Automated derivation and parallel execution of finite difference models on CPUs, GPUs and Intel Xeon Phi processors using code generation

Christian T. Jacobs, Satya P. Jammy,
David J. Lusher, Neil D. Sandham

University of Southampton

11 October 2017

Background

- SBLI: CFD code developed at Soton. Written in F95.

- Solves compressible N-S equations:
  - Multi-block
  - 2D & 3D curvilinear grids
  - 4th order central differencing
  - 3rd & 4th order explicit Runge-Kutta timestepping
  - DNS, LES, shock capturing, filtering
  - ~60K lines of code

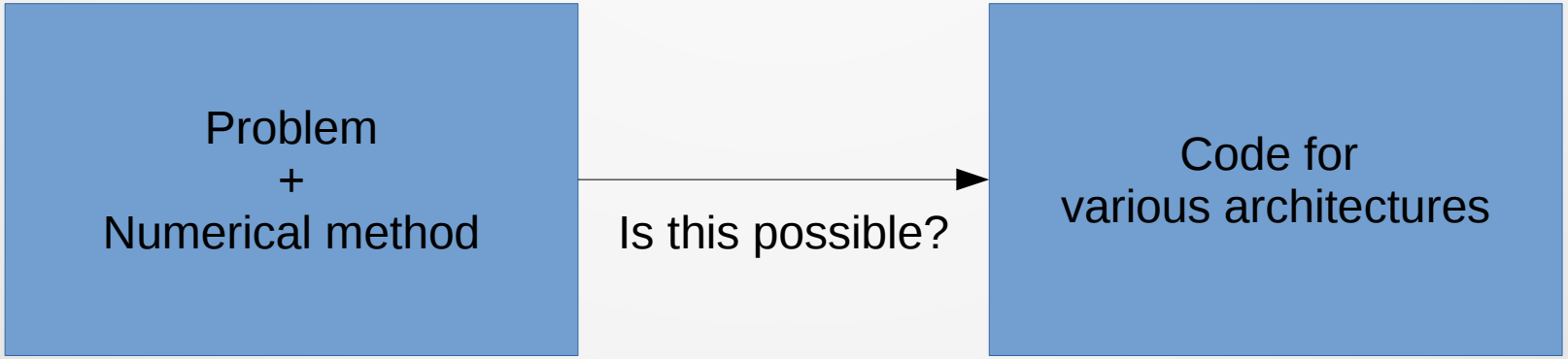
- Currently capable of running on CPU clusters
  - Scaling to 200,000+ cores (Yao et al., 2009)

Motivation

- Existing archs: multicore CPUs, GPUs, Intel Xeon Phi.
  - Future energy efficient systems from ARM and friends
- Newer hardware has the potential to reduce runtime, but…
- Most models not in a position to readily exploit such architectures to their full potential.
  - Porting SBLI requires **non-trivial rewrite**.
  - **Burden** on model devs to not only be domain specialists, but also experts in numerial methods, parallel computing paradigms, and their efficient implementation.
  - Newer architecture might arrive during porting.
  - **How do we future-proof codes?**

Approach

- Key components of numerical solution are
  - Problem description
  - Numerical method
  - Model code (discretisation, solver, etc) for various architectures



Future-proofing with OPS

- **OPS**: **O**xford **P**arallel library for **S**tructured-mesh computations
- Multi-block structured applications
- Source-to-source translation for parallel implementations on various architectures
- Very little overhead with the automation process for similar applications (CloverLeaf mini app - Mudalige et al. 2014)

Example for simple stencil averaging

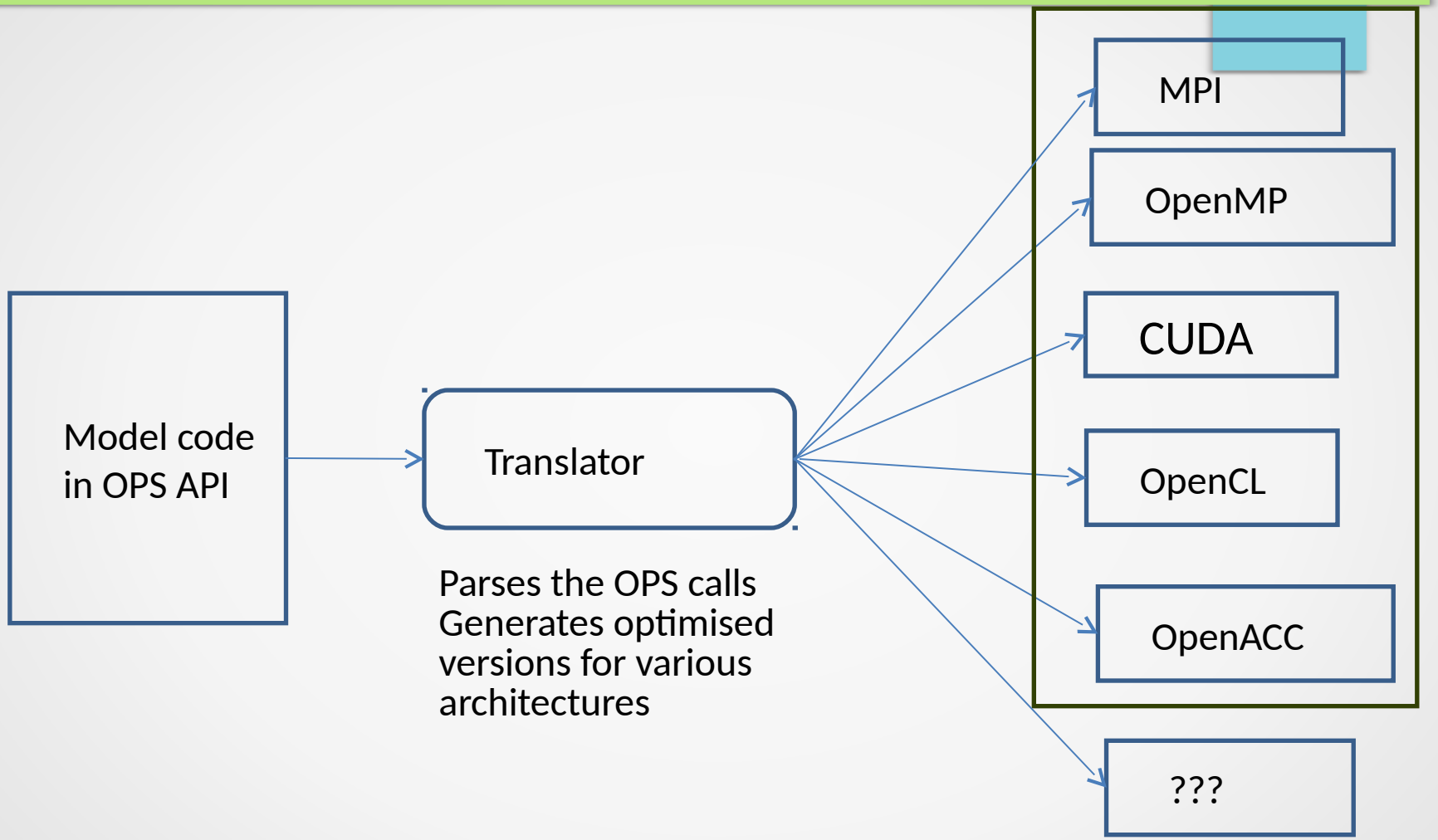
ops_par_loop:

```
int range[4] = {imin,imax,jmin,jmax};
ops_par_loop(calc, block, 2, range,
ops_arg_dat(a,S2D_0,"double",OPS_WRITE),ops_arg_dat(b,S2D_1,"double",OPS_READ));
```

