Division of Scientific Computing
Department of Information Technology
Uppsala University

NGSSC: HPC II
December 15, 2011

Hands-on: solving numerical PDEs: experience with *deal.II* and *Trillinos*

The task of this computer lab is to try some basic functionalities of the packages *deal.II* and *Trillinos*.

To play with, you are provided with some ready codes in C++ (serial and parallel). These are to be downloaded from `http://user.it.uu.se/~maya/Courses/NGSSC/HPCII/Files_2011/dealII`.

1 Introduction

In the lab, we consider the well-known Poisson's problem in two dimensions:
Find $u(x, y)$ such that

$$\begin{aligned} -\Delta u &= f, \quad in\,\Omega \subset \mathbb{R}^2 \\ u &= 0, \quad on\,\partial\Omega \end{aligned}$$

where $\Omega = [-1, 1]^2$, and $f = 1$.

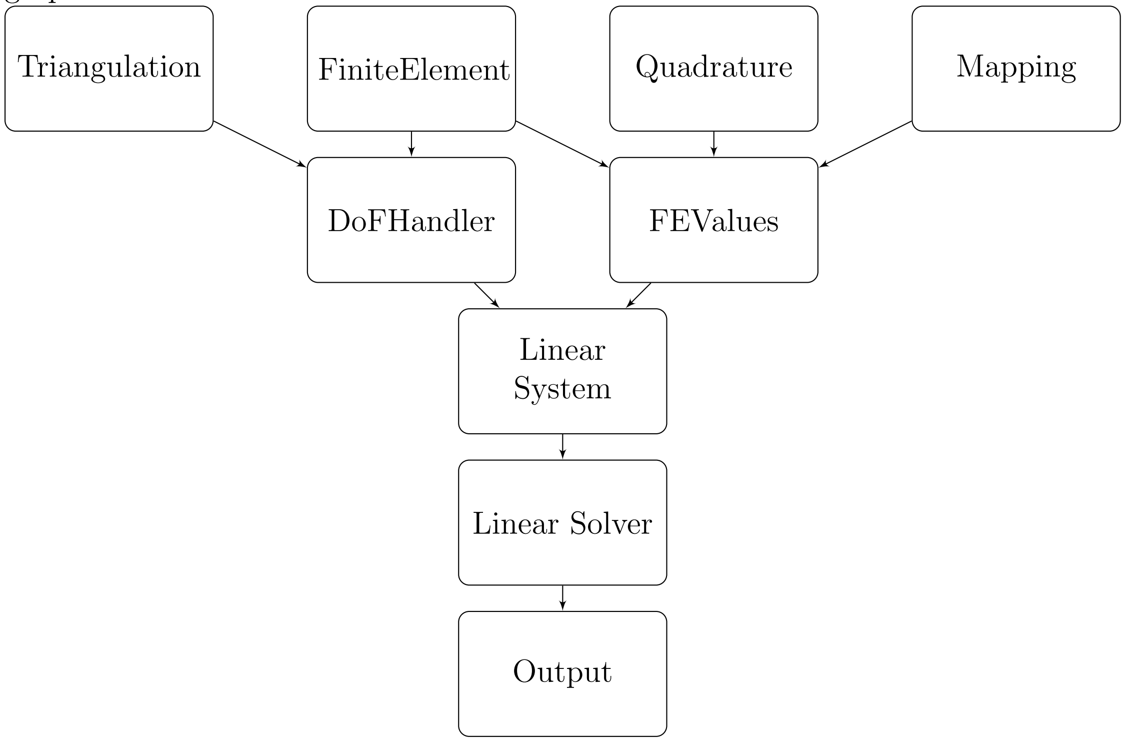
We discretize the problem using quadrilateral mesh (square in this case) and bilinear basis functions, see also the slides.

