JOHANNES KEPLER
UNIVERSITY LINZ

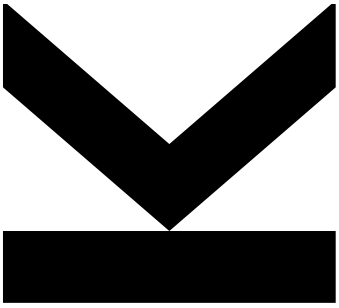
Submitted by
Christoph Plakolm

Submitted at
Institute of Computational Mathematics

Supervisor
O.Univ.-Prof. Dipl.-Ing. Dr. Ulrich Langer

12.06.2019

Parallel CG Method



Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Technische Mathematik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

This thesis is devoted to the parallel solution of large-scale systems of algebraic equations, arising from the finite element discretization of elliptic boundary value problems, by means of a parallel version of the conjugate gradient method. Starting from the classical Poisson equation as a model problem, we use the Galerkin finite element method in order to discretize the model boundary value problem. The resulting system of linear algebraic equations is symmetric and positive definite. Therefore, it can be solved by the conjugate gradient method on a serial computer. In order to make use of the computational power of a massive parallel computer, the conjugate gradient method needs to be parallelized. This is achieved by domain decomposition, where the computational domain is decomposed into P subdomains that are then assigned to P processing units. We provide a `C++` implementation. Finally, we present some numerical results obtained on serial and parallel computers.

Zusammenfassung

Die Parallelisierung von Lösungsverfahren in der Numerischen Mathematik ist ein mächtiges Werkzeug, um großdimensionierte Probleme zeitsparend und energieeffizient zu lösen. Diese Bachelorarbeit beschäftigt mit einer mathematischen Theorie zum Parallelisieren von Finite-Elemente-Methoden für elliptische Randwertprobleme und im Speziellen mit der parallelen Anwendung des Verfahrens der konjugierten Gradienten. Das klassische Poisson Problem wird mit dem Galerkin Verfahren diskretisiert. Das dabei entstehende System linearer algebraischer Gleichungen ist dann symmetrisch und positiv definit, sodass das Verfahren der konjugierten Gradienten seriell angewendet werden kann. Um nun die Rechenleistung von Supercomputern wirklich nutzen zu können, muss das Lösungsverfahren parallelisiert werden. Dazu wird das Rechengebiet in P sich nicht überlappende Teilgebiete zerlegt, die dann P Prozessoren zugeordnet werden. Jeder der Prozessoren löst ein lokales Teilproblem auf dem zugehörigen Teilgebiet. Im Rahmen dieser Arbeit stellen wir eine `C++` Implementierung vor und präsentieren numerische Resultate aus Berechnungen auf großen Parallelrechnern.

Acknowledgements

I thank my supervisor Ulrich Langer for giving me the possibility to write this thesis and for providing literature to me as well as answering my many questions. I would also like to thank Martin Neumüller for raising my interest in this topic.

Finally a great word of thanks goes to my family who always supported me during my studies.

Christoph Plakolm
Linz, June 2019

Contents

Chapter 1

Introduction

The conjugate gradient method (CG method) is a well-known algorithm used for the numerical computation of the solution of linear systems of algebraic equations with a corresponding symmetric and positive definite matrix. Although the CG method is a direct solver, it is more commonly used as an iterative method for large-scale sparse systems. Such types of linear systems typically arise when solving partial differential equations numerically. They are often too large to be solved with direct algorithms such as the Gaussian elimination algorithm or the Cholesky decomposition, where the latter also requires positive definiteness and symmetry. In 2016, Nicholas J. Higham published a top ten rating of the most influential algorithms in applied mathematics in the 20th and 21st century on his popular website (https://nickhigham.wordpress.com/2016/03/29/the-top-10-algorithms-in-applied-mathematics/), ranking the CG method among other Krylov-subspace methods at place six. The growing demand for fast and efficient solution procedures of massively large systems of algebraic equations as well as the increasing importance of efficiency in terms of energy inspired the use of huge clusters of computers over the last decades. Algorithms that are executed on so-called supercomputers require parallelization. The aim of this thesis is to provide a mathematical approach to the parallelization of the CG method.

In Chapter 2, we will introduce the famous Poisson equation as our model boundary value problem, which is interesting not only from a mathematical but also from a practical point of view, as it is used to model for example heat conduction and diffusion problems. Then we will derive the variational formulation, which serves as a starting point for the classical finite-element-discretization. Using a uniform triangulation and the Galerkin principle, we obtain a large-scale sparse system of algebraic equations to which we can apply the CG method.

Chapter 3 is devoted to the parallelization of the CG method. First the discretized computational domain is decomposed into P subdomains, where P is the number of processing units. Then each subdomain will be allocated to a processing unit. Finally, each unit will solve a system of algebraic equations locally on the corresponding subdomain. In order to compute the arithmetic operations required for the CG methods in parallel, we will introduce two types of vectors: accumulated vectors, that hold the exact values on interfaces of subdomains, and distributed vectors, that hold only a part of the original value on interfaces. With this concept

we will see that it is quite easy to derive the parallel CG method from the serial algorithm.

The C++ implementation of the parallel CG method is described in Chapter 4. The one-dimensional Poisson equation was implemented using the OpenMP API specification for parallel programming (see: www.openmp.org) and the C++-class "Vector.h" by Dr. Clemens Pechstein for numerical computations with vectors (see: Appendix). The source code is attached to the Appendix as well. The data structure used is of particular interest, as it resembles the theory that has been worked out in Chapter 3. Finally, we will examine tests and numerical results obtained by computations on serial and parallel computers. The latter have been executed on the high performance computing cluster RADON 1 at the Johannes Kepler University in Linz; see: https://www.ricam.oeaw.ac.at/hpc/.

Chapter 2

Model Problem and Numerical Solution

The goal for this chapter is to apply the finite element method to discretize a partial differential equation, and to solve the resulting system of linear equations by means of the CG method. The serial conjugate gradient method can be applied to any system of equations with a positive definite and symmetric system matrix. The famous Poisson equation will fullfill these properties.

2.1 Model Problem: The Poisson Equation

Let $f : \Omega \longrightarrow \mathbb{R}$ be a given function, and let $\Omega := (a, b) \times (a, b) \subset \mathbb{R}^2$ be a square. We consider the model boundary value problem:

find $u : \bar{\Omega} \longrightarrow \mathbb{R}$ such that

$$-\Delta u = f \quad \text{in } \Omega, \qquad u = g_D \quad \text{on } \Gamma_D, \qquad \frac{\partial u}{\partial n} = g_N \quad \text{on } \Gamma_N, \tag{2.1}$$

witz $\partial\Omega = \Gamma_D \cup \Gamma_N$ and $\text{meas}_{d-1}(\Gamma_D) > 0$. The Poisson equation (2.1) is a partial differential equation with, in this case, a Dirichlet and a Neumann boundary condition. In order to use the finite element method, we have to convert the classical formulation into the variational (weak) formulation.

2.2 Variational Formulation

Multiplying (2.1) by a test function $v \in V_0 \subset V := H^1(\Omega)$ and integrating over the whole domain Ω, we get

$$-\int_\Omega \Delta u \cdot v \, \mathrm{d}x = \int_\Omega f \cdot v \, \mathrm{d}x,$$

where $V_0 := \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\}$. $H^1(\Omega)$ denotes the Sobolev space $W_2^1(\Omega)$, that is a Hilbert space with the inner product $(u, v)_{H^1(\Omega)} := \int_\Omega (uv + \nabla u \nabla v) \mathrm{d}x$ and the norm $||v||_{H^1(\Omega)} := (u, v)^{1/2}_{H^1(\Omega)}$ for $u, v \in H^1(\Omega)$. Applying Gauß' theorem

(partial integration) to the left-hand side and incorporating the Neumann boundary condition, we obtain the variational form

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f \cdot v \, \mathrm{d}x + \int_{\Gamma_N} g_N \cdot v \, \mathrm{d}s_x.$$

All in all we can summerize the variational formulation as follows.
Find $u \in V_g := \{v \in H^1(\Omega) : v = g_D \text{ on } \Gamma_D\}$ such that

$$a(u,v) = \langle F, v\rangle \quad \forall v \in V_0 := \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\}, \tag{2.2}$$

where the bilinear form and the linear form are defined as

$$a(u,v) := \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x \tag{2.3}$$

and

$$\langle F, v\rangle := \int_\Omega f \cdot v \, \mathrm{d}x + \int_{\Gamma_N} g_N \cdot v \, \mathrm{d}s_x \tag{2.4}$$

with given $f \in L_2(\Omega)$ and $g_N \in L_2(\Gamma_N)$.

For a more detailed derivation of the variational formulation for Poisson's equation (in the one-dimensional case), see [2].

Furthermore the following properties hold [2]:

Boundeness of the linear functional $F \in V^*$, i. e. $\exists c = const > 0$ s. t.

$$|\langle F, v\rangle| \leq c\|v\|_V \quad \forall v \in V,$$

where $||\cdot||_V = ||\cdot||_{H^1(\Omega)}$. Indeed, with the trace theorem,
i. e. $\exists c_T < 0 : ||u||_{L_2(\Gamma_N)} \leq c_T||u||_{H^1(\Omega)}$, we get

$$|\langle F, v\rangle| \leq ||f||_{L_2(\Omega)}||v||_{L_2(\Omega)} + ||g_N||_{L_2(\Gamma_N)}||v||_{L_2(\Gamma_N)} \leq \left(||f||_{L_2(\Omega)} + c_T||g_N||_{L_2(\Gamma_N)}\right) ||v||_{H^1(\Omega)}.$$

Coercivity of bilinearform $a : V \times V \longrightarrow \mathbb{R}$, i. e. $\exists c_1^a = const > 0$ s. t.

$$a(v,v) \geq c_1^a \|v\|_V^2 \quad \forall v \in V_0.$$

Indeed, using Friedrich's inequality, i. e. $\exists c_F > 0 : ||v||_{H^1(\Omega)} \leq c_F|v|_{H^1(\Omega)}$, we obtain

$$a(v,v) = \int_\Omega |\nabla v|^2 \mathrm{d}x = |v|^2_{H^1(\Omega)} \geq \frac{1}{c_F^2}||v||^2_{H^1(\Omega)},$$

thus $c_1^a = 1/c_F^2$. Here $|v|_{H^1(\Omega)} := ||\nabla v||_{L_2}$ denotes the H^1-seminorm.

Boundness of bilinearform $a : V \times V \longrightarrow \mathbb{R}$, i. e. $\exists c_2^a = const > 0$ s. t.

$$a(u,v) \leq c_2^a \|u\|_V \|v\|_V \quad \forall u, v \in V.$$

The boundness easily follows from Cauchy's inequality with the constant $c_2^a = 1$.

The Lemma of Lax-Milgram [2] delivers existence and uniqueness of the solution u as well als the a priori estimate

$$\frac{1}{c_2^a}\|F\|_{V^*} \leq \|u\|_V \leq \frac{1}{c_1^a}\|F\|_{V^*}.$$

The bilinearform $a(\cdot,\cdot)$ is obviously symmetric, i. e. $a(u,v) = a(v,u)\ \forall u, v \in V$.

2.3 Finite-Element-Discretization

In this part, we will use an admissible uniform triangulation of the domain Ω and apply the Galerkin principle to the weak formulation (2.2) as described in [1] and [2].

