Parallel Algorithms

Maya Neytcheva[1] Ali Dorostkar[2]

[1]Department of Information Technology
Uppsala University

[2]Department of Information Technology
Uppsala University

May 22, 2013

Outline

Outline

What do we want to do?

Solve a PDE numerically.

Test example: Find u 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 $f = 1$.

What do we want to do

For setting up the spatial FE approximation, the first step is to rewrite the above equation in variational form.
Let $V = \{v : \|\nabla v\| + \|v\| < \infty,\ v|_{\partial\Omega} = 0\}$. Multiplying the equation with a test-function $v \in V$ and integrating over Ω using Green's formula with the homogeneous Dirichlet boundary conditions, we obtain

$$\begin{aligned}\int_\Omega f\,v\,dx &= -\int_\Omega \Delta u\,v\,dx \\ &= \int_\Omega \nabla u\cdot\nabla v\,dx - \int_{\partial\Omega} \mathbf{n}\cdot\nabla u\,v\,dx \\ &= \int_\Omega \nabla u\cdot\nabla v\,dx\end{aligned}$$

What do we want to do

The variational form is thus defined to be the following problem:
Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f \, v \, dx, \quad \forall v \in V$$

What do we want to do

In order to formulate the discretization in space, we decompose the infinite-dimensional computational domain Ω into finite-dimensional subsets (elements) with a characteristic size h. Let $\mathcal{K}$ be a triangulation of Ω, and let $V_h \subset V$ be the subspace of continuous piecewise linears on $\mathcal{K}$

$$V_h = \{v \in V,\ v|_{\partial\Omega} = 0\}$$

With this choice of approximation space, the discrete space counterpart of the equation reads:
Find $U \in V_h$ such that

$$\int_\Omega \nabla U \cdot \nabla v \, dx = \int_\Omega f\, v \, dx, \quad \forall v \in V_h$$

What do we want to do

Next, to compute the finite element approximation U we let $\{\varphi_i\}_{i=1}^N$ be the basis for the subspace V_h. Since U belong to V_h it can be written as:

$$U = \sum_{j=1}^{N} \mathbf{u}_j \varphi_j$$

with N unknowns $\mathbf{u}_j, j = 1, 2, \cdots, N$, to be found.
This equation can be rewritten as a linear system by inserting the representation $U = \sum_{j=1}^{N} \mathbf{u}_j \varphi_j$. Using the notation

$$A_{ij} = \int_\Omega \nabla\varphi_i \nabla\varphi_j \, dx, \quad b_i = \int_\Omega f \varphi_i \, dx, \quad i = 1, 2, \cdots, N$$

we have

$$b_i = \sum_{j=1}^{N} A_{ij} \mathbf{u}_j, \quad i = 1, 2, \cdots, N$$

What do we want to do

The linear system for the unknowns $\mathbf{u}_j$ in matrix form:

$$A\mathbf{u} = \mathbf{b}$$

Now we know what we are going to solve, and we can look at how to compute A_{ij} and $\mathbf{b}_i$ (form the integrals).
In the finite element method, this is most commonly done using some quadrature.

What do we want to do

We first split the integral over the whole domain into integrals over all cells,

$$A_{ij}^K = \sum_{K \in \mathcal{K}} \int_K \nabla \varphi_i \, \nabla \varphi_j$$

$$b_i^K = \sum_{K \in \mathcal{K}} \int_K f \, \varphi_i$$

and then approximate the integrals in each cell K by quadrature

$$A_{ij}^K \approx \sum_q \int_K \nabla \varphi_i(x_q^K) \, \nabla \varphi_j(x_q^K) \, \omega_j^K$$

$$b_i^K \approx \sum_q \int_K f(x_q^K) \, \varphi_i(x_q^K) \, \omega_j^K$$

What do we want to do

After A and $\mathbf{b}$ are made available, we have to choose a suitable numerical solution to solve the system.

- fast
- accurate
- robust

Outline

What is deal.ii

DEAL.II[1]: A Finite Element Differential Equations Analysis Library

- a C++ program library targeted at the computational solution of partial differential equations using adaptive finite elements.
- aims: to enable rapid development of modern finite element codes, using among other aspects adaptive meshes and a wide array of tools classes often used in finite element program.
- seemles using 1D, 2D or 3D programs.
- locally refined grids, adaptive refinement strategies and error indicators and error estimators.

