Performance evaluation of explicit finite difference algorithms with varying amounts of computational and memory intensity

Satya P. Jammy[a,*], Christian T. Jacobs[a], Neil D. Sandham[a]

[a] *Aerodynamics and Flight Mechanics Group, Faculty of Engineering and the Environment, University of Southampton, University Road, Southampton, SO17 1BJ, United Kingdom*

Abstract

Future architectures designed to deliver exascale performance motivate the need for novel algorithmic changes in order to fully exploit their capabilities. In this paper, the performance of several numerical algorithms, characterised by varying degrees of memory and computational intensity, are evaluated in the context of finite difference methods for fluid dynamics problems. It is shown that, by storing some of the evaluated derivatives as single thread- or process-local variables in memory, or recomputing the derivatives on-the-fly, a speed-up of $\sim$2 can be obtained compared to traditional algorithms that store all derivatives in global arrays.

Keywords: Computational Fluid Dynamics; Finite Difference Methods; Algorithms; Exascale; Parallel Computing; Performance

1. Introduction

Explicit finite difference methods are an important class of numerical methods for the solution of partial or ordinary differential equations. For example, they are used for numerically solving the governing equations in computational fluid dynamics (CFD), astrophysics, seismic wave simulations, financial simulations, etc.

In CFD they are used by many researchers for the Direct Numerical Simulation (DNS) or Large Eddy Simulation (LES) of compressible flows. DNS is often performed to study boundary layers, aerofoils (involving both

*Corresponding author.
E-mail address: s.p.jammy@soton.ac.uk

hydrodynamics and noise computations) [1], mixing analysis [2], shock-wave boundary layer interactions [3] or benchmark test cases such as the Taylor-Green vortex [4], decaying homogeneous isotropic turbulence, etc. Even with the advances in computing hardware during the past decade, the current capabilities of DNS are limited to moderate Reynolds number flows [5].

It is expected that computing architectures will be capable of exaFLOPs (10^{18} Floating Point Operations) by 2018 and 30 exaFLOPs by 2030 [6]. Exascale architectures have the capability to perform DNS of the aforementioned examples (amongst others) at higher Reynolds numbers, or potentially wall-modelled LES of the full model of an aircraft at operating Reynolds numbers. However, while there is a consensus [6] that future architectures would not look much like the present IBM Blue Gene, Cray, or IBM Productive, it is hard to predict the architectural design of such exascale systems. For example, they are expected to comprise less memory per core than the existing architectures. Exploiting the full potential of the exascale architectures poses many challenges to researchers, such as the sustainability of the solver's implementation with the uncertainty of architectures, the need for new revolutionary algorithms/numerical methods, increasing computation to communication ratio and the likelihood of I/O bottlenecks.

To address the problem of sustainability, taking into account the uncertainty in future architectures, one solution adopted by the CFD community involves decoupling the work of a domain scientist and a computational/-computer scientist [7]. In this approach, Domain Specific Languages (DSL) are developed by the computational/computer scientists, and the specifics of the problem and the numerical solution method are specified in the DSL by the domain scientist. Using source-to-source translation the numerical solver is targetted towards different parallel hardware backends (e.g. MPI, CUDA, OpenMP, OpenCL, and OpenACC) [8, 9]. This ensures that, for new architectures, only the backend that interfaces with the new architecture needs to be written and supported by the translator. The underlying implementation of the solver remains the same, thereby introducing a separation of concerns.

On the algorithms front, a lot of effort has gone into rewriting CFD solvers to exploit the available FLOPS of existing architectures. While the architectures have changed drastically in the last decade, algorithms have not advanced at a similar pace [6]. Some algorithmical changes have been attempted by [10, 11] to reduce the data transfer on Graphics Processing Units (GPUs), but a complete and detailed study on the performance of such algorithms on the existing CPU-based architectures is currently lacking. A first step towards exascale computing would be to evaluate the performance of algorithms characterised by varying intensities of memory usage

and computational cost on current CPU-based architectures for a relevant hydrodynamic test case, solved using a finite difference scheme.

