High Performance Multi-Physics Simulations with FEniCS/DOLFIN

eCSE03-10 Report

Chris N. Richardson[*] Garth N. Wells[†]

27th May 2016

Abstract

This report presents new software developments implemented in the FEniCS/DOLFIN libraries to support efficient block preconditioning for linear systems, and the assembly of discontinuous Galerkin methods in parallel. For block preconditioning, support for the PETSc `MatNest` framework has been implemented. This provides memory savings of a factor of more than three over the previous implementation, as well as reduced run times. For discontinuous Galerkin methods, the DOLFIN distributed mesh data structure has been extended to allow ghost/halo layers of cells. These layers are necessary to compute, on process boundaries, the numerical fluxes that characterise discontinuous Galerkin methods. During the development of the above functionality, a number of other performance improvements (in memory and run time) have been made, and some new platform-independent profiling tools have been added to DOLFIN. These are detailed in this report. Performance data for the newly developed functionality on ARCHER are reported, and show very good scaling over a large range of process/core counts.

The primary objectives of this project have been achieved, and the developed code is publicly available in the FEniCS Project development repositories. The functionality will be included in future releases of the FEniCS Project libraries. Developments from this project have already been used in EPSRC and NERC funded projects, as well as being made available through third-party tools that build on FEniCS.

[*]BP Institute, University of Cambridge (`chris@bpi.cam.ac.uk`)
[†]Department of Engineering, University of Cambridge (`gnw20@cam.ac.uk`)

Contents

1 Introduction

This project developed enhancements to the open source FEniCS/DOLFIN [3] library to extend its functionality for multi-physics/multi-field simulations on parallel computers. The main topics addressed in the project were: (a) memory efficient block preconditioning of linear systems via PETSc; and (b) support for overlapping domain and discontinuous Galerkin methods in parallel. This report summarises the new developments and provides performance data on ARCHER.

1.1 Background

The FEniCS Project allows users to express partial differential equations (PDEs) in a concise mathematical form, which is then compiled into generated code for finite element analysis by a 'form compiler'. This has the advantage of freeing users from writing low-level code and enables them to quickly develop complex models. Special optimisations can be automatically applied in the code generation phase.

As model sizes increase, users inevitably need to run on larger machines, and ultimately large distributed memory High Performance Computing (HPC) systems, such as ARCHER. The FEniCS problem solving environment, DOLFIN, has for some years supported distributed memory parallel simulation. However, some important functionalities that have recently been in high demand from users were not supported (efficiently) in parallel, namely discontinuous Galerkin methods and block preconditioning.

1.2 Scope

In this report, we first consider multi-field problems, such as Stokes' equations, which yield matrices with a block structure. Often the blocks in such equations may be duplicated, or empty. When solving with iterative solvers (almost always the case for large problems in parallel), the preconditioner may also share elements with the operator matrix, and memory can be saved by sharing matrix blocks. The second main topic addresses discontinuous Galerkin (DG) methods. DG methods involve integration over cell facets, and require data from cells on either side of a facet. In parallel, this requires a layer of ghost cells on process boundaries. Support for this has been added, and it is now possible to use DG methods in parallel. During the course of the code development, numerous performance improvements were made based on performance profiling. Performance of the new functionality is supported by performance data on ARCHER.

1.3 Objectives

The primary objectives of the project were to:

1. Provide support in DOLFIN for the efficient application of block preconditioners via PETSc.
2. Provide support in DOLFIN for the assembly of discontinuous Galerkin matrix operators and vectors by extending the DOLFIN distributed mesh with a ghost/halo layer.
3. Apply generic optimisations as as result of the profiling performed in this work.

The main objectives of this work are the provision of new functionality rather than optimising existing code. Hence, the performance testing focuses on parallel scaling rather than comparison with old implementations.

1.4 Libraries

The FEniCS libraries (`http://fenicsproject.org`) are available under open source licenses and are developed publicly at `https://bitbucket.org/fenics-project/`. This work made extensive use of PETSc [4, 5], which is hosted at `https://bitbucket.org/petsc/petsc`. We use the development version of PETSc in all tests.

2 Parallel block preconditioning (WP1)

We summarise in this section the block preconditioning support that has been developed. To demonstrate the approach, we use Stokes' equations as a canonical example. As equation-specific code is generated in FEniCS, users can easily apply the developed code to their own multi-field equations. The matrix block structure supports $m \times m$ blocks.