Consider the variational problem: find $u \in V_g$ such that

$$a(u, v) = \langle F, v \rangle \quad \forall v \in V_0,$$

where $V_g \subset V$, $V_0 \subset V$ and the dimension of V_g and V_0 is infinite, such as in (2.2). We will now choose the finite-dimensional subspaces $V_h \subset V$ such that $\dim(V_h) < \infty$ and $V_{0h} := V_h \cap V_0$, and a finite-dimensional linear submanifold $V_{gh} := V_h \cap V_g$ where we define an approximation $u_h \in V_{gh}$ of $u \in V_g$ as the solution of the finite-dimensional variational problem (Galerkin scheme): find $u \in V_{gh}$ such that

$$a(u_h, v_h) = \langle F, v_h \rangle \quad \forall v_h \in V_{0h}. \tag{2.5}$$

Note that V_{gh} has to be chosen as a translation of V_{0h}, i. e. $V_{gh} = g_h + V_{0h}$. In order to compute the Galerkin solution, we consider a basis $\{\varphi_i : i \in \bar{\omega}_h\}$ for the space V_h such that

$$V_h := \operatorname{span}\{\varphi_i : i \in \bar{\omega}_h\} = \left\{ v_h = \sum_{i \in \bar{\omega}_h} v_i \varphi_i \right\}$$

as done in [1]. Here $\bar{\omega}_h$ denotes an index set that provides the enumeration of the basis functions. Note, that for the dimension of the space V_h, it holds that

$$\dim(V_h) = |\bar{\omega}_h| = \bar{N}_h = N_h + \partial N_h < \infty,$$

where $N_h = \dim(V_{0h})$ and $\bar{N}_h$ is the number of nodes of the triangulation $\mathcal{T}_h(\bar{\Omega}) := \{\delta_r\}_{r=1}^{R_h}$ of the domain $\bar{\Omega}$ with R_h elements. The triangulation $\mathcal{T}_h(\bar{\Omega})$ is also called a mesh. We choose a uniform triangulation, i. e. all elements δ_r are of the same size.

Example 2.1. Figure 2.1 shows a uniformly triangulized square with $N_h = 81$ nodes and $R_h = 128$ elements.

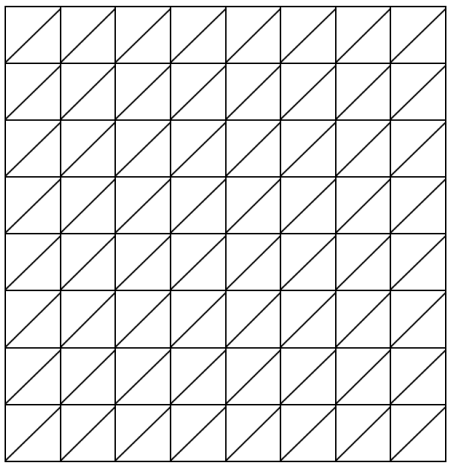


Figure 2.1: Example of a simple uniform mesh

Choosing a set of basis (hat-)functions $\{\varphi_i\}_{i=1}^{N_h}$ as described in [1] and [2], we obtain a system of linear equations (Galerkin system) equivalent to (2.5),

$$K_h \underline{u}_h = \underline{f}_h, \tag{2.6}$$

where

$$\underline{u}_h[j] := u_j, \quad \underline{f}_h[j] := \langle F, \varphi_j \rangle - a(g_h, \varphi_j), \quad K_h[i,j] := a(\varphi_j, \varphi_i), \quad 1 \leq i, j \leq N_h.$$

For the association of vectors and elements of the discrete spaces (the finite-element isomorphism), we write

$$\underline{u}_h \in \mathbb{R}^{N_h} \leftrightarrow u_h \in V_{gh} \subset H^1(\Omega), \quad \underline{v}_h \in \mathbb{R}^{N_h} \leftrightarrow v_h \in V_{0h} \subset H^1(\Omega).$$

In [2], we can see that, for the stiffness matrix K_h, the identity

$$a(w_h, v_h) = (K_h \underline{w}_h, \underline{v}_h)_{\ell_2} \ \ \forall w_h, v_h \in V_{0h}$$

holds, and, for the load vector, we have

$$\langle \hat{F}, v_h \rangle := \langle F, v_h \rangle - a(g_h, v_h) = (\underline{f}_h, \underline{v}_h)_{\ell_2} \ \ \forall v_h \in V_{0h}.$$

The symmetry of the stiffness matrix K_h follows from the symmetry of the bilinearform $a(u, v)$, and since

$$(K_h \underline{v}_h, \underline{v}_h)_{\ell_2} = a(v_h, v_h) \overset{coercive}{\geq} c_1^a ||v_h||^2 > 0 \quad \forall \underline{v}_h \neq 0 \leftrightarrow \ \forall v_h \in V_{0h} : v_h \neq 0$$

we have that K_h is positive definite. Hence, the standard CG method is applicable to (2.6).

2.4 Serial CG Method

The CG method is used to solve symmetric and positive definite systems of linear equations [2].

Algorithm 2.2 (**Serial CG Method**).
Input: matrix K_h, vector $\underline{f}_h$, tolerance ε
Output: vector $\underline{u}^k$ that hods approximate solution
choose $\underline{u}^0$
$\underline{r}^0 = \underline{f}_h - K_h\underline{u}^0$
$\sigma^0 = (\underline{r}^0, \underline{r}^0)$
if $\sigma^0 = 0$ **then stop**
for $k = 0, \quad \sqrt{\sigma^k/\sigma^0} \geq \varepsilon, \quad k = k + 1$ **do**
  $\sigma^k = \left(\underline{r}^k, \underline{r}^k\right)$
  if $k = 0$ **then**
    $\underline{p}^k = \underline{r}^k$
  end
  else
    $\underline{p}^k = \underline{r}^k + \beta^{k-1}\underline{p}^{k-1}$ with $\beta^{k-1} = \sigma^k/\sigma^{k-1}$
  end
  $\underline{v}^k = K_h\underline{p}^k$
  $\underline{u}^{k+1} = \underline{u}^k + \alpha^k\underline{p}^k$ with $\alpha^k = \frac{\sigma^k}{\left(\underline{v}^k, \underline{p}^k\right)}$
  $\underline{r}^{k+1} = \underline{r}^k - \alpha^k\underline{v}^k$
end

Remark 2.3. Although the CG method is a direct solver it is commonly used to solve systems iteratively. The following theorem about its convergence rate holds.

Theorem 2.4. (Convergence Rate of Serial CG)
For the k^{th} iteration, the error $||\underline{u}^k - \underline{u}||_{K_h}$ can be bounded by the initial error $||\underline{u}^0 - \underline{u}||_{K_h}$ as

$$||\underline{u}^k - \underline{u}||_{K_h} \leq 4\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^{2k} ||\underline{u}^0 - \underline{u}||_{K_h}, \tag{2.7}$$

with the condition number $\kappa = \Lambda(K_h)/\lambda(K_h)$, where $\Lambda(K_h)$ and $\lambda(K_h)$ denote the largest and the smallest eigenvalues of the stiffness matrix K_h, respectively. Here

$$||\cdot||_{K_h}^2 = (K_h\cdot, \cdot)_{\ell_2} = a(\cdot, \cdot)$$

is the K_h-energy norm.

For the proof, we refer to [2].

Chapter 3

Parallel CG

This chapter is devoted to the parallelization of the CG method. The first step consists in decomposing the computational domain Ω into P non-overlapping subdomains, where $P \in \mathbb{N}$ is the number of processing units (threads) $\mathbb{P}_i$, $1 \leq i \leq P$. Then each subdomain Ω_i will be allocated to a thread $\mathbb{P}_i$. Finally, each thread will locally solve the system (2.6).

3.1 Domain Decomposition

Let $P \in \mathbb{N}$ be the number of processing units (threads). We want to decompose the already triangulized domain Figure 2.1 into P uniform subdomains Ω_i that are non-overlapping, i. e.

$$\bigcup_{i=1}^{P} \overline{\Omega}_i = \overline{\Omega} \quad \text{and} \quad \Omega_i \cap \Omega_j = \emptyset \quad \text{for } i \neq j.$$

Such subdomains Ω_i are called pairwise disjoint.

Remark 3.1. If Ω is uniformly triangulized such that the length of the catheti of each elements

$$h = \frac{b-a}{2^{N_h}},$$

where $N_h \in \mathbb{N}$ is the total number of elements, we obtain $2^n = P$ subdomains with $n \in \mathbb{N}, n < N_h$, i. e. with a triangulation like this we can ensure that the number of subdomains is a power of 2.

Example 3.2. If we choose $P = 4$ for the mesh in Figure 2.1, we obtain four subdomains Ω_i like it is shown in Figure 3.1.

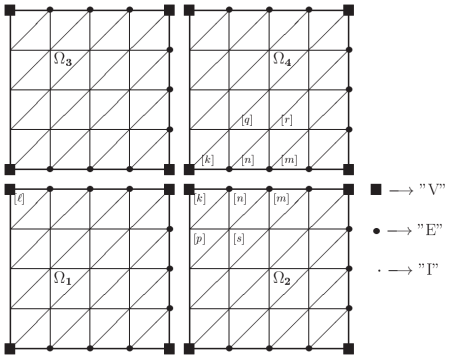


Figure 3.1: Example of a uniformly decomposed non-overlapping mesh [1]

.

Remark 3.3. By numbering the nodes as

- all vertices of each subdomain (crosspoints) N_V,
- all nodes on the edges of each subdomain $N_E := \bigcup_{i=1}^{n_{ed}} N_{E,i}$, where $n_{ed} \in \mathbb{N}$ is the number of nodes on all edges (excluding the nodes N_V) and $N_{E,i}, 1 \leq i \leq n_{ed}$ are the nodes on the edges,
- all inner points $N_I := \bigcup_{i=1}^{n_{in}} N_{I,i}$, where $n_{in} \in \mathbb{N}$ is the number of all inner nodes and $N_{I,i}, 1 \leq i \leq n_{in}$ are the inner nodes,
- all connected nodes $N_C := N_V \cup N_E$,

such that

$$\underbrace{N_V \cup N_E}_{=: N_C} \cup N_I = \bigcup_{i=1}^{P} N_i = N_h$$

is the total number of nodes as pictured above in Figure 3.1, we naturally obtain a block structure in all vectors and matrices [1]. Vectors are now of the form

$$\underline{v}^T = (\underline{v}_V,\ \underline{v}_{E,1}, \ldots, \underline{v}_{E,n_e},\ \underline{v}_{I,1}, \ldots, \underline{v}_{I,P})^T,$$

and the system of linear equations (2.6) can now be rewritten as

$$\begin{pmatrix} K_{VV} & K_{VE} & K_{VI} \\ K_{EV} & K_{EE} & K_{EI} \\ K_{IV} & K_{IE} & K_{II} \end{pmatrix} \begin{pmatrix} \underline{u}_V \\ \underline{u}_E \\ \underline{u}_I \end{pmatrix} = \begin{pmatrix} \underline{f}_V \\ \underline{f}_E \\ \underline{f}_I \end{pmatrix}$$

where $K_{VV}, K_{VE}, K_{EE}, K_{EI}$ are block matrices, and

$$K_{II} = \begin{pmatrix} K_{I,1} & 0 & \ldots & 0 \\ 0 & K_{I,2} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \ldots & 0 & K_{I,P} \end{pmatrix}$$

is a block-diagonal matrix.