2 The blocks of a finite element program

The basic structure of a general finite element program to compute the solution of the problem numerically is as follows:

- Create a mesh
- Set up the linear system, i.e., assemble the matrix and the right hand side vector
- Solve the linear system
- Output the solution

An outline of how the primary groups of classes in deal.II interact is given by the following graph:



To ease the implementation of such a problem, *deal.II* provides a number of building blocks (modules), namely,

- Triangulation: Triangulation objects are used to define and create mesh. The triangulation stores geometric and topological properties of a mesh.
- Finite Element: Finite element classes describe the properties of a finite element space as defined on the unit cell.
- Quadrature: Quadrature objects are defined on the unit cell and only describe the location of quadrature points on the unit cell, and the weights of quadrature points thereon.
- DoFHandler: DoFHandler objects are the confluence of triangulations and finite elements. These enumerate all the degrees of freedom on the mesh and manage which degrees of freedom live where.
- Mapping: Mappings between reference and real cell.
- FEValues: The FEValues classes offer exactly this information: Given finite element, quadrature, and mapping objects.

- Linear Systems: In this module, classes are used to store and manage the entries of associated matrices, vectors, and the solution of linear systems.
- Linear Solvers: This module groups iterative and direct solvers, eigenvalue solvers, and some control classes.
- Output: deal.II generates output files in a variety of graphics formats understood by widely available visualization tools.

More details about these classes can be found at
`http://www.dealii.org/developer/doxygen/tutorial/index.html`

3 Tasks

3.1 Get started

Log onto the parallel computer kalkyl: ssh -AX kalkyl.uppmax.uu.se
The provided test programs are 'laplace.cc' (the sequential code) and 'laplace-p.cc' (the parallel code).

3.2 Check the *.bashrc* file

The *.bashrc* file has to contain the following lines:

```
export PATH=$HOME/bin:$PATH:/bubo/home/h1/petia/sw/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/bubo/home/h1/petia/sw/petsc-3.1-p6/linux-gnu-c-opt/lib
export PETSC_DIR=/bubo/home/h1/petia/sw/petsc-3.1-p6/
export PETSC_ARCH=linux-gnu-c-opt

module unload pgi
module load gcc/4.5 openmpi
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/bubo/sw/comp/intel/Compiler/11.1/073/mkl/lib/em64t/:
/bubo/home/h1/petia/sw/petsc-3.1-p6/linux-gnu-c-opt/lib/:
/bubo/home/h1/petia/sw/slepc-3.1-p4/linux-gnu-c-opt/lib/
```

3.3 Compile and run programs

Study the Makefile to see how to compile programs.

3.3.1 Serial runs

- Compile the program 'laplace.cc',
  ```
  make
  ```
  Run the program with n times of refinements of mesh,
  ```
  ./laplace n
  ```

Study the program source file and re-run the program and compare the output.

- Test the behaviour of the three solvers for a different size of the problem. Write down the timing results and compare those with the timings below.

3.3.2 Parallel runs

- Compile the program 'laplace-p.cc' (similarly as for 'laplace.cc')

  ```
  make
  ```

  Notice that we use mpicc and mpicxx. The runs are then handled with the command

  ```
  mpirun -n x laplace-p n
  ```

  requesting x processes to run program laplace-p with n times of refinements of mesh.

- For the parallel runs we want to see the strong scalability and the weak scalability of the problem. How would you do that?

