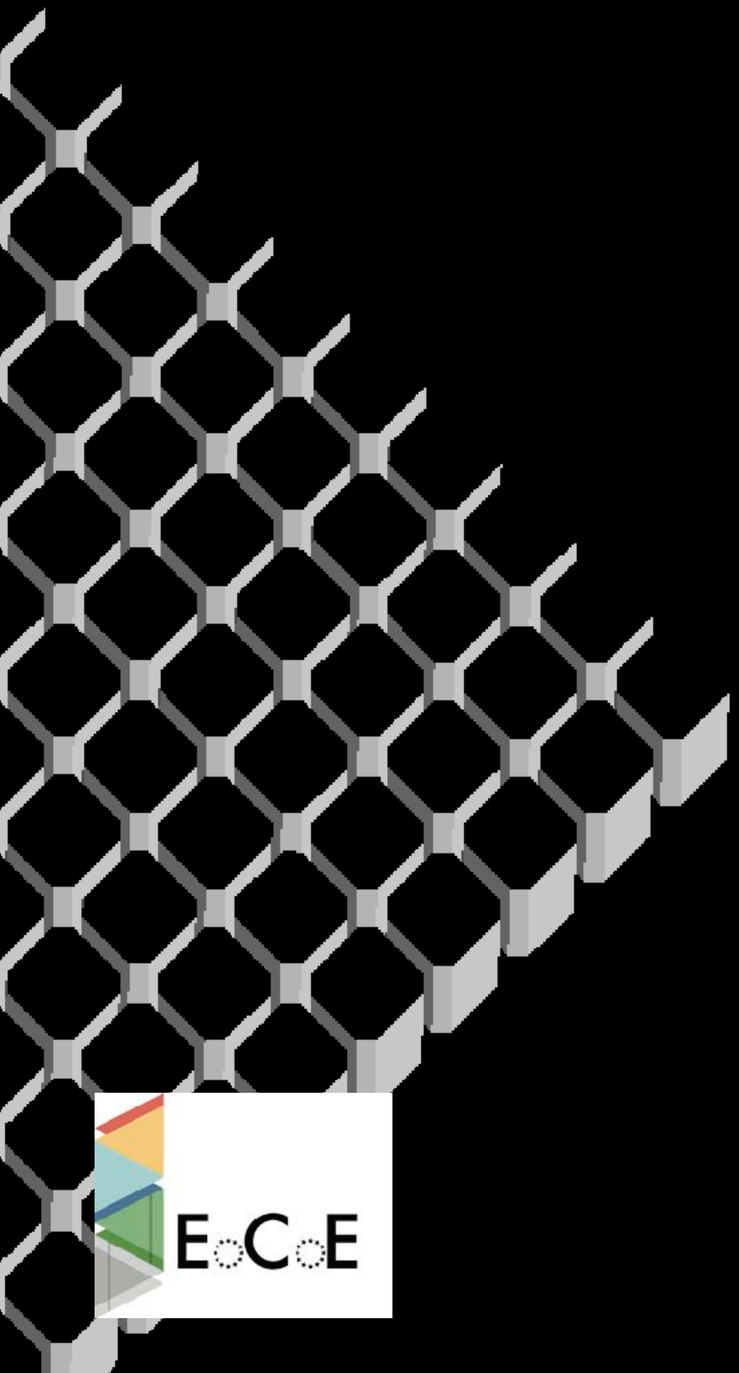


Alya - CFD on exascale GPU hardware for the Wind community.