3.2 Vector-Types

Each of the nodes in the decomposed mesh now lies on one of the P subdomains $\bar{\Omega}_i$. Every subdomain $\bar{\Omega}_i$ directly corresponds to a thread $\mathbb{P}_i$. Analogously we distribute the entries of the global vectors and matrices to all processing units, such that each thread $\mathbb{P}_i$ now directly corresponds to a local vector $\underline{v}_i$.

The relation between a global vector $\underline{v}$ and a local vector $\underline{v}_i$ can be symbolically represented by the multiplication with a Boolean $N_i \times N$ matrix A_i, where $N = \dim(\underline{v})$ denotes the total number of nodes, and $N_i = \dim(\underline{v}_i)$ is the number of nodes in $\bar{\Omega}_i$ [1].

One possibility to store vectors for parallel computation is to keep the global vector $\underline{v}$ and to multiply with A_i in order to obtain the local vector $\underline{v}_i$.

Definition 3.4. (Type I: accumulated vectors)
A vector $\underline{\mathbf{v}}$ is an accumulated vector if it is stored globally, i. e. the local vector is computed as

$$\underline{\mathbf{v}}_i := A_i \underline{\mathbf{v}} \text{ for all } \mathbb{P}_i;$$

see [1].

Example 3.5. Let us consider the following simple decomposed non-overlapping mesh as shown in Figure 3.2.

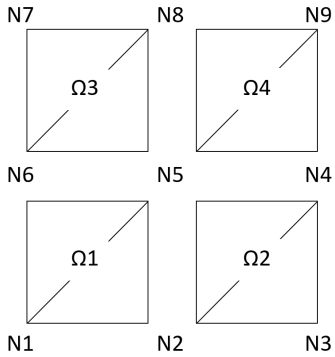


Figure 3.2: Example of a simple decomposed mesh

We can represent the type I vector $\underline{\mathbf{v}}_1$ on Ω_1 by multiplying the global vector $\underline{\mathbf{v}}$ with the corresponding representation matrix A_1:

$$\begin{pmatrix} v_1 \\ v_2 \\ v_5 \\ v_6 \end{pmatrix} = \underline{\mathbf{v}}_1 = A_1 \underline{\mathbf{v}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \end{pmatrix}$$

Remark 3.6. For each inner node, there is exactly one 1-entry in A_i for all i. Each connected node has as many 1-entries in A_i for all i as the number of subdomains it is part of, i. e., 1, 2 or 4.
The diagonal matrix

$$R := \sum_{i=1}^{P} A_i^T A_i$$

represents the amount of occurances of each node in different subdomains.

Alternatively, we can store a parallel vector $\underline{f}$ locally for each processing unit and revert it to obtain a global vector.

Definition 3.7. (Type II: Distributed Vectors)
A vector $\underline{\mathbf{f}}$ is a distributed vector if it is stored locally for each thread $\mathbb{P}$ as $\underline{\mathbf{f}}_i$, i. e., the global vector is computed as

$$\underline{\mathbf{f}} := \sum_{i=1}^{P} A_i^T \underline{\mathbf{f}}_i;$$

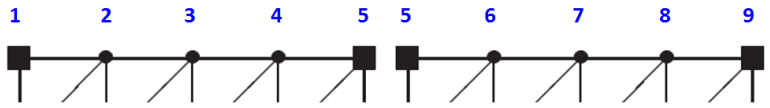
see [1].

Remark 3.8. There are two important differences between these two types of vectors:

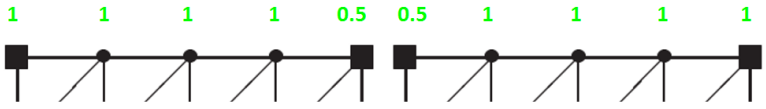
1. A type I vector $\underline{\mathbf{v}}$ holds the complete value of each node, whereas a type II vector $\underline{\mathbf{f}}$ holds only a part of any nodal value for nodes on the interface between subdomains.

2. For type I vectors, the corresponding elements from $H^1(\Omega)$ are functions ($\underline{\mathbf{u}}$ and $\underline{\mathbf{v}}$). For type II vectors, the corresponding elements are functionals ($\underline{\mathbf{f}}$ and $\underline{\mathbf{r}}$).

Example 3.9. For simplicity, we assume a one-dimensional problem with $P = 2$. Let $\underline{\mathbf{v}} = (1, 2, \ldots, 9)^T$ and $\underline{\mathbf{r}} = (1, \ldots, 1)^T$. The accumulated vector would be stored as



resulting in the two subvectors $\underline{\mathbf{v}}_1 = (1, 2, 3, 4, 5)^T$ and $\underline{\mathbf{v}}_2 = (5, 6, 7, 8, 9)^T$, whereas the distributed vector would be stored as



and thus give $\underline{\mathbf{r}}_1 = (1, 1, 1, 1, 0.5)^T$ and $\underline{\mathbf{r}}_2 = (0.5, 1, 1, 1, 1)^T$. Note that the value on the interface does not need to be halved. In fact any sum adding up to the original value is possible.

Definition 3.10. (Matrix Types)
A matrix $\mathbf{M}$ is accumulated if it is stored globally, i. e. the local matrix is computed as

$$\mathbf{M}_i := A_i \mathbf{M} A_i^T \text{ for all } i.$$

A matrix $\mathbf{K}$ is distributed if it is stored locally for each thread $\mathbb{P}$, i. e. the global matrix is computed as

$$\mathbf{K} := \sum_{i=1}^{P} A_i^T \mathbf{K}_i A_i$$

[1].

Remark 3.11. In order to compute the solution of (2.6) in parallel, the processing units $\mathbb{P}_i$ assemble the load vectors $\underline{\mathbf{f}}_i$ and the stiffness matrices $\mathbf{K}_i$ locally. This naturally gives a type II vector and a type II matrix as a result.

In order to derive the parallel CG method, we will need to transform one type of vector into the other.

Property 3.12. (Transformation of Vector-Types)
Let $\underline{W}_i$ be a subvector of any vector $\underline{w}$.

Let $\underline{\mathbf{w}}$ be the corresponding distributed vector. Then the corresponding accumulated vector is given by

$$\underline{\mathbf{w}}_i = A_i \cdot \sum_{i=1}^{P} A_i^T \underline{\mathbf{w}}_i \quad \text{for all } i.$$

Let $\underline{\mathbf{w}}$ be the corresponding accumulated vector. Then the corresponding distributed vector is given by

$$\underline{\mathbf{w}}_i = R^{-1} \underline{\mathbf{w}}_i = \left(\sum_{j=1}^{P} A_j^T A_j \right)^{-1} \underline{\mathbf{w}}_i \quad \text{for all } i.$$

Note that this transformation is not unique. As already mentioned in Example 3.9 any distribution of the original value to the interface values of a type II vector is possible as long as they add up to the original value [1].

Remark 3.13. The transformation of vector-types requires communication between the processing units. Hence this step will consume time if used in an algorithm. Communication is always indicated by the occurence of summations.

3.3 Parallel Arithmetic Operations

After having defined the two different vector-types, we want to examine the arithmetic operations required for the CG method. For the algorithm, we need

- the addition of vectors,
- the multiplication of vectors with a scalar value,
- the (ℓ_2-)inner product of two vectors,
- the multiplication of matrix times vector.

Remark 3.14. Obviously, type I or II vectors can be added just like standard vectors as long as they are of the same type. Adding an accumulated vector to a distributed vector would require transformation of any one of them into the other which would then need communication.
The scalar multiplication of parallel vectors is also not different from the multiplication of standard vectors. However, the other two arithmetic operations need to be studied in more detail.

Property 3.15. (Parallel Inner Product)
The inner product of two parallel vectors is well-defined if and only if the two vector-types are different.

Proof: It can be easily shown that an inner product of same vector-types is not well-defined by the use of the vectors $\underline{\mathbf{v}}$ and $\underline{\mathbf{r}}$ from Example 3.9. It holds that

$$(\underline{\mathbf{v}},\underline{\mathbf{v}}) = \sqrt{\sum_{i=1}^{N} \underline{\mathbf{v}}[i]^2} \neq \sqrt{\sum_{j=1}^{P}\sum_{i=1}^{n} \underline{\mathbf{v}}_j[i]^2} \overset{\text{Ex. } 3.9}{=} \sqrt{\sum_{i=1}^{n} \underline{\mathbf{v}}_1[i]^2 + \sum_{i=1}^{n} \underline{\mathbf{v}}_2[i]^2}$$

as an error occurs at the interface, where the interface value is added twice. Analogously there is an error at the interface of $(\underline{\mathbf{r}},\underline{\mathbf{r}})$, where the interface value is added less than once.

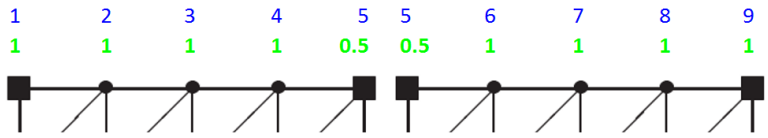
Let $\underline{\mathbf{v}}$ and $\underline{\mathbf{r}}$ be arbitrary, then

$$(\underline{\mathbf{v}},\underline{\mathbf{r}}) = \underline{\mathbf{v}}^T\underline{\mathbf{r}} = \underline{\mathbf{v}}^T \sum_{i=1}^{P} A_i^T \underline{\mathbf{r}}_i = \sum_{i=1}^{P} (\underline{\mathbf{v}}A_i)^T \underline{\mathbf{r}}_i = \sum_{i=1}^{P} (\underline{\mathbf{v}}_i,\underline{\mathbf{r}}_i)\,.$$

Hence the ℓ_2-inner product for a type I vector and a type II vector is well-defined [1].

Remark 3.16. As indicated by the sum over all processing units P, we can see that communication is required for this operation. The expense matches the summation of a real number. Calculating the inner product for vectors of the same type requires transformation and thus additional communication.

Example 3.17. For this example, we will again use the vectors $\underline{\mathbf{v}}$ and $\underline{\mathbf{r}}$ from (Example 3.9). It is clear that for the inner product of the corresponding standard vectors $\underline{v} = (1, 2, \ldots, 9)^T$ and $\underline{r} = (1, \ldots, 1)^T$ the result $(\underline{v}, \underline{r}) = 45$. The same holds true for the parallel vectors. This is pictured below.



We can see that the interface values of the parallel vectors $5 \cdot 0.5 \; + \; 5 \cdot 0.5$ result in the same value as the standard vectors $5 \cdot 1$ at this node.

Property 3.18. (Matrix-Vector Multiplication)
The result of a multiplying a type II matrix $\mathbf{K}$ with a type I vector $\underline{\mathbf{v}}$ is a type II vector $\underline{\mathbf{r}}$.

Proof: This follows from the definitions of the type II matrix (Def. 3.10) and the type I vector (Def. 3.4) [1],

$$\mathbf{K}\underline{\mathbf{v}} \overset{\text{Def 3.10}}{=} \sum_{i=1}^{P} A_i^T \mathbf{K}_i A_i \underline{\mathbf{v}} \overset{\text{Def 3.4}}{=} \sum_{i=1}^{P} A_i^T \underbrace{\mathbf{K}_i \underline{\mathbf{v}}_i}_{:=\underline{\mathbf{r}}_i} = \underline{\mathbf{r}}.$$

Remark 3.19. Analogous to the calculation of the inner product, a type II matrix can only be multiplied with a type I vector, as problems on the interfaces would occur otherwise. For the same reason, the multiplication of a local type II matrix with a local type I vector naturally gives a local type II vector.
If we need to multiply a type II matrix with a type II vector, we have to transform first which requires communication.