To facilitate these investigations, the capabilities of the recently developed OpenSBLI framework [12] are extended to easily generate algorithms with varying amounts of computational and memory intensity. OpenSBLI is a framework for the automated derivation and parallel execution of finite difference-based models. It is written in Python and uses SymPy to generate a symbolic representation of the governing equations and discretisation. The framework generates OPS-compliant C code that is targetted towards MPI via the OPS active library [9]. A similar approach can also be applied to any set of compute-intensive equations solved using finite difference methods.

The aims of this paper are to: (a) study the performance of various algorithms on current multi-core CPU-based architectures, (b) identify the best possible algorithm for the solution of explicit finite difference methods on current multi-core CPU-based architectures, and (c) demonstrate the ease at which algorithmic manipulations can be achieved with OpenSBLI framework to overcome the challenges exascale architectures can pose.

The rest of the paper is organised as follows. The various algorithms are described in section 2. The validation of the algorithms is presented in section 3. The performance and scaling results are presented in section 4. Some conclusions are drawn in section 5.

2. Algorithms

All the algorithms presented in this paper solve the three-dimensional unsteady compressible Navier-Stokes equations, with constant viscosity, given by

$$\frac{\partial \rho}{\partial t} = -\frac{\partial}{\partial x_j}\left[\rho u_j\right], \tag{1}$$

$$\frac{\partial \rho u_i}{\partial t} = -\frac{\partial}{\partial x_j}\left[\rho u_i u_j + p\delta_{ij} - \tau_{ij}\right], \tag{2}$$

and

$$\frac{\partial \rho E}{\partial t} = -\frac{\partial}{\partial x_j}\left[\rho E u_j + u_j p - q_j - u_i \tau_{ij}\right], \tag{3}$$

for the conservation of mass, momentum and energy, respectively. The quantity ρ is the fluid density, u_i is the velocity vector, p is pressure and E the total energy. The stress tensor τ_{ij} is defined as,

$$\tau_{ij} = \frac{1}{\mathrm{Re}}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial u_k}{\partial x_k}\right), \tag{4}$$

where δ_{ij} is the Kronecker Delta and Re is the Reynolds number. The heat flux term q_j is given by,

$$q_j = \frac{1}{(\gamma - 1)\ \mathrm{M}^2\ \mathrm{Pr}\ \mathrm{Re}} \frac{\partial T}{\partial x_j}, \tag{5}$$

where, T is temperature, M is the Mach number of the flow, Pr is Prandtl number and γ is the ratio of specific heats. The pressure and temperature are given by,

$$p = (\gamma - 1)\left(\rho E - \frac{1}{2}\rho u_j^2\right), \tag{6}$$

and

$$T = \frac{\gamma \mathrm{M}^2 p}{\rho}, \tag{7}$$

respectively. The variables that are advanced in time $(\rho, \rho u_i, \rho E)$ are referred to as the conservative variables, and the right-hand sides (RHS) in the mass, momentum and energy equations are referred to as the residuals of the equations.

The mass, momentum and energy equations are discretised in space using a fourth-order central finite-difference scheme and a low storage Runge-Kutta (RK) scheme with three stages of temporal discretisation. For improved stability, the convective terms in the governing equations are rewritten using the formulation of [13],

$$\frac{\partial}{\partial x_j}\rho\phi u_j = \frac{1}{2}\left(\frac{\partial}{\partial x_j}\rho\phi u_j + u_j\frac{\partial}{\partial x_j}\rho\phi + \rho\phi\frac{\partial}{\partial x_j}u_j\right), \tag{8}$$

where ϕ is 1, u_i or internal energy (e) for the mass, momentum and energy equations, respectively. To improve the stability of the present scheme, the viscous terms in the momentum and energy equations are expanded to second derivatives as used by [2, 10, 14].