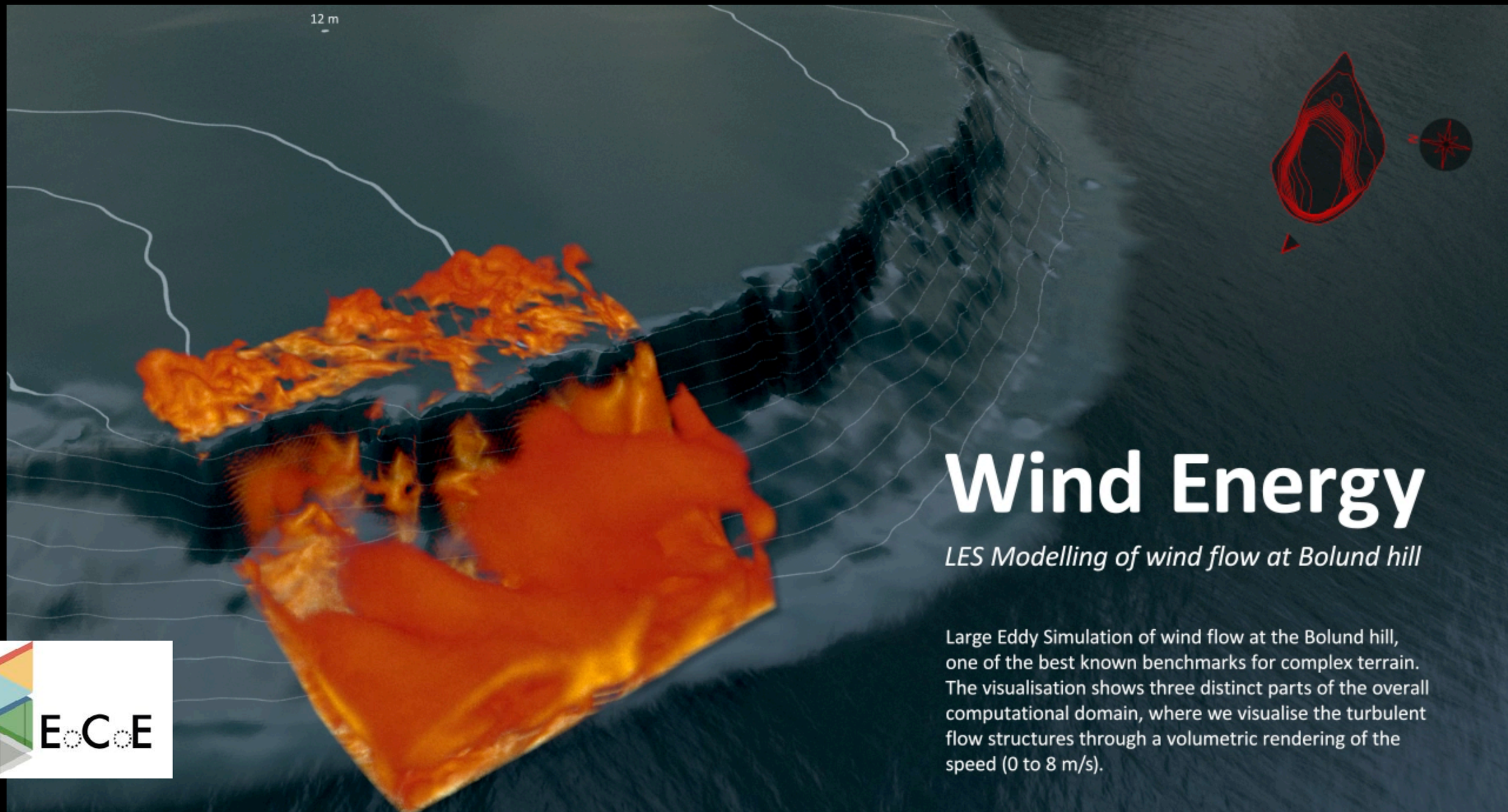
ISC 2022

Herbert OWEN and Dominik ERNST

Barcelona Supercomputing Center (BSC) - FAU University

with help from Filippo Spiga(Nvidia)
& the PSC Toolkit

Alya - Wind energy - Energy oriented Center of Excellence



Wind Energy

LES Modelling of wind flow at Bolund hill

Large Eddy Simulation of wind flow at the Bolund hill, one of the best known benchmarks for complex terrain. The visualisation shows three distinct parts of the overall computational domain, where we visualise the turbulent flow structures through a volumetric rendering of the speed (0 to 8 m/s).

E·C·E

EoCoE

Wind is a mature technology but plant-level energy losses —> 20% on flat terrain.

A better understanding of the turbulent flow is needed to reduce these losses, make wind energy cheaper, and enable widespread adoption.

Exascale simulations will enable an unprecedented understanding of a wind farm's wide range of scales.

The US is investing 6 million dollars per year in the A2e and Exawind projects. (20X EoCoE-wind)

Today, to Out-Compute is to Out- Compete. Europe needs to react accordingly.

Alya

HPC-based multi-physics simulation code

Developed at BSC to run efficiently in parallel supercomputers.

Scalability has been tested up to 100K cores.

Part of UEABS - Unified European Bench Suite for CPUs & GPUs..

Numerical Model: Alya - LES

Galerkin discretisation with explicit (RK3/4) treatment of convective and diffusive terms.

EMA - Energy, momentum and angular momentum conserving convective term.

Stabilisation for the p-v interaction coming from Laplacian approximation in Fractional Step Method.

Physical based SGS modelling (Vreman in current work).

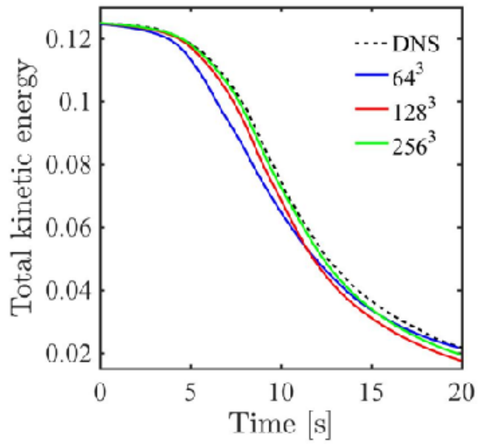
SIMPLE and no user defined numerical parameters. 