The sum over all threads is in this case part of the definition of a type II matrix, and, therefore, does not indicate communication.

Multiplications with type I matrices are not required for parallelizing the CG method as the only occuring matrix is the stiffness matrix $\mathbf{K}$.

3.4 Parallel CG Method

In order to construct the parallel CG method from Algorithm 2.2, we need to use the parallel structures for vectors and matrices from Section 3.2 and the arithmetic operations discussed in Section 3.3.

As already mentioned, we know that functions in the Sobolev space $H^1(\Omega)$ correspond to accumulated vectors, and that functionals are the equivalent of distributed vectors. Further, the stiffness matrix $\mathbf{K}$ is naturally of type II. With this knowledge, we can adjust our input parameters $\mathbf{K}_h, \underline{\mathbf{f}}_h$ and $\underline{\mathbf{u}}^0$ as well as the residual $||\underline{\mathbf{r}}^k||$.

It is also necessary to adjust the search direction $\underline{p}^k$ and the intermediate result $\underline{v}$. Since $\underline{p}^k \leftrightarrow p_h^k \in V_{0h} \subset H^1(\Omega)$ is a function, we choose it as type I $\underline{\mathbf{p}}^k$. As $\underline{v}$ is now the result of a multiplication of a type II matrix with a type I vector, we naturally obtain a type II vector $\underline{\mathbf{v}}$.

In order to compute $||\underline{\mathbf{r}}^k||^2 = (\underline{\mathbf{r}}^k, \underline{\mathbf{r}}^k)$ we need to transform $\underline{\mathbf{r}}^k$ to a type I vector $\underline{\mathbf{s}}^k$. Both computing the inner product and transforming $\underline{\mathbf{r}}^k$ requires communication between the processing units. The allocation $\underline{p}^k = \underline{r}^k$ and $\underline{p}^k = \underline{r}^k + \beta^{k-1}\underline{r}^{k-1}$ in Algorithm 2.2 for the type I vector $\underline{\mathbf{p}}^k$ can now be executed with $\underline{\mathbf{s}}^k$.

The considerations above lead to the parallel CG method; c. f. [1].

Algorithm 3.20. (Parallel CG Method)

Input: matrix $\mathbf{K}_h$, vector $\underline{\mathbf{f}}_h$, tolerance ε
choose $\underline{\mathbf{u}}^0$
$\underline{\mathbf{r}}^0 = \underline{\mathbf{f}}_h - \mathbf{K}_h\underline{\mathbf{u}}^0 \qquad \underline{\mathbf{s}}^0 = \sum_{i=1}^P A_i^T \underline{\mathbf{r}}_i^0$
$\sigma^0 = (\underline{\mathbf{r}}^0, \underline{\mathbf{s}}^0)$
if $\sigma^0 = 0$ **then stop**
for $k = 0, \quad \sqrt{\sigma^k/\sigma^0} \geq \varepsilon, \quad k = k+1$ **do**
  $\sigma^k = \left(\underline{\mathbf{r}}^k, \underline{\mathbf{s}}^k\right)$
  if $k = 0$ **then**
    $\underline{\mathbf{p}}^k = \underline{\mathbf{s}}^k$
  end
  else
    $\underline{\mathbf{p}}^k = \underline{\mathbf{s}}^k + \beta^{k-1}\underline{\mathbf{p}}^{k-1}$ with $\beta^{k-1} = \sigma^k/\sigma^{k-1}$
  end
  $\underline{\mathbf{v}}^k = \mathbf{K}_h\underline{\mathbf{p}}^k$
  $\underline{\mathbf{u}}^{k+1} = \underline{\mathbf{u}}^k + \alpha^k\underline{\mathbf{p}}^k$ with $\alpha^k = \frac{\sigma^k}{(\underline{\mathbf{v}}^k, \underline{\mathbf{p}}^k)}$
  $\underline{\mathbf{r}}^{k+1} = \underline{\mathbf{r}}^k - \alpha^k\underline{\mathbf{v}}^k \qquad \underline{\mathbf{s}}^k = \sum_{i=1}^P A_i^T \underline{\mathbf{r}}_i^k$
end

3.5 Preconditioning

We will now study preconditioning matrices $C^{-1} \in \mathbb{R}^{N_h \times N_h}$ in a parallel setting. We want to apply a preconditioner as a type II matrix $\mathbf{C}^{-1}$. It holds that

$$\mathbf{C}^{-1}\underline{\mathbf{r}} \overset{\text{Def 3.10}}{=} \sum_{i=1}^P A_i^T \mathbf{C}_i^{-1} A_i \sum_{j=1}^P A_j^T \underline{\mathbf{r}}_j = \sum_{i=1}^P A_i^T \mathbf{C}_i^{-1} \underbrace{\left(A_i \sum_{j=1}^P A_j^T \underline{\mathbf{r}}_j \right)}_{\text{Type II} \to \text{Type I}}$$

$$= \sum_{i=1}^P A_i^T \underbrace{\mathbf{C}_i^{-1}\underline{\mathbf{r}}_i}_{=:\underline{\mathbf{s}}_i} = \sum_{i=1}^P A_i^T \underline{\mathbf{s}}_i \overset{\text{Def 3.7}}{=} \underline{\mathbf{s}}.$$

We can see that the multiplication with a preconditioner requires transformation and, thus, also communication. In order to compute $(\mathbf{C}^{-1}\underline{\mathbf{r}}, \underline{\mathbf{r}})$ as the next step in the algorithm, we need another transformation, as well as for computing the search direction $\underline{\mathbf{p}}$.

Algorithm 3.21. (Preconditioned Parallel CG Method) [1]

Input: matrices $\mathbf{K}_h$, $\mathbf{C}$, vector $\underline{\mathbf{f}}_h$, tolerance ε

choose $\underline{\mathbf{u}}^0$

$\underline{\mathbf{r}}^0 = \underline{\mathbf{f}}_h - \mathbf{K}_h\underline{\mathbf{u}}^0$

$\underline{\mathbf{s}}^0 = \sum_{i=1}^{P} A_i^T \underline{\mathbf{r}}_i^0$

$\sigma^0 = \left(\mathbf{C}^{-1}\underline{\mathbf{r}}^0, \underline{\mathbf{s}}^0\right)$

if $\sigma^0 = 0$ **then stop**

for $k = 0, \quad \sqrt{\sigma^k/\sigma^0} \geq \varepsilon, \quad k = k + 1$ **do**

$\underline{\mathbf{w}}^k = \mathbf{C}^{-1}\underline{\mathbf{r}}^k$ //communication: transformation

$\sigma^k = \left(\underline{\mathbf{w}}^k, \underline{\mathbf{s}}^k\right)$ //communication: inner product

if $k = 0$ **then**

$\underline{\mathbf{p}}^k = \sum_{i=1}^{P} A_i^T \underline{\mathbf{w}}_i^k$ //communication: transformation

end

else

$\beta^{k-1} = \sigma^k/\sigma^{k-1}$

$\underline{\mathbf{p}}^k = \sum_{i=1}^{P} A_i^T \underline{\mathbf{w}}_i^k + \beta^{k-1}\underline{\mathbf{p}}^{k-1}$ //communication: transformation

end

$\underline{\mathbf{v}}^k = \mathbf{K}_h\underline{\mathbf{p}}^k$

$\alpha^k = \sigma^k/\left(\underline{\mathbf{v}}^k, \underline{\mathbf{p}}^k\right)$ //communication: inner product

$\underline{\mathbf{u}}^{k+1} = \underline{\mathbf{u}}^k + \alpha^k\underline{\mathbf{p}}^k$

$\underline{\mathbf{r}}^{k+1} = \underline{\mathbf{r}}^k - \alpha^k\underline{\mathbf{v}}^k$

$\underline{\mathbf{s}}^k = \sum_{i=1}^{P} A_i^T \underline{\mathbf{r}}_i^k$ //communication: transformation

end

By observing this final algorithm we can see that communication is needed five times in each iteration.

Chapter 4

C++ Implementation

In this chapter, we will discuss the implementation of a parallel CG method in the sense of Chapter 3. For the sake of simplicity, the implementation was done for the one-dimensional Poisson equation:

Let $f : \Omega \longrightarrow \mathbb{R}$ be a given function, and let $\Omega = (0, 1) \subset \mathbb{R}$ be the unit interval. We consider the boundary value problem:

find $u : \bar{\Omega} \longrightarrow \mathbb{R}$ such that

$$-u''(x) = f(x) \text{ for } x \in \Omega \qquad + \text{ boundary conditions at } x = 0 \text{ and } x = 1.$$

The program supports the incorporation of Dirichlet, Neumann and Robin type boundary conditions.

4.1 Implementation

The program was implemented in C++ using the OpenMP API specification for parallel programming (see: www.openmp.org). The C++-class "Vector.h" by Dr. Clemens Pechstein was used for numerical computations with vectors; see Appendix.The source code is attached to the Appendix as well.

The input-variables for the program are

- $\dim(V_h)$ (for the discretization of $\Omega = (0, 1)$),
- a given real function $f \in L_2(\Omega)$,
- some error threshold $\varepsilon > 0$,
- an initial guess $\underline{u}^0 \leftrightarrow u^0 \in V_g$,
- the number of threads $P \in \mathbb{N}$,
- two boundary conditions g_D, g_N or (g_R, α) at $x = 0$ and $x = 1$.

The output is an approximation of the solution as well as the computed L_2-error provided that the exact solution is known.

4.2 Architecture

This section is about the architecture of the implemented objects, i. e., the data structure used in order to store parallel vectors $\underline{\mathbf{u}}$ and $\underline{\mathbf{v}}$ of type I and type II, respectively. For this part, input-variables are formatted in bold.

First, the domain $\bar{\Omega} = [0, 1]$ is uniformly discretized into $\mathbf{dim}(\mathbf{V_h}) - 1$ subintervals of width $h := 1/\mathbf{dim}(\mathbf{V_h})$. Then the corresponding nodes ($N_h = \mathbf{dim}(\mathbf{V_h})$) are distributed among the $\mathbf{P}$ threads $\mathbb{P}_i$, $1 \leq i \leq \mathbf{P}$, creating $\mathbf{P}$ subdomains $\bar{\Omega}_i$. Assume $P \geq 2$. Note that, for the number of nodes in each of the subdomains $n_i < N_h$ it holds that

$$\sum_{i=1}^{\mathbf{P}} n_i > N_h,$$

since nodes on interfaces are contained in each subdomain. We choose n_i such that $n_i = n_j =: l$ and $r := n_{\mathbf{P}} \leq n_i$ for all $1 \leq i, j < \mathbf{P}$. This distribution is carried out in parallel. The data structure used is a $\mathbf{P}$-dimensional vector, where the i-th entry corresponds to the thread $\mathbb{P}_i$, holding a vector of length n_i that contains the nodes of the subdomain $\bar{\Omega}_i$.

Example 4.1. The vector $\underline{v} \in \mathbb{R}^{N_h}$ in Figure 4.1 holds the nodes of a mesh. Each of threads $\mathbb{P}_i$ holds the nodes of the corresponding subdomain Ω_i that has l nodes, except for the thread $\mathbb{P}_{\mathbf{P}}$, which holds the subdomain $\Omega_{\mathbf{P}}$ that has r nodes.