The Stokes' equations read:

$$-\nabla^2 u + \nabla p = f, \tag{1}$$
$$\nabla \cdot u = 0, \tag{2}$$

where u is the velocity field, p is the pressure field and f is the prescribed body force. Translating the equations into a weak form and using a finite element basis leads to a linear system of algebraic equations, which may be represented as a block matrix:

$$\underbrace{\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & 0 \end{bmatrix}}_{A} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}. \tag{3}$$

For a problem with n degrees-of-freedom, the matrix A will have dimensions $n \times n$.

For large problems, it is necessary to use preconditioned iterative methods to solve the above system. For Stokes' equations, a simple yet optimal preconditioner P has a block diagonal structure,

$$P = \begin{bmatrix} A_{00} & 0 \\ 0 & M \end{bmatrix}, \tag{4}$$

where M is the pressure mass matrix. The iterative solver then solves the modified problem

$$\begin{bmatrix} A_{00} & 0 \\ 0 & M \end{bmatrix}^{-1} \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} A_{00} & 0 \\ 0 & M \end{bmatrix}^{-1} \begin{bmatrix} f \\ 0 \end{bmatrix}. \tag{5}$$

A multigrid method applied to each block of P to approximate P^{-1} is known to be optimal, which in turn leads to a method with (close to) $O(n)$ cost, where n is the number of degrees-of-freedom.

For efficient computation and memory use, rather than assembling the entire problem 'monolithically' as a single matrix A which then subsequently needs to be split into blocks to apply a block solver, it is preferable to assemble the sub-matrices A_{00}, A_{01} and A_{10} directly to avoid the copying. Moreover, the operator and the preconditioner can share matrix blocks (in the case of Stokes, the A_{00} block) and the structure of each block is simpler, which allows optimisations such as local degree-of-freedom (DOF) blocking. Using the PETSc `MatNest` interface, it is possible to create a 'nested matrix' which effectively holds pointers to each sub-block. The PETSc `MatNest` type matrix behaves in most ways as a regular matrix, but the underlying storage is a collection of sparse matrices.

2.1 Interface

We show here some code extracts for creating nested matrices from finite element forms. Complete examples are available in the code repository.

We have generalised the DOLFIN `dolfin::SystemAssembler` class to support the creation of nested matrices. It can now assemble a collection of forms, together with boundary conditions, into a corresponding set of matrices and right-hand side vectors. When Dirichlet boundary conditions are applied (algebraically), for a given boundary condition both the diagonal and off-diagonal block(s) must be be modified.

Previously in DOLFIN, in a mixed formulation (not blocked) for Stokes' equations we would have written (in the C++ interface):

```
auto a = std::make_shared<Stokes::Form_a>(W, W);
auto L = std::make_shared<Stokes::Form_L>(W);
dolfin::SystemAssembler assembler(a, L, bcs);

auto A = std::make_shared<dolfin::PETScMatrix>();
auto b = std::make_shared<dolfin::PETScVector>();
assembler.assemble(*A, *b);
```

to create the global matrix A and the vector b. In the low-level version of the new interface, it becomes:

```
// Create forms
auto a00 = std::make_shared<Stokes::Form_a00>(V, V);
auto a01 = std::make_shared<Stokes::Form_a01>(Q, V);
auto a10 = std::make_shared<Stokes::Form_a10>(V, Q);

auto L0 = std::make_shared<Stokes::Form_L0>(V);
auto L1 = std::make_shared<Stokes::Form_L1>(Q);

// Assemble all blocks
dolfin::SystemAssembler assemblerA0({a00, a01, a10, NULL}, {L0, L1},
                                    {{bc}, {}});

auto b0 = std::make_shared<dolfin::PETScVector>();
auto b1 = std::make_shared<dolfin::PETScVector>();
auto A00 = std::make_shared<dolfin::PETScMatrix>();
auto A01 = std::make_shared<dolfin::PETScMatrix>();
auto A10 = std::make_shared<dolfin::PETScMatrix>();
assemblerA0.assemble({A00, A01, A10, NULL}, {b0, b1});
```

From the blocks, a nested matrix can be created and used in a PETSc solver. An analogous interface to the `SystemAssembler` is also available in Python.

At first, the process appears more complex, but it is simply a deconstructed version of the previous code. The `FunctionSpace W` is now separated into `V` and `Q`, and the mathematical forms are expressed separately. This mirrors standard mathematical notation. Previously, we might have written the form description (in UFL [2] syntax) for '`a`' as follows:

```
a = inner(grad(u), grad(v))*dx - div(v)*p*dx - div(u)*q*dx
```