3.4 For FEM connoisseurs

Consider the convection-diffusion problem

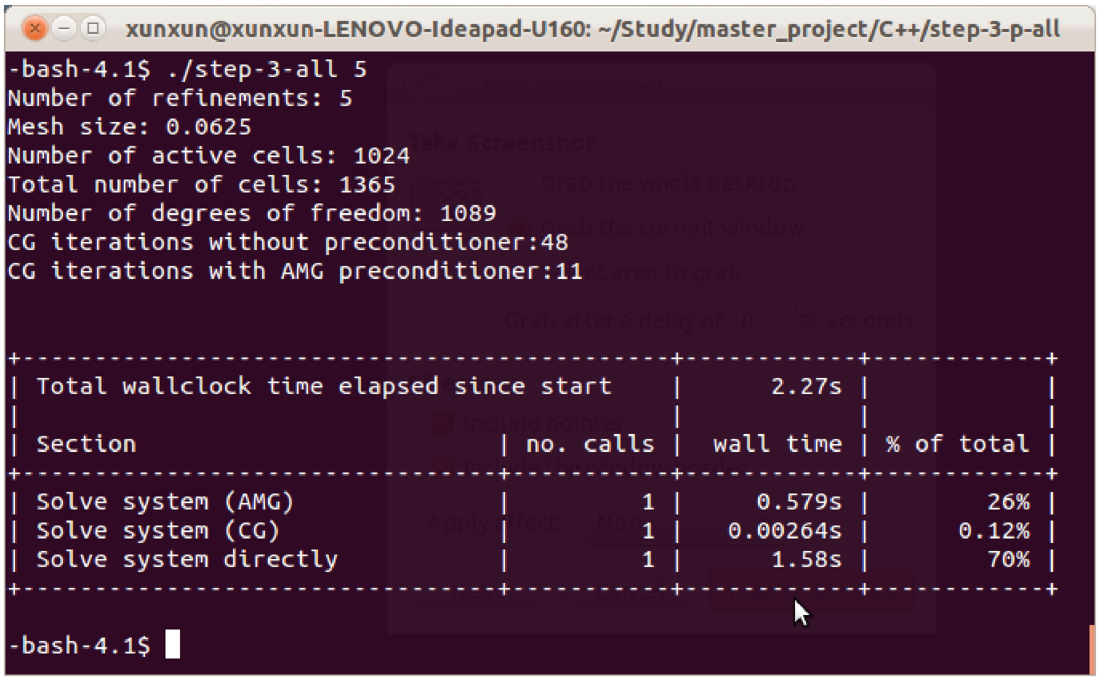
$$\begin{aligned} -\varepsilon\Delta u + (\mathbf{b}\cdot\nabla)u &= f, \quad in\,\Omega \subset \mathbb{R}^2 \\ u &= 0, \quad on\,\partial\Omega \end{aligned}$$

where $(\mathbf{b}\cdot\nabla)u = b_1\,u_x + b_2\,u_y$, $\mathbf{b} = [1,1]$ and $\varepsilon = 1, 0.1, 0.01$.
The necessary changes in the code are that the discrete Laplacian matrix has to be multiplied by ε and one new matrix has to be assembled, corresponding to the convection term.
As a hint, below you see the `Matlab` code to assembly the diffusion and the convection element matrices. (Note, that it is simpler to do the same in *deal.II*, check the assembly in the provided codes.)

```
for k=1:nip                              % nip = number of integration points
    FUN    = shape_fun_trian(Gauss_point,k);  % FUN(1,3)
    DER    = shape_der_trian;  % DER(2x3) %linear b.f.
    Jac    = DER *Coord;
    Det    = determinant2_m(Jac);
    IJac   = inv(Jac);
    Deriv  = IJac*DER;            % (2x3)=(2x2)*(2x3)
    L    = Deriv'*(epsilon*Deriv);         % anisotropic Laplacian
    C    = b(1)*FUN'*Deriv(1,:) + b(2)*FUN'*Deriv(2,:); % conv-diff
    M    = FUN'*FUN;
    A_elem = A_elem + Det*Gauss_weight(k)*(L+C);
end
```

4 Results

The output of the program looks as follows:


xunxun@xunxun-LENOVO-Ideapad-U160: ~/Study/master_project/C++/step-3-p-all

```
-bash-4.1$ ./step-3-all 5
Number of refinements: 5
Mesh size: 0.0625
Number of active cells: 1024
Total number of cells: 1365
Number of degrees of freedom: 1089
CG iterations without preconditioner:48
CG iterations with AMG preconditioner:11


+---------------------------------------------+------------+------------+
| Total wallclock time elapsed since start    |      2.27s |            |
|                                             |            |            |
| Section                         | no. calls |  wall time | % of total |
+---------------------------------+-----------+------------+------------+
| Solve system (AMG)              |         1 |     0.579s |        26% |
| Solve system (CG)               |         1 |   0.00264s |      0.12% |
| Solve system directly           |         1 |      1.58s |        70% |
+---------------------------------+-----------+------------+------------+

-bash-4.1$ 
```

In the sequential code, the direct solver is sparse direct solver UMFPACK, using the Unsymmetric-pattern MultiFrontal method and direct sparse LU factorization.

		Direct	CG		AMG	
DOF	Mesh Size h	time (s)	time (s)	iteration	time (s)	iteration
4225	0.03125	0.0438	0.00771	96	0.0346	13
66049	0.0078125	0.788	0.336	380	0.427	13
1050625	0.00195312	26.9	32.7	1446	7.07	14

Table 1: Sequential code

In the parallel code, the direct solver is the KLU direct solver, provided by Trilinos.