A generic pseudo-code of the solution algorithm is shown in figure 2. The time loop is the most computationally expensive part of the algorithm. It consists of evaluating the primitive variables (p, u_i, T), spatial derivatives, the residual for each equation and advancing the solution in time. This is achieved by iterating over the solution points of the grid, referred to as the grid loop in the rest of the paper. Various algorithms used for the evaluation of the residual of the equations are presented herein. Starting with a memory-intensive algorithm representing a typical handwritten CFD solver, the amount of memory used and the computational intensity are varied, either by re-evaluating the derivatives on-the-fly or evaluating the

```
set-the-initial-condition
for each-iteration do
  save-state
  for each-rk-substep do
    evaluate-u_i,p,T
    evaluate-the-derivatives
    evaluate-the-residual-of-the-equations
    boundary-conditions
    advance-solution-in-time
  end for // end of rk sub loop
end for // end of iteration loop
```

Figure 1: Pseudo-code for the solution of the compressible Navier-Stokes equations.

derivatives using process-local variables. In all the algorithms presented, the primitive variables are evaluated and stored in memory.

Baseline algorithm (BL). This algorithm incorporates features similar to a typical handwritten static algorithm (i.e. the derivatives in the residual of each equation are evaluated and stored in memory as arrays of grid point values; these are referred to as work arrays in the rest of the paper) on CPUs, to run as a sequential or parallel using MPI or OpenMP. For multi-threaded parallel programs, this requires the algorithm to be thread-safe in order to avoid race conditions; these occur when a variable is updated in the grid loop and the updated variable is used to update another variable in the same loop. For example, in the evaluation of the primitive variables from the conservative variables, the equation for pressure (6) is dependent on the evaluated velocity components, and the equation for temperature (7) is dependant on the evaluated pressure. When running on threaded architectures, this potentially results in race conditions. This means that temperature could be evaluated before evaluating the pressure, and pressure could be evaluated before the velocity components are evaluated. Similar candidates for race conditions exist in the update equations (which advance the conservative variables forward in time) of the RK scheme.

To remove the race conditions, the code is generated such that no variable is updated and used in the same loop. This is achieved by separating the evaluations into multiple loops over grid points. For example, in the evaluation of primitive variables, the velocity components (u_0, u_1, u_2) are grouped into a single loop as the evaluations are independent, but the pressure and temperature are evaluated in different loops.

When generating the code that implements the BL algorithm, the first and second derivatives in the equations are evaluated and stored in work arrays in order to compute the RHS residual. The evaluation of the derivative of a combination of variables (e.g $\partial(\rho u_0 u_0)/\partial x_0$) is achieved in two stages. In the first stage the function $\rho u_0 u_0$ is evaluated and stored in a work array. In the second stage the derivative of the work array is evaluated using the central finite difference formula, and this result is stored in a new work array. The work array used in the first stage is not freed in memory, but is overwritten/reused when evaluating other quantities.