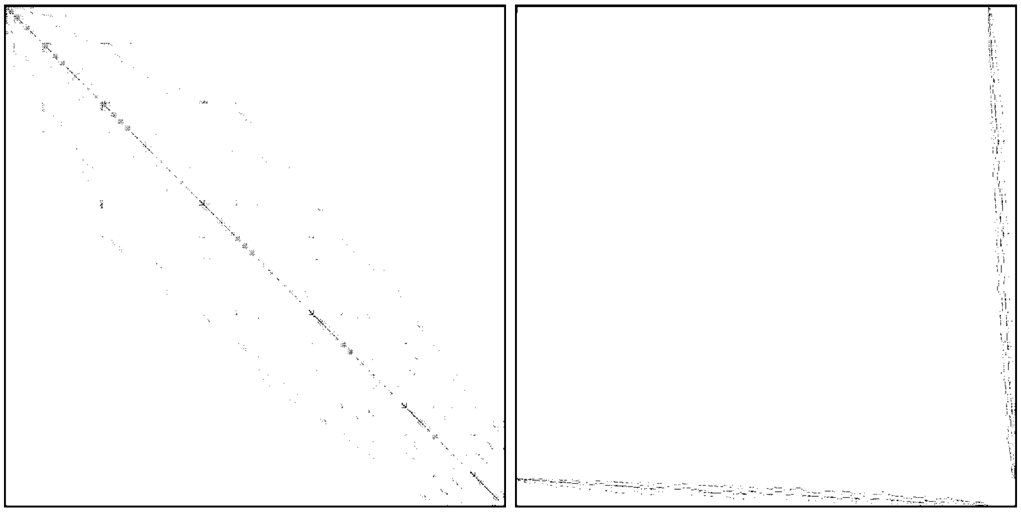


Figure 1: Matrix sparsity pattern for a small 3D Stokes problem with (a) monolithic/non-blocked construction and (b) with nested matrix structure.

but that now becomes:

```
a00 = inner(grad(u), grad(v))*dx
a01 = -div(v)*p*dx
a10 = -div(u)*q*dx
```

The low-level interface is not a major challenge for users and offers full control. In addition, we have added a new function `block_split` operator to the form language UFL to automatically 'split' variational forms, built on mixed function spaces, into components. Each component of the split form corresponds to a block in the matrix operator. We have implemented the necessary tools for the high-level interface and are receiving user feedback before finalising for a future FEniCS release.

2.2 Matrix structure

The monolithic approach for the Stokes problem with Taylor–Hood elements in three dimensions leads to the a matrix sparsity pattern shown in Figure 1(a). The graph re-ordering that DOLFIN applies interlaces the velocity and pressure degrees-of-freedom, and attempts to cluster the non-zeroes close to the diagonal. Since the velocity and pressure degrees-of-freedom are not collocated, it is not possible to exploit the node-wise block structure of the degrees-of-freedom (three velocity components at each 'node'), which means that memory locality at each node cannot be exploited in matrix-vector products.

The sparsity structure for the new nested matrix implementation is shown in Figure 1(b). The very small empty block in the lower right-hand corner is the zero A_{11} block, which we no longer need to store with the nested format. While it cannot be easily perceived from the figure, the large A_{00} block now exploits local velocity degree-of-freedom blocking, with groups of three velocity degree-of-freedoms blocked together. This is possible as the velocity and pressure degrees-of-freedom are not interlaced. Moreover, the large A_{00} block is shared by the operator A and the preconditioner P, which leads to large memory savings.

2.3 Memory usage

A motivation for adding support for PETSc nested matrices (`MatNest`) was to reduce memory usage compared to the automatic copying of matrices into a nested format internally by PETSc. We had previously observed that the high memory usage of the PETSc 'automatic' splitting severely limited the utilisation of compute nodes.

Figure 2 shows the memory usage on one of the compute nodes on ARCHER, used during a simulation, against time for solving Stokes' equations using: (a) the PETSc automated splitting ('AutoFieldSplit'); and (b) the new interface to PETSc `MatNest`. The plots include all phases of the solution process, from initial mesh construction and partitioning, through to the solution of the linear system. It is clear from the plots that the peak memory usage for the `MatNest` implementation is reduced by a factor of approximately three compared to the auto-split case for 96 processes, and by considerably more for higher process counts. The dashed red line indicates the maximum available memory on a node. Due to the memory requirements exceeding the available memory on a node, we could not complete the larger simulations using the automatic PETSc splitting, whereas we could perform simulations using the new functionality. This clearly illustrates how the new interface allows machines like ARCHER to be more effectively and efficiently used.

The memory spike during the simulations for the auto-splitting implementation coincides with the construction of the solver and algebraic multigrid preconditioners, at which point matrix copies are being made. Moreover, since we cannot set the local degree-of-freedom block size, the multigrid construction becomes very costly in terms of memory. The nested matrix approach allows the local block size to be set on matrix blocks, which reduces the memory requirements substantially during construction of the preconditioner.