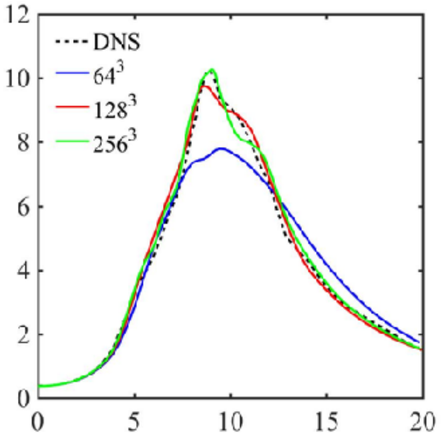
Lehmkuhl el al. A low-dissipation finite element scheme for scale resolving simulations of turbulent flows J. Comput. Phys., 308:51–65, 2019

Numerical Model: Alya - LES

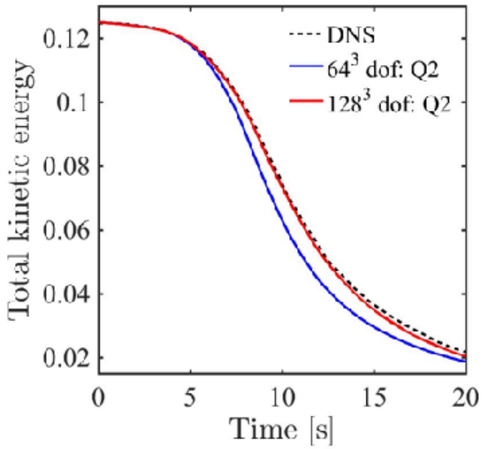
Test case: Taylor-Green vortex Re = 1600 *

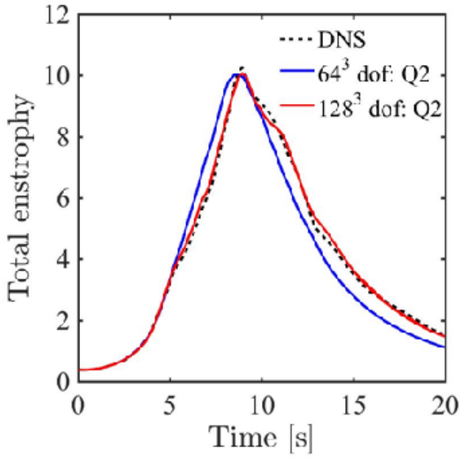
EMA approximation: Q1

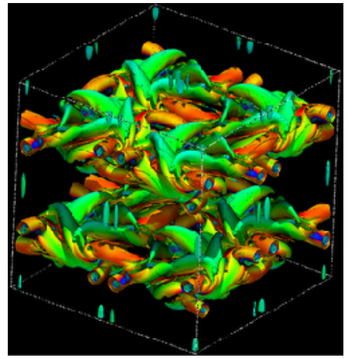




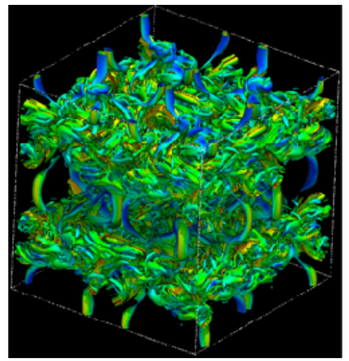
EMA approximation: Q2



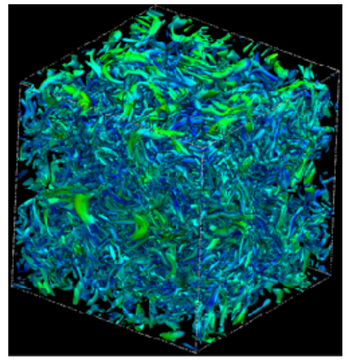




t = 5



t = 10



t = 20

A low-dissipation finite element scheme for scale resolving simulations of turbulent flows, Lehmkuhl et al. submitted to Journal of Computational Physics

For reference see: Comparison between several approaches to simulate the Taylor-Green vortex case, Moulinec et al. PARCFD 2016

FRACTIONAL STEP SCHEME - with Runge Scheme time discretization

For each substep:

Obtain F^j assembly process — close to 80% on typical run on MN4

$$\mathsf{U}^{i,*} = \mathsf{U}^n + \delta t \mathsf{M}^{-1} \sum_{j=1}^{i} a_{ij} \mathsf{F}^j,$$

Multiply by inverse of Lumped Mass Matrix (cheap)

$$\mathsf{D}\mathsf{M}^{-1}\mathsf{G}\left(\Phi^i\right) = \frac{1}{c_i \delta t}\left(\mathsf{D}\mathsf{U}^{i,*} - \mathsf{R}_c\right),$$

Multiply by Divergence Matrix — D

Solve Linear System for Laplacian (2nd most costly step)

$$\mathsf{U}^i = \mathsf{U}^{i,*} - c_i \delta t \mathsf{M}^{-1}\mathsf{G}\left(\Phi^i\right),$$

Multiply by Gradient Matrix — G

Multiply by inverse of Lumped Mass Matrix (cheap)

Other Operations : multiply by scalars; add vectors; apply dirichlet BCs

A low-dissipation finite element scheme for scale resolving simulations of turbulent flows, O. Lehmkuhl, G. Houzeaux, H. Owen, G. Chrysokentis, I. Rodriguez , JCP 2018