The baseline algorithm is optimised such that computationally-expensive divisions are minimised. Rational numbers (e.g. finite difference stencil weights) and all the negative powers of constants in the equations are evaluated and stored at the start of the simulation. Typically, these are $\gamma^{-1}, \mathrm{Pr}^{-1}$, Re^{-1}, and so on.

```
ndim=3 # Problem dimension
# Define the compressible Navier-Stokes in Einstein notation.
mass="Eq(Der(rho,t),- Skew(rho*u_j,x_j))"
momentum="Eq(Der(rhou_i,t),-Skew(rhou_i*u_j,x_j)-Der(p,x_i)+Der(
    tau_i_j,x_j))"
energy="Eq(Der(rhoE,t),-Skew(rhoE*u_j,x_j)-Conservative(p*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/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/((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 - (1/2)*rho*(u_j*u_j)))"
temperature="Eq(T,gama*Minf*Minf/rho)"
formulas=[velocity,pressure,temperature]
# Create the problem and expand the equations.
problem=Problem(equations,substitutions,ndim,constants,
    coordinate_symbol,metrics,formulas)
ex_eq=problem.get_expanded(problem.equations)
ex_form=problem.get_expanded(problem.formulas)
ss=Central(4) # Fourth-order central differencing
ts=RungeKutta(3) # Third-order RK scheme
# Create a numerical grid of solution points
```

```
np=[64]*ndim;deltas=[2.0*pi/np[i] for i in range(len(length))]
grid=Grid(ndim,{'delta':deltas,'number_of_points':np})
# Perform the discretisation
sd=SpatialDiscretisation(ex_eq,ex_form,grid,ss)
td=TemporalDiscretisation(ts,grid,constant_dt=True,sd)
# Boundary conditions
bc=PeriodicBoundaryCondition(grid)
for dim in range(ndim):
    bc.apply(td.prognostic_variables,dim)
# Constant initial conditions
ics=["Eq(grid.work_array(rho),1.0)","Eq(grid.work_array(rhou0)
    ,1.0)","Eq(grid.work_array(rhou1),0.0)","Eq(grid.work_array(
    rhou2),0.0)","Eq(grid.work_array(rhoE),1.0)"]
ics=GridBasedInitialisation(grid,initial_conditions)
io=FileIO(td.prognostic_variables)# I/O
# Simulation parameters
var=['niter','Re','Pr','gama','Minf','precision','name','deltat'
    ]
values=[1000,1600,0.71,1.4,0.1,"double","test",3.385*10**-3]
sp=dict(zip(vars,values))# dictionary
code=OPSC(grid,sd,td,bc,ics,io,sp)#code generation
```

Listing 1: Key lines of the setup file for obtaining the BL algorithm.

A sample setup file used to generate an implementation of this algorithm in OpenSBLI is shown in listing 1. All the algorithms presented next are also optimised to reduce computationally expensive divisions. The setup file for other algorithms is similar to the BL algorithm with extra attributes to control the combinations of memory used and computational intensity.

Recompute All algorithm (RA). In contrast to the BL algorithm, the evaluation of pressure and temperature are first rewritten using the conservative variables,

$$p = (\gamma - 1)\left(\rho E - \frac{1}{2}\rho\left(\frac{\rho u_j}{\rho}\right)^2\right), \tag{9}$$

and

$$T = \frac{\gamma \mathrm{M}^2 p}{\rho} = \frac{\gamma\left(\gamma - 1\right)\mathrm{M}^2\left(\rho E - \frac{1}{2}\rho\left(\frac{\rho u_j}{\rho}\right)^2\right)}{\rho}, \tag{10}$$

within the code to avoid race condition errors while fusing loops for the evaluation of the primitive variables. Then, to evaluate the residual of the equation, all the continuous spatial derivatives in the residual are replaced by their respective finite difference formulas in the code generation stage. This differs from the BL algorithm in that, instead of evaluating the derivatives to work arrays and using them to compute the RHS residual, the code

```
for each-solution-point do
  double t1 = central difference formula for ∂u0/∂x0
  double t2 = central difference formula for ∂u1/∂x1
  ⋮
  residual = t1 + t2 ...
end for
```

Figure 2: Pseudo-code for residual evaluation using SN algorithm.

generation process directly replaces the derivatives by their respective finite difference formulas such that they are recomputed every time.

This algorithm results in a code in which no work arrays are used for storing the derivatives. The memory required for this algorithm is therefore the least of all algorithms and the computational intensity is the highest of all. The control parameters to generate code for this algorithm are shown in listing 2.

```
grid = Grid(ndim,{"delta":deltas,"number_of_points":np})
grid.store_derivatives = False # Do not store derivatives
```

Listing 2: Control parameters to generate the code for RA algorithm.

Store None algorithm (SN). This algorithm is similar to the RA algorithm. The difference is that, in the loop over grid points where the residuals are evaluated, each derivative in the RHS is evaluated to a single thread- or process-local variable. These variables are then used to update the residuals on a point-by-point basis, rather than storing all evaluations in a global-scope, grid-sized work array. To generate the code that implements this algorithm in OpenSBLI, the grid attribute `local_variables` should be set to `True` along with the control parameters given in listing 2. The pseudo-code for the residual evaluation as described here is provided in figure 2.

