Technical report for eCSE-0507: Developing Fluidity for high-fidelity coastal modelling

Angus Creech, University of Edinburgh[1]
Adrian Jackson, EPCC
20 March 2017

1 Abstract

This project aimed to improve the performance of Fluidity for both general computational fluid dynamics and tidal modelling problems. A secondary objective was to add features to Fluidity to improve its ease of use. Both of these were successful, with Fluidity running noticeably faster even at high core counts. This changes the level of detail that fluids problems can be studied with Fluidity, and impacts upon research that examines high Reynolds number, turbulent flows – particularly in areas such as engineering aerodynamics, wind energy, marine energy, and environmental/pollution modelling.

2 Introduction

Energetic tidally-driven flow in coastal areas such as straits represents a challenge for high fidelity simulation, as the turbulent flow processes to be modelled range from less than 1m to greater than 1km (eg. tidal jets). More than that, such turbulence is highly anisotropic, in a domain that may be only 100-200 metres vertically, but have a horizontal surface area (the sea surface) of 1000s of km^2. Coastal domains also have an irregular shape, with undulating bathymetry and complex coastlines; this makes them well suited to the unstructured mesh approaches used in finite-element computational fluid dynamics.

Fluidity is an open source finite-element computational fluid dynamics code that solves the non-hydrostatic Navier-Stokes momentum equation and continuity equation, which is capable of modelling free surfaces and is designed for irregularly shaped, unstructured grids. Fluidity is written in C++ and Fortran, with the majority written in Fortran. It utilises both OpenMP and MPI for parallel computation, and has been the subject of previous development for efficient scaling to thousands of cores on both HECTOR and ARCHER. It also supports a stable velocity-pressure element pair, P1DG-P2, which is first-order Discontinuous Galerkin velocity and second-order Continuous Galerkin pressure. This pair is LBB stable and suitable for high Reynolds number flow.

That said, assembly of the momentum equations in DG is expensive, taking a substantial portion of each timestep. Optimisation of this was required for a substantial performance improvements. Each timestep in Fluidity consists of several non-linear Picard iterations, which are very similar in nature to the SIMPLE family of algorithms of Patankar (Patankar 1980). These follow a sequence shown in Table 1.

Step #	Action
1	Calculate any fields dependent upon velocity
2	Assemble global momentum matrix from local element data
3	Solve the momentum equation for tentative velocity solution $\underline{u}^*$
4	Calculate the pressure correction Δp
5	Correct $\underline{u}^*$ to u^{n+1} using Δp, and update p^{n+1}
6	Either perform next Picard iteration (go to step 1), or finish timestep

Table 1. Outline of Picard iteration within Fluidity.

[1] Principal Investigator and corresponding author. Email: a.creech@ed.ac.uk

Typically Fluidity requires at least two Picard iterations per timestep to achieve satisfactory convergence of the velocity and pressure fields. Substantial performance improvements were made to the DG assembly routines, alone giving noticeable improvements in overall runtimes from low to 1000+ core counts. Other miscellaneous optimisations and bug fixes were made, in particular: graph reordering of mesh files to produce more coherent meshes; enabling the Fluidity to run with 32-bit floating point; load balancing improvements for extruded meshes; faster sub-cycling routines; simplified checkpointing.

Secondly, two new Large Eddy Simulation (LES) turbulence models were developed, one for isotropic grids (Deardroff 1970) and one for highly anisotropic grids (Roman, et al. 2010), which utilised the naturally stable Compact DG method (Peraire and Persson 2008). The anisotropic method in particular is necessary for coastal modelling, which typically feature grids with a horizontal/vertical element aspect ratio of around (20:1). As part of this, an eddy-viscosity wall model implemented. This was verified with against experimental data for a well-known test case.

The third part involved simplifying the configuration of new simulations, particularly those involving DG. Originally in 'stock' Fluidity, fields and field options useful to DG simulations had to be all created and set manually, which is a complex and time-consuming task; now, many options trigger default behaviour, such as the creation of default options and required additional fields. In particular, creating a DG velocity field now automatically creates projected CG velocity fields for results output, which leads an order of magnitude reduction in file size.

Lastly, there were the creation of tools and utilities to simplify the process of tidal modelling within Fluidity. These varied from additional Fortran routines and configuration scheme changes, to Python utilities designed to be embedded within simulations via the configuration file.

3 Development setup

There were two test systems used in development, with different properties. Firstly, was an Opteron server (herein called 'rack server') which represented the low-end of the computing scale. Much of the rudimentary testing and development work was conducted there. The second system was ARCHER, which has 100 000+ cores, ie. the high-end: the profiling and benchmarking was undertaken there. The individual configurations of each are detailed below.

3.1 ARCHER

Hardware

The full specifications of ARCHER are detailed elsewhere; we will cover them briefly. The thousands of compute nodes each consist of two 12-core Xeon E5-2650 v2 CPUs, giving a total of 24 cores per node. Previous simulation work on ARCHER with Fluidity has shown that memory contention can be an issue, so that, as with the rack server, optimum performance occurs when Fluidity is run in hybrid OpenMPI/MPI mode as before. This means that there are 2 OpenMP threads for each MPI tasks, giving a total of 12 MPI tasks per node.

Software

A new environment module file *fluidity-gcc* was created for Fluidity, which loaded the ARCHER module PrgEnv-gnu whilst swapping out *gcc* for *gcc/4.9.3*. The full list of modules can be found in the appendix. PETSc 3.5.4 was compiled for use. Profiling information was generated using Allinea MAP; binaries were compiled with the additional *–g* flag for profiling. An automatic generator for PBS scripts was written, which automatically sets up all the necessary environment flags and module files, copies across any required Python files, parses the simulation for the latest checkpoint file, and submits the job to the desired queue. This greatly eased the use of running Fluidity on ARCHER, and works on any general Fluidity simulation.

4 Discontinuous Galerkin matrix assembly optimisation

The P1DG-P2 velocity-pressure element pair used within Fluidity (first order Discontinuous velocity; second order continuous pressure) is provably LBB stable (Ladyzhenskaya 1969) (Babuška 1973) (Brezzi 1974), meaning that it is suitable for simulation of high-Reynolds number, turbulent flow.

However, the Discontinuous Galerkin (DG) finite-element assembly routines in Fluidity are considerably slower than the Continuous Galerkin ones, making them prohibitively expensive for large simulations: increasing the number of cores to counteract the lack of performance, often leads to communication overheads being significant. Thus the size of simulations are limited, even on supercomputers such as ARCHER. Fortunately, most of the code related to DG assembly exists in one file, and so performance enhancement efforts can be focused solely upon this one file.