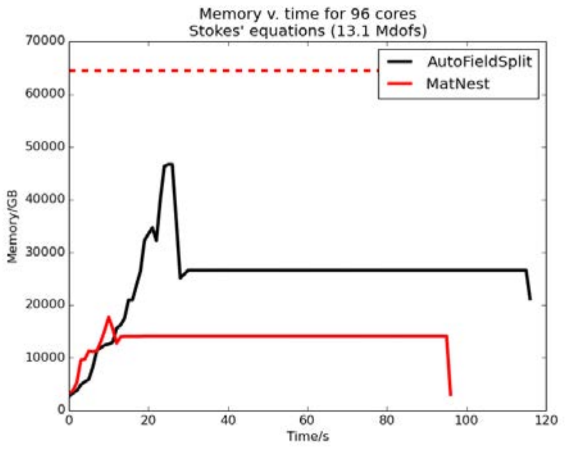
2.4 Scaling

Weak scaling results are summarised in Figure 3, using the PETSc GAMG (algebraic multigrid) preconditioner for the A_{00} block, and Hypre BoomerAMG (algebraic multigrid) for the M block. The matrix assembly phase scales well. The modest growth in the time for the preconditioned Krylov solver can be attributed in part to a modest increase (roughly 10%) in the iteration count over the 16 fold increase in problem size. We have put limited effort into tuning the multigrid preconditioners for this problem as it falls outside the scope of work. Some improvements may be possible through tuning of the preconditioners.

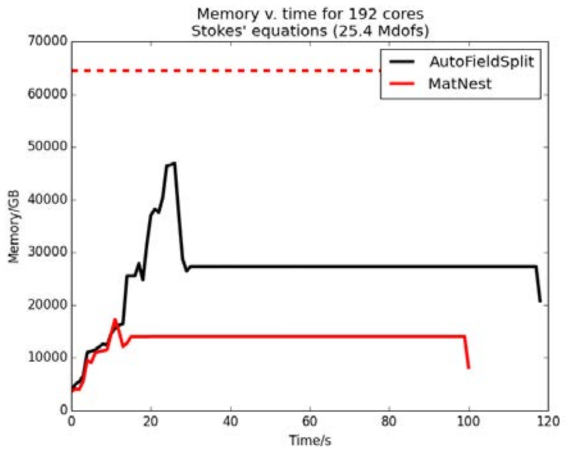
3 Discontinuous Galerkin Methods in parallel (WP2)

Discontinuous Galerkin (DG) methods are finite element-like methods which, in addition to the usual loop over mesh cells, also involve a loop over mesh facets. To compute facet contributions to the global matrix, data from the two adjoining cells/elements is required for each facet. DG methods are particularly appealing for transport equations as they allow the natural evaluation of fluxes across cell boundaries, with the choice of an appropriate flux expression being essential for stability and accuracy of a method.

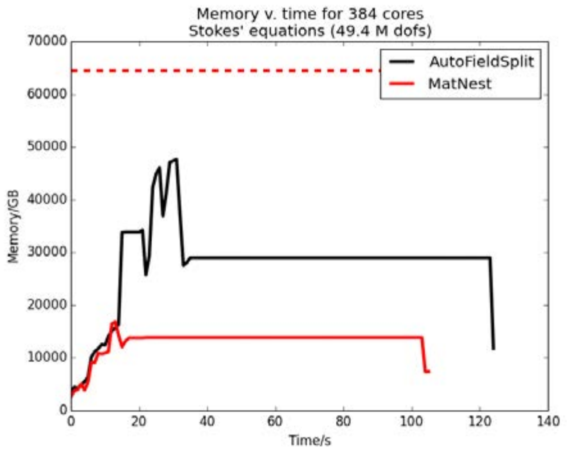
DOLFIN has supported DG methods in serial for some years. However, DG methods were not supported in parallel on distributed memory computers due to the absence of a mesh ghost layer at process boundaries. Without a mesh ghost layer, it was not possible to compute fluxes on facets that are shared by two processes. In this project, DOLFIN distributed meshes have been extended to support layers of ghost/halo cells. The parallel performance of the new developments is presented below.