The memory footprint of this algorithm is similar to that of the RA algorithm, but is slightly less computationally-intensive. This is because, for example, if a derivative is evaluated to a process-local variable then it can be reused if that derivative appears in any of the equations more than once.

Recompute Some algorithm (RS). In this algorithm, some of the derivatives (in this case, the first derivatives of the velocity components) are stored in work arrays and the remaining derivatives are replaced by their respective finite difference formulas in the residual. The evaluation of primitive variables follows the same procedure as the RA algorithm. Listing 3 shows the

control parameters used to generate code for the RS algorithm. The memory usage for this algorithm is moderate, i.e. it is more than the RA algorithm but less than the BL algorithm.

```
grid = Grid(ndim,{"delta":deltas,"number_of_points":np})
grid.store_derivatives = False # Do not store derivatives
grid.derivatives_to_store = set(problem.
    get_expanded_term_in_equations("Der(u_i,x_j)"))
```

Listing 3: Control parameters to generate the code for the RS algorithm

Store Some algorithm (SS). This algorithm is a fusion of the RS and SN algorithms, such that the derivatives that are not stored in the RS algorithm are evaluated and stored in thread- or process-local variables as per the SN algorithm. Listing 4 shows the control parameters used to generate code for this algorithm. Compared to the SN algorithm, an additional nine work arrays would be required for the SS algorithm for the 3D cases, and an additional four work arrays for 2D cases, since the first derivatives of the velocity components are now stored.

```
grid = Grid(ndim,{"delta":deltas,"number_of_points":np})
grid.store_derivatives = False
grid.derivatives_to_store = set(problem.
    get_expanded_term_in_equations("Der(u_i,x_j)"))
grid.local_variables = True
```

Listing 4: Control parameters in setup file to generate the code for the SS algorithm.

3. Validation

The baseline (BL) algorithm is validated for a 3D compressible Taylor-Green vortex problem, to check the correctness of the solver. The initial conditions and the post-processing procedure are described in [4]. The simulations are performed in a cube of non-dimensional length 2π, with periodic boundary conditions in all three directions for grids containing $64^3, 128^3, 256^3$ and 512^3 solution points. The Mach number, Prandtl number and Reynolds number of the flow are taken as 0.1, 0.71 and 1600, respectively. The non-dimensional time-step for the 64^3 grid size was set to 3.385×10^{-3}, and was halved for each increase in the grid size by a factor of 2^3. Double-precision is used throughout all simulations presented in this paper.

Figure 3 shows the evolution of kinetic energy and enstrophy compared with the reference data [15]. The results from the BL algorithm agree very well with the reference data for the 512^3 case. For computational expedience, the other algorithms are validated on the 128^3 grid. For each one, the

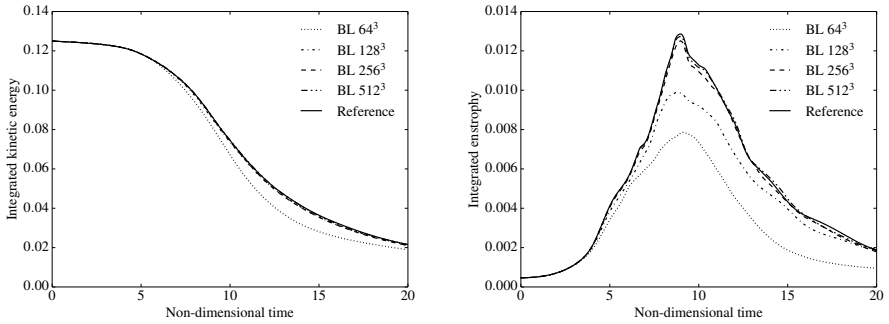

Figure 3: Left: Evolution of the integral of kinetic energy. Right: Evolution of the integral of enstrophy.

results relative to the BL algorithm are found to be the same up to machine precision.

