

High Performance Computing – Blatt 6

(Präsenzübung 3. Juni 2013)

Diskussion

- *Finite Elemente:*

- Was ist die schwache Formulierung einer PDE und wie wird sie hergeleitet? Was ist mit schwacher/starker Lösung gemeint?
- Was ist die Idee des Ritz-Galerkin-Verfahrens? Wie kommt man von der unendlich-dimensionalen schwachen Formulierung auf ein numerisch lösbares Problem?
- Wie sieht die Hutbasis aus? Malen Sie ein 1D-Gitter und die zugehörigen Basisfunktionen.
- Was für ein Gleichungssystem muss dann numerisch gelöst werden? Was sind die Unbekannten?
- Zur Behandlung von nicht-homogenen Dirichlet-Randbedingungen, also einem Problem

$$\begin{aligned}\mathcal{A}(u) &= f && \text{auf } \Omega, \\ u &= g && \text{auf } \Gamma_D,\end{aligned}$$

mit $g \neq 0$ und linearem Operator \mathcal{A} , führt man normalerweise eine Homogenisierung durch: Wir lösen das Problem für $\tilde{u} = u - g$, also

$$\begin{aligned}\mathcal{A}(\tilde{u}) &= f && \text{auf } \Omega, \\ \tilde{u} &= 0 && \text{auf } \Gamma_D.\end{aligned}$$

Wie verändert sich dadurch das zu lösende Gleichungssystem? Wie kann man aus \tilde{u} die gewünschte Lösung u wieder rekonstruieren?

- *Vorbereitung Aufgabe 1:*

- Lesen Sie die Aufgabenstellung und die Hinweise durch (!!)
- Schauen Sie sich die einzelnen Klassen an und versuchen Sie, zu verstehen, was wofür zuständig ist (also warum die Aufteilung so ist, wie sie ist).
- Notieren Sie Ihre Fragen, die sich dabei ergeben!

- *Vorbereitung Aufgabe 2:*

- Lesen Sie das beschriebene Vorgehen durch (!!)
- Welche Längen haben jeweils die lokalen Vektoren?
- Wie kann das kleine Tridiagonalsystem (Schritt 5) auf jedem Prozessor assembliert werden? Welche Datenstrukturen braucht man dafür?

Aufgabe 1: FEM 1D (seriell)

Solve the 1D Poisson problem

$$\begin{aligned} -u_{xx} &= f(x) && \text{on } \Omega = (0, 1), \\ u(0) &= g_0, u(1) = g_1. \end{aligned}$$

using linear Lagrangian Finite Elements (the “hat basis”). On the homepage, you find files for the following classes:

- `EllipticFEM1D`: Contains the finite element structures, i.e. the 1D mesh, the solution vector, information about the Dirichlet boundaries etc. Also has a reference to a class that provides the PDE which we want to solve.
- `PDE1D`: The (abstract) class for a PDE problem which we want to solve. This is nothing but an interface, specifying what a PDE problem class should look like. There can never be an object of this class, but we can derive different classes for different PDEs from this one. The abstract function `assemble_on_element` has to be implemented in each of these derived classes, so that it is guaranteed that this function always exists.
- `Poisson1D`: One example of a concrete PDE problem. This class provides the assembly of the Laplace operator on a given finite element (here in 1D specified by the interval boundaries x_i, x_{i+1}). It requires function pointers describing the parameters of the PDE: f for the right hand side and g for the Dirichlet boundaries (with $g(0) = g_0, g(1) = g_1$).

Moreover, you find a file `test_fem1d.cpp` that uses the above classes to set up a Poisson problem and solve this PDE.

Implement the missing function implementations:

- `Poisson1D::assemble_on_element(..)`: Computes the contributions to **A** and **F** for one single element (i.e. $A^{(k)}, b^{(k)}$, see hints below). These entries can be computed by hand (see lecture notes).
- `EllipticFEM1D::assemble()`: Loops over all elements of the mesh, calls the function `assemble_on_element(..)` for each element and adds the results into the correct positions of the stiffness matrix **A** and the right hand side vector **F**. Also handles the modifications of both **F** and the solution vector `solution` for (non-homogeneous) Dirichlet boundary conditions.

Solve the problem for $f(x) = 1, g_0 = g_1 = 0$, as well as for $f(x) = 1, g(x) = -\frac{1}{2}x$ (with solution $u(x) = -\frac{1}{2}x^2$). Plot your solutions u .