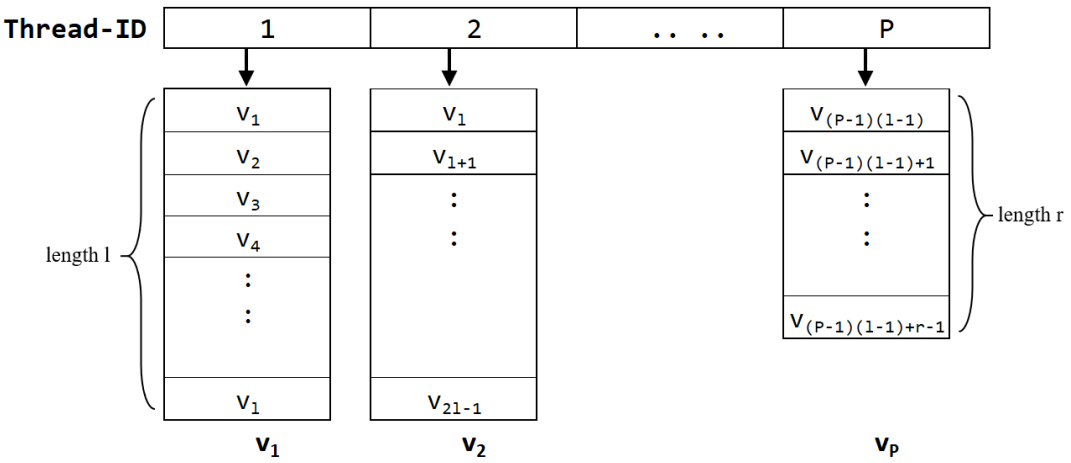


Figure 4.1: Visualization of a vector $v \in \mathbb{R}^{N_h}$ in the data structure used

The data structure above is used for vectors $\underline{\mathbf{u}}$ of type I and vectors $\underline{\mathbf{v}}$ of type II, their data-type differs only semantically. Analogously, matrices $\mathbf{K}$ of type II are stored as submatrices in a $\mathbf{P}$-dimensional vector. Since we consider the one-dimensional problem, the stiffness matrices are tridiagonal matrices, which is used in order to save storage.

The routines for assembling the stiffness matrix $\mathbf{K}_h$ and the load vector $\underline{\mathbf{f}}_h$ (for a given function $\mathbf{f}$) remain sequential. Each thread $\mathbb{P}_i$ assembles a local stiffness

matrix $\mathbf{K}_h^i$ and a local load vector $\underline{\mathbf{f}}_h^i$ for its corresponding subdomain Ω_i in parallel. The boundary conditions are incorporated only on the first and the last thread, as we have a one-dimensional problem. Then, for given inputs $\mathbf{u^0}$ and $\boldsymbol{\varepsilon}$, the parallel CG method is executed using the parallel arithmetic operations described in Section 3.3.

4.3 Tests and Numerical Results

The implementation of the parallel CG method has been tested on the high performance computing cluster RADON 1 at the Johannes Kepler University in Linz; see https://www.ricam.ac.at/hpc/.

For the test cases, the following one-dimensional boundary value problem problem was considered:

find $u : [0, 1] \to \mathbb{R}$ such that

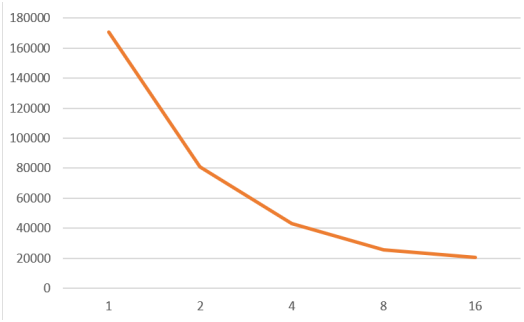
$$-u''(x) = \pi^2 \sin(\pi x), \quad x \in (0, 1)$$

with zero-Dirichlet boundary conditions

$$u(0) = 0 \quad \text{and} \quad u(1) = 0.$$

For this easy example, we know that the exact solution is $u = \sin(\pi x)$. For the numerical solution, the interval [0,1] was discretized into $N_h = 5 \cdot 10^7$ nodes. $\underline{\mathbf{u}}^0 = \underline{0}$ was chosen as initial guess. After assembling the stiffness matrix $\mathbf{K}_h$ and the load vector $\underline{\mathbf{f}}_h$, the parallel CG method was run until the tolerance of $\varepsilon = 10^{-4}$ was reached for the residual error.

The following table and graph illustrate the achieved scaling of the implemented parallel CG method.



number of threads	time in seconds
1	170621
2	80948
4	43130
8	25812
16	20513

Figure 4.2 & Table 4.1: Scaling of test problem

We can see that the we could achieve good (strong) scaling for up to 8 threads, thereafter the scaling effect decreases (weak scaling). This is due to the amount of communication required by the algorithm. Naturally, the expenses of communication worsen for an increasing number of threads. The number of iterations remains constant with respect to the number of threads. For the test problem the CG method took a total of 160560 steps.

We can also see that the CG method for this test problem is much slower than a direct method. Using the Thomas algorithm we would be able to solve this problem within a fractional amount of the time needed by the CG method. The reason for this is the one-dimensional problem itself. Increasing the dimension of the test problem will greatly worsen the time duration of direct methods such as Gaussian elimination, whereas the time duration of the CG method will not increase significantly.

Another reason for the slow behaviour of the CG method is that the model problem is ill-conditioned and we did not use a preconditioner for the sake of simplicity. Applying a parallel preconditioner (e. g. multigrid) to the model problem reduces the number of iterations and therefore the speed of the parallel CG method considerably; see [1] and [3].

Appendix

Source Code

Vector.h

This is the vector class for numerical computations by Dr. Clemens Pechstein.

```
#pragma once
#ifndef _MYVECTOR_H
#define _MYVECTOR_H
/*!
\file     vector.hh
\brief    dynamic numerical vector
\date     30 April 2010
\author   Clemens Pechstein

Institute of Computational Mathematics
Johannes Kepler University Linz, Austria

Tutorials -- Numerical methods for elliptic PDEs
summer semester 2010
*/
#include <cassert>
#include <valarray>
#include <iostream>

class Vector
{
public:
   /// constructor
   Vector(int size = 0) : data_(size) {}

   /// copy constructor
   Vector(const Vector& v) : data_(v.data_) {}

   /// assignment
   Vector& operator= (const Vector& v)
   {
      if (&v != this)
      {
         data_.resize(v.size());
         data_ = v.data_;
      }
      return *this;
```

```
38    }
39
40    /// destructor
41    ~Vector() {}
42
43    /// size/length/dimension of vector
44    int size() const { return (int)(data_.size()); }
45
46    /// change size and set all entries to value
47    void resize(int newsize, const double value = 0) { data_.resize((↩
          size_t)newsize, value); }
48
49    /// const element access
50    const double operator[] (int i) const { assert(0 <= i && i < size↩
          ()); return data_[i]; }
51
52    /// element access
53    double& operator[] (int i) { assert(0 <= i && i < size()); return ↩
          data_[i]; }
54
55    /// set all elements to value
56    Vector& operator= (const double value) { for (int i = 0; i<size();↩
           ++i) { data_[i] = value; } return *this; }
57
58    /// increment
59    Vector& operator+= (const Vector& v)
60    {
61       for (int i = 0; i<size(); ++i) data_[i] += v.data_[i];
62       return *this;
63    }
64
65    /// decrement
66    Vector& operator-= (const Vector& v)
67    {
68       for (int i = 0; i<size(); ++i) data_[i] -= v.data_[i];
69       return *this;
70    }
71
72    /// scale
73    Vector& operator*= (const double factor)
74    {
75       for (int i = 0; i<size(); ++i) data_[i] *= factor;
76       return *this;
77    }
78
79    /// output
80    void print(std::ostream& os) const
81    {
82       os << "[";
83       for (int i = 0; i<size(); ++i) os << ' ' << data_[i];
84       os << " ]";
85    }
86
87 private:
88    std::valarray<double> data_;
89
```

```
90 }; // class Vector
91
92
93
94    /// vector addition
95 inline
96 Vector operator+ (const Vector& v, const Vector& w)
97 {
98    Vector res(v); // copy
99    res += w;
100   return res;
101 }
102
103
104 /// vector subtraction
105 inline
106 Vector operator- (const Vector& v, const Vector& w)
107 {
108   Vector res(v); // copy
109   res -= w;
110   return res;
111 }
112
113
114 /// scale vector
115 inline
116 Vector operator* (const double factor, const Vector& v)
117 {
118   Vector res(v);
119   res *= factor;
120   return res;
121 }
122
123
124 /// Euclidean scalar product
125 inline
126 double inner_product(const Vector& v, const Vector& w)
127 {
128   double res = 0;
129   for (int i = 0; i<(int)v.size(); ++i) res += v[i] * w[i];
130   return res;
131 }
132
133
134
135 /// Euclidean norm
136 inline
137 double l2norm(const Vector& v)
138 {
139   double res = 0.;
140   for (int i = 0; i<(int)v.size(); ++i) res += v[i] * v[i];
141   return sqrt(res);
142 }
143
144
145
```

```
146 /// output
147 inline
148 std::ostream& operator<< (std::ostream& os, const Vector& v)
149 {
150     v.print(os);
151     return os;
152 }
153
154 #endif // _MYVECTOR_H
```

Mesh.h

```
1  /*!
2  \file     Mesh.h
3  \brief    Bachelor Thesis: Parallel CG
4  \date     September 2018
5  \author   Christoph Plakolm
6
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9
10 Mesh.h generates a one-dimensional grid on the domain [0, 1] and ↩
       distributes
11 the domain amongst one or more threads.
12 */
13
14 #pragma once
15 #include <iostream>
16 #include "Vector.h"
17 #include <omp.h>
18 #include <time.h>
19
20 //array that holds vectors
21 typedef Vector* pvec;
22
23 class Mesh
24 {
25 private:
26     int nThr;
27     int nNod;
28     int nEl;
29     double hk;
30     pvec Nodes;
31     double totaltime = 0;
32
33 public:
34     //constructor
35     Mesh(); //default
36     Mesh(int numEl, int numThr); //number of elements in interval ↩
           [0,1] and number of threads
37     ~Mesh() {  }
38
39     //create output
```

```
40     void Out();
41
42     //return length of subvector p
43     int Len(int p)
44     {
45        assert(p >= 0 && p < nThr);
46        return Nodes[p].size();
47     }
48
49     //return time
50     double getTime()
51     {
52        return totaltime;
53     }
54
55     //return width of element
56     double Width()
57     {
58        return hk;
59     }
60
61     //return subvector p
62     Vector SubVec(int p)
63     {
64        assert(p >= 0 && p < nThr);
65        return Nodes[p];
66     }
67
68     //return number of nodes
69     int NumNodes() { return nNod; }
70
71     //return number of threads
72     int NumThrds() { return nThr; }
73
74 };
```