GPU implementation and optimization

GPU2 → GPU40 → ●●● → GPU49 → GPU51 → GPU61

Initial GPU Optimization

NAME	time from .log	improvement	Clasification	Work done
GPU2_17	39,61		code reformats	Alya from master
40	33,64	1,18	code reformats	Reincorporate non optimized miniapp + and e
41	19,64	1,71	algorithmic changes	Incorporate optimized miniapp
42-43	18,83	1,04	code reformats	Switch to just P1 - Eliminate call to ker_prope
44-45	4,35	4,33	privatization	all thread private
46-49	3,95	1,09	code reformats	Cleaning- add and use vreman-
51		3,50	code reformats	avoid elmat – obtain directly elrbu – improvem

email 5 nov 21

GPU71

CPU Optimization using experience from initial GPU

MAIL 30 nov 21

71) I start from alya/Sources/modules/nastin_private/mod_nsi_element_operations_fast_dev.f90 and remove dt_rho_nsi & mass_rho_nsi.

Thus, 71 is very similar to 40, which you typicaly take as the base. Actually what we have in Alya currently is slower due to dt_rho_nsi & mass_rho_nsi.

72 & 73 are useless -- I did them again in 74 and 75 but in a better order

74) I send assemble directly in the RHS - That is eliminate elauu

75) I eliminate the call to ker_proper to obtain teh turbulent viscosity.

Up to 75 it is valid for any kind of element.

---

76) here I start specialising for linear tetrahedra.
use pnode (that is a parameter) instead of mnode. This is the change you found

Just that mini change

77) calculate shape functions and derivatives in a way that is only valid for tetras.

78) gpcar without pgaus. Also gpvol xmile gpgve tampoco xnutu, Aalph, Bbeta G__ij
In finite elements you need to do volume integrals over the elements. The integral is performed numerically by evaluating over certain gauss points. The number of gauss points depends on the element. Typically it is equal to the number of nodes. Thus typically pgaus = pnode.

A particular feature of the linear tetrahedra is that the derivatives are constant within the element. It is linear, the derivatives of a linear function are constant. This does not happen for other elements. For general elements we have a different value of gpcar at each gauss point. Thus if you program for general elements gpcar /the cartesian derivatives are of size pgaus.
If you are going to work only with linear tetrahedra it does not make sense to store 4 times the same value. This is the improvement I introduce here.
The other variables depend on gpcar and teh same applies for them.

This change is important for the GPU because it reduces the registers.

79) Here I eliminate gpden & gpvis, the density and viscosity, which are up to now obtained by ker_proper and set them as parameters. This is a small change to get closer to what I had done in previous version. In 99% of the cases we run density and vsicosity are constant. Thus this makes a lot of sense.

80) Cleaning to make it as close as possible to what I have in the GPU (51) but still valid on CPU.

GPU80 → GPU91

Final GPU version

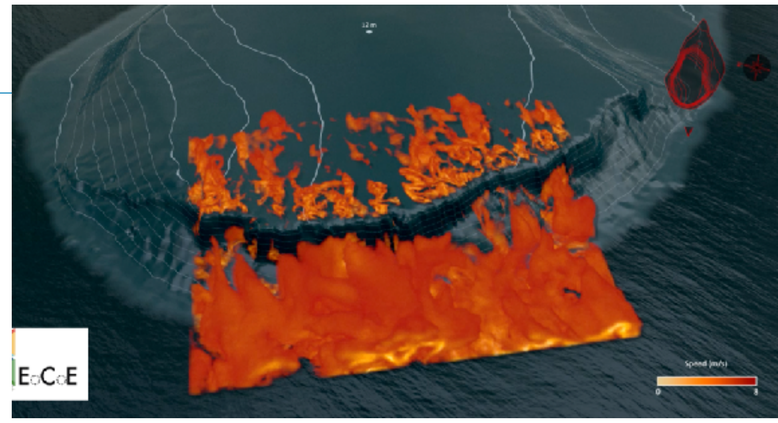
CPU4

Final CPU version with some generalisations not still available on GPU - it uses ker_proper so it could be faster by calculating gmut on the fly

Times for assembly (& properties)

	Initial - 2	Final - 80CPU / 91 GPU	Improvement
1A100 GPU + 9mpi	9.8 s	0.07 s	140 X
MN4 46 mpi Intel Xeon Platinum 8160 - Skylake	1.93 s	0.49 s	3.9 X
	CPU 5 X faster	GPU 7 X faster	

5.6 M nodes and
32 M elements



Maximum assembly time obtained from Alya's .log file

Energy estimates from Top 500

The GPU node we have used is very similar to a Perlmutter node
From top 500 - 2589KW/1536nodes - 1685W per node
We use approx. 1/4 node —> 421 W x 0.07s - 29.5J

….for 1 MN-IV node —> 511W x 0.49s - 250J

Our assembly is 8.5 times more efficient on the GPU!!
30 (8.5x3.9) times more efficient than our current CPU implementation!!