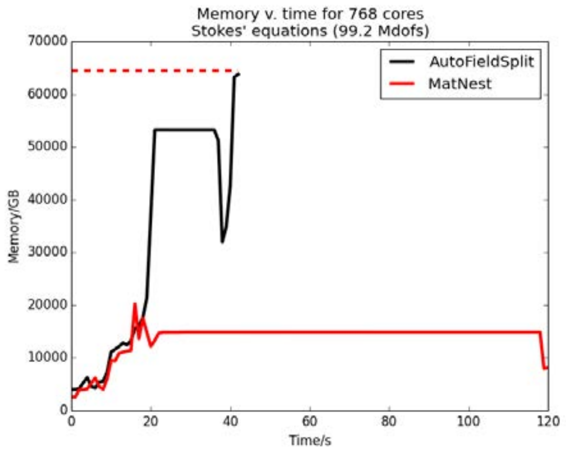
(a)



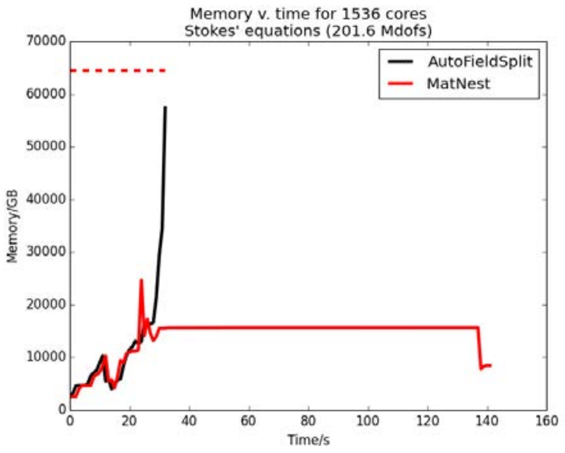
(b)



(c)



(d)



(e)

Figure 2: Memory usage for the auto-split and the new nested matrix implementation on ARCHER for the Stokes' equations with approximately 130k DOFS per process/core.

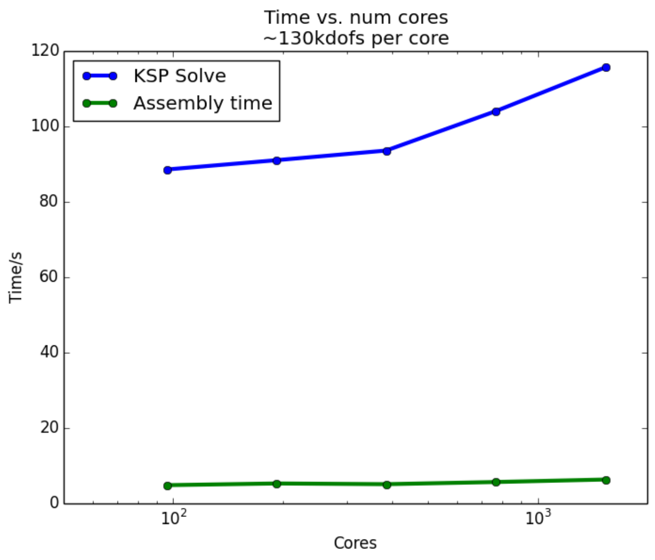


Figure 3: Weak scaling on ARCHER for a Taylor–Hood Stokes simulation in three dimensions with approximately 130k degrees-of-freedom per process.

3.1 Construction and partitioning of ghosted meshes

DOLFIN has been extended to permit distributed meshes to be constructed with different types of ghost/halo cell layers, such as cells connected by facet or cells connected by vertex (see Figure 4 for an illustration). In Figure 5 we compare the timing for construction of ghosted and non-ghosted meshes. The timing includes the graph partitioning step, which is performed by a third-party library and is a very difficult problem to make scale well. We see that the addition of ghost layers has minimal impact on the time required for mesh construction, compared to the 'non-ghosted' case. Users can specify the mode of mesh ghosting by setting a parameter, '`ghost_mode`' to '`shared_facet`' or '`shared_vertex`'.

To manage memory effectively, it was decided that it is best for the distributed layout of a mesh to be specified by the user at construction. The avoids the complexity of lazy automated construction of a ghosted mesh once the user equation has been analysed, and which can lead to memory management difficulties. Therefore we decided to require the user to specify the type of ghosts/halo at mesh construction. If a mesh does not have a suitably constructed ghost region for a given equation/method type, an informative error message is printed.

3.2 Assembly of forms on ghosted meshes

We now show some timings for assembling the matrix operator on a ghosted mesh for two DG methods. This was not previously possible with DOLFIN, hence we do not compare with a previous version of the library.

Figure 6 shows the assembly time for the Poisson operator using the interior penalty method with up to over 2.5 billion degrees-of-freedom. The assembly phase clearly scales well. Figure 7 shows the assembly time for an upwinded DG method for the scalar advection operator. Again, the assembly process scales well.