Mesh.cpp

```
1  /*!
2  \file    Mesh.cpp
3  \brief   Bachelor Thesis: Parallel CG
4  \date    September 2018
5  \author  Christoph Plakolm
6
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9
10 Contains routines for Mesh.h.
11 */
12
13 #include "Mesh.h"
14
15
```

```
16
17 Mesh::Mesh()
18 {
19
20 }
21
22
23 Mesh::Mesh(int numEl, int numThr)
24 {
25    time_t tstart, tend;
26    tstart = time(0);
27
28    nEl = numEl;
29    nNod = numEl + 1;
30    hk = 1 / (double)numEl;
31    nThr = numThr;
32
33    //length of each subvector
34    int np = nEl / nThr + 1; //rounded up
35    if (nEl % nThr > 0) np++;
36
37    //length of last subvector
38    int nl = (nEl + nThr) % np;
39    if (nl == 0) nl = np;
40
41    //number of subvectors is equal to the number of threads
42    Nodes = new Vector[nThr];
43
44
45    omp_set_num_threads(nThr);
46    #pragma omp parallel
47    {
48       int id = omp_get_thread_num();
49
50       if (id == nThr - 1)
51       {
52          //fill last subvector
53
54          if (nl == 0) //if last subvector is of size np
55          {
56             //allocate size np
57             Nodes[id] = Vector(np);
58             for (int i = 0; i < np; i++)
59             {
60                Nodes[nThr - 1][i] = hk*id*(np - 1) + i*hk;
61             }
62          }
63          else //last subvector is smaller than other subvectors
64          {
65             //allocate size nl
66             Nodes[id] = Vector(nl);
67             for (int i = 0; i < nl; i++)
68             {
69                Nodes[nThr - 1][i] = hk*id*(np - 1) + i*hk;
70             }
71          }
```

```
72        }
73        else
74        {
75           //fill other subvectors
76
77           //allocate size np
78           Nodes[id] = Vector(np);
79
80           for (int i = 0; i < np; i++)
81           {
82              Nodes[id][i] = hk*id*(np - 1) + i*hk;
83           }
84        }
85     }
86     tend = time(0);
87     totaltime = difftime(tend, tstart);
88  }
89
90
91  void Mesh::Out()
92  {
93     for (int j = 0; j < nThr; j++)
94     {
95        std::cout << "[ ";
96
97        for (int i = 0; i < Len(j); i++)
98        {
99           std::cout << Nodes[j][i];
100          if (i != (Len(j) - 1))
101             std::cout << ", ";
102       }
103
104       std::cout << " ]" << std::endl;
105    }
106    std::cout << std::endl;
107
108
109 }
```

StiffMat.h

```
1  /*!
2  \file     StiffMat.h
3  \brief    Bachelor Thesis: Parallel CG
4  \date     September 2018
5  \author   Christoph Plakolm
6
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9
10 StiffMat.h is the header for a class that generates a stiffness ↩
      matrix for the
11 Poisson problem -u''(x) = f(x).
```

```
12 */
13
14 #pragma once
15 #include <iostream>
16 #include "Vector.h"
17 #include <cmath>
18
19 typedef double Vec2[2];
20 typedef double Mat22[2][2];
21
22 class StiffMat
23 {
24 private:
25    //number of nodes
26    int nNod;
27
28    //Diagonal, Upper diagonal and Lower diagonal entries
29    Vector D, U, L;
30
31
32
33 public:
34
35    //constructor, assembles stiffness matrix
36    StiffMat(Vector pmesh); //pmesh is a subvector of a mesh
37    StiffMat(int numNodes); //create empty StiffMat
38
39    StiffMat(){}
40    ~StiffMat();
41
42    //output
43    void Out();
44
45    //set element
46    void Set(int i, int j, double val);
47
48    //add to element
49    void Add(int i, int j, double val);
50
51    //element access
52    const double operator() (int i, int j) const;
53
54    //get number of Nodes
55    int NumNod() { return nNod; }
56
57    //matrix-vector-multiplication
58    Vector Mult(const Vector& Vec);
59
60 private:
61    //computes element stiffness matrix
62    void ElementStiff(const Vec2& elMesh, Mat22& elMat);
63
64
65 };
```

StiffMat.cpp

```
1  /*!
2  \file     StiffMat.cpp
3  \brief    Bachelor Thesis: Parallel CG
4  \date     September 2018
5  \author   Christoph Plakolm
6
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9
10 Contains the routines for StiffMat.h
11 */
12
13 #include "StiffMat.h"
14
15
16 StiffMat::StiffMat(Vector pmesh)
17 {
18    //number of nodes equals size of vector
19    nNod = pmesh.size();
20
21    //allocate size of diagonal vectors
22    D = Vector(nNod);
23    L = Vector(nNod - 1);
24    U = Vector(nNod - 1);
25
26    Vec2 elMesh;
27    Mat22 elMat;
28
29    //loop over all elements
30    for (int k = 1; k < nNod; k++)
31    {
32       //each element
33       elMesh[0] = pmesh[k - 1];
34       elMesh[1] = pmesh[k];
35
36       //calculate element stiffness matrix for each element
37       ElementStiff(elMesh, elMat);
38
39       //add to diagonal
40       D[k - 1] += elMat[0][0];
41       D[k] += elMat[1][1];
42
43       //set secondary diagonals
44       U[k - 1] = elMat[0][1];
45       L[k - 1] = elMat[1][0];
46    }
47 }
48
49 StiffMat::StiffMat(int numNodes)
50 {
51    nNod = numNodes;
52    D = Vector(numNodes);
```

```
53     U = Vector(numNodes - 1);
54     L = Vector(numNodes - 1);
55     //create empty StiffMat using only the number of nodes
56 }
57
58
59 StiffMat::~StiffMat()
60 {
61 }
62
63
64 void StiffMat::ElementStiff(const Vec2& elMesh, Mat22& elMat)
65 {
66     double hk = elMesh[1] - elMesh[0];
67
68     elMat[0][0] = 1. / hk;
69     elMat[0][1] = -1. / hk;
70     elMat[1][0] = -1. / hk;
71     elMat[1][1] = 1. / hk;
72 }
73
74
75 void StiffMat::Out()
76 {
77     for (int i = 0; i < nNod; i++)
78     {
79         std::cout << "|  ";
80
81         for (int j = 0; j < nNod; j++)
82         {
83             if (abs(i - j) <= 1)
84             {
85                 if (i == j)
86                 {
87                     if (abs(D[i]) < 1e-11)
88                         std::cout << "0";
89                     else std::cout << D[i];
90                 }
91                 if (i == j + 1)
92                 {
93                     if (abs(U[j]) < 1e-11)
94                         std::cout << "0";
95                     else std::cout << U[j];
96                 }
97                 if (j == i + 1)
98                 {
99                     if (abs(L[i]) < 1e-11)
100                        std::cout << "0";
101                    else std::cout << L[i];
102                }
103                std::cout << ",  ";
104            }
105            else
106                std::cout << ".,  ";
107        }
108
```

```
109         std::cout << "  |" << std::endl;
110     }
111     std::cout << std::endl;
112 }
113 
114 
115 void StiffMat::Set(int i, int j, double val)
116 {
117     //check if index is in range and if index is within tridiagonal ↩
            matrix
118     assert(0 <= i && i < nNod && 0 <= j && j < nNod && (abs(i - j) <= ↩
            1));
119 
120     //set value
121     if (i == j + 1)  U[j] = val;
122     if (j == i + 1)  L[i] = val;
123     if (j == i)  D[i] = val;
124 }
125 
126 
127 void StiffMat::Add(int i, int j, double val)
128 {
129     //check if index is in range and if index is within tridiagonal ↩
            matrix
130     assert(0 <= i && i < nNod && 0 <= j && j < nNod && (abs(i - j) <= ↩
            1));
131 
132     //add value
133     if (i == j + 1)  U[j] += val;
134     if (j == i + 1)  L[i] += val;
135     if (j == i)  D[i] += val;
136 }
137 
138 
139 const double StiffMat::operator() (int i, int j) const
140 {
141     //check if index is in range and if index is within tridiagonal ↩
            matrix
142     assert(0 <= i && i < nNod && 0 <= j && j < nNod && (abs(i - j) <= ↩
            1));
143 
144     //return value
145     if (i == j + 1) return U[j];
146     if (j == i + 1) return L[i];
147     if (j == i) return D[i];
148 }
149 
150 Vector StiffMat::Mult(const Vector& Vec)
151 {
152     //check if input vector is of same size as matrix
153     assert(Vec.size() == nNod);
154 
155     //initialize Product-Vector
156     Vector Prod(nNod);
157 
158     //compute first and last entry
```

```
159     Prod[0] = D[0] * Vec[0] + U[0] * Vec[1];
160     Prod[nNod - 1] = L[nNod - 2] * Vec[nNod - 2] + D[nNod - 1] * Vec[↩
            nNod - 1];
161 
162     for (int i = 1; i < nNod - 1; i++)
163     {
164         //compute each entry in between
165         Prod[i] = L[i - 1] * Vec[i - 1] + D[i] * Vec[i] + U[i] * Vec[i ↩
                + 1];
166     }
167 
168     return Prod;
169 }
```

LoadVec.h

```
1  /*!
2  \file     LoadVec.h
3  \brief    Bachelor Thesis: Parallel CG
4  \date     September 2018
5  \author   Christoph Plakolm
6 
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9 
10 LoadVec.h is the header for a class that computes the load vector for
11 the Poisson problem -u''(x) = f(x).
12 */
13 
14 #pragma once
15 #include <iostream>
16 #include "Vector.h"
17 #include <cmath>
18 
19 
20 typedef double(*RealFunction)(double x);
21 typedef double Vec2[2];
22 
23 class LoadVec
24 {
25 private:
26 
27     //number of nodes
28     int nNod;
29 
30     //real function f
31     RealFunction f;
32 
33     //load vector
34     Vector lVec;
35 
36 public:
37 
```

```
38     //constructor, assembles load vector
39     LoadVec(Vector pmesh, RealFunction fun); //pmesh is a subvector of↩
           a mesh
40
41     LoadVec();
42     ~LoadVec();
43
44     //output
45     void Out();
46
47     //set element
48     void Set(int i, double val);
49
50     //element access
51     const double operator[] (int i) const;
52
53     //get Load Vector
54     Vector GetLoad() { return lVec; }
55
56
57 private:
58
59     //approximate given function f by trapezoid rule
60     void ElementLoadVec(const Vec2& elMesh, Vec2& elLoVec);
61 };
```

LoadVec.cpp

```
1  /*!
2  \file     LoadVec.cpp
3  \brief    Bachelor Thesis: Parallel CG
4  \date     September 2018
5  \author   Christoph Plakolm
6
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9
10 Contains routines for LoadVec.h.
11 */
12
13 #include "LoadVec.h"
14 #include <cmath>
15
16
17 LoadVec::LoadVec(Vector pmesh, RealFunction fun)
18 {
19     f = fun;
20     nNod = pmesh.size();
21     lVec = Vector(nNod);
22
23     Vec2 elMesh;
24     Vec2 elLoVec;
25
```

```
26     //loop over all elements
27     for (int k = 1; k < nNod; k++)
28     {
29        //each element
30        elMesh[0] = pmesh[k - 1];
31        elMesh[1] = pmesh[k];
32
33        //calculate approximation of f on each element for element load↩
              vector
34        ElementLoadVec(elMesh, elLoVec);
35
36        //fill load vector
37        lVec[k - 1] += elLoVec[0];
38        lVec[k] += elLoVec[1];
39     }
40  }
41
42
43  LoadVec::LoadVec()
44  {
45  }
46
47
48  LoadVec::~LoadVec()
49  {
50  }
51
52
53  void LoadVec::ElementLoadVec(const Vec2& elMesh, Vec2& elLoVec)
54  {
55     double hk = elMesh[1] - elMesh[0];
56
57     //approximate
58     elLoVec[0] = hk / 2 * f(elMesh[0]);
59     elLoVec[1] = hk / 2 * f(elMesh[1]);
60  }
61
62
63  void LoadVec::Out()
64  {
65     std::cout << "[  ";
66     for (int i = 0; i < nNod - 1; i++)
67     {
68        std::cout << lVec[i] << ",  ";
69     }
70     std::cout << lVec[nNod - 1] << "  ]" << std::endl;
71     std::cout << std::endl;
72  }
73
74
75  const double LoadVec::operator[] (int i) const
76  {
77     //check if index is in range
78     assert(0 <= i && i < nNod);
79
80     //return value
```

```
81     return lVec[i];
82 }
83
84
85 void LoadVec::Set(int i, double val)
86 {
87     //check if index is in range
88     assert(0 <= i && i < nNod);
89
90     //set value
91     lVec[i] = val;
92 }
```