From Green 500:
Perlmutter 27.374 GFlops/watts MN-IV 3.965 GFlops/watts

Power Efficiency Ratio 6.90 - We are getting 8.5!!!!

Optimization

- Thread private instead of 1:VECTOR_SIZE

```
real(rp)                                :: elvel(VECTOR_DIM,ndime,pnode)                ! u
real(rp)                                :: elcod(VECTOR_DIM,ndime,pnode)                ! x
```

```
real(rp)                                :: gpvel(VECTOR_DIM,ndime,pgaus)                ! u
real(rp)                                :: gpgve(VECTOR_DIM,ndime,ndime,pgaus)          ! grad(u)
```

```
      gprhs(DEF_VECT,idime,igaus) = gprhs(DEF_VECT,idime,igaus) + FACT2X * gravi_nsi(idime) &
                                                                - gppor(DEF_VECT,igaus) * gpvel(DEF_VECT,idime,igaus)
```

- Specialization 1: Only tetrahedra: FE Shape functions and its derivatives easier to calculate. Gradient homogeneous for all gauss points -> only one needed
- Specialization 2: Obtain elemental RHS directly - only valid for explicit.
- Specialization 3: Turbulent viscosity calculated on the fly - nearly for free.
- Pnode, Pgaus, Mnode as Fortran parameters
- Reduction of registries. Better memory use.

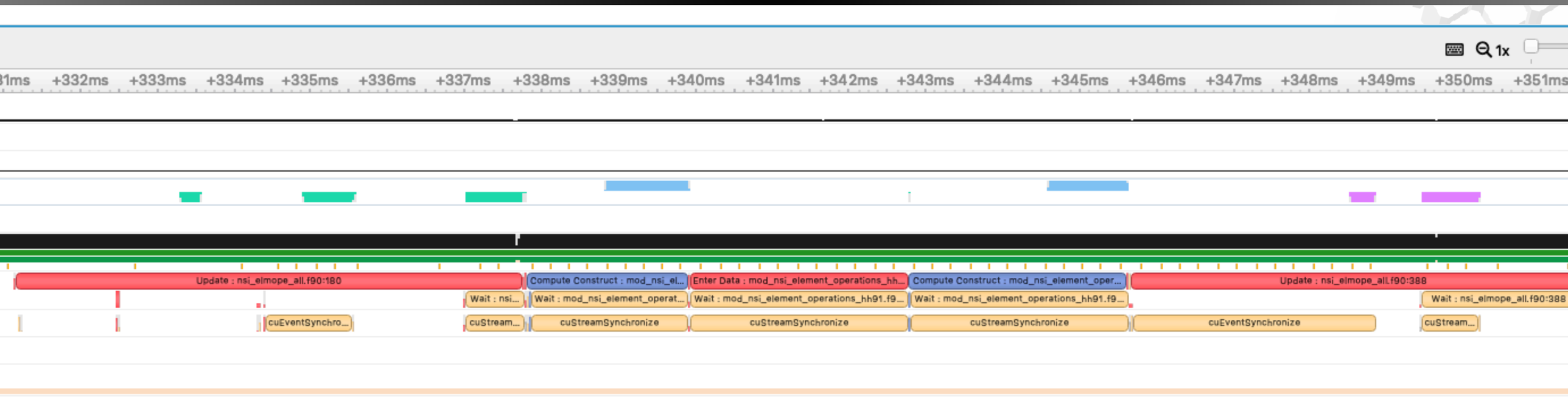
NSYS profile — final version 9 mpi

31ms 20ms 20ms

3 Runge Kutta substesp —> total 71ms
matches 0.07s from previous slide

The total GPU compute time from ncu (nsight compute) is 50ms
with 9 mpi overhead is only 21 ms

NSYS profile— final version 9 mpi



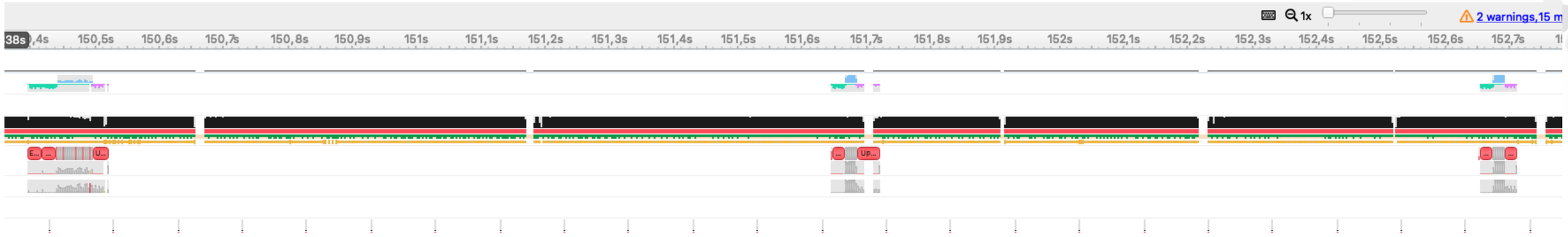
2nd substep 20ms

Blend between compute and copyin/out

Using 9 MPIs allows to hide part of the copyin/out

Copyin/out y mainly velocity and rhsid once we have everything on the GPU those times will disappear!!!