[1] http://www.dealii.org/

What is deal.ii

DEAL.II[1]: A Finite Element Differential Equations Analysis Library

- *h*, *p*, *hp* refinement.
- continuous and discontinuous elements.
- support for a variety of finite elements.
- complete stand-alone linear algebra library.
- interface to other packages such as Trilinos, PETSc, METIS and P4est.
- smooth transition from serial to parallel.
- online documentation.

[1] http://www.dealii.org/

What is deal.ii

DEAL.II[1]: A Finite Element Differential Equations Analysis Library

- Modern software techniques that make access to the complex data structures and algorithms as transparent as possible.
- Support for several output formats.
- Portable support for a variety of computer platforms and compilers.
- Free source code under an Open Source license.
- open to contributors.

For its creation, its principal authors have received the 2007 J. H. Wilkinson Prize for Numerical Software.

[1] http://www.dealii.org/

What is Trilinos

The Trilinos[2] Project is an effort to develop and implement robust algorithms and enabling technologies using modern object-oriented software design, while still leveraging the value of established libraries such as PETSc, Metis/ParMetis, SuperLU, Aztec, the BLAS and LAPACK. It emphasizes abstract interfaces for maximum flexibility of component interchanging, and provides a full-featured set of concrete classes that implement all abstract interfaces.

[2] http://trilinos.sandia.gov/about.html

What is Trilinos

Deal.II has interface with two packages from Trilinos:

- Stratimikos: Thyra-based strategies for linear solvers
- Sacado: Automatic Differentiation Tools for C++ Codes

Programming in deal.ii

- mesh (choice of finite elements)
- assembly of matrices (choice of basis functions)
- solution methods (nonlinear and linear)
- paraelization tools
- visualization

Mesh- and finite-element related:

```
#include <deal.II/grid/tria.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/dofs/dof_accessor.h>

#include <deal.II/fe/fe_values.h>
#include <deal.II/base/quadrature_lib.h>

#include <deal.II/base/function.h>
```

Matrix/vector data structure and

```
#include <deal.II/lac/vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/compressed_sparsity_pattern.h>

#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/sparse_direct.h>

#include <deal.II/lac/trilinos_precondition.h>
```

Making the grid:

```
void laplace_problem::make_grid ()
{
  GridGenerator::hyper_cube (triangulation, -1, 1);

  triangulation.refine_global (n_refinement_steps);
  std::cout << "Total number of cells: "
    << triangulation.n_cells()
    << std::endl;
}
```

```
void laplace_problem::setup_system ()
{
  dof_handler.distribute_dofs (fe);
  std::cout << "Number of degrees of freedom: "
    << dof_handler.n_dofs()
    << std::endl;
  CompressedSparsityPattern c_sparsity(dof_handler.n_dofs())
     ;
  DoFTools::make_sparsity_pattern (dof_handler, c_sparsity);
  sparsity_pattern.copy_from(c_sparsity);

  system_matrix.reinit (sparsity_pattern);

  solution.reinit (dof_handler.n_dofs());
  system_rhs.reinit (dof_handler.n_dofs());
}
```

```
void laplace_problem::assemble_system ()
{
  QGauss<2>  quadrature_formula(2);
  FEValues<2> fe_values (fe, quadrature_formula,
 update_values | update_gradients | update_JxW_values);
  const unsigned int   dofs_per_cell = fe.dofs_per_cell;
  const unsigned int   n_q_points    = quadrature_formula.
      size();

  FullMatrix<double>   cell_matrix (dofs_per_cell,
      dofs_per_cell);
  Vector<double>       cell_rhs (dofs_per_cell);

  std::vector<unsigned int> local_dof_indices (dofs_per_cell
      );
```

```
DoFHandler<2>::active_cell_iterator
  cell = dof_handler.begin_active(),
  endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    fe_values.reinit (cell);
    cell_matrix = 0;
    cell_rhs = 0;
    for (unsigned int i=0; i<dofs_per_cell; ++i)
for (unsigned int j=0; j<dofs_per_cell; ++j)
  for (unsigned int q_point=0; q_point<n_q_points; ++q_point
      )
    cell_matrix(i,j) += (fe_values.shape_grad (i, q_point) *
 fe_values.shape_grad (j, q_point) *
 fe_values.JxW (q_point));
```

```
      for (unsigned int i=0; i<dofs_per_cell; ++i)
for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
  cell_rhs(i) += (fe_values.shape_value (i, q_point) *
  1 *
  fe_values.JxW (q_point));

      cell->get_dof_indices (local_dof_indices);

      for (unsigned int i=0; i<dofs_per_cell; ++i)
for (unsigned int j=0; j<dofs_per_cell; ++j)
  system_matrix.add (local_dof_indices[i],
     local_dof_indices[j],
     cell_matrix(i,j));

      for (unsigned int i=0; i<dofs_per_cell; ++i)
system_rhs(local_dof_indices[i]) += cell_rhs(i);
    }
```

```
std::map<unsigned int,double> boundary_values;
VectorTools::interpolate_boundary_values (dof_handler,
MatrixTools::apply_boundary_values (boundary_values,
    system_matrix,
    solution,
    system_rhs);
}
```

