeonPhi 5110P



- Performance on XeonPhi is not great[6] [7] (even Intel says so[8])

[6] T. Cramer, D. Schmidl, M. Klemm, D. an Mey: Programming on Intel®Xeon Phi™Coprocessors: An Early Performance Comparison, RWTH Aachen University, 2012

[7] Vectorizing Unstructured Mesh Computations for Many-core Architectures I. Z. Reguly, E. Laszlo, G. R. Mudalige, M. B. Giles, Proceedings of Programming Models and Applications on Multicores and Manycores, 2014

[8] https://software.intel.com/en-us/articles/running-minife-on-intel-xeon-phi-coprocessors

Conclusion, future work

- OpenMP works well for source-transformed unstructured solvers (in principal), with some technical issues
- For more speed, we need to rethink the colouring: Apply colouring to larger patches to preserve some cache efficiency and allow data reusage
- XeonPhi alone is not promising: Bottlenecks similar to CPU
- More promising: Hybrid approach: OpenMP on CPU, OpenMP on XeonPhi, MPI in between
- A robust way that will not break with the next Tapenade update needs to be found

Acknowledgement

This project 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's MidPlus computational facilities, supported by QMUL Research-IT and funded by EPSRC grant EP/K000128/1.