Kernel:

```
void calc(double *a, const double *b) {
 a[OPS_ACC0(0,0)] = 0.5*(b[OPS_ACC1(1,0)] + b[OPS_ACC1(-1,0)];)
}
```

Future-proofing with OPS



Source code remains unchanged

Newer architectures backend translator needs to be written

Proof of concept: Shu-Osher case

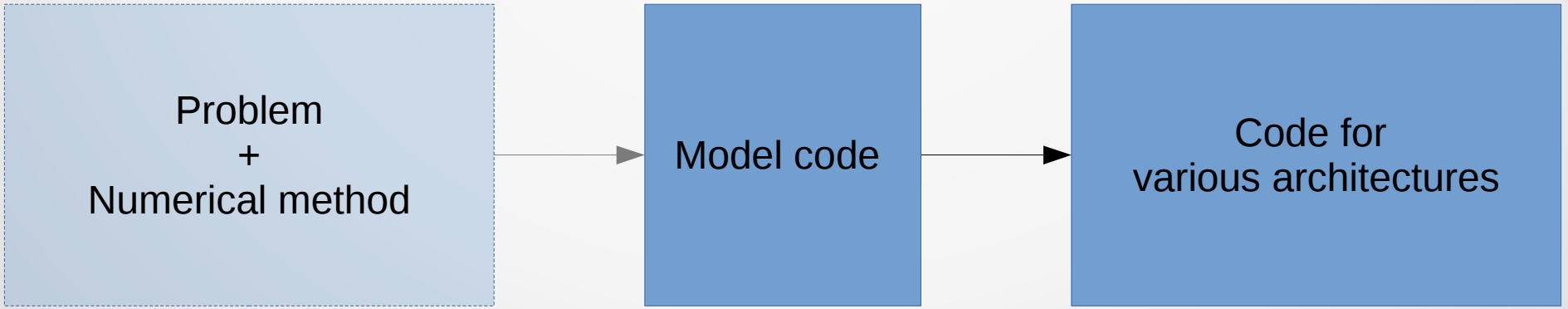
CPU: Intel® Xeon®
E5-2640 @2.5GHz 12 cores

GPU: NVIDIA Tesla K20c
2946 CUDA cores 5GB
memory

	CPU		GPU		Speedup
Grid Size (Millions)	12 MPI	12 OpenMP	OpenCL on GPU	CUDA	CPU/GPU
0.0025	2.92	4.61	3.75	3.37	0.89
0.05	33.75	52.89	10.81	9.34	3.61
0.1	66.05	76.47	16.69	16.13	4.09
0.2	136.79	167.13	30.09	29.15	4.69
2	1738.8	2429.44	271.51	264.59	6.57

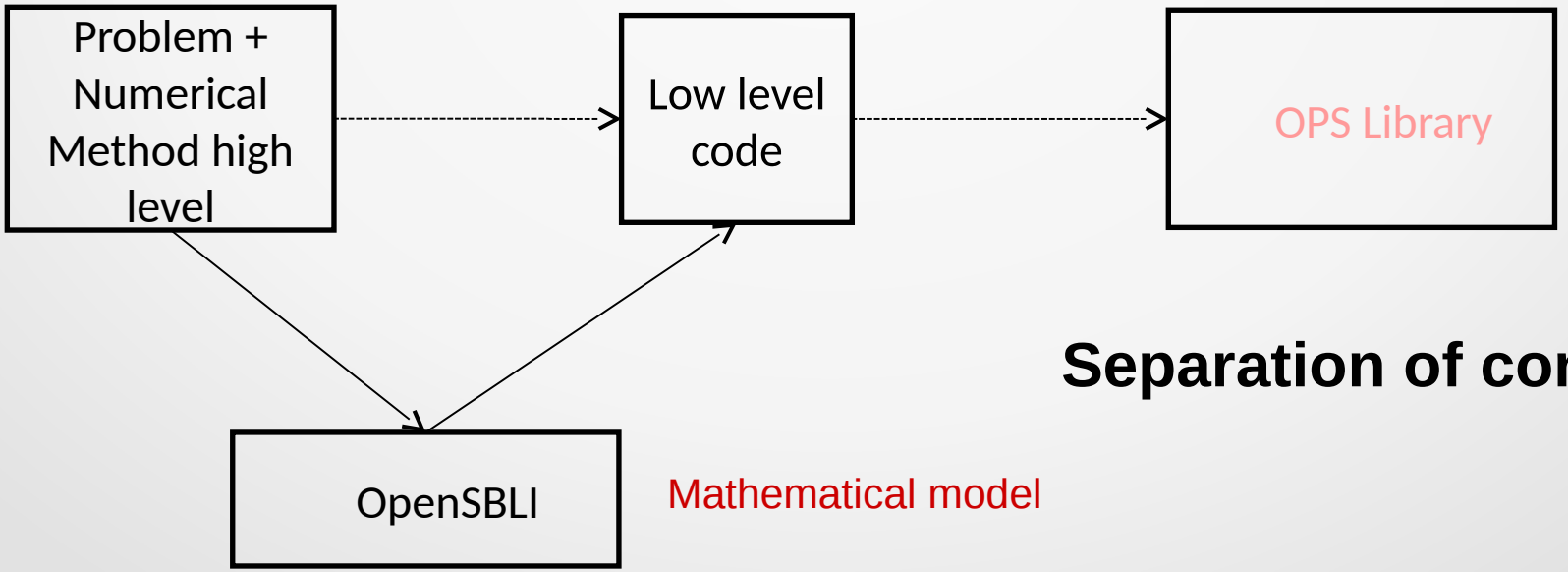
OPS

- OPS is verbose in nature.
- Error prone.
- No flexibility of numerical method or equations to be solved.