4.1 Simple test case

This was based upon an idealised, rectilinear channel with a logarithmic velocity profile as a Dirichlet condition at the inlet, an open boundary at the outlet, a free-slip boundary condition on the top and sides, and a quadratic drag on the bottom. No normal flow was allowed on the top, bottom or sides. The channel measured 1000m x 250m x 50m, as shown in Figure 1. The velocity solves used the GMRES solver with SSOR preconditioning; the pressure solves used Conjugate Gradient with SSOR for preconditioning. The simulation was set to run for 100 time-steps, with 2 Picard iterations per timestep. This meant the DG assembly would be called 200 times, which was deemed sufficient for performance measurements.

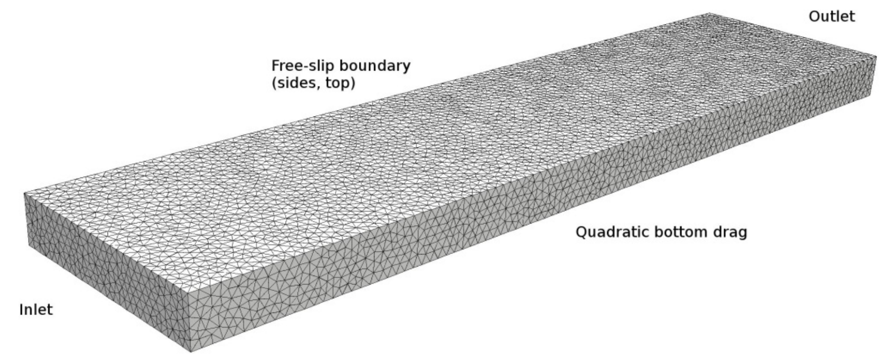


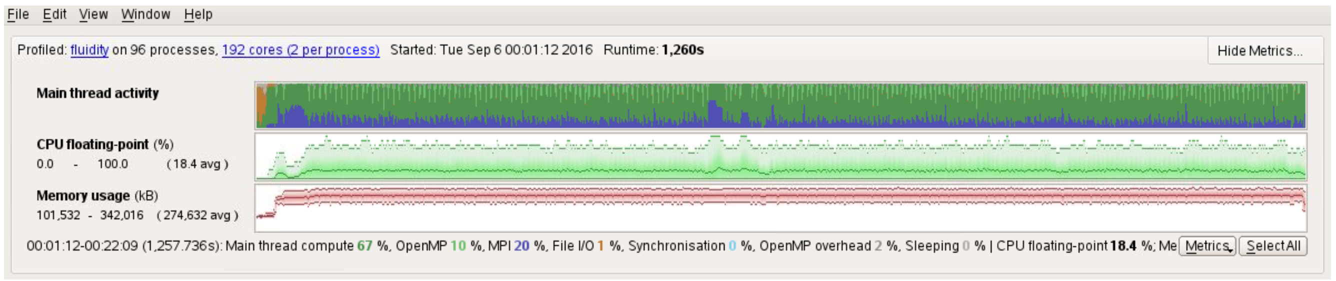
Figure 1. Channel test case for DG optimisation, here showing the mesh for 24 MPI tasks. The mesh was generated using the GMSH meshing utility.

MPI tasks	Cores (2xOpenMP)	Nodes	Max. Δx (m)	Elements	Elements / MPI task
96	192	8	4.05665	968 664	10 090
384	768	32	2.4730	4 169 234	10 857
768	1536	64	1.9309	8 569 453	11 158

Table 2. Soft-scaling configurations for profiling on ARCHER.

For profiling on ARCHER, a weak-scaling approach was used, since partitioning the problem into ever-smaller chunks results in tiny partitions and a problem dominated by communication overhead; this does not represent a realistic use case. The complexity of the mesh was controlled through the maximum allowable size of the elements (max. Δx), and whilst the number of MPI tasks scaled exponentially (see Table 2), the number of elements was kept approximately constant. Allinea MAP was used for profiling, with the *–g* option added to the compiler flags.

Upon running the simulations, it was clear from the time-series of CPU and memory usage (see Figure 2) there was a spin-up period at the beginning of execution before the simulation would be running at full tilt. It was not certain whether this would skew performance figures unduly, however, MAP does allow the deselection of the start-up period when calculating performance figures.



Profiled: fluidity on 96 processes, 192 cores (2 per process) Started: Tue Sep 6 00:01:12 2016 Runtime: **1,260s**

00:01:12-00:22:09 (1,257.736s): Main thread compute 67 %, OpenMP 10 %, MPI 20 %, File I/O 1 %, Synchronisation 0 %, OpenMP overhead 2 %, Sleeping 0 % | CPU floating-point **18.4** %; Me

Figure 2. View of thread activity, CPU usage, and memory usage time-series in Allinea MAP, for test case using 192 cores. Note the 'spin-up' period at the start (far left of graphs).

4.2 Unoptimised performance results

Weak scaling was calculated from the following equation, ie. $s = t_1/t_N$, where t_1 is the time taken to completion on 1 processing unit, and t_N the time taken on N processing units. However, due to memory and processor limitations, it is difficult to run a meaningful problem small enough for 1 core in this context, we shall use t_{192} instead as our base case. This gives us the scaling shown in Figure 3.