		Direct	CG		AMG	
DOF	Cores n	time (s)	time (s)	iteration	time (s)	iteration
66049	1	1.57	0.734	380	0.385	13
263169	1	15.5	6.62	751	1.61	13
1050625	1	134	57.9	1446	6.89	14

Table 2: Results obtained from the parallel code on one core (mpirun -n 1)

		Direct	CG		AMG	
DOF	Cores n	time (s)	time (s)	iteration	time (s)	iteration
66049	1	1.57	0.734	380	0.385	13
263169	4	15.2	2.76	751	0.484	13
1050625	8	137	22.6	1446	1.62	13

Table 3: Results obtained from the parallel code

5 Visualization

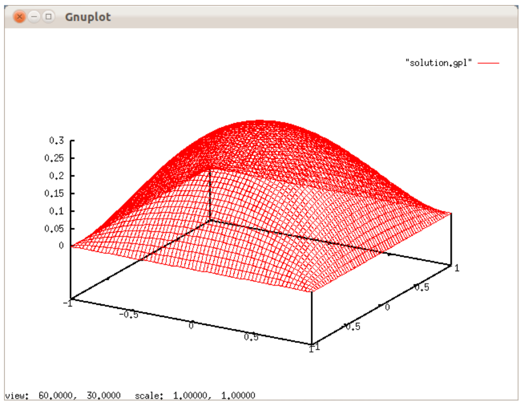
The program 'laplace.cc' generates the output file solution.gpl, which is in GNUPLOT format. It can be viewed by typing the following commands:
invoke GNUPLOT:

```
gnuplot
```

type commands at GNUPLOT prompt:

```
gnuplot> set style data lines
gnuplot> splot "solution.gpl"
```

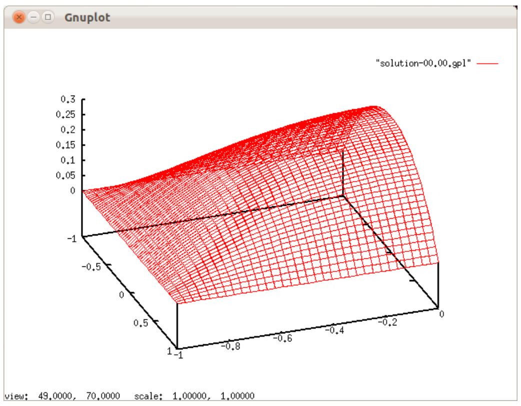
The result looks as follows:



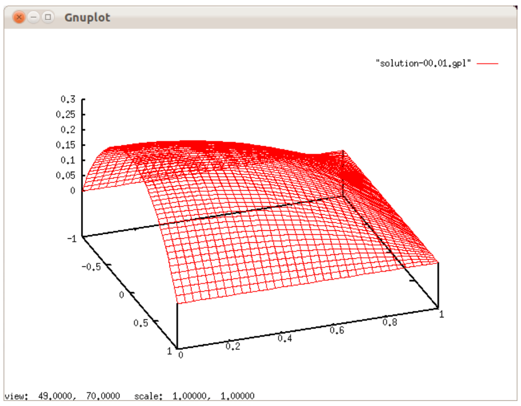
The program 'laplace-p.cc' also generates the output file in GNUPLOT format. All processors will write their own files. We could visualize them individually in GNUPLOT. And there is also a whole set of solution. If we test with the following command:

```
mpirun -n 2 laplace-p 6
```

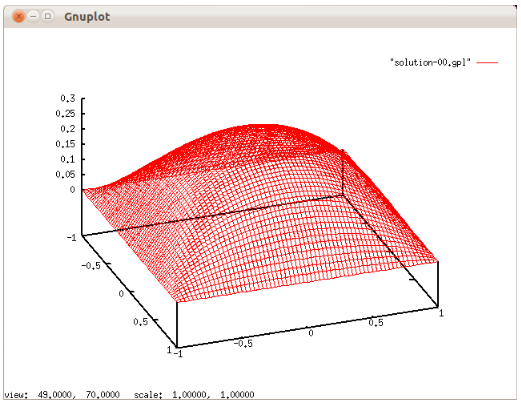
We will get two individual files written by two processors, namely 'solution-00.00.gpl' and 'solution-00.01.gpl', and a whole solution file, namely 'solution-00.gpl'.
The result looks as follows:



(a) processer 1



(b) processer 2



(c) whole solution