OpenSBLI

- User describes the problem at a higher level.
- Numerical analyst develops the numerical algorithm which generates a sequential model code in OPS-compliant C.
- Computer scientist handles parallel backend implementation.



Separation of concerns.

OpenSBLI: Design

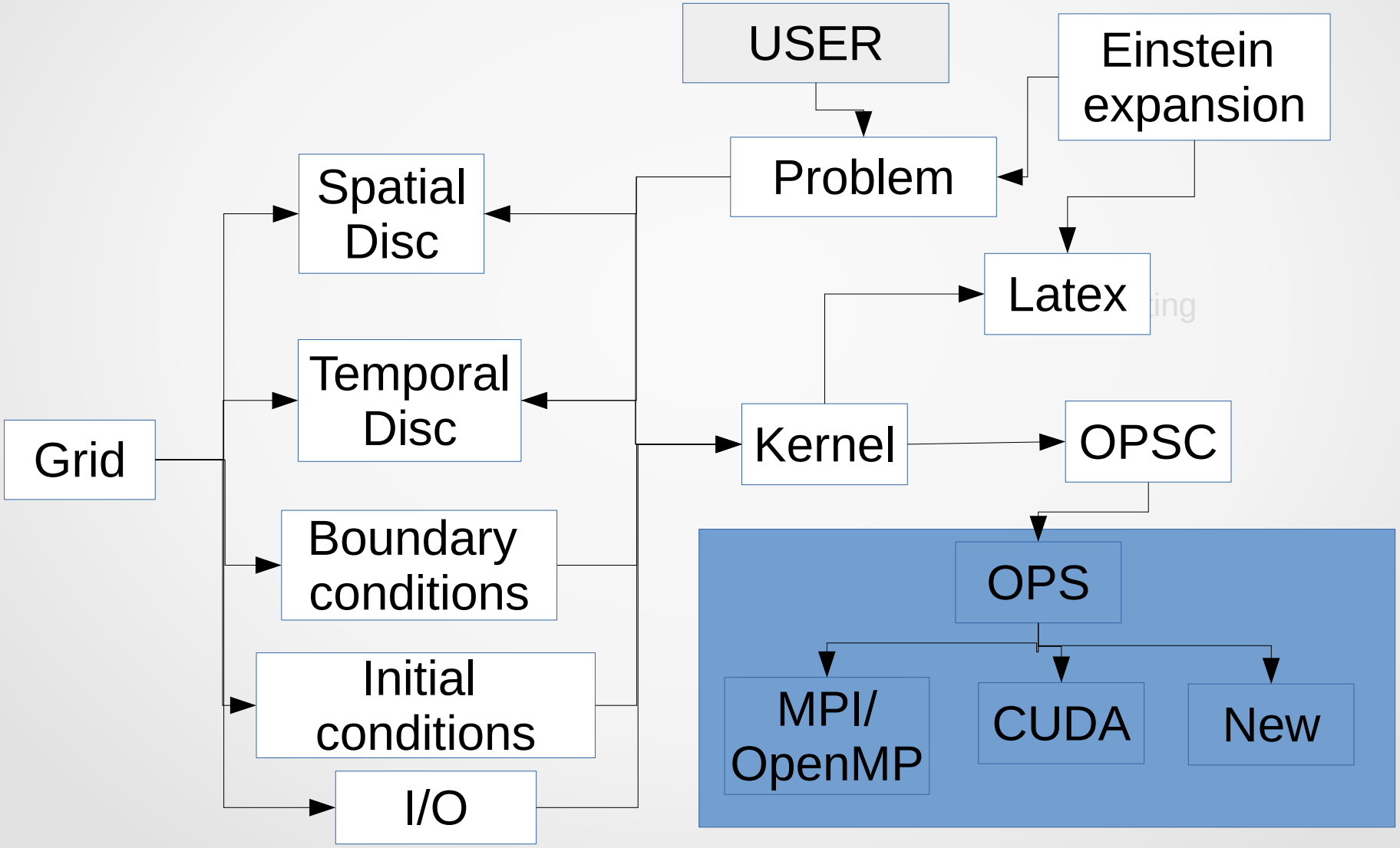
- Written in Python and uses SymPy as building blocks

- User specifies problem in a Python ‘problem spec. file’
  - PDEs in Einstein/index notation
  - Spatial scheme and order of accuracy (finite difference)
  - Time-stepping scheme (RK3)
  - Boundary conditions
  - I/O and other parameters for the simulation

OpenSBLI: Design

- Finds and expands the summation indices of PDEs
  - Applies spatial, temporal and boundary schemes
- Creates computational kernels.
- Generates model code. Translation to different arch. with OPS.
- Outputs LaTeX files of the computational kernels for debugging.

OpenSBLI: Design



Example

- 50 line high-level problem definition for the 3D compressible Navier-Stokes equations
- 2000 line generated sequential OPS C code
- 20K lines of generated code for MPI and CUDA

```
# Number of dimensions for the problem
ndim = 3

# Define the compresible Navier-Stokes equations in Einstein notation.
mass = "Eq(Der(rho,t), - Conservative(rho*u_j,x_j))"
momentum = "Eq(Der(rhou_i,t) , -Conservative(rhou_i*u_j + KD(_i,_j)*p ,x_j) + Der(tau_i_j,x_j))"
energy = "Eq(Der(rhoE,t), - Conservative((p+rhoE)*u_j,x_j) + Der(q_j,x_j) + Der(u_i*tau_i_j ,x_j))"
equations = [mass, momentum, energy]

# Substitutions
stress_tensor = "Eq(tau_i_j, (1.0/Re)*(Der(u_i,x_j)+ Der(u_j,x_i)- (2/3)* KD(_i,_j)* Der(u_k,x_k)))"
heat_flux = "Eq(q_j, (1.0/((gama-1)*Minf*Minf*Pr*Re))*Der(T,x_j))"
substitutions = [stress_tensor, heat_flux]

# Define all the constants in the equations
constants = ["Re", "Pr","gama", "Minf"]

# Define coordinate direction symbol (x) this will be x_i, x_j, x_k
coordinate_symbol = "x"

# Formulas for the variables used in the equations
velocity = "Eq(u_i, rhou_i/rho)"
pressure = "Eq(p, (gama-1)*(rhoE - rho*(1/2)*(u_j*u_j)))"
temperature = "Eq(T, p*gama*Minf*Minf/(rho))"
formulas = [velocity, pressure, temperature]
```