CG.h

```
1  /*!
2  \file    CG.h
3  \brief   Bachelor Thesis: Parallel CG
4  \date    September 2018
5  \author  Christoph Plakolm
6
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9
10 CG.h is the header for a class that computes the solution of a ↩
       discretized system of linear equations
11 using the conjugate gradient method in parallel.
12 */
13
14 #pragma once
15 #include <iostream>
16 #include <omp.h>
17 #include "Vector.h"
18 #include "Mesh.h"
19 #include "StiffMat.h"
20 #include "LoadVec.h"
21 #include <math.h>
22 #include <time.h>
23
24 //Reminder:
25 //type 1 holds accumulated values at interfaces
26 //type 2 holds distributed values at interfaces
27
28 //neutral array that holds Vectors
29 typedef Vector* pvec;
30
31 //type 1 vectors
32 typedef Vector* type1;
33
34 //type 2 vectors
35 typedef Vector* type2;
36
37 //array that holds matrices of type 2
```

```
38 typedef StiffMat* pmat2;
39
40
41 class CG
42 {
43 private:
44
45    Mesh mesh;
46
47    pmat2 stiff;
48    type2 load;
49    RealFunction f;
50
51    //starting point for CG
52    type1 u0;
53
54    //number of threads
55    int nThr;
56
57    //tolerance
58    double tol;
59
60    //solution
61    type1 sol;
62
63    //flag if active dirichlet boundary
64    bool d0 = false;
65    bool d1 = false;
66    double v0, v1; //dirichlet values
67
68    double totaltime = 0;
69
70 public:
71
72    //constructor
73    CG(Mesh mesh1D, RealFunction fun, double tolerance, Vector start);
74
75    CG();
76    ~CG();
77
78    //output
79    void printSolution();
80    void pVecOut(pvec& vec)
81    {
82       for (int j = 0; j < nThr; j++)
83       {
84          std::ostream& os = std::cout;
85          vec[j].print(os);
86       }
87       std::cout << std::endl;
88    }
89    void pVecOut(const pvec& vec)
90    {
91       for (int j = 0; j < nThr; j++)
92       {
93          std::ostream& os = std::cout;
```

```
94          vec[j].print(os);
95       }
96       std::cout << std::endl;
97    }
98    void pMatOut(pmat2& mat)
99    {
100      for (int j = 0; j < nThr; j++)
101      {
102         mat[j].Out();
103      }
104      std::cout << std::endl;
105   }
106
107   //calculate a posteriori error
108   double L2error(RealFunction exact);
109
110
111
112private:
113
114   //Parallel computations↩
         ==========
115   pvec pvecAlloc(); //allocate storage space for pvec
116   pmat2 pmatAlloc(); //allocate storage space for pmat
117
118   //conversions
119   void Vector_to_Type1(const Vector& vec, type1& vec_type1);
120   void type2_to_type1(type2& vec_conv);
121
122   //arithmetic operations
123   void Mult(const pmat2& mat, const type1& vec, type2& vec_prod);
124   double L2_IP(const type2& vec2, const type1& vec1);
125   void Add(const pvec& vec1, pvec& vec2);
126   void Mult(const pvec& vec, double alpha, pvec& vec_prod);
127
128   //assembling
129   void AssembleStiff();
130   void AssembleLoad();
131   //↩
         ==========↩
132
133
134
135   //enter boundary conditions via console
136   void EnterBC();
137
138   //apply boundary conditions
139   void Robin0(double alpha, double g);
140   void Robin1(double alpha, double g);
141   void Neumann0(double du);
142   void Neumann1(double du);
143   void Dirichlet0(double u);
144   void Dirichlet1(double u);
145
146   //run CG method
```

```
147     void Run();
148
149 };
```