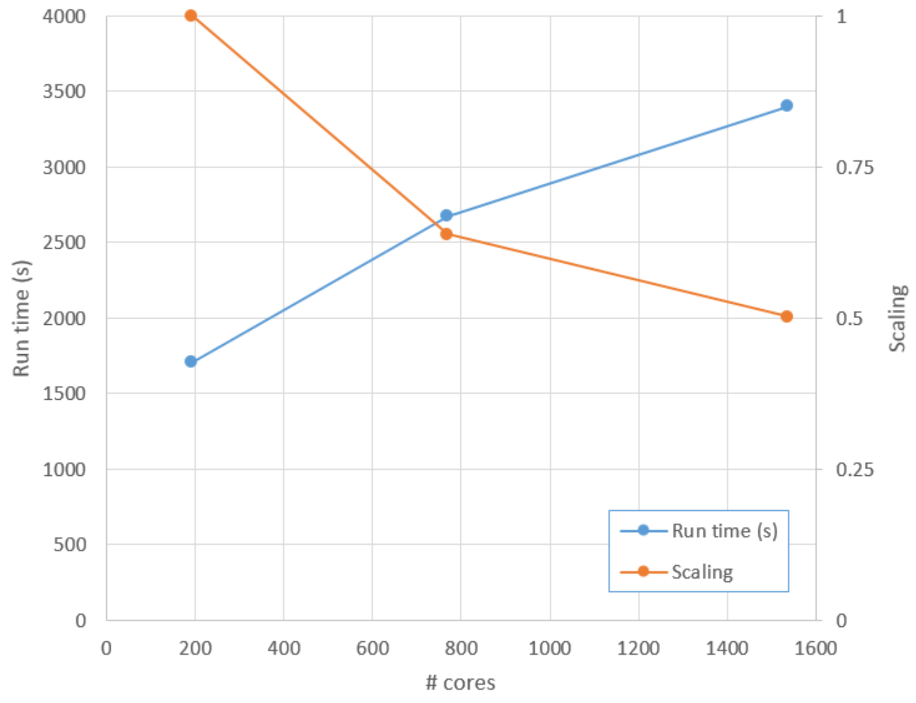


Figure 3. Scaling and run times of unoptimised Fluidity code

This can be broken down into the most expensive high level routines, shown in Table 3. Note, that any direct PETSC library calls or wrappers have been removed, and that these figures are inclusive (i.e. routines times include the time of any routine called from with that routine. What should be clear is that removing the spin-up phase of the simulation can change the performance figures by more than 1 %. Therefore, only the % runtimes without spin-up shall be used hereon in. These are represented in Figure 4.

	% runtime (with / without spin-up)					
# cores	192		*768*		*1536*	
Function name						
solve_momentum	90.6	92.20	91.6	92.3	93.2	94.1

correct_pressure	37.6	38.4	47.0	48.3	52.7	55.9
construct_momentum_dg	28.3	29.3	20.3	21.1	17.1	17.6
construct_momentum_element_dg	27.8	28.7	19.9	20.8	16.8	17.3
advance_velocity	16.2	15.8	17.1	15.5	17.1	14.9
construct_momentum_interface_dg	13.5	14.0	9.5	9.9	8.1	8.3
subcycle_momentum_dg	13.4	13.0	13.7	11.8	14.5	12.3
assemble_cmc_dg	6.3	6.6	5.0	5.2	4.2	4.2
local_assembly_cdg_face	6.6	6.9	4.6	4.8	3.9	4.0
calculate_courant_number	4.9	5.4	4.1	4.4	4.2	4.3

Table 3. % run times of unoptimised code

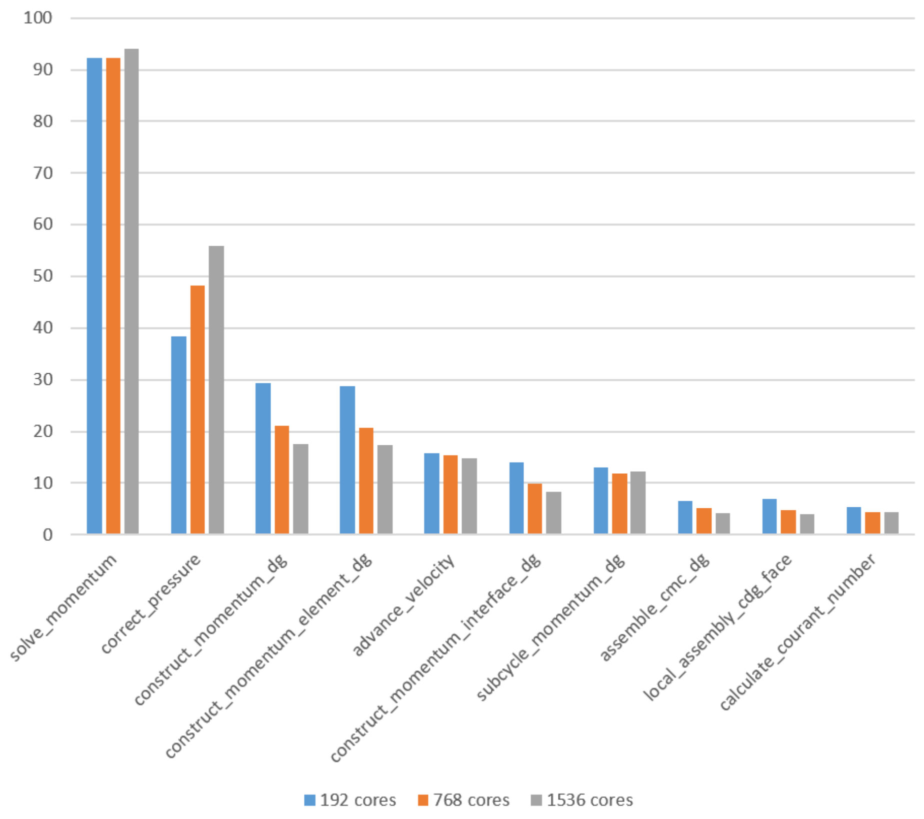


Figure 4. Percentage runtimes for unoptimised code

As the main piece of code, *solve_momentum* dominates the calculations, representing over 90% of execution time: it calls all of the proceeding functions. *correct_pressure* is nearly 38-55% of execution time, virtually all of which is spent inside PETSC solves; the same goes for *advance_velocity*. As optimisation of PETSC use was beyond the scope of this project, the next largest valid target was *construct_momentum_dg*. This is what assembles the global momentum matrix for solution. It does this calling *construct_momentum_element_dg* CME_DG for short) within a tight loop, where it spends about 98% of its time. It is interesting is that, aside from taking a substantial amount of execution time (17.6–29.3%), two functions called from CME_DG, *construct_momentum_interface_dg* and *local_assembly_cdg_face*, occupy 2/3 of that time. This was important in deciding on optimisation strategy for the assembly code.

4.3 Optimisation work

4.3.1 Rationale

The goal of *construct_momentum_dg* is to assembly a global momentum matrix for solving, which it does by looping over each element in the local submesh, creating a local matrix which is then added the global matrix as it progresses. Work in a previous dCSE project gave some speedup through splitting the loop across OpenMP threads (Guo, et al. 2012), however there were strong suggestions that further improvements could be made.

The basic outline of execution in *construct_momentum_dg* is:

```
construct_momentum_element_dg:

Set relevant options from FLML tree
For each element e in submesh:
     call construct_momentum_element_dg:
          For each face f in element e:
               call construct_momentum_interface_dg:
                    call local_assembly_cdg_face
```

As previously mentioned, *construct_momentum_dg* spends approximately 98% of its time inside *construct_momentum_element_dg* (*CME_DG*), so it ought to be the main target for optimisation. Inside CME_DG, the code is quite complex: Fluidity is designed to support 1 to 3 dimensions, four different viscosity schemes (Bassi-Rebay, Interior Penalty, Arbitrary Upwind, and Compact Discontinuous Galerkin), along with many other options that affect element assembly. The original design of *construct_momentum_dg* had each of these options being parsed at run-time within *CME_DG*, for each element. Moreover, there are tens of smaller arrays that are defined and allocated/deallocated at run-time, for each element, and for each face of each element. Lastly, there were many finite-element utility functions such as *shape_shape_vector()*, *ele_val_at_quad()* and *face_val_at_quad()* from the files *FETools.F90* and *Fields_base.F90*, that consist of little more than matrix multiplication, but each of which allocates and deallocates small temporary work arrays upon being called and when being exited.