Example

```
void taylor_green_vortex_block0_69_kernel(const double *wk20 , const double *wk47 , const double *wk21 , const double
*wk28 , const double *u1 , const double *wk29 , const double *wk19 , const double *wk0 , const double *wk15 , const
double *wk35 , const double *wk18 , const double *wk11 , const double *wk12 , const double *wk31 , const double *wk8 ,
const double *wk37 , const double *wk34 , const double *wk10 , const double *wk30 , const double *wk39 , const double
*wk44 , const double *u0 , const double *wk40 , const double *wk46 , const double *wk45 , const double *wk41 , const
double *wk25 , const double *wk3 , const double *wk7 , const double *wk1 , const double *wk2 , const double *wk33 ,
const double *wk6 , const double *wk32 , const double *wk38 , const double *wk14 , const double *wk42 , const double
*wk26 , const double *wk43 , const double *u2 , const double *wk22 , const double *wk24 , const double *wk27 , const
double *wk5 , const double *wk23 , const double *wk9 , const double *wk4 , const double *wk17 , const double *wk13 ,
const double *wk36 , const double *wk16 , double *wk49 , double *wk48 , double *wk50 , double *wk51 , double *wk52)
{
   wk48[OPS_ACC52(0,0,0)] = -wk11[OPS_ACC11(0,0,0)] - wk14[OPS_ACC35(0,0,0)] - wk2[OPS_ACC30(0,0,0)];
    wk49[OPS_ACC51(0,0,0)] = rinv11*(wk0[OPS_ACC7(0,0,0)] + wk44[OPS_ACC20(0,0,0)]) +
      rinv11*(wk3[OPS_ACC27(0,0,0)] + wk47[OPS_ACC1(0,0,0)]) + rinv11*((rc4)*wk16[OPS_ACC50(0,0,0)] -
      rc6*wk44[OPS_ACC20(0,0,0)] - rc6*wk47[OPS_ACC1(0,0,0)]) - wk18[OPS_ACC10(0,0,0)] - wk20[OPS_ACC0(0,0,0)] -
      wk29[OPS_ACC5(0,0,0)] - wk39[OPS_ACC19(0,0,0)];
    wk50[OPS_ACC53(0,0,0)] = rinv11*(wk13[OPS_ACC48(0,0,0)] + wk42[OPS_ACC36(0,0,0)]) +
      rinv11*(wk43[OPS_ACC38(0,0,0)] + wk5[OPS_ACC43(0,0,0)]) + rinv11*((rc4)*wk26[OPS_ACC37(0,0,0)] -
      rc6*wk42[OPS_ACC36(0,0,0)] - rc6*wk43[OPS_ACC38(0,0,0)]) - wk21[OPS_ACC2(0,0,0)] - wk27[OPS_ACC42(0,0,0)] -
      wk31[OPS_ACC13(0,0,0)] - wk41[OPS_ACC25(0,0,0)];
    wk51[OPS_ACC54(0,0,0)] = rinv11*(wk22[OPS_ACC40(0,0,0)] + wk45[OPS_ACC24(0,0,0)]) +
      rinv11*(wk46[OPS_ACC23(0,0,0)] + wk7[OPS_ACC28(0,0,0)]) + rinv11*((rc4)*wk4[OPS_ACC46(0,0,0)] -
      rc6*wk45[OPS_ACC24(0,0,0)] - rc6*wk46[OPS_ACC23(0,0,0)]) - wk28[OPS_ACC3(0,0,0)] - wk32[OPS_ACC33(0,0,0)] -
      wk36[OPS_ACC49(0,0,0)] - wk9[OPS_ACC45(0,0,0)];
    wk52[OPS_ACC55(0,0,0)] = rinv11*rinv12*rinv13*rinv14*wk19[OPS_ACC6(0,0,0)] +
      rinv11*rinv12*rinv13*rinv14*wk30[OPS_ACC18(0,0,0)] + rinv11*rinv12*rinv13*rinv14*wk35[OPS_ACC9(0,0,0)] +
      rinv11*(wk0[OPS_ACC7(0,0,0)] + wk44[OPS_ACC20(0,0,0)])*u0[OPS_ACC21(0,0,0)] +
      rinv11*(wk1[OPS_ACC29(0,0,0)] + wk23[OPS_ACC44(0,0,0)])*wk1[OPS_ACC29(0,0,0)] +
      rinv11*(wk1[OPS_ACC29(0,0,0)] + wk23[OPS_ACC44(0,0,0)])*wk23[OPS_ACC44(0,0,0)] +
      rinv11*(wk12[OPS_ACC12(0,0,0)] + wk37[OPS_ACC15(0,0,0)])*wk12[OPS_ACC12(0,0,0)] +
      rinv11*(wk12[OPS_ACC12(0,0,0)] + wk37[OPS_ACC15(0,0,0)])*wk37[OPS_ACC15(0,0,0)] +
      rinv11*(wk13[OPS_ACC48(0,0,0)] + wk42[OPS_ACC36(0,0,0)])*u1[OPS_ACC4(0,0,0)] +
      rinv11*(wk15[OPS_ACC8(0,0,0)] + wk8[OPS_ACC14(0,0,0)])*wk15[OPS_ACC8(0,0,0)] +
      rinv11*(wk15[OPS_ACC8(0,0,0)] + wk8[OPS_ACC14(0,0,0)])*wk8[OPS_ACC14(0,0,0)] +
      rinv11*(wk22[OPS_ACC40(0,0,0)] + wk45[OPS_ACC24(0,0,0)])*u2[OPS_ACC39(0,0,0)] +
      rinv11*(wk3[OPS_ACC27(0,0,0)] + wk47[OPS_ACC1(0,0,0)])*u0[OPS_ACC21(0,0,0)] +
      rinv11*(wk43[OPS_ACC38(0,0,0)] + wk5[OPS_ACC43(0,0,0)])*u1[OPS_ACC4(0,0,0)] +
      rinv11*(wk46[OPS_ACC23(0,0,0)] + wk7[OPS_ACC28(0,0,0)])*u2[OPS_ACC39(0,0,0)] +
      rinv11*((rc4)*wk16[OPS_ACC50(0,0,0)] - rc6*wk44[OPS_ACC20(0,0,0)] -
      rc6*wk47[OPS_ACC1(0,0,0)])*u0[OPS_ACC21(0,0,0)] + rinv11*(-rc6*wk17[OPS_ACC47(0,0,0)] -
      rc6*wk25[OPS_ACC26(0,0,0)] + (rc4)*wk34[OPS_ACC16(0,0,0)])*wk34[OPS_ACC16(0,0,0)] +
      rinv11*(-rc6*wk17[OPS_ACC47(0,0,0)] + (rc4)*wk25[OPS_ACC26(0,0,0)] -
      rc6*wk34[OPS_ACC16(0,0,0)])*wk25[OPS_ACC26(0,0,0)] + rinv11*((rc4)*wk17[OPS_ACC47(0,0,0)] -
      rc6*wk25[OPS_ACC26(0,0,0)] - rc6*wk34[OPS_ACC16(0,0,0)])*wk17[OPS_ACC47(0,0,0)] +
      rinv11*((rc4)*wk26[OPS_ACC37(0,0,0)] - rc6*wk42[OPS_ACC36(0,0,0)] -
      rc6*wk43[OPS_ACC38(0,0,0)])*u1[OPS_ACC4(0,0,0)] + rinv11*((rc4)*wk4[OPS_ACC46(0,0,0)] -
      rc6*wk45[OPS_ACC24(0,0,0)] - rc6*wk46[OPS_ACC23(0,0,0)])*u2[OPS_ACC39(0,0,0)] - wk10[OPS_ACC17(0,0,0)] -
      wk24[OPS_ACC41(0,0,0)] - wk33[OPS_ACC31(0,0,0)] - wk38[OPS_ACC34(0,0,0)] - wk40[OPS_ACC22(0,0,0)] -
      wk6[OPS_ACC32(0,0,0)];
}
```