CG.cpp

```
1  /*!
2  \file    CG.cpp
3  \brief   Bachelor Thesis: Parallel CG
4  \date    September 2018
5  \author  Christoph Plakolm
6
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9
10 Contains definitions of memberfunctions of CG.h as well as most of ↩
       the parallel routines.
11 */
12
13 #include "CG.h"
14
15
16 CG::CG(Mesh mesh1D, RealFunction fun, double tolerance, Vector start)
17 {
18     //check if length of vector equals number of Nodes
19     assert(start.size() == mesh1D.NumNodes());
20
21     time_t t_assemble, tass_end, t_cg, tcg_end;
22
23     t_assemble = time(0);
24
25     mesh = mesh1D;
26     nThr = mesh.NumThrds();
27     f = fun;
28     tol = tolerance;
29
30     sol = pvecAlloc();
31
32     u0 = pvecAlloc();
33     Vector_to_Type1(start, u0);
34
35     AssembleStiff();
36     AssembleLoad();
37
38     tass_end = time(0);
39     totaltime += difftime(tass_end, t_assemble);
40
41     EnterBC();
42
43     t_cg = time(0);
44
45     Run();
46
```

```
47    tcg_end = time(0);
48    totaltime += difftime(tcg_end, t_cg);
49    totaltime += mesh.getTime();
50
51    //printSolution();
52    std::cout << "done in " << totaltime << " seconds with L2 error ";
53 }
54
55
56 void CG::Run()
57 {
58    //r0 = fh - Kh*u0
59    type2 residual = pvecAlloc();
60    Mult(stiff, u0, residual);
61    Mult(residual, -1, residual);
62    Add(load, residual);
63
64    //set size of solution-vector
65    sol = pvecAlloc();
66
67    //convert residual to type 1 vector (required for inner product)
68    type1 s = pvecAlloc();
69    omp_set_num_threads(nThr);
70    #pragma omp parallel
71    {
72       int id = omp_get_thread_num();
73       s[id] = residual[id];
74    }
75    type2_to_type1(s);
76
77    //set p = s
78    type1 p = pvecAlloc();
79    omp_set_num_threads(nThr);
80    #pragma omp parallel
81    {
82       int id = omp_get_thread_num();
83       p[id] = s[id];
84    }
85
86    //calculate (r,r) = |r|^2
87    double sigma0 = L2_IP(residual, s);
88
89    double sigma = sigma0;
90    double sigmaOld;
91
92    for (int k = 0; sqrt(sigma / sigma0) >= tol; k++)
93    {
94       if (k != 0)
95       {
96          sigmaOld = sigma;
97
98          //sigma = (r,r) = |r|^2
99          sigma = L2_IP(residual, s);
100
101         if(k%200 == 0) //test
```

```
102             std::cout << "sigma = " << sigma << " | crit = " << sqrt(↩
                    sigma / sigma0) << std::endl;
103
104         double beta = sigma / sigmaOld;
105
106         //p = s + beta*p
107         Mult(p, beta, p);
108         Add(s, p);
109
110       }
111       // v = Kh*p
112       type2 v = pvecAlloc();
113       Mult(stiff, p, v);
114
115       double alpha = sigma / L2_IP(v, p);
116
117       //std::cout << "alpha " << alpha << std::endl;
118       //std::cin >> stop;
119
120       //u = u + alpha*p
121       type1 u_temp = pvecAlloc();
122       omp_set_num_threads(nThr);
123       #pragma omp parallel
124       {
125          int id = omp_get_thread_num();
126       }
127       Mult(p, alpha, u_temp);
128       Add(u_temp, sol);
129
130       //r = r - sigma*v;
131       //use u_temp as temporary space for r
132       Mult(v, (-1)*alpha, u_temp);
133
134       Add(u_temp, residual);
135       delete[] u_temp;
136
137       //convert residual to type 1
138       omp_set_num_threads(nThr);
139       #pragma omp parallel
140       {
141          int id = omp_get_thread_num();
142          s[id] = residual[id];
143       }
144       type2_to_type1(s);
145
146       delete[] v;
147    }
148    //test
149    std::cout << "sigma = " << sigma << " | crit = " << sqrt(sigma / ↩
          sigma0) << std::endl;
150    delete[] s;
151    delete[] p;
152
153    //add dirichlet values to solution vector
154    if (d0) //if there is an active dirichlet boundary at 0
155    {
```

```
156         sol[0][0] = v0;
157     }
158     if (d1) //if there is an active dirichlet boundary at 1
159     {
160         int len = mesh.Len(nThr - 1);
161         sol[nThr - 1][len - 1] = v1;
162     }
163 }
164 
165 
166 void CG::printSolution()
167 {
168     pVecOut(sol);
169 }
170 
171 
172 void CG::EnterBC()
173 {
174     //order of implementing boundary conditions does not matter in 1D
175 
176     int b0 = 0;
177     int b1 = 0;
178     double u, alpha;
179 
180     std::cout << "Enter boundary conditions at 0" << std::endl;
181     std::cout << "Press 1 for Dirichlet boundary" << std::endl;
182     std::cout << "Press 2 for Neumann boundary" << std::endl;
183     std::cout << "Press 3 for Robin boundary" << std::endl;
184     std::cin >> b0;
185     assert(b0 > 0 && b0 < 4 && (b0 % 1 == 0)); //integers from 1 to 3
186     if (b0 == 1) //Dirichlet
187     {
188         std::cout << "u(0) = ";
189         std::cin >> u;
190         std::cout << std::endl;
191         Dirichlet0(u);
192         v0 = u;
193         d0 = true;
194     }
195     if (b0 == 2) //Neumann
196     {
197         std::cout << "u'(0) = ";
198         std::cin >> u;
199         std::cout << std::endl;
200         Neumann0(u);
201     }
202     if (b0 == 3) //Robin
203     {
204         std::cout << "-u'(0) + alpha * u(0) = g0" << std::endl;
205         std::cout << "alpha = ";
206         std::cin >> alpha;
207         std::cout << std::endl;
208         std::cout << "g0 = ";
209         std::cin >> u;
210         Robin0(alpha, u);
211     }
```

```
212
213
214     std::cout << "Enter boundary conditions at 1" << std::endl;
215     std::cout << "Press 1 for Dirichlet boundary" << std::endl;
216     std::cout << "Press 2 for Neumann boundary" << std::endl;
217     std::cout << "Press 3 for Robin boundary" << std::endl;
218     std::cin >> b1;
219     assert(b0 > 0 && b0 < 4 && (b0 % 1 == 0));
220     if (b1 == 1) //Dirichlet
221     {
222        std::cout << "u(1) = ";
223        std::cin >> u;
224        std::cout << std::endl;
225        Dirichlet1(u);
226        v1 = u;
227        d1 = true;
228     }
229     if (b1 == 2) //Neumann
230     {
231        std::cout << "u'(1) = ";
232        std::cin >> u;
233        std::cout << std::endl;
234        Neumann1(u);
235     }
236     if (b1 == 3) //Robin
237     {
238        std::cout << "-u'(1) + alpha * u(1) = g1" << std::endl;
239        std::cout << "alpha = ";
240        std::cin >> alpha;
241        std::cout << "g1 = ";
242        std::cin >> u;
243        Robin1(alpha, u);
244        std::cout << std::endl;
245     }
246 }
247
248
249 void CG::Dirichlet0(double u)
250 {
251     //set first column and first row of stiffness matrix
252     stiff[0].Set(0, 0, 1);
253     stiff[0].Set(0, 1, 0);
254     stiff[0].Set(1, 0, 0);
255
256     //set first entry of load vector
257     load[0][0] = 0;
258
259     //add to second entry of load vector
260     load[0][1] += u / mesh.Width();
261
262     //flag dirichlet boundary
263     d0 = true;
264 }
265 void CG::Dirichlet1(double u)
266 {
267     int len = mesh.Len(nThr - 1); //length of last subvector
```

```
268
269     //set last column and last row of stiffness matrix
270     stiff[nThr - 1].Set(len - 1, len - 1, 1);
271     stiff[nThr - 1].Set(len - 1, len - 2, 0);
272     stiff[nThr - 1].Set(len - 2, len - 1, 0);
273
274     //set last entry of load vector
275     load[nThr - 1][len - 1] = 0;
276
277     //add to second last entry of load vector
278     load[nThr - 1][len - 2] += u / mesh.Width();
279
280     //flag dirichlet boundary
281     d1 = true;
282 }
283
284 void CG::Neumann0(double du)
285 {
286     //add to first entry of load vector
287     load[0][0] += du;
288 }
289 void CG::Neumann1(double du)
290 {
291     int len = mesh.Len(nThr - 1); //length of last subvector
292
293     //add to last entry of load vector
294     load[nThr - 1][len - 1] += du;
295 }
296
297 void CG::Robin0(double alpha, double g)
298 {
299     //add to first entry of load vector
300     load[0][0] += g;
301
302     double addval = alpha*mesh.Width();
303
304     //add to first entry of stiffness matrix
305     stiff[0].Set(0, 0, addval);
306 }
307 void CG::Robin1(double alpha, double g)
308 {
309     int len = mesh.Len(nThr - 1); //length of last subvector
310
311     //add to last entry of load vector
312     load[nThr - 1][len - 1] += g;
313
314     double addval = alpha*mesh.Width();
315
316     //add to last entry of stiffness matrix
317     stiff[nThr - 1].Set(len - 1, len - 1, addval);
318 }
319
320
321
322 CG::CG()
323 {
```

```
}


CG::~CG()
{
    delete[] u0;
    delete[] sol;
    delete[] load;
    delete[] stiff;
    //mesh deconstruct
}



//Parallel computations↩
    ==========================================================


//allocate storage for pvec or pmat
pvec CG::pvecAlloc()
{
    pvec newVec = new Vector[nThr];

    omp_set_num_threads(nThr);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        newVec[id] = Vector(mesh.Len(id));
    }
    return newVec;
}
pmat2 CG::pmatAlloc()
{
    pmat2 newMat = new StiffMat[nThr];

    omp_set_num_threads(nThr);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        newMat[id] = StiffMat(mesh.Len(id));
    }
}

//conversion of standard Vector into Vector1
void CG::Vector_to_Type1(const Vector& vec, type1& vec_type1)
{
    omp_set_num_threads(nThr);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();

        int start = 0;
        int j = 0;

        //define starting point for each subVector
        for (j; j < id; j++)
        {
```

```
379             start += mesh.Len(j) - 1;
380         }
381
382         //fill subVectors
383         for (int i = 0; i < mesh.Len(id); i++)
384         {
385             vec_type1[id][i] = vec[start + i];
386         }
387     }
388 }
389
390
391 //Conversion of type2 into type1
392 void CG::type2_to_type1(type2& vec_conv)
393 {
394     omp_set_num_threads(nThr);
395     #pragma omp parallel
396     {
397         int id = omp_get_thread_num();
398
399         //add first entries of i-th subvector to last entries of (i-1)-th subvector
400         if (id != 0)
401         {
402             //vec_conv[id - 1][mesh.Len(id-1) - 1] += vec_conv[id][0];
403             vec_conv[id - 1][mesh.Len(id - 1) - 1] += vec_conv[id][0];
404         }
405
406         //wait
407         #pragma omp barrier
408
409         //add last entries of i-th subvector to first entries of (i+1)-th subvector
410         if (id != nThr - 1)
411         {
412             //vec_conv[id + 1][0] += vec_conv[id][mesh.Len(id) - 1]/2;
413             vec_conv[id + 1][0] = vec_conv[id][mesh.Len(id)-1];
414         }
415     }
416 }
417
418
419 //Matrix2 times Vector1 Multiplication
420 void CG::Mult(const pmat2& mat, const type1& vec, type2& vec_prod)
421 {
422     omp_set_num_threads(nThr);
423     #pragma omp parallel
424     {
425         //check, if dimensions of matrix and vector are equal
426         int id = omp_get_thread_num();
427         assert(vec[id].size() == mat[id].NumNod());
428
429         //multiply matrix times vector for each thread
430         vec_prod[id] = mat[id].Mult(vec[id]);
431     }
432 }
```

```
433
434
435 //L2 inner product
436 double CG::L2_IP(const type2& vec2, const type1& vec1)
437 {
438    double ip = 0.;
439
440    omp_set_num_threads(nThr);
441    #pragma omp parallel // private(i)
442    {
443       //check, if both vectors are of equal size
444       int id = omp_get_thread_num();
445       assert(vec2[id].size() == vec1[id].size());
446
447
448
449       #pragma omp for reduction (+ : ip)
450       for (int i = 0; i < nThr; i++)
451       {
452          ip = ip + inner_product(vec2[i], vec1[i]);
453       }
454    }
455    return ip;
456 }
457
458
459 //addition of two vectors
460 void CG::Add(const pvec& vec1, pvec& vec2)
461 {
462    omp_set_num_threads(nThr);
463    #pragma omp parallel
464    {
465       int id = omp_get_thread_num();
466
467       //check if both vectors are of equal size
468       assert(vec2[id].size() == vec1[id].size());
469
470       for (int i = 0; i < mesh.Len(id); i++)
471       {
472          vec2[id][i] = vec2[id][i] + vec1[id][i];
473       }
474    }
475 }
476
477
478 //scalar multiplication of a vector
479 void CG::Mult(const pvec& vec, double alpha, pvec& vec_prod)
480 {
481    omp_set_num_threads(nThr);
482    #pragma omp parallel
483    {
484       int id = omp_get_thread_num();
485
486       for (int i = 0; i < mesh.Len(id); i++)
487       {
488          vec_prod[id][i] = alpha*vec[id][i];
```

```
489         }
490     }
491 }
492 
493 
494 void CG::AssembleLoad()
495 {
496     load = new Vector[nThr];
497 
498     omp_set_num_threads(nThr);
499     #pragma omp parallel
500     {
501         int id = omp_get_thread_num();
502 
503         //assemble load vector for each thread
504         LoadVec LV(mesh.SubVec(id), f);
505         load[id] = LV.GetLoad();
506     }
507 }
508 
509 
510 void CG::AssembleStiff()
511 {
512     stiff = new StiffMat[nThr];
513 
514     omp_set_num_threads(nThr);
515     #pragma omp parallel
516     {
517         int id = omp_get_thread_num();
518 
519         //assemble stiffness matrix for each thread
520         stiff[id] = StiffMat(mesh.SubVec(id));
521     }
522 }
523 
524 double CG::L2error(RealFunction exact)
525 {
526     double err = 0;
527 
528     //Vector function(mesh.NumNodes());
529 
530     //pvec fun = Vector_to_Type1(function);
531 
532     omp_set_num_threads(nThr);
533     #pragma omp parallel
534     {
535         int id = omp_get_thread_num();
536         int len = mesh.Len(id);
537         double dist = 0;
538 
539         //as type 1 vectors hold the same value twice, we do not sum ↩
                over the first entry of each subvector
540         for (int i = 1; i < len; i++)
541         {
542             dist += (exact(mesh.SubVec(id)[i]) - sol[id][i])*(exact(mesh↩
                    .SubVec(id)[i]) - sol[id][i]);
```

```
543         }
544
545         //add the difference of the first entries of the first ↩
                subvectors
546         if (id == 0)
547         {
548             dist += (exact(mesh.SubVec(0)[0]) - sol[0][0])*(exact(mesh.↩
                    SubVec(0)[0]) - sol[0][0]);
549         }
550
551         #pragma omp for reduction (+ : err)
552         for (int i = 0; i < nThr; i++)
553         {
554             err = err + dist;
555         }
556     }
557     return sqrt(err);
558 }
```

FEM_parallel.cpp

This is the file containing `int main()`.

```
1  /*!
2  \file     FEM_parallel.cpp
3  \brief    Bachelor Thesis: Parallel CG
4  \date     September 2018
5  \author   Christoph Plakolm
6
7  Institute of Computational Mathematics
8  Johannes Kepler University Linz, Austria
9
10 FEM_parallel.cpp contains int main() and is opening point for the ↩
       console application.
11 */
12
13 #include <iostream>
14 #include "Vector.h"
15 #include "Mesh.h"
16 #include "StiffMat.h"
17 #include "LoadVec.h"
18 #include <cmath>
19 #include "CG.h"
20 #include <time.h>
21
22 typedef double(*RealFunction)(double x);
23
24 double PI = 355. / 113;
25
26 double fun(double x)
27 {
28     //return PI*PI*sin(PI*x); //1
29     return 10000*PI*PI*sin(100*PI*x); //2
```

```
30     //return x; //3
31 }
32
33 double exact(double x)
34 {
35     //return sin(PI*x); //1, u(0) = 0 = u(1)
36     return sin(100*PI*x); //2, u(0) = 0 = u(1)
37     //return x / 6. - x*x*x / 6.; //3, u(0) = 0 = u(1)
38     //return 7 * x / 6. - x*x*x / 6.; //3, u(0) = 0, u(1) = 1
39 }
40
41 using namespace std;
42
43 void testCG(int NumberNodes, int NumberThreads, double eps)
44 {
45     Vector v1(NumberNodes);
46     for (int i = 0; i < NumberNodes; i++) v1[i] = .0;
47
48     Mesh m1(NumberNodes - 1, NumberThreads);
49     CG test(m1, fun, eps, v1);
50     cout << test.L2error(exact) << endl;
51 }
52
53
54
55 int main()
56 {
57     int NumberNodes;
58     cout << "enter number of nodes: "; cin >> NumberNodes;
59     int NumberThreads;
60     cout << "enter number of threads: "; cin >> NumberThreads;
61     double eps;
62     cout << "enter stopping criterion: "; cin >> eps;
63     cout << endl;
64
65     testCG(NumberNodes, NumberThreads, eps);
66
67
68
69     int stop;
70     cin >> stop;
71      return 0;
72
73
74     //Tests:
75     // 1,000,001 Nodes, .00001 tolerance, 4 threads, 2298 sec (38.3 ↩
          min), .00300693 error
76     //
77 }
```

Bibliography

[1] Craig C. Douglas, Gundolf Haase, Ulrich Langer. *A Tutorial on Elliptic PDE Solvers and their Parallelization.* SIAM Series "Software, Environments, Tools", Philadelphia, 2003

[2] Walter Zulehner. *Numerische Mathematik. Eine Einführung anhand von Differentialgleichungsproblemen. Band 1: Stationäre Probleme.* Birkhäuser, Basel, 2008

[3] Wolfgang Hackbusch. *Multi-Grid Methods and Applications.* Springer, Berlin, Heidelberg, 1985