Hints:

- **Element-wise assembly**: Each entry $A_{i,j}, i, j = 1, \dots, N$, in the stiffness matrix A_h can be decomposed into the contributions on the individual elements (intervals in 1D):

$$A_{i,j} = \int_0^1 \nabla \psi_i(x) \cdot \nabla \psi_j(x) dx = \sum_{k=0}^N \int_{x_k}^{x_{k+1}} \nabla \psi_i(x) \cdot \nabla \psi_j(x) dx =: \sum_{k=0}^N A_{i,j}^{(k)}.$$

As $A_{i,j}^{(k)} \neq 0$ only for $i, j \in \{k, k+1\}$, we have to assemble on each interval only the 2×2 -matrix $A^{(k)} := \begin{pmatrix} A_{k,k}^{(k)} & A_{k,k+1}^{(k)} \\ A_{k+1,k}^{(k)} & A_{k+1,k+1}^{(k)} \end{pmatrix}$, which can be done “by hand” (cf. lecture).

Therefore, we can assemble A_h by once looping over all intervals, calculating the individual contributions $A^{(k)}$ and adding them to the corresponding entries in the global stiffness matrix A_h .

- **Assembly of right-hand side:** As for the stiffness matrix, we perform an element-wise assembly for each entry:

$$(f, \psi_i)_0 = \int_0^1 f(x)\psi_i(x)dx = \sum_{k=0}^N \int_{x_k}^{x_{k+1}} f(x)\psi_i(x)dx =: \sum_{k=0}^N b_i^{(k)}, \quad i = 1, \dots, N.$$

We use the trapezoidal rule to approximate the integrals: $b_i^{(k)} = \frac{f(x_k)\psi_i(x_k) + f(x_{k+1})\psi_i(x_{k+1})}{2}(x_{k+1} - x_k)$. Again, on each interval, we only have to compute the vector $b^{(k)} := (b_k^{(k)}, b_{k+1}^{(k)})^T$, which is easily done using the nodal structure of our basis.

- **Dirichlet boundary conditions:** In order to homogenize the problem, we choose a function $g \in S_h$ as $g = g_0\psi_0 + g_1\psi_{N+1}$. Then $g(0) = g_0$, $g(1) = g_1$. The homogeneous equation system $Lw = f - Lg$ then reads in the discrete space

$$\sum_{j=1}^N u_j a(\psi_j, \psi_i) = (f, \psi_i)_0 - [g_0 a(\psi_0, \psi_i) - g_1 a(\psi_{N+1}, \psi_i)], \quad i = 1, \dots, N,$$

and $u_h := \sum_{i=1}^N u_i \psi_i + g$ is a solution of the original boundary value problem. Note that most additional terms $g_0 a(\psi_0, \psi_i)$, $g_1 a(\psi_{N+1}, \psi_i)$ on the right hand side are zero. Which ones are not?

- **Data structures I (Mesh):** The linear system itself is only solved for the unknown coefficients u_j , $j = 1, \dots, N$ (the so-called *degrees of freedom*). However, we want a solution representation for **all** mesh points $\{x_0, \dots, x_{N+1}\}$, including the boundary. The easiest solution (which avoids unnecessary copy operations or separate structures for inner and boundary values) is to just assemble a system of size $(N+1) \times (N+1)$ and fill the rows and columns corresponding to Dirichlet boundary points with zeros. For our cg-solver, this causes no problem at all. Note, however, that one usually sets the diagonal entries $A_{i,i} = 1$ in the Dirichlet rows/columns, so that the matrix has full rank. This is important for some solvers and most preconditioners.
- **Data structures II (Matrix and Vector classes):** In the material, you’ll find a folder *LinearAlgebra* containing matrix and vector classes which we’ll be using for the rest of the semester. Inside this folder, call `make` once to generate the library `linAlg` to which we can then link our programs.
- **Compilation:** This code is still completely serial, so we don’t need MPI. But we have to link against the library *LinAlg*, so that the compilation command looks as follows:

```
g++ -Wall -o test_fem1d test_fem1d.cpp ellipticfem1d.cpp poisson1d.cpp
-L./LinearAlgebra -lLinAlg
```


2. Berechne $\mathbf{U} = \mathbf{VW}$, also die lokalen Vektoren $\tilde{f}^{(k)}, g^{(k)}$ für $k = 0, \dots, n_{\text{procs}}$ und die Einträge $\tilde{d}_{n_k-1}^{(k)}$ für $k < n_{\text{procs}}$.

