Numerical Solution of PDEs Using the Finite Element Method

Exercise 7 & 8: MPI parallelisation:
parallel::shared::Triangulation and parallel::distributed::Triangulation

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

21 March 2018

Some useful resources

https://www.dealii.org/8.5.1/doxygen/deal.II/step_17.html
https://www.dealii.org/8.5.1/doxygen/deal.II/step_18.html
https://www.dealii.org/8.5.1/doxygen/deal.II/step_40.html
https://www.dealii.org/8.5.1/doxygen/deal.II/group__distributed.html
https://www.dealii.org/8.5.1/doxygen/deal.II/group__TrilinosWrappers.html
https://www.dealii.org/8.5.1/doxygen/deal.II/group__PETScWrappers.html

---

1. Using the supplied modified version of `step-6` as a base:

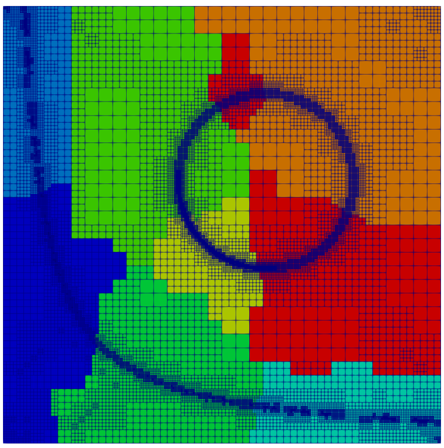
   (a) For the first version of this code, use `parallel::shared::Triangulation` and solve the 2d non-homogeneous Poisson equation

$$-\alpha(\mathbf{x})\Delta u(\mathbf{x}) = f(\mathbf{x}) \quad \text{in} \quad \Omega \in [0,1]^2, \quad \text{with}$$
$$u(\mathbf{x}) = 0 \quad \text{on} \quad \partial\Omega,$$

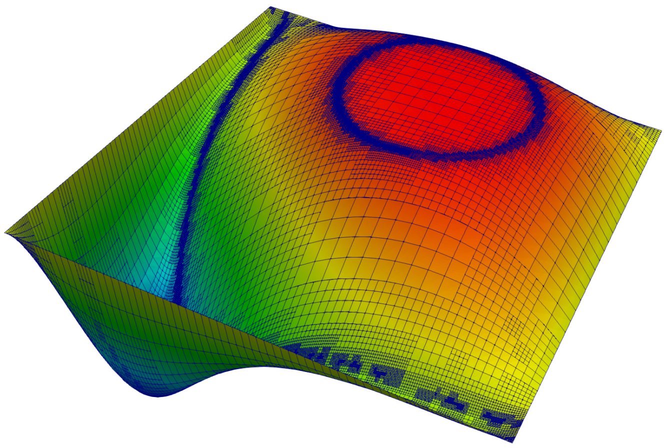
where

$$\alpha(\mathbf{x}) = \begin{cases} 5 & \text{if} \quad |\mathbf{x}-\mathbf{c}| < 0.2 \\ 1 & \text{otherwise,} \end{cases} \quad \text{and} \quad f(\mathbf{x}) = \begin{cases} 1 & \text{if} \quad |x_1 x_2| > 0.05 \\ -10 & \text{otherwise,} \end{cases}$$

and $\mathbf{c} = [0.6, 0.6]^T$. The result similar to the following:



(a) Mesh and partitioning



(b) Warped solution

Figure 1: Result produced from 3 initial global refinements and 8 refinement cycles, as visualised in `Paraview`.

These are the rough steps that you'll need to take to achieve this (look for the `TODO`'s listed in the minimal code):

i. Implement the functions defining the material coefficient $\alpha(\mathbf{x})$ and forcing function $f(\mathbf{x})$. Extra credit for using a `dealii::Function` for this purpose.

ii. Initialise the MPI environment correctly in the `main` function.

iii. In the `Step6` class constructor, initialise the class member variables correctly.

iv. In the `setup_system` function, initialise the sparsity pattern, system matrix, solution and RHS vectors correctly.

v. In the `assemble_system` function:

   α) Configure the range of cells over which the assembly is performed.

   β) Implement the forcing function.

   γ) Ensure synchronisation of the elements of the linear system at the end of the assembly loop.

vi. In the `solve` function, correctly choose the template parameter for the conjugate gradient solver, and select an appropriate preconditioner.

vii. In the `refine_grid` function, create the vector with entries required by the `KellyErrorEstimator`.

viii. In the `output_results` function, correctly construct the solution vector to be passed to `DataOut` for later processing and visualisation.

(b) Repeat the above using a `parallel::distributed::Triangulation`.

2. Additional tasks

   (a) Compare the distribution of cells across the processes for the two implementations. Is there a difference and, if so, why?

   (b) For this problem, measure the performance difference between the two implementations. What, do you think, are the primary factors affecting any differences you notice?

   (c) Investigate some of the various options for solvers (direct and iterative) and preconditioners. For example, `Trilinos` offers a direct solver, and its own implementation of iterative solvers, and numerous preconditioners. Consider the properties of the linear system when deciding which options/combinations to test.

   (d) Similar to the previous task, investigate the use of `PETSc` as the parallel linear algebra library.