The inclusion of cell ghost layers opens the possibility of eliminating communication during matrix assembly in favour of redundant computation at process boundaries. We have chosen at this stage to not implement support for communication-less matrix assembly. Based on assembly

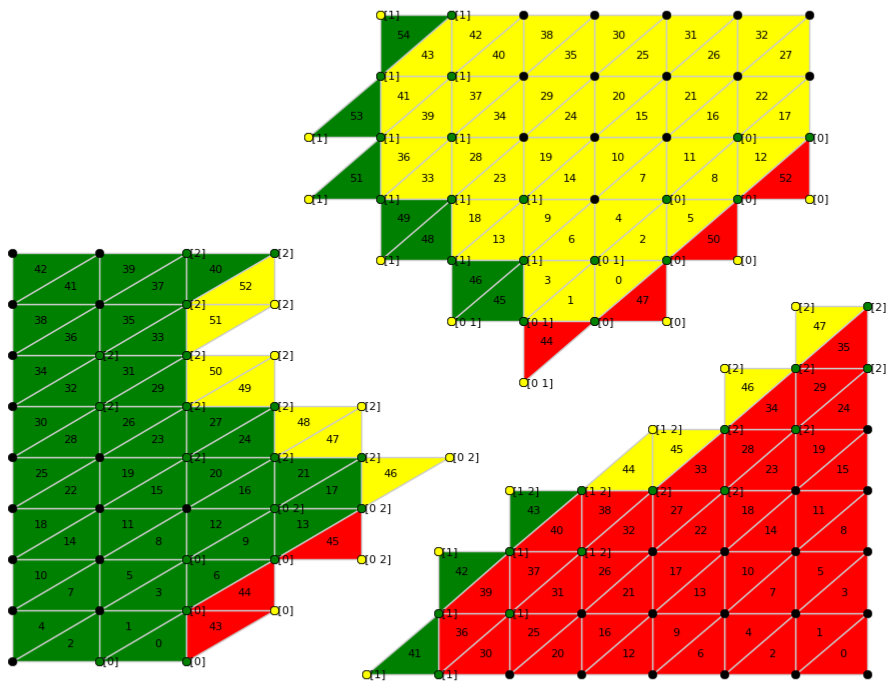


Figure 4: Example of mesh ghosting on 3 processes in two dimensions. Colours represent partitioning, with local cell indices shown on cells, and sharing processes at vertices. Yellow vertices are ghosts, green are shared, and black are exclusively local.

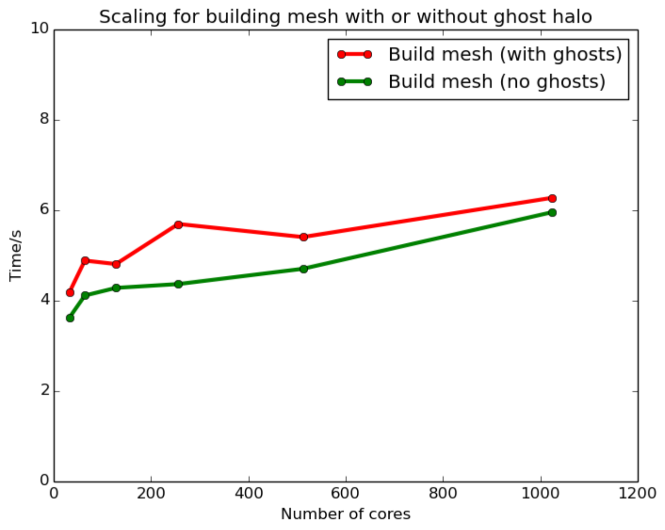


Figure 5: Time to partition and build the mesh for ghosted and non-ghosted meshes. The meshes constructed here were used for the results shown in Figure 6, with ∼40,000 cells per core.

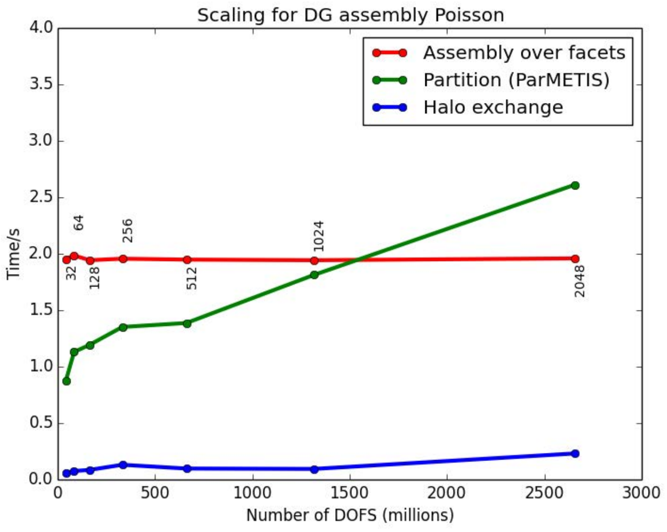


Figure 6: Weak scaling for the assembly of the Poisson matrix operator using an interior penalty DG method, to over 2 billion dofs. The numbers at each point show the number of processes. Note the halo exchange process is fast and scalable. The graph partitioner (ParMETIS) is an external dependency.