What is important to note here is that:

1) For each core, there are many hundreds of thousands of small allocs/deallocs per iteration for a reasonable-sized problem. This could easily results in millions memory allocations for each NUMA region on modern computer hardware.
2) The sizes of all of these small arrays do not change over the course of the simulation, as Fluidity only supports one type of element per mesh.
3) None of the element discretisation options change over the mesh for a given simulation.

From this, it was clear that compile-time parsing of these options could provide an answer. This would allow:

1) Compiling out of unnecessary option parsing using #ifdef pre-processor directives
2) Compile-time defining of common size parameters through #define directive, eg. number of dimensions and number of element nodes, which would allow static allocation of all of the small arrays
3) Compile-time vectorisation of small, tight loops (eg. looping over dimensions, nodes) by using #define.

Unfortunately, due to the extensive use of the *FETools.F90* and *Fields_base.F90* within Fluidity, compile-time optimisation of these was not possible.

4.3.2 Summary of completed tasks

The first task was the construction of *construct_momentum_elements_dg_opt*, a per-element local matrix assembly routine which relies on compile-time definitions for dimension, element quadrature, number of element nodes, etc. Through this, many further optimisations were undertaken to improve the performance over the original unoptimised subroutine, *construct_momentum_element_dg*. These were, namely:

- Conversion of all small dynamic array allocations to static allocations
- Optimisation of tight loops to use compile-time length definitions, to allow compile-time vectorization of the loops
- Inlining of calls to finite-element utility subroutines (in FETools.F90 and Fields_base.F90), eg. *ele_val_at_quad*, *face_val_at_quad*.
- Inlining of code element face assembly subroutines, such as *construct_momentum_interface_dg*. All code was moved inside *construct_momentum_elements_dg_opt* and all dependent arrays declared statically there.
- Rearrangement of decision logic to minimise expensive recalculation of array values.
- Addition of logic in *construct_momentum_elements_dg* to only call optimised code if run-time element configuration for dimension, quadrature, etc. matches compile-time configuration; otherwise, it runs non-optimised original *construct_momentum_element_dg* code.

The second task was the creation of command-line tools to parse the simulation options file (.flml) for discretisation options, and generate an include file *compile_opt_defs.h*, which contained compile-time definitions of element dimension, no. faces, no. element nodes, etc. This would then be used to compile the optimised Fluidity routines. This consisted of two parts:

- The creation of a simple script called *tools/generate_optimised_defines.sh*, which requires no special tools or libraries to parse XML.
- Altering *configure.in*, *configure* to accept *–enable-cto=FLML_file* option to call above script for the simulation options file *FLML_file*.

Lastly, we created the idealised channel test case for P1DG-P2 formulation, and set up the soft-scaling tests on ARCHER to investigate the performance benefits of these optimisations.

4.4 Optimised results

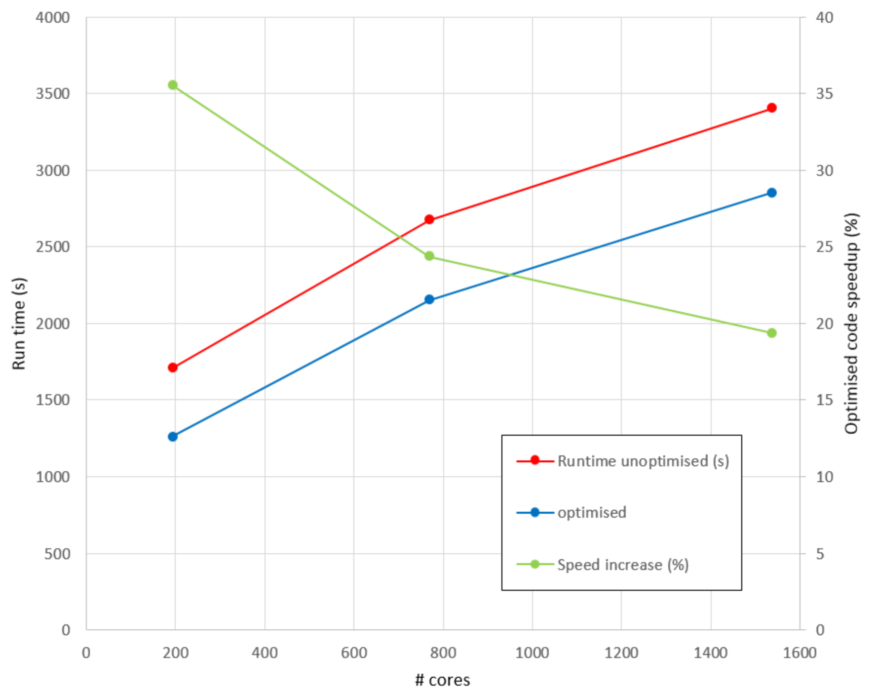


Figure 5. Optimised versus unoptimised run times, and speed up. Note that, even at the highest core counts, just optimising the DG assembly code has resulted in a 19-24% performance increase overall. This would be higher in longer simulation runs.

Figure 5 shows the overall run times and speed up for the whole simulation, using the assembly-optimised code. What is clear that, even though only the DG assembly subroutines have optimised, there has been a substantial overall improvement in performance. From 192 cores (8 compute nodes), we see a 36% speed up; when this is scaled up to 1536 cores (64 nodes), we still see a 19% speed up. This perhaps can be expected, as DG assembly does not rely on any MPI communication, which can be expected to dominate the rest of the program at very high core counts. It is also clear that the scalings of unoptimised and optimised versions follow a similar progression.

More detail can be seen in Table 4, which shows the % run times of the most heavily used subroutines in the optimised code. As before, all PETSc calls and direct wrappers have been removed. The subroutine for *construct_momentum_interface_dg* was manually inlined, and so it no longer appears in the profiler log; *local_assembly_cdg_face* is now so quick it barely registers in the profiler timings. As expected, *construct_momentum_dg* now occupies substantially less of the execution time: for 192 cores, it is 2.69x less of overall execution time; at 1536 cores, it is 3.03x less.