Dabei können zunächst alle Einträge von $\tilde{f}^{(k)}$ und alle bis auf den letzten Eintrag von $g^{(k)}$ ohne Kommunikation berechnet werden:

```

 $\tilde{f}_{n_k-1} = f_{n_k-1}, \tilde{f}_{n_k-2} = f_{n_k-2}, g_{n_k-2} = u_{n_k-1}.$ 
for  $i = n_k - 3, \dots, 0$  do
     $\tilde{f}_i = f_i - \frac{u_i \tilde{f}_{i+1}}{d_{i+1}}, g_i = -\frac{u_i g_{i+1}}{d_{i+1}}.$ 
end for

```

Um die letzte Zeile in die richtige Form zu bringen (also $u_{n_k-1}^{(k)}$ zu eliminieren), muss allerdings kommuniziert werden:

```

if  $k > 0$  then
    Schicke  $d_0$  und  $\tilde{f}_0$  an Prozess  $k - 1$ .
    Empfange  $u_{\text{recv}}$  von Prozess  $k - 1$ .
    Berechne  $g_{n_k-1} = -\frac{g_0}{d_0} \cdot u_{\text{recv}}$ .
end if
if  $k < n_{\text{procs}}$  then
    Schicke  $u_{n_k-1}$  an Prozess  $k + 1$ .
    Empfange  $d_{\text{recv}}$  und  $f_{\text{recv}}$  von Prozess  $k + 1$ .
    Berechne  $\tilde{d}_{n_k-1} = d_{n_k-1} - \frac{f_{\text{recv}}}{d_{\text{recv}}} \cdot u_{n_k-1}$ .
end if

```

3. Löse $\mathbf{Ly} = \mathbf{F}$ durch Vorwärtseinsetzen, hierfür ist keine Kommunikation notwendig.
4. Löse $\mathbf{Vz} = \mathbf{y}$ durch Rückwärtseinsetzen. Ohne Kommunikation kann man dabei $z_{n_k-2}^{(k)}, \dots, z_0^{(k)}$ berechnen. Für die Berechnung von $z_{n_k-1}^{(k)}$ muss noch mit dem nachfolgenden Prozess $k + 1$ kommuniziert werden, um $z_0^{(k+1)}$ zu erhalten.
5. Löse $\mathbf{Wx} = \mathbf{z}$. Dazu muss erst das kleine Tridiagonal-System

$$\begin{pmatrix} \tilde{d}_{n_0-1}^{(0)} & g_{n_1-1}^{(1)} & \\ \tilde{f}_{n_1-1}^{(1)} & \tilde{d}_{n_1-1}^{(1)} & g_{n_2-1}^{(2)} \\ & \tilde{f}_{n_2-1}^{(2)} & \tilde{d}_{n_2-1}^{(2)} \end{pmatrix} x = \begin{pmatrix} z_{n_0-1}^{(0)} \\ z_{n_1-1}^{(1)} \\ z_{n_2-1}^{(2)} \end{pmatrix}$$

auf jedem Prozessor assembliert und gelöst werden. Zum Lösen steht der Solver `void solveTridiag(Vector &ldiag, Vector &diag, Vector &udiag, Vector &x, Vector &b);` zur Verfügung.

Danach können die restlichen Einträge von \mathbf{x} durch Rückwärts-Substitutionen lokal auf jedem Prozess berechnet werden (Achtung: Hier muss man zwischen Prozess 0 und den anderen Prozessen unterscheiden).

6. Zum Schluss müssen noch die lokalen Lösungen an Prozess 0 zurückgesendet werden, der sie zusammensetzt und dann auch z.B. in eine Datei ausgibt.

Aufgabe:

Ergänzen Sie die fehlenden Code-Teile in der Funktion `EllipticFEM1D::solve_parallel()`, um den obigen Algorithmus zu implementieren.

Kompiliert wird mit dem folgenden Befehl:

```

openmpic++ -Wall -o test_fem1d_parallel test_fem1d_parallel.cpp ellipticfem1d.cpp
poisson1d.cpp -L./LinearAlgebra -lLinAlg

```