4. Performance evaluation

After checking that the results from the various algorithms match, the performance of the different algorithms were evaluated using the same Taylor-Green vortex test case described in section 3. All simulations are performed on ARCHER (the UK National Supercomputing Service) and the code that implements the various algorithms is compiled using the Cray C compiler (version 2.4.2) with the -O3 optimisation flag. Each ARCHER node comprises 24 cores, with each MPI process being mapped to its own individual core. All simulations for performance evaluation purposes are run in parallel using 24 MPI processes/cores (one ARCHER node). The run-time of the time iteration loop was recorded for 500 iterations and is summarised in table 1 for a range of grid sizes.

Nx	**Ny**	**Nz**	**BL**	**RA**	**RS**	**SN**	**SS**
64	64	64	16.21	9.29	10.76	8.44	9.78
128	128	128	182.55	98.18	97.36	90.72	88.95
256	256	256	1561.52	765.42	802.76	693.66	685.25

Table 1: Total run-time in seconds for different grid sizes for all algorithms on ARCHER using 24 MPI processes.

The data in table 1 is plotted in figure 4; from this figure it can be inferred that when the amount of memory access is reduced, the current CPU-based

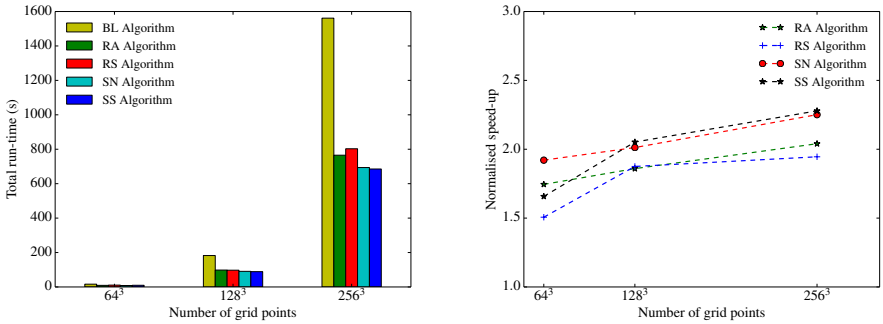

Figure 4: Left: Figure showing data in table 1. Right: speed-up of algorithms normalised with the BL algorithm.

architectures perform better, even though the computational intensity of such algorithms is higher. The baseline algorithm is a factor of ∼2 slower than all the other algorithms presented in this paper. For larger grid sizes the benefit of the SS algorithm becomes more pronounced.

4.1. Scaling

Strong scaling tests were performed for the best performing algorithm (i.e. the SS algorithm) on ARCHER for the test problem with a total of 1.07×10^9 grid points and the runtime of the time iteration loop was recorded for 10 iterations. Figure 5 shows the strong scaling results on ARCHER up to 73,728 MPI processes/cores (i.e. 3,072 ARCHER nodes). The minimum number of processes required for running the problem is 120. The algorithm shows a near-linear scaling (speed-up of 2) until 36,864 MPI processes (i.e. 1,536 ARCHER nodes) and thereafter the speed-up is ~ 1.5 as the process count doubles.

Weak scaling tests were also performed for the SS algorithm. Here, the number of MPI processes was varied from 192 to 65,856 (i.e. 8 to 2,744 ARCHER nodes), while the number of grid points per MPI process was kept fixed at 64^3 and the runtime of the time iteration loop was recorded for 10 iterations. The largest grid size considered comprises ∼17 billion solution points. Figure 6 demonstrates that the normalised run-time is near-ideal.