	% runtime (without spin-up)		
# cores	192	*768*	*1536*
Function name			
solve_momentum	89.8	92.7	99.3
correct_pressure	54.0	64.6	69.7
advance_velocity	13.7	11.6	10.6
subcycle_momentum_dg	11.3	9.9	8.9
construct_momentum_dg	10.9	8.0	5.8
assemble_cmc_dg	8.9	6.8	5.2
construct_momentum_element_dg	8.4	6.4	5.0
construct_momentum_interface_dg	-	-	-
local_assembly_cdg_face	-	-	-
calculate_courant_number	3.7	3.0	2.5

Table 4. % run times of optimised code. The blank indicate code that no longer exists or registers in the profiler output.

As Allinea MAP does not conveniently give the absolute run times of individual routines, to give better idea of the impact on run times of the DG assembly optimisation on each of the major subroutines, Figure 4 was repeated but with the optimised % run times scaled by the relative change in the runtimes, ie. with a scaling factor $s = R_{opt}(N)/R_{unopt}(N)$ where $R_{xxx}(N)$ is the overall run time for N cores for case *xxx* (ie. *opt* or *unopt*). This would give a scaled % run time of

$$R^*(N) = s(N)R(N)$$

The scaled % run times of the optimised code can be thought of as the run time relative to that of the unoptimised code.

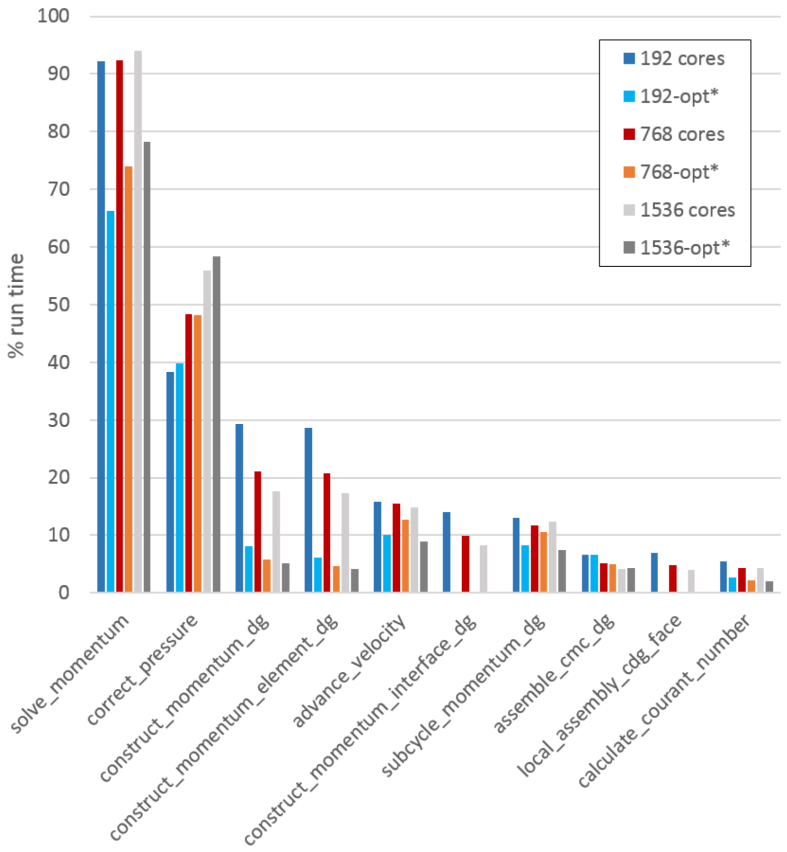


Figure 6. Comparison of % run times for unoptimised code, versus optimised code scaled by relative run time for each case. The asterisk (*) indicates a scaled run time.

The graph of the scaled results are shown in Figure 6. As expected, DG assembly has a much shorter absolute run time: at 96 cores, there is a 3.64x absolute speed up; at 1536 cores, the absolute speed up is 3.39x. This compares well with the 2-3x target speed up in the original proposal (Creech and Jackson 2015). Some subroutines independent of the assembly code (eg. *correct_pressure* and *assembly_cmc_dg*) have almost the same execution time before and after the DG optimisation, but others (*advance_velocity*) show a substantial improvement. Speculatively, it could be said that the static allocation of the small arrays in the assembly routines has a relatively benign effect on memory allocation in other cores in the same NUMA region, but this would require further investigation.

5 Tidal modelling optimisations

5.1 Test configuration

A realistic test case was chosen, to be as close as possible as to the type of simulation the code has been optimized for: ie. a tidal coastal model. This was developed from a pre-existing prototype of the Sound of Islay, spanning 21.4 km east-west and 36.6km north-south as shown in Figure 7.

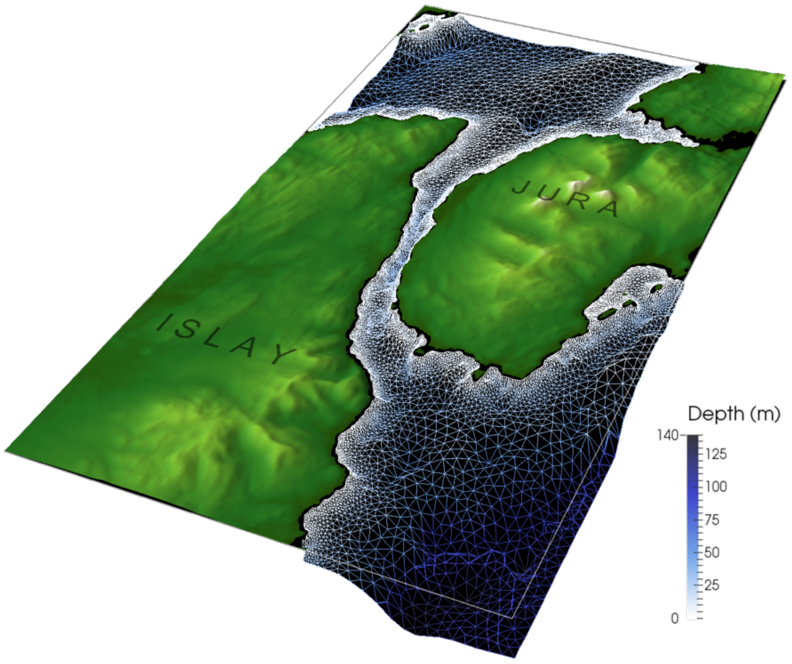


Figure 7. Sound of Islay overview

The mesh in this simulation, as with all tidal simulations within Fluidity, is a semi-structured 3D mesh. This means that an initial 2D mesh which matches the coastlines and open boundaries is extruded downwards to a depth which matches the specified bathymetry. The mesh is still tetrahedral, but the tetrahedral are arranged in vertical columns.