Example of auto-generated kernel for computing RHS of compressible Navier-Stokes equation.

OpenSBLI: Advantages

- Flexible choice of equations
- Spatial order can be easily varied
- Implementing new numerical method requires symbolic represetation
  - To generate a Fortran or code in another language, only a new OPS backend need be written.

Verification & Validation

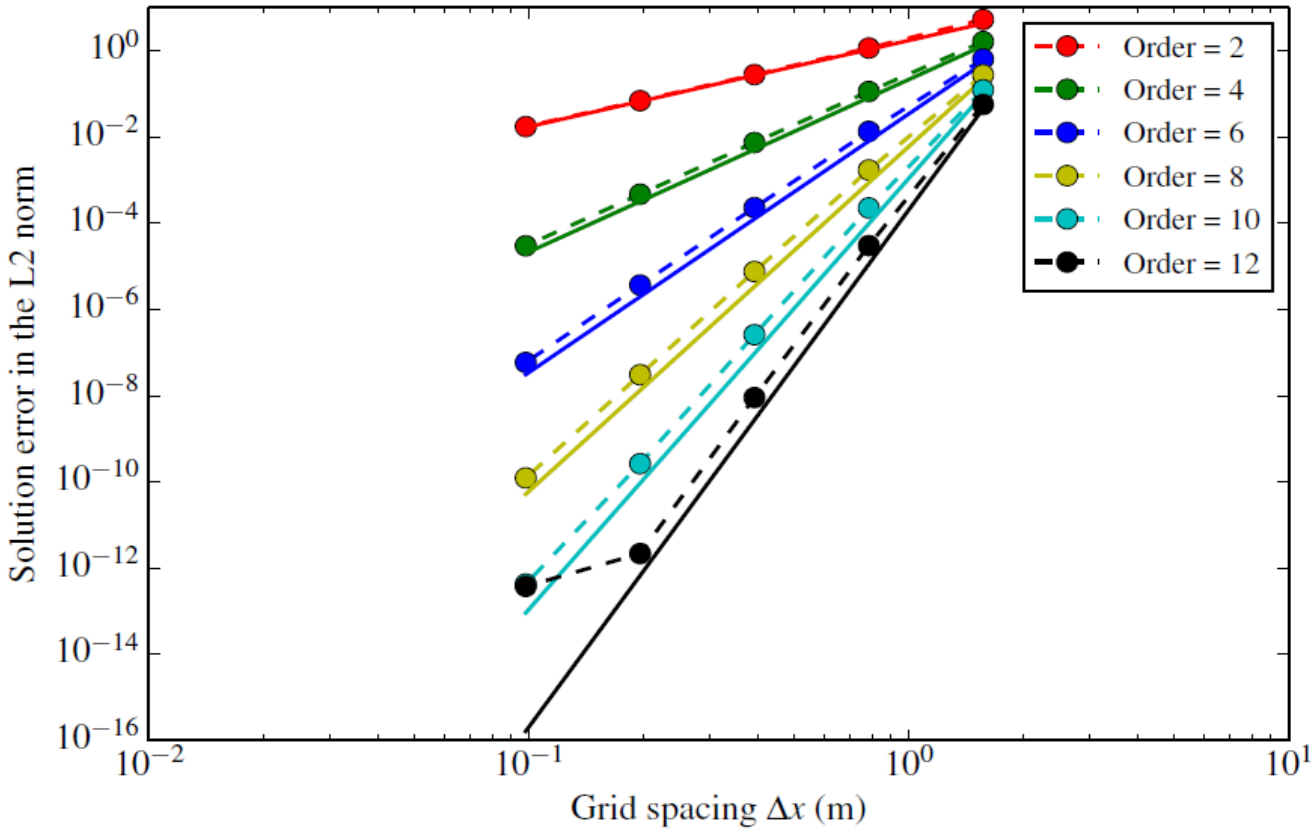
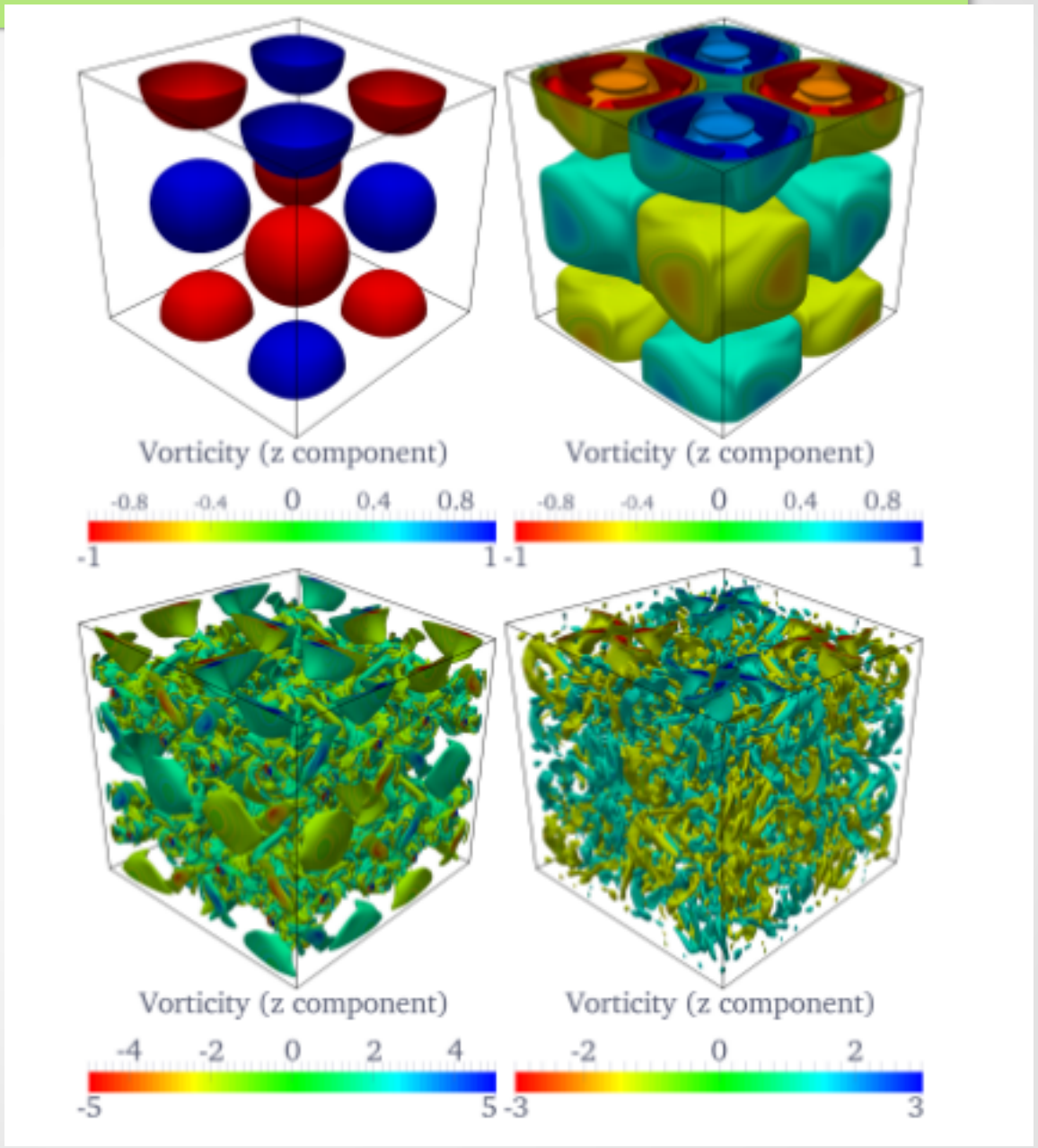


Figure 7: The absolute error (in the L2 norm) between the numerical solution ϕ and the exact/manufactured solution ϕ_m, from the suite of MMS simulations. The solid lines represent the expected convergence rate for each order.

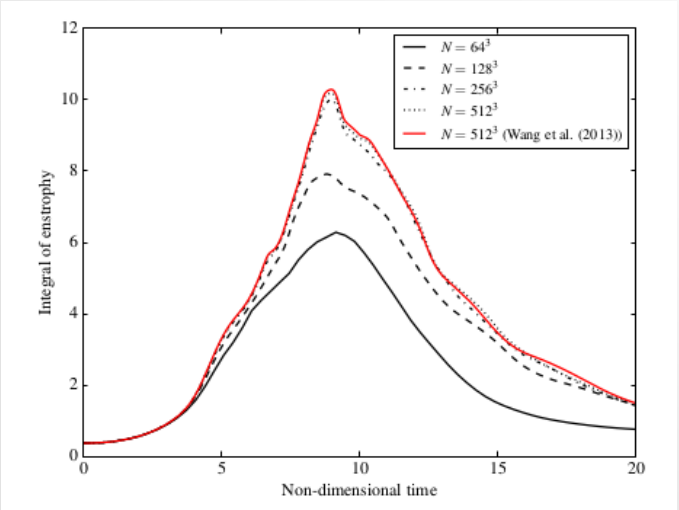
Solution convergence for the 2D advection-diffusion equation (method of manufactured solutions).

Verification & Validation

- 3D Taylor-Green vortex problem N-S Equations
- Up to 1 billion grid points,
- Re = 1600
- CPU(ARCHER), GPU(K40c)



Images by Jacobs et al. (2017). Used under CC-BY licence.



Flexibility of algorithms

- Future HPC machines expected to deliver exascale capabilities
- Theoretical flops of many-core processors (e.g. GPUs, Xeon Phi cards) have increased but RAM is limited.
- Requires novel algorithmic changes to exploit the flops
- Reducing memory usage also reduces the amount of data to be transferred between CPU and GPU.

- A **detailed study** of the algorithm's performance in the context of compressible Navier-Stokes equations is important because of the uncertainty of these architectures.