5. Conclusion

In this paper the automated code generation capabilities of the OpenSBLI framework have been extended to easily modify the memory usage and

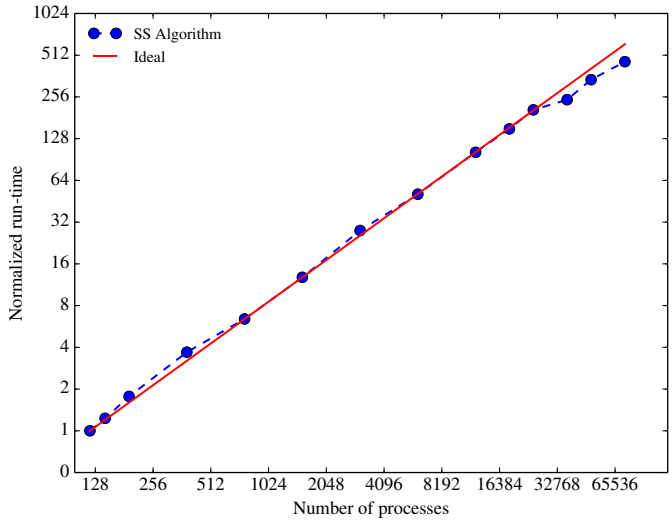

Figure 5: Strong scaling of the SS algorithm on ARCHER up to 73,728 cores using 1.07×10^9 grid points. The run-time has been normalised by that of the 120-process case.

computational intensity of the solution algorithm. It was found that the baseline (BL) algorithm featured in traditional CFD codes, in which all derivatives are evaluated and stored in work arrays, is not the best algorithm in terms of performance on current multi-core CPU-based architectures. Re-computing all or some of the derivatives performs better than the baseline algorithm. The best algorithm found here for the solution of the compressible Navier-Stokes equations is to store only the first derivatives of velocity components in work arrays, and compute the remaining spatial derivatives and store them in thread- or process-local variables. The run-time of such an algorithm has been shown to be $\sim$2 times smaller than the BL algorithm. Through the use of modern code generation techniques in the OpenSBLI framework, it has been demonstrated that by changing just a few attributes (three in this case) in the problem setup file, different algorithms with varying degrees of memory and computational intensity can be readily generated automatically. The methodology presented in this paper can also be used to find the best possible algorithm for other existing architectures such as GPUs or Intel Xeon Phi coprocessors. Moreover, existing numerical models that use finite difference methods for the solution of any governing equations can be optimised for the current CPU-based architectures. When exascale systems

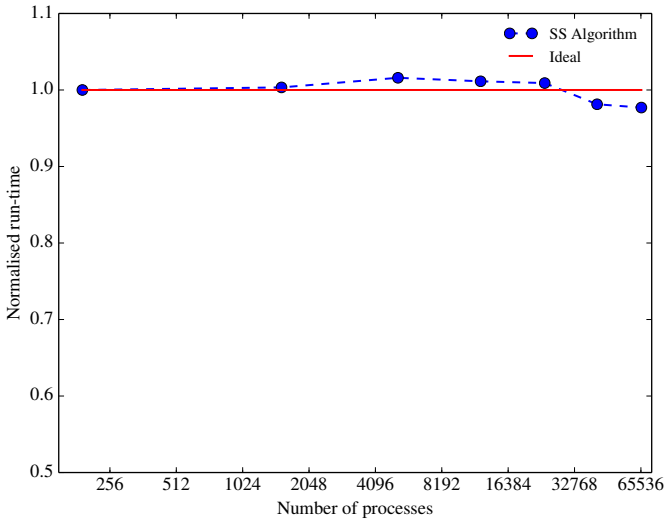

Figure 6: Weak scaling of the SS algorithm on ARCHER with 64^3 grid points per MPI process up to 65,856. The results have been normalised by the run-time from the 192-process case.

become available, depending on their architecture and amount of available memory, users can readily tune the memory and computational intensity in the OpenSBLI framework to determine the best performing algorithm on such systems.

6. Acknowledgements

SPJ was supported by an EPSRC grant entitled "Future-proof massively-parallel execution of multi-block applications" (EP/K038567/1). CTJ was supported by a European Union Horizon 2020 project grant entitled "ExaFLOW: Enabling Exascale Fluid Dynamics Simulations" (grant reference 671571). The data behind the results presented in this paper will be available from the University of Southampton's institutional repository. The authors acknowledge the use of the UK National Supercomputing Service (ARCHER), with computing time provided by the UK Turbulence Consortium (EPSRC grant EP/L000261/1).