The simulation was set to run for 100 timesteps; there were two non-linear iterations for each timestep, which meant the DG assembly code would be called 200 times. To track the improvements that each section of optimisation work made to performance, the model was benchmarked at four stages of development: i) the code with no optimisations, ii) with DG assembly optimisation, iii) using the 2D (pre-extruded) mesh reordering tool, and iv) load balancing tweaks for extruded meshes. The technical details of the last two are in Section 5.2.

The mesh configuration used for all tests is shown in Table 5.

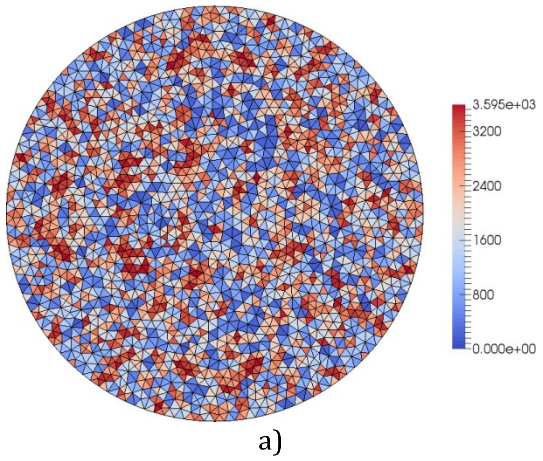
MPI tasks	Cores (2xOpenMP)	Nodes	Min-Max. Δx (m)	Elements	Elements / MPI task
384	768	32	8-80	3 784 390	9855

Table 5. Realistic test case configuration for profiling on ARCHER.

5.2 *Tidal model performance optimisations*

5.2.1 Mesh reordering

The first performance improvement was made by tackling the relatively unordered nature of the underlying 2D mesh that Fluidity extrudes into a 3D mesh, using a solution derived from code developed by Dr. J. Maddison in the School of Mathematics. Typically, the finite element assembly code performs calculations on an element-by-element basis. Each calculation uses data for the element considered, and typically also uses data for neighbouring elements. However the meshes used by Fluidity (as created by mesh generators prior to execution) typically do not order mesh data in an optimal way for such calculations. For example, Figure 8a shows a mesh generated by Gmsh. The colour scale here indicates the element number. In Fluidity calculations such as finite element assembly, the elements are processed in order of their element number and, as neighbouring elements typically have very different element numbers in this mesh, which could lead to a loss of performance. By comparison, Figure 8b shows the same mesh with elements reordered using the Gibbs-Poole–Stockmeyer algorithm implemented in SCOTCH (using code derived from DOLFIN 1.5.0). The ordering in this latter case means that elements that are nearby (in the sense of the distance as defined by the element-element graph) are more likely to have smaller differences in element number.



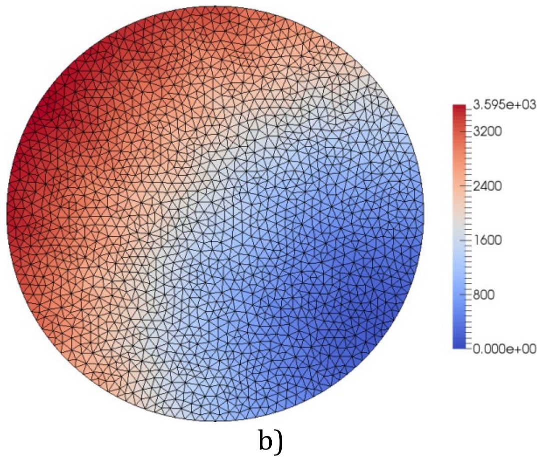


Figure 8. a) A mesh generated using Gmsh. The elements are coloured according to their element number. b) The same mesh, with elements reordered using the Gibbs-Poole–Stockmeyer algorithm.

5.2.2 Load balancing improvements

The second performance improvement was made by improving the load balancing algorithm for extruded meshes. In the unoptimised code, Fluidity would decompose extruded meshes with considerable imbalance in their relative size, even after running Fluidity's redecomposition tool, *flredecomp*, on an extruded mesh. Changes were made to Fluidity's interface to the Zoltan redecomposition and load-balancing routines, in particular *zoltan_cb_get_owned_nodes()* in *Zoltan_callbacks.F90*, so that weighting of each individual surface (pre-extruded mesh) node is weighted by the number of nodes below it, and that this is contributes towards a global weighting for each mesh partition. This was arguably more of a bug-fix that a strategic performance development, albeit one that took considerable time to diagnose and correct.

This improvement now meant that Fluidity more effectively balanced mesh sizes across subdomains, even when there are spatially variable numbers of element layers. Subdomain mesh sizes from the tidal test case before and after this improvement are shown in Table 6.

Case	**Min.**	**Max.**	**Imbalance**	**Mean**	**Standard dev.**
Pre-optimisation	10088	64538	6.398	16058.979	7727.234
Load balancing	10930	25365	2.321	16369.620	2803.807

Table 6. Statistics on elements for partitioned subdomains before and after load balancing enhancements.

It can be seen that before optimisation, the largest subdomain contained almost 6.4x the elements of the smallest subdomain. This severe load imbalance resulted in smaller partitions waiting on blocked MPI communications (ie. mpi_allreduce) until the larger partitions finish. The code with the enhanced load balancing brought the imbalance down to 2.3x – approaching 1/3 of the pre-optimised imbalance – so significant speedups were expected. The test case performance results bear this out, and these are detailed in Section 5.3.

5.3 Results

Table 7 and Figure 9 show the effect of each successive optimisation on the performance of Fluidity on the tidal case. Interestingly, the DG assembly optimisation has an even more dramatic effect on code performance than the unstructured 3D case – 1.24x versus 1.74x for tidal. Running the 2D mesh reordering increases the performance over the pre-optimised code to 1.95x, a further 11% increase, which is impressive considering only the 2D mesh is being reordered, and no internal Fluidity code was rewritten.

Case (n768)	**Run time (min)**	**Speed-up**
Pre-optimisation	80	-
+DG assembly optimisation	46	1.74 x
+2D mesh reordering	41	1.95 x
+Load balancing optimisation	25	3.20 x

Table 7. Values for speed-ups from successive optimisations for tidal modelling case with 768 cores.