NSYS profile — final version serial

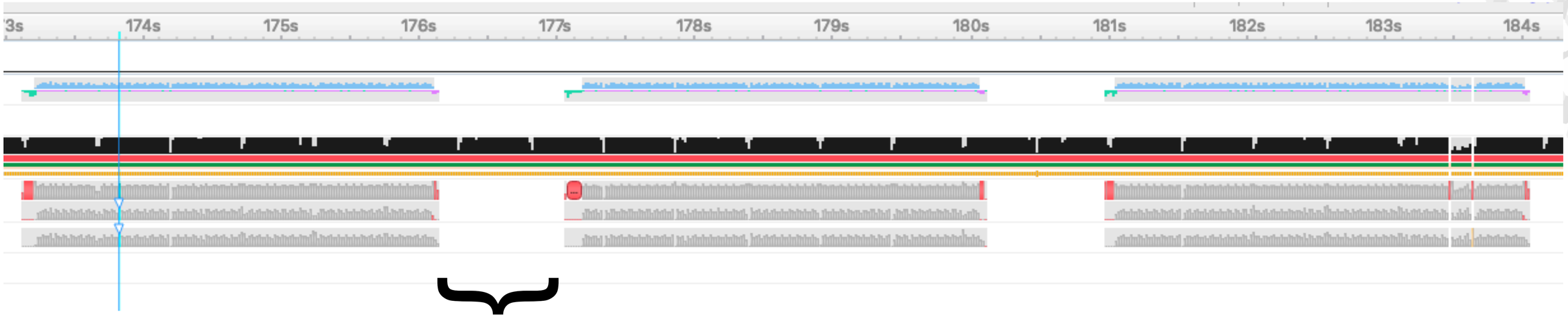


116ms 59ms 60ms

Total 235ms - much slower than with 9mpi (71ms)
copyin/out become dominant

In any case all copyin/out will disappear when we port ‘the rest’ to GPU
The rest of the code will be much faster on the GPU

NSYS profile — original version serial



‘The rest’ looked much smaller because GPU computation was very inefficient

What is being done there? Mainly:

Multiplication by Gradient and Divergence Matrices

Solve Linear System for Laplacian (2nd most costly step)

Nsight Compute

38% of FP64 Peak — before 0%

NVIDIA Nsight Compute

⚠	**FP64/32 Utilization**	The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1.The kernel achieved 0% of this device's fp32 peak performance and 38% of its fp64 peak performance.If Compute Workload Analysis determines that this kernel is fp64 bound, consider using 32-bit precision floating point operations to improve its performance.

Original

ⓘ	**Roofline Analysis**	The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1.The kernel achieved 0% of this device's fp32 peak performance and 0% of its fp64 peak performance.

Explanation, that I nearly believed

Low order FE not suited for GPUs (still trying to find papers)

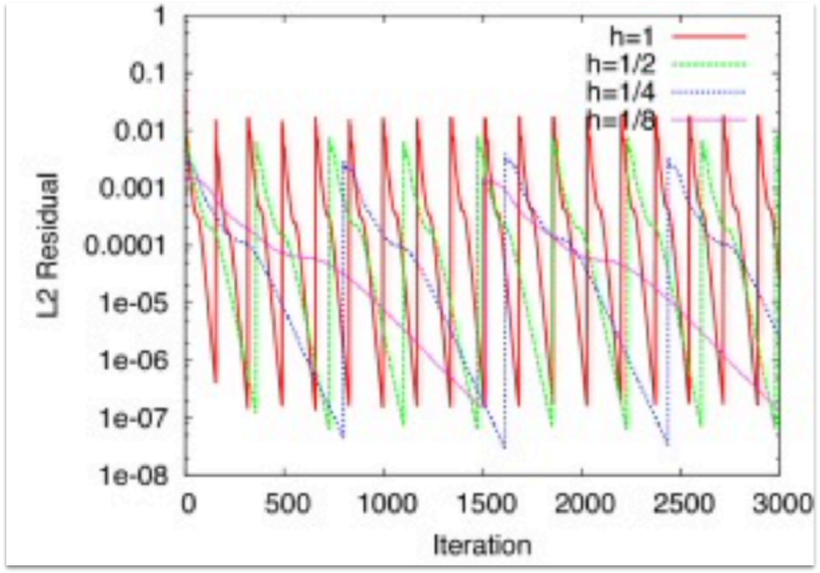
High order, Finite Difference and LBM - YES

Alya - linear solver for the pressure

Journal of Computational Science
Volume 14, May 2016, Pages 15-27

Alya: Multiphysics engineering simulation toward exascale

Mariano Vázquez [a, b], Guillaume Houzeaux [a], Seid Koric [c], Antoni Artigues [a], Jazmin Aguado-Sierra [a], Ruth Arís [a], Daniel Mira [a], Hadrien Calmet [a], Fernando Cucchietti [a], Herbert Owen [a], Ahmed Taha [c], Evan Dering Burness [c], José María Cela [a], Mateo Valero [a]