References

[1] L. E. Jones, R. D. Sandberg, Acoustic and hydrodynamic analysis of the flow around an aerofoil with trailing-edge serrations, Journal of Fluid Mechanics 706 (2012) 295–322. doi:10.1017/jfm.2012.254.

[2] S. Pirozzoli, M. Bernardini, S. Marié, F. Grasso, Early evolution of the compressible mixing layer issued from two turbulent streams, Journal of Fluid Mechanics 777 (2015) 196–218. doi:10.1017/jfm.2015.363.

[3] B. Wang, N. D. Sandham, Z. Hu, W. Liu, Numerical study of oblique shock-wave/boundary-layer interaction considering sidewall effects, Journal of Fluid Mechanics 767 (2015) 526–561. doi:10.1017/jfm.2015.58.

[4] J. DeBonis, Solutions of the Taylor-Green Vortex Problem Using High-Resolution Explicit Finite Difference Methods, 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition (February) (2013) 1–9. doi:10.2514/6.2013-382.

[5] W. C. Reynolds, Whither Turbulence? Turbulence at the Crossroads: Proceedings of a Workshop Held at Cornell University, Ithaca, NY, March 22–24, 1989, Springer Berlin Heidelberg, Berlin, Heidelberg, 1990, Ch. The potential and limitations of direct and large eddy simulations, pp. 313–343. doi:10.1007/3-540-52535-1_52.

[6] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, D. Mavriplis, CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences, Nasa Cr-2014-21878 (March) (2014) 1–73.

[7] C. T. Jacobs, M. D. Piggott, Firedrake-Fluids v0.1: numerical modelling of shallow water flows using an automated solution framework, Geoscientific Model Development 8 (3) (2015) 533–547. doi:10.5194/gmd-8-533-2015.

[8] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, P. H. Kelly, PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes, in: High Performance Computing, Networking Storage and Analysis, SC Companion, IEEE Computer Society, 2012, pp. 1116–1123.

[9] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, S. McIntosh-Smith, The OPS Domain Specific Abstraction for Multi-Block Structured Grid Computations, in: Proceedings of the 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, IEEE Computer Society, 2014, pp. 58–67. doi:10.1109/WOLFHPC.2014.7.

[10] F. Salvadore, M. Bernardini, M. Botti, GPU accelerated flow solver for direct numerical simulation of turbulent flows, Journal of Computational Physics 235 (2013) 129–142. doi:10.1016/j.jcp.2012.10.012.

[11] J. C. Thibault, Implementation of a Cartesian Grid Incompressible Navier-Stokes Solver on Multi-Gpu Desktop Platforms Using Cuda, New Horizons (May) (2009) 1–15.

[12] C. T. Jacobs, S. P. Jammy, N. D. Sandham, OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures, Journal of Computational Science, SubmittedarXiv:1609.01277.

[13] G. A. Blaisdell, E. T. Spyropoulos, J. H. Qin, The effect of the formulation of nonlinear terms on aliasing errors in spectral methods, Applied Numerical Mathematics 21 (3) (1996) 207–219. doi:10.1016/0168-9274(96)00005-0.

[14] N. Sandham, Q. Li, H. Yee, Entropy Splitting for High-Order Numerical Simulation of Compressible Turbulence, Journal of Computational Physics 178 (2) (2002) 307–322. doi:10.1006/jcph.2002.7022.

[15] Z. Wang, K. Fidkowski, R. Abgrall, F. Bassi, D. Caraeni, A. Cary, H. Deconinck, R. Hartmann, K. Hillewaert, H. Huynh, N. Kroll, G. May, P.-O. Persson, B. van Leer, M. Visbal, High-order cfd methods: current status and perspective, International Journal for Numerical Methods in Fluids 72 (8) (2013) 811–845. doi:10.1002/fld.3767.