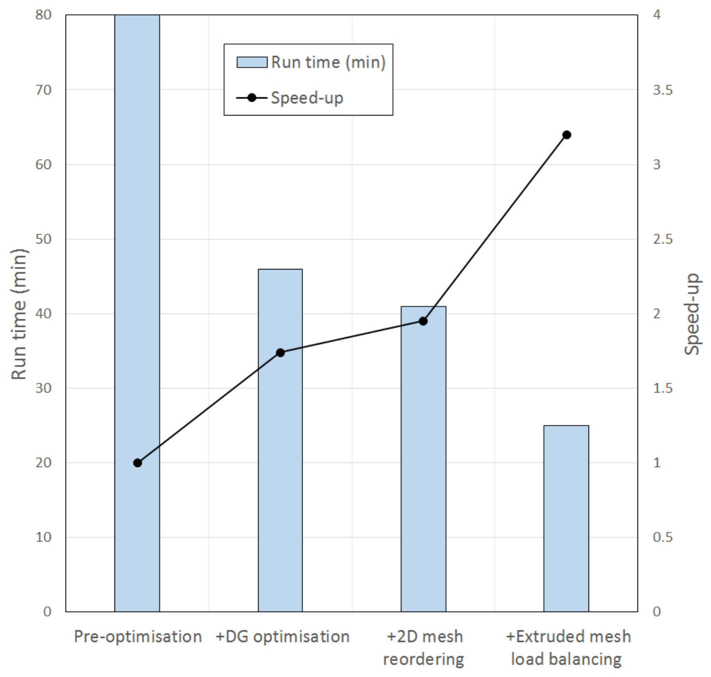


Figure 9. Graph of speed-ups for tidal case with 768 cores.

Finally, as expected, addressing the load balancing issues gives an additional 64% speedup, results in the tidal test case running 3.2x faster with the optimised version of Fluidity, than it did with the original, pre-optimised code. This is significant, as this changes the level of physical modelling that can be achieved with Fluidity, putting it clearly in the 'new science' category.

6 Further optimisations

Outside of the assembly code, several other optimisations were made to the Fluidity code.

6.1 Advection subcycling

Fluidity implements advection subcycling for stability at high Reynolds numbers, and for adaptive timestepping under DG velocity elements. It does this by splitting the global simulation timestep into smaller timesteps to satisfy a specified CFL number for the sub-timesteps (Zheng, et al. 2015). However, Fluidity's original implementation of subcycling in the routine *subcycle_momentum_dg* inside *Momentum_DG.F90* used a vertex-base slope limiter, *limit_vb* inside *Slope_limiters.F90*, which treats each of the velocity field dimensions as a scalar, and applies the limiter to them separately. This meant that for each dimension, expensive memory allocation/deallocation and value initialisation took place, ie. from the top of *limit_vb*:

```
! Allocate copy of field
call allocate(T_limit, T%mesh,trim(T%name)//"Limited")
call set(T_limit, T)

! returns linear version of T%mesh (if T%mesh is periodic, so is vertex_mesh)
call find_linear_parent_mesh(state, T%mesh, vertex_mesh)

call allocate(T_max, vertex_mesh, trim(T%name)//"LimitMax")
call allocate(T_min, vertex_mesh, trim(T%name)//"LimitMin")

call set(T_max, -huge(0.0))
call set(T_min, huge(0.0))
```

This meant that subcycling could be quite expensive: in tests using the channel case from Section 4.1 and the unoptimised code, over 100 timesteps it occupied 11.3% of execution time. This was completely rewritten as a new routine *limit_vb_opt* within *Momentum_DG.F90* so that limiting the vector field (always velocity in this case) so was done by limiting it as a vector rather than individual scalar components: this meant that expensive initialisation was only done once per field; secondly, compile-time optimisations (CTO) were used so that small, tight loops, such as those over local element nodes or dimensions were known at compile time, and so the compiler could take advantage of vectorisation. Eg. The calculation of the alpha variable in the slope limiter, per element:

```
!loop over nodes, adjust alpha
do node = 1, opNloc
      do concurrent (i=1:opDim)
            !check whether to use max or min, and avoid floating point algebra
      errors due to round-off and underflow

            if(T_val(i,node)>Tbar(i)*(1.0+sign(1.0e-12,Tbar(i))) .and.
      T_val_minus_bar(i,node) > tiny(0.0)*1e10) then
                  alpha(i) = min(alpha(i),(T_val_max(i,node)-
            Tbar(i))/T_val_minus_bar(i,node) )
            else if(T_val(i,node)<Tbar(i)*(1.0-sign(1.0e-12,Tbar(i))) .and.
      T_val_minus_bar(i,node)  < -tiny(0.0)*1e10) then
                  alpha(i) = min(alpha(i),(T_val_min(i,node)-
            Tbar(i))/T_val_minus_bar(i,node) )
            end if
      end do
      . . .
      . . .
end do
```

Test case	% run time	Subcycling speed-up	Program speed-up
Pre-optimised	11.3	-	-
After optimisation	8.0	1.412	1.065

Table 8. Performance figures for unoptimised and optimised DG subcycling, for channel test case (1 compute node).

Note that, should dimensions not match those used in the CTO, the subcycling code defaults to using *limit_vb* as before. The performance improvement can be seen here in Table 8 using a single compue node version of the DG assembly channel flow test case detailed in Section 4.1. There was a 6.5% increase in overall performance of the code, which when coupled with the assembly optimisations, is significant, as subcycling adds stability when solving for high Reynolds number, turbulent flows.

7 Feature additions

Several sets of features were added to Fluidity to better facilitate general fluid dynamics modelling, with some emphasis on tidal modelling. These will be described in detail in the updated Fluidity manual. Briefly, these are:

Default options generation. Configuring Fluidity through the Diamond GUI can be a lengthy, complex process, as there are no default options and few default behaviours. This is particularly the case for the P1DG-P2 velocity-pressure element pair. A new set of preprocessing routines were added in *DG_prep.F90*, which parse the FLML options tree prior to simulation initialisation and all other parsing routines. These include setting up the projection of the discontinuous velocity to continuous velocity field, which is then written to the parallel VTU results files. The CG velocity can be used for a variety of purposes (see DG LES paper below), however one of its main benefits is that if it is written out to the results files, and all DG fields output is suppressed, then file size drops by an order of magnitude. Other options include: setting out a DG Courant number field for adaptive time-stepping with the output suppressed, and DG LES defaults, which will be detailed below. Note, all default behaviours can be overridden: if the automatic fields already exist, then they will not be overwritten.

Discontinuous Galerkin Large Eddy Simulation (DG LES) was implemented, from scratch. In particular, this was targetted for the Compact Discontinuous Galerkin (CDG) formulation. A partial-stress implementation of CDG was developed, as was a unique, never before used LES turbulence model, in which CG velocity and eddy viscosity fields were used rather than DG, before being applied to the momentum equation. This mixed-mode implementation gave a much more efficient and simple formulation than would have been possible by solely using DG spaces. This approach was validated using the well-known backward-facing step problem, and gave excellent agreement with publish experimental results. This hybrid DG LES technique has now been submitted as paper entitled ‘Efficient Large Eddy Simulation for the Discontinuous Galerkin Method’ to the journal *Computers & Fluids* (Creech, Jackson and Maddison, et al. submitted). A preprint is available on *arxiv.org* for reference.