Solving the linear system: direct method

```
void laplace_problem::solve_direct ()
{
    SparseDirectUMFPACK direct_solver;
    direct_solver.initialize(system_matrix);
    direct_solver.vmult (solution, system_rhs);
}
```

Solving the linear system: unpreconditioned CG

```
void laplace_problem::solve_cg ()
{

  SolverControl   solver_control (system_matrix.m(), 1e-12);

  SolverCG<>      solver (solver_control);

  solver.solve (system_matrix, solution, system_rhs,
PreconditionIdentity());
  std::cout<< "CG iterations without preconditioner:"...
                <<solver_control.last_step()<<std::endl;
}
```

Solving the linear system: AMG-preconditioned CG

```
void laplace_problem::solve_amg ()
{
  Amg_preconditioner.reset ();
  Amg_preconditioner = std_cxx1x::shared_ptr<TrilinosWrappers::PreconditionAMG>
                            (new TrilinosWrappers::PreconditionAMG());

  std::vector<std::vector<bool> > constant_modes;
  std::vector<bool>  components (3,true);
  components[2] = false;
  DoFTools::extract_constant_modes (dof_handler, components,
                                    constant_modes);
  TrilinosWrappers::PreconditionAMG::AdditionalData Amg_data;
  Amg_data.constant_modes = constant_modes;
  Amg_data.elliptic = true;
  Amg_data.higher_order_elements = true;
  Amg_data.smoother_sweeps = 2;
  Amg_data.aggregation_threshold = 0.02;
  Amg_preconditioner->initialize(system_matrix, Amg_data);

  SolverControl            solver_control (system_matrix.m(), 1e-12);
  SolverCG<>               solver (solver_control);
  solver.solve (system_matrix, solution, system_rhs,
*Amg_preconditioner);
  std::cout<< "CG iterations with AMG preconditioner:"<<solver_control.last_step()<<std::
        endl;
}
```

The actual execution part:

```
void laplace_problem::run (int n_refs)
{
  Vector<double>  init_sol;
  Vector<double>  init_rhs;
  double mesh_size;
  n_refinement_steps = n_refs;
  std::cout<< "Number of refinements: " <<n_refinement_steps
       << std::endl;

  mesh_size = 2*std::pow(0.5, double(n_refinement_steps));
  pcout << "Mesh size: " << mesh_size<< std::endl;
  make_grid ();
  setup_system();
  assemble_system ();
  init_rhs = system_rhs;
  init_sol = solution;
}
```

```
 computing_timer.enter_section("Solve system directly");
  solve_direct ();
  computing_timer.exit_section("Solve system directly");

  solution = init_sol;
  system_rhs = init_rhs;
  computing_timer.enter_section("Solve system (CG)");
  solve_cg ();
  computing_timer.exit_section("Solve system (CG)");

  solution = init_sol;
  system_rhs = init_rhs;
  computing_timer.enter_section("Solve system (AMG)");
  solve_amg ();
  computing_timer.exit_section("Solve system (AMG)");

  output_results ();
}
```