Flexibility of algorithms

Using the OpenSBLI framework we can compare different algorithmic choices without need to rewrite the code

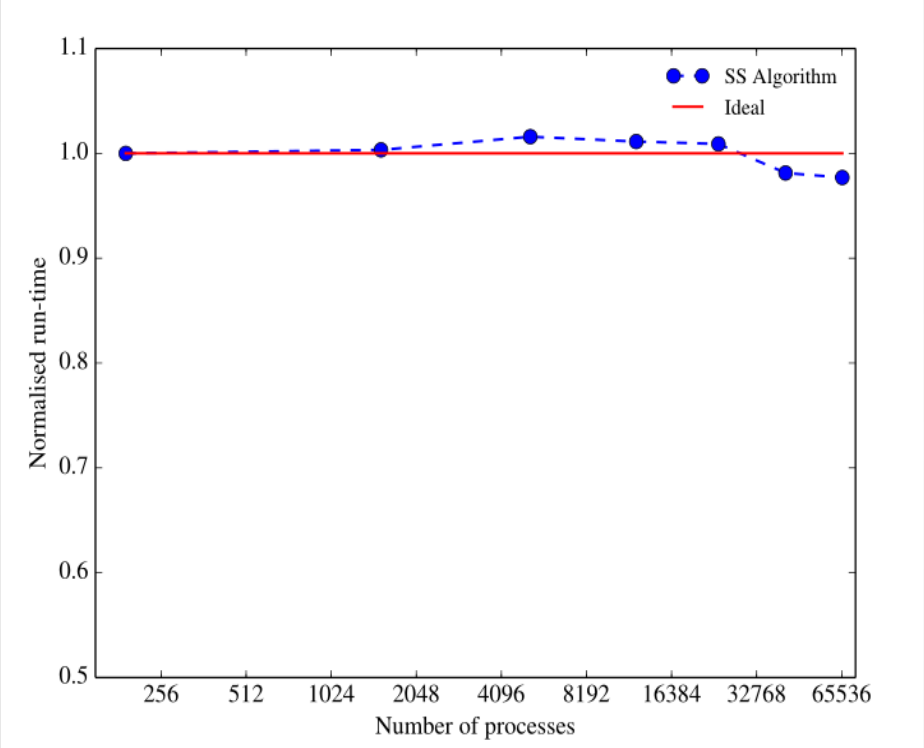
- **Baseline (BL)** – all the derivatives are stored in work arrays
- **Recompute All (RA)** – all continuous derivatives in the governing equations are replaced by their discretised formula
- **Recompute Some (RS)** – only the first derivatives of velocity are stored and the rest recomputed
- **Store None (SN)** – all the derivatives are evaluated as thread/process local variables
- **Store Some (SS)** – only the first derivatives of velocity are stored and the rest are evaluated as thread/process local variables

Algorithmic performance

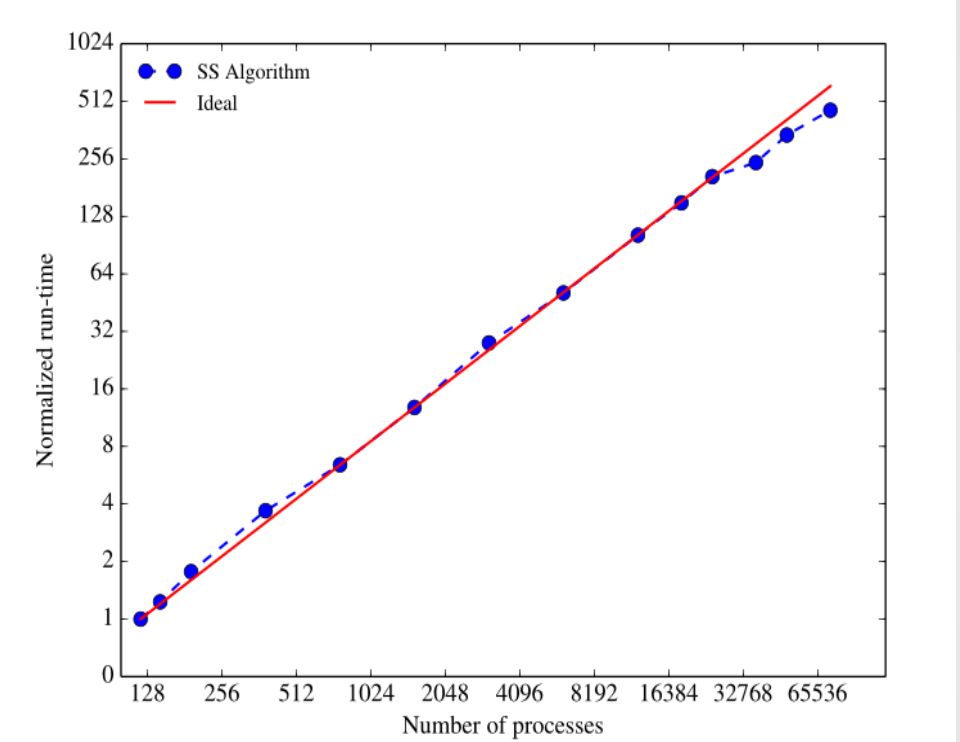
	ARCHER node (24 MPI processes), runtime (s)				
Grid Size (Millions)	Baseline	Recompute All	Recompute Some	Store None	Store Some
0.2	16	9	11	8	10
2.09	183	98	97	91	89
16.77	1562	765	803	694	685

	CUDA Tesla K40c, runtime (s)				
Grid Size (Millions)	Baseline	Recompute All	Recompute Some	Store None	Store Some
0.2	9	6	6	6	5
2.09	57	35	35	41	33
16.77	495	259	256	302	246