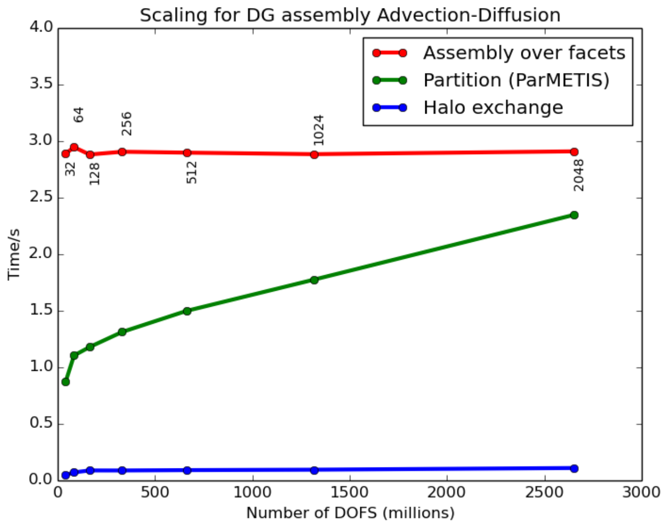


Figure 7: Weak scaling for the assembly of the advection matrix operator for an upwinded DG method over 2 billion dofs. The numbers at each point show the number of processes.

profiling data, we have seen that the present assembly scales well, so it was decided that adding communication-less assembly would be a premature optimisation. It would increase code complexity, and severely complicate and burden the automated testing systems which would need to perform all tests with and without communication-less assembly. However, the necessary hooks to implement communication-less assembly have been developed (with switches to iterators to iterate over just over local cells, or over both local and ghost cells). If it proves necessary, the activation of communication-less assembly will be straightforward in the future.

4 Profiling, performance and tuning (WP3)

As part of this work, some new platform-independent profiling tools for FEniCS/DOLFIN have been implemented, and a number of general performance and memory usage improvements have been made.

4.1 New profiling tools

We have developed here some new techniques for profiling the DOLFIN core library. A question which often arises is how much memory is being used at any particular point in the code. Usually, we want to work near the limits of the machine with the memory high-water mark for a node close to the maximum available memory on that node. Conversely, we would like to optimise the code so that we can reduce the high-water mark of memory usage. The package `atop` [6] allows us to continuously log memory usage for a machine (along with other statistics). We can do this by running the command: '`atop -w atop.dat 1`' to log to the file '`atop.dat`' once per second. During, or after a run, we can check the memory usage with '`atopsar -m -r atop.dat`'. On ARCHER it is difficult to launch multiple codes at the same time, so we need to employ a shell script to launch `atop` at the same time as the main code under test:

```
#!/bin/bash
if [ $ALPS_APP_PE -eq 0 ]; then
 atop -w atop.dat 1 &
fi
./code_under_test $*
```

This script is then launched in the usual way with `aprun`. The memory/time plots presented in this report were generated in this way. In principle, this technique is code independent, and can be used with any executable, without the need for recompilation with special profiling tools.

Memory profiling has proved very useful for identifying points in time when usage is high. However, to understand and improve code it is important to be able to identify what the code is doing at a particular time and to match this to the memory being used. In order to do this, we applied a small patch to the `Timer` class in DOLFIN. At present, the `Timer` is an object which measures the time between its instantiation and destruction in the code. By adding a simple logging statement to the constructor on process 0, we can get 'tickmarks' indicating when the timer was triggered. It is then possible to align events with the memory-time graph. An example is presented in Figure 8.

4.2 Improved distributed mesh memory usage and construction

The reading, partitioning and construction of unstructured meshes in parallel is a challenging problem since very little can be assumed about the structure of the input data. Partitioning a mesh