Kiln furnace. Convergence of the DCG solver for the continuity equation

Computational Mechanics solvers have two main steps:

1) Assembly of a Matrix & RHS

2) Solution of a Linear System

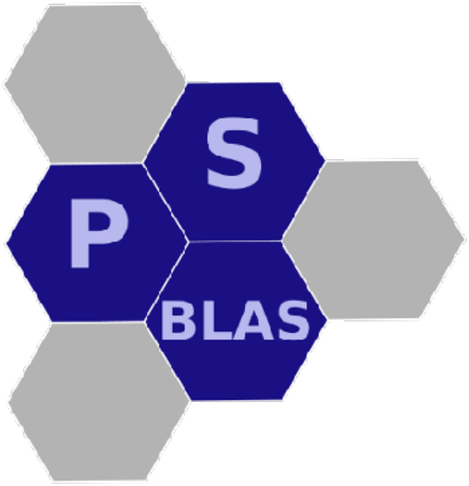
For the solution of the linear System Alya has used in house developed code until recently.

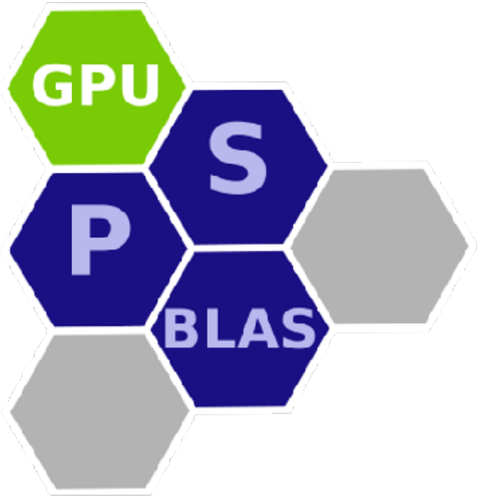
Main solvers:

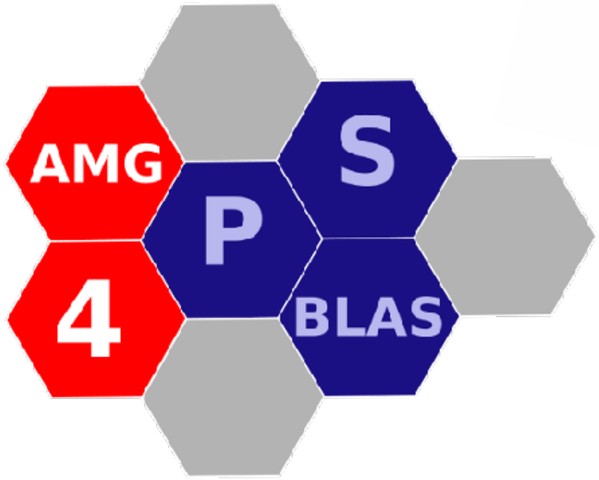
- GMRES
- CG
- Deflated CG

AMG4PSBLAS

PSCToolkit (Parallel Sparse Computation Toolkit)