Tidal boundary conditions have been implemented for Fluidity using Python. Strictly speaking, Fluidity already supports tidal forcing on open boundaries through the use of a NetCDF file containing a longitudinal/latitudinal map for each tidal constituent’s amplitude and phase. However, this is extremely hard to work with, since the format of the NetCDF is not specified in the manual, and moreover, it does not support the ‘ramping up’ of tidal constituent amplitudes from 0 over a specified period of time. Ramping up boundary conditions is often necessary at the start of a tidal simulation, as applying an elevated free-surface height at the boundaries of a tidal domain, where the interior free surface is initially at rest, can cause instabilities in the free surface moving mesh, if one is being used, or pressure wave reflection off coastlines. Instead, a Python utility called *tidal_bcs.py* was developed, which reads in tidal constituent data from a CSV file and then derives into the required hydrostatic pressure boundary conditions. This also supports the necessary ramping time condition, which experimental has been shown to be around 48 hours. This forms part of a growing set of Python utilities called *Tidetools* which are designed to assist in coastal modelling in Fluidity.

8 Summary

The main goal of this eCSE project was increase the performance on ARCHER of Fluidity for high Reynolds numbers, turbulent flow simulations, including tidal models. This goal achieved: general CFD problems achieved approximately a 1.2-1.4x speed-up on ARCHER, even for +1500 core counts. This was primarily achieved by optimising the DG assembly code, which itself runs 3.4-3.6x faster than the original subroutines. A new, unique LES turbulence model was developed, again with efficiency in mind. This was successfully validated against published experimental data, and has been submitted as a journal paper.

Further optimisations were made specifically for tidal models, achieving a 3.2x speedup for coastal simulations over the original version of Fluidity. This changes the level of scientific investigation possibly with Fluidity – putting it in the 'new science' bracket. The changes and additions were made to make Fluidity easier to use with for general CFD simulations, as well as for coastal modelling.

All the changes to Fluidity done in this project have been provided back to the code developers at Imperial, and we are working with them on integrating them back into the main Fluidity releases. At the same time, we will separately document and release as open source the modifications to GitHub in the coming months, so that the fluid dynamics modelling community may immediately benefit from this optimised version of Fluidity.

Appendix

List of modules used

```
  1) modules/3.2.10.2
  2) eswrap/1.3.3-1.020200.1278.0
  3) switch/1.0-1.0502.57058.1.58.ari
  4) craype-network-aries
  5) craype/2.4.2
  6) pbs/12.2.401.141761
  7) craype-ivybridge
  8) cray-mpich/7.2.6
  9) packages-archer
 10) bolt/0.6
 11) nano/2.2.6
 12) leave_time/1.0.0
 13) quickstart/1.0
 14) ack/2.14
 15) xalt/0.6.0
 16) epcc-tools/6.0
 17) gcc/4.9.3
 18) cray-libsci/13.2.0
 19) udreg/2.3.2-1.0502.9889.2.20.ari
 20) ugni/6.0-1.0502.10245.9.9.ari
 21) pmi/5.0.7-1.0000.10678.155.25.ari
 22) dmapp/7.0.1-1.0502.10246.8.47.ari
 23) gni-headers/4.0-1.0502.10317.9.2.ari
 24) xpmem/0.1-2.0502.57015.1.15.ari
 25) dvs/2.5_0.9.0-1.0502.1958.2.55.ari
 26) alps/5.2.3-2.0502.9295.14.14.ari
 27) rca/1.0.0-2.0502.57212.2.56.ari
 28) atp/1.8.3
```

29) PrgEnv-gnu/5.2.56
30) perftools-base/6.3.0
31) perftools
32) cray-tpsl/1.5.2
33) cray-netcdf/4.3.3.1
34) cray-hdf5/1.8.14
35) cmake/2.8.12
36) zlib/1.2.8
37) vtk/5.10.1
38) zoltan/3.8
39) python-compute/2.7.6
40) pc-numpy/1.9.2-libsci
41) pc-scipy/0.15.1-libsci
42) fluidity-gcc

References

Babuška, I. "The finite element method with lagrangian multipliers." *Numerische Mathematik* 20 (1973): 179-192.

Brezzi, F. "On the existence, uniqueness and approximation of saddle-point problems arising from Lagrangian multipliers." *Revue Française d'Automatique, Informatique et Recherche Opérationelle* 8 (1974): 129-151.

Creech, A., A. Jackson, J. Maddison, J. Percival, and T. Bruce. "Efficient Large Eddy Simulation for the Discontinuous Galerkin Method." *Computers & Fluids*, submitted.

Creech, A., and A. Jackson. *eCSE05-7: Optimisation of Large Eddy Simulation (LES) turbulence modelling within Fluidity.* Project proposal, eCSE, 2015.

Deardroff, J. "A numerical study of three-dimensional turbulent channel flow at large Reynolds numbers." *Journal of Fluid Mechanics*, 1970: 453-480.

Guo, X., G Gorman, M. Lange, A. Sunderland, and M. Ashworth. *Developing Hybrid OpenMP/MPI Parallelism for Fluidity-ICOM - Next Generation Geophysical Fluid Modelling Technology.* Technical report, dCSE, 2012.

Ladyzhenskaya, O.A. *The mathematical theory of viscous incompressible flow.* 2. Gordon and Breach Science Publishers, 1969.

Pain, C., et al. "Three-dimensional unstructured mesh ocean modelling." *Ocean Modelling*, 2005: 5-33.

Patankar, S.V. *Numerical Heat Transfer and Fluid Flow.* CRC Press, 1980.

Peraire, J., and P.-O. Persson. "The compact Discontinuous Galerkin method for elliptic problems." *SIAM J. Sci. Comput.*, 2008: 1806-1824.

Roman, F., G. Stipcich, R. Armenio, and S. Corsini. "Large eddy simulation of mixing in coastal areas." *Int. J. of Heat and Fluid Flow*, 2010: 327–341.

Zheng, J., J. Zhu, Z. Wang, F. Fang, C.C. Pain, and J. Xiang. "Towards a new multiscale air quality transport model using fully unstructured anisotropic adaptive mesh technology of Fluidity (version 4.1.9)." *Geosci. Model Dev.*, 2015: 3421-3440.