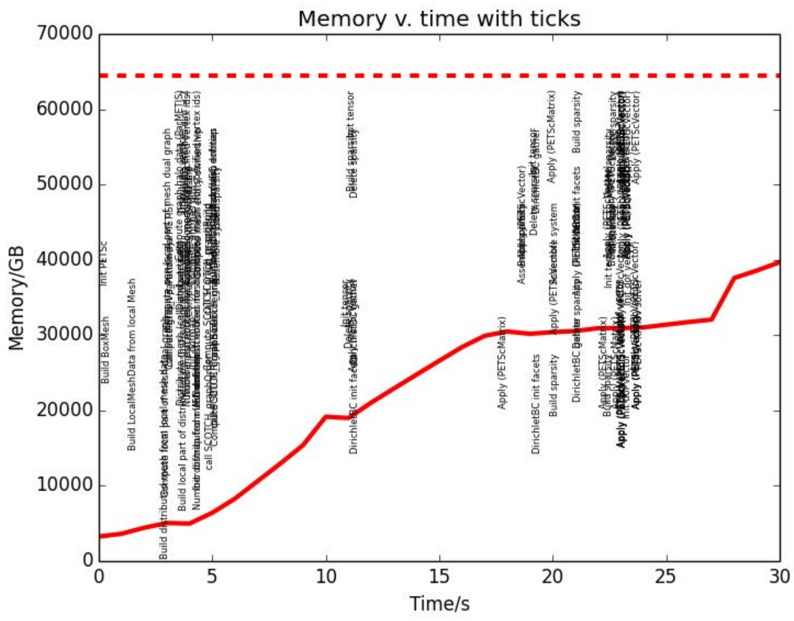


Figure 8: Memory versus time with text showing what step/action the code is performing at at particular time. These graphs are intended for developer inspection rather than public presentation.

requires non-trivial algorithms to construct a dual-graph for partitioning, and to subsequently number mesh entities globally.

We have made a number of significant improvements to dual graph construction, partitioning and distributed mesh construction. In particular:

- New algorithm for the construction of the process-wise dual graph. The new algorithm, which avoids the use of unordered maps is approximately two times faster with gcc and four times faster with clang. It also reduces the memory used, which is now independent of the quality of the input ordering (the degree of the memory improvement depends on how ordered/disordered the input mesh is for the old implementation). We do not present performance plots as this step is entirely local.

- The dual graph that DOLFIN constructs to pass to a mesh partitioner has been re-designed such that the graph can now be shared with the graph partitioners rather than copied. This reduces the memory used during mesh construction. This change required some templating to match the partitioning library integer type. The memory required for the dual graph is reduced by half.

4.3 Ordering algorithms

We have made it possible to apply local graph ordering strategies for the ordering of cells and vertices on processes, using the library SCOTCH. This can impact the speed of iterating over mesh entities by enhancing data locality. The effect of these operations is problem dependent, and in particular depends on the quality of the input mesh ordering. Users can switch the re-orderings on and off, depending on what best suits their problem.

5 Documentation and testing (WP4)

The newly implemented interfaces are documented in the code, and unit tests and example programs have been added. The new developments are in a branch in the main DOLFIN Git repository. Once merged into the main branch, the tests will be run upon any changes to the DOLFIN code (as part of the DOLFIN continuous integration system). Additional tests will be added as the high-level user interface evolves with user feedback.

6 Conclusions

The technical objectives of the work have been successfully achieved. The code developments are available in the development version of the relevant libraries, and are accessible via the public Git repositories (`https://bitbucket.org/fenics-project/`). The ghosted mesh functionality is available in the FEniCS 1.6 release, and other developments will be incorporated into a future release version of the FEniCS libraries. Developments in the project have already been used in published EPSRC and NERC supported scientific works, e.g. [7, 1], and are already being made available to users through the third-party library TerraFERMA (`http://terraferma.github.io/`).

Acknowledgement

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (`http://www.archer.ac.uk`).

References

[1] L. Alisic, S. Rhebergen, J. F. Rudge, R. F. Katz, and G. N. Wells. Torsion of a cylinder of partially molten rock with a spherical inclusion: theory and simulation. *Geochemistry, Geophysics, Geosystems*, 17(1), 2016. URL `http://dx.doi.org/10.1002/2015GC006061`.

[2] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans Math Software*, 40(2):9:1–9:37, 2014. URL `http://dx.doi.org/10.1145/2566630`.

[3] M. S. Alnæs, J. Blechta, J. Hake, J. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3 (100):9–23, 2015. doi: 10.11588/ans.2015.100.20553. URL `http://dx.doi.org/10.11588/ans.2015.100.20553`.

[4] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2016. URL `http://www.mcs.anl.gov/petsc`.

[5] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016. URL `http://www.mcs.anl.gov/petsc`.

[6] G. Langeveld and J. C. van Winkel. Atop Web page. `http://www.atoptool.nl`, 2015. URL `http://www.atoptool.nl`.

[7] H. J. T. Unwin, G. N. Wells, and A. W. Woods. CO_2 dissolution in background hydrological flow. *Journal of Fluid Mechanics*, 789:768–784, 2016. URL `http://dx.doi.org/10.1017/jfm.2015.752`.