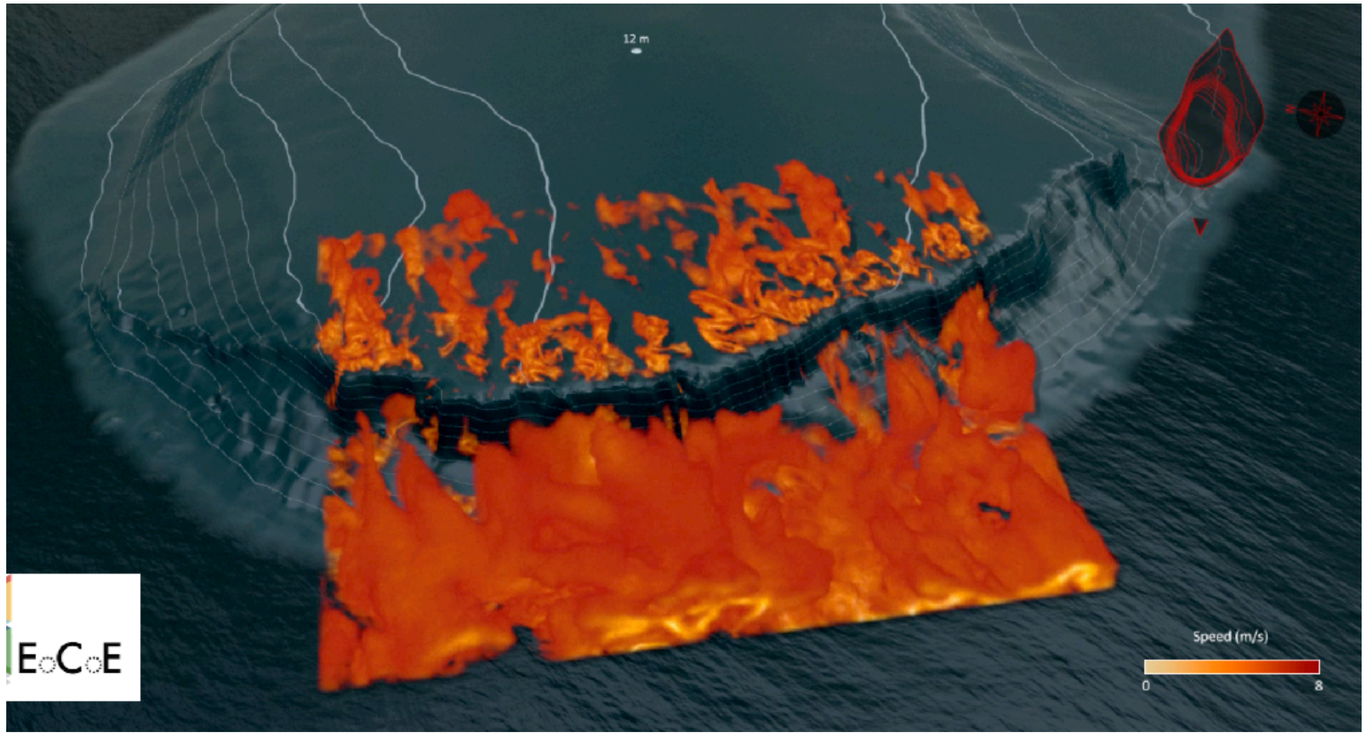


A software framework for scalable solution of sparse linear systems by Krylov methods coupled with Algebraic Multigrid (AMG) Preconditioners on hybrid CPU/NVIDIA-GPU architectures

Available at PSCTOOLKIT | Parallel Sparse Computation Toolkit

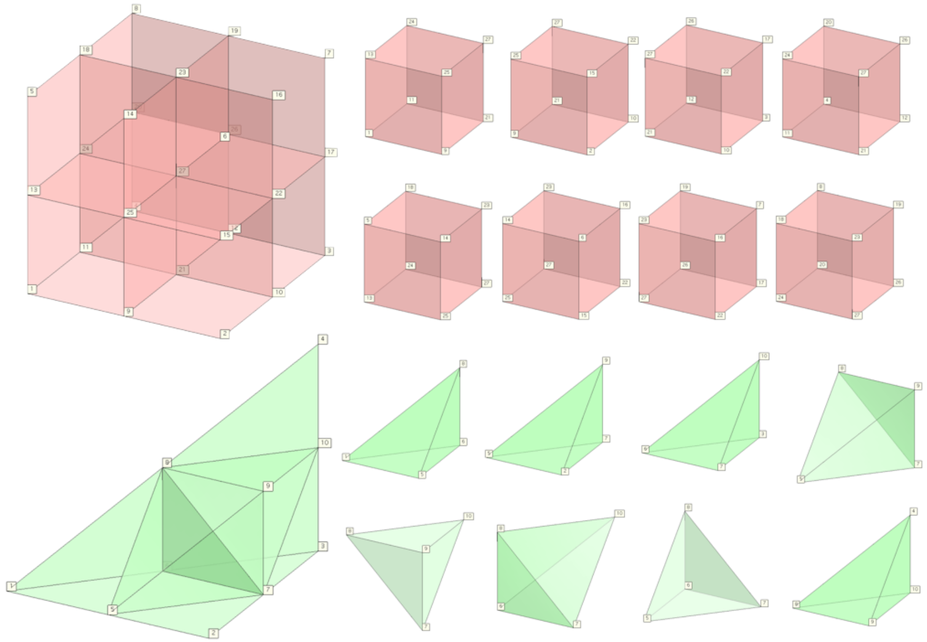
Main reference: P. D'Ambra, F. Durastante, S. Filippone, AMG Preconditioners for Linear Solvers towards Extreme Scale, to appear.
Preprint available at [2006.16147] (arxiv.org)

Bolund case



Original mesh with 5.6 M nodes
and
32 M elements

Apply divisor up to 3 times
(X 512)
2850M nodes
16000M elements



Guillaume Houzeaux, Raúl de la Cruz, Herbert Owen, and Mariano Vázquez. Parallel uniform mesh multiplication applied to a Navier- Stokes solver. *Computers and Fluids*

Excellent weak and algorithmic scalability from original mesh to 3 divisors

	cores	average iterations	time per iteration	av. time [ms]	av. time w/ Alya CG
Original mesh	46	6	133	800	1568
1 divisor	368	6	134	802	1198
2 divisors	2944	4.8	150	716	1939
3 divisors	23552	4	159	635	4658

Conclusions and next step

- Low order FE can run efficiently on GPUs !!!!
- Algebraic multigrid - weak scalability up to 16000 million elements

- Put the whole fractional step on the GPU.
- Generalise (other elements, richer physics) without losing too much efficiency.

- If Europe does not want to be out-computed/out-competed it must react soon.

Thanks for your attention!

Horizon 2020 grant N° 824158

The author thankfully acknowledges the computer resources at MareNostrum and the technical support provided by Barcelona Supercomputing Center (RES-AECT-IM-2021-2-0011)

E·C·E