Parallel scaling

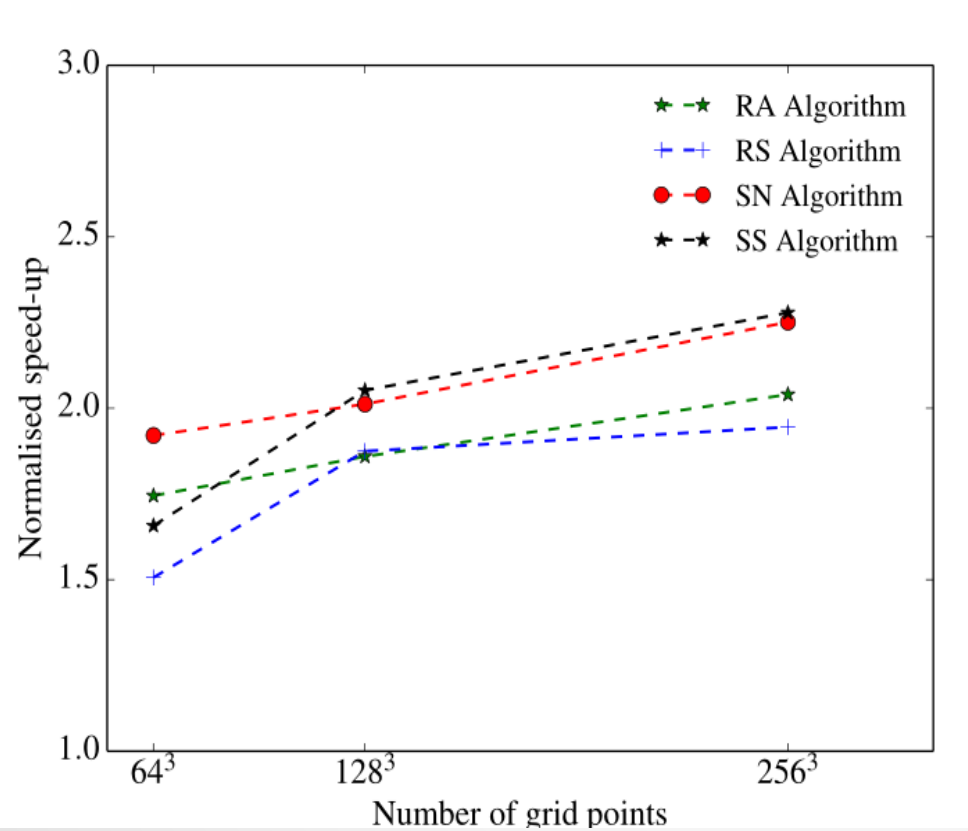


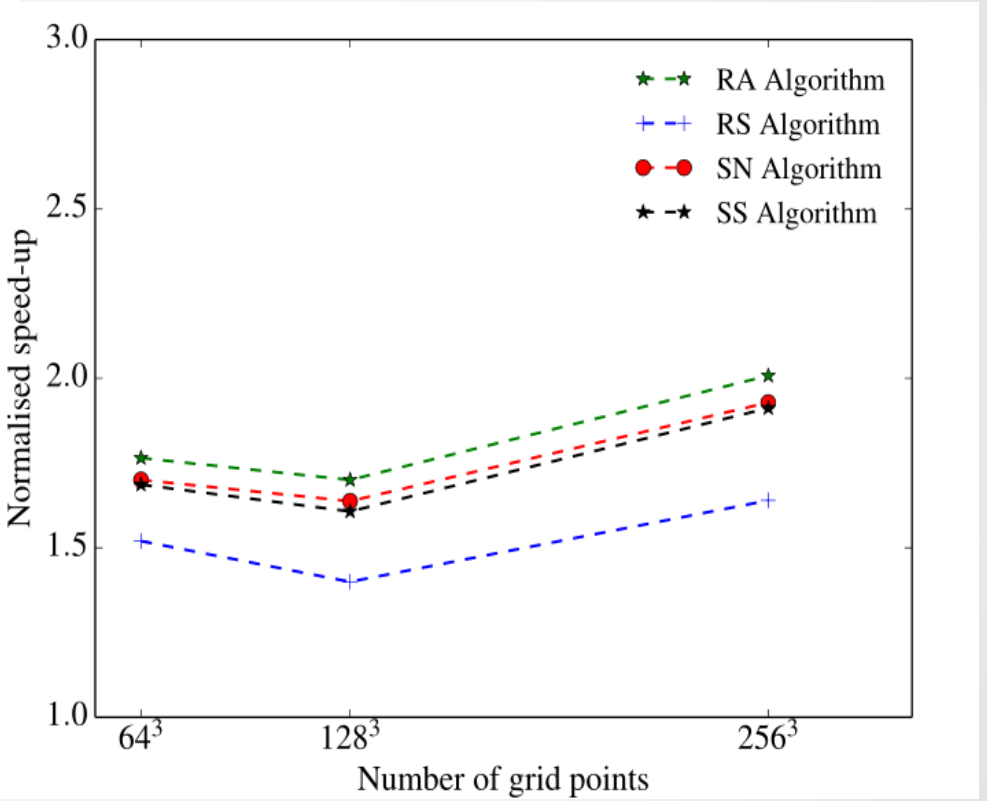
Weak scaling, 64^3 grid points per process



Strong scaling, 1024^3 grid points per process

CPU to GPU speed-up





1 ARCHER CPU node

NVIDIA Tesla K40c

CPU to GPU speed up 2.7~3.15

Intel Xeon Phi (KNL) performance

	ARCHER node (24 MPI processes), runtime (s)				
Grid Size (Millions)	Baseline	Recompute All	Recompute Some	Store None	Store Some
0.2	16	9	11	8	10
2.09	183	98	97	91	89
16.77	1562	765	803	694	685

	Intel KNL (64 MPI processes), runtime (s)				
Grid Size (Millions)	Baseline	Recompute All	Recompute Some	Store None	Store Some
0.2	20	13	14	13	13
2.09	148	92	107	87	102
16.77	688	675	647	610	562

Advantages and limitations

- New DSLs can be readily integrated in OpenSBLI to aid future-proofing
- Flexibility of algorithms, methods and equations.
- Use of external libraries like FFTW & implementation of implicit solvers needs to be done at OpenSBLI and the backend OPS level.
- Debugging for errors at different levels (partly helped by the OpenSBLI LaTeX debugging).

Conclusions & Acknowledgements

- New framework for the automated solution of finite difference methods on various architectures is developed and validated.
- Separation of concerns enables better model maintainability, and future-proofs the code as newer architectures arrive.
- Storing the first derivatives of velocity in the context of compressible Navier-Stokes solution is optimal across architectures.
- Funded by EPSRC grants EP/K038567/1 and EP/L000261/1, and European Commission H2020 grant 671571 “ExaFLOW: Enabling Exascale Fluid Dynamics Simulations”

Resources

- OpenSBLI released under GNU GPL: https://opensbli.github.io

- C. T. Jacobs, S. P. Jammy, N. D. Sandham (2017). OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *Journal of Computational Science*. DOI: 10.1016/j.jocs.2016.11.001

- S. P. Jammy, C. T. Jacobs, N. D. Sandham (In Press). Performance evaluation of explicit finite difference algorithms with varying amounts of computational and memory intensity. *Journal of Computational Science*. DOI: 10.1016/j.jocs.2016.10.015