VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Applied Mathematics

Parallelizations of TFETI–1 coarse problem

Paralelizace hrubého problému TFETI–1 metody

2016 Jakub Kružík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Zadání bakalářské práce

Student: **Jakub Kružík**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 1103R031 Výpočetní matematika

Téma: Paralelizace hrubého problému TFETI-1 metody
Parallelizations of TFETI-1 coarse problem

Jazyk vypracování: čeština

Zásady pro vypracování:

Metody FETI typu jsou velmi úspěšné k řešení obrovských inženýrských úloh. Důvodem je to, že dualita redukuje velkou primární úlohu na menší duální, relativně dobře podmíněnou konvexní iteračně řešenou úlohu kvadratického programování (QP). Total-FETI-1 (TFETI-1) zjednodušuje výpočet inverze matice tuhosti podoblastí tím, že používá Lagrangeových multiplikátorů nejen k lepení podoblastí na rozhraních rozřezání, nýbrž i k předepsání Dirichletových okrajových podmínek. Tato metoda může být dokonce efektivnější než původní FETI-1. Náš výzkum se zabývá vývojem škálovatelných FETI algoritmů pro kontaktní úlohy kombinujíce klasický FETI přístup s algoritmy pro úlohy QP s jednoduchým omezením se známou rychlostí konvergence dané číslem podmíněnosti (QPMPGP, SMALBE) a jejich paralelní testování. Jednou z nejobtížnějších fází je řešení úloh jednotlivých podoblastí, které můžou být prováděny paralelně bez jakékoliv koordinace, díky čemuž může být dosaženo vysoké paralelní škálovatelnosti. Rostoucí počet podoblastí sice snižuje velikost jednotlivých podúloh, což vede ke kratším časům pro faktorizaci matic tuhostí podoblastí a následných přímých a zpětných substitucí během násobení pseudoinverzí, ale na druhou stranu zvyšující se počet podoblastí při fixovaném diskretizačním parametru zvětšuje duální dimenzi a dimenzi jádra matice tuhosti a tím pádem i dimenzi hrubého problému, což vede k nárůstu časů pro tzv. duální operace a aplikace ortogonálních projektorů.
Cílem práce je analýza různých způsobů paralelizací TFETI-1 hrubého problému, distribuce dat a paralelní implementace nejnáročnějších operací.

Seznam doporučené odborné literatury:

[1] Dostál, D. Horák: Scalability and FETI based algorithm for large discretized variational inequalities, Mathematics and Computers in Simulation 61, (3-6), 2003, 347-357.
[2] Z. Dostál, D. Horák a R. Kučera: Total FETI - an easier implementable variant of the FETI method for numerical solution of elliptic PDE, Commun. in Numerical Methods in Engineering 22, 2006, 1155-1162.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. David Horák, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016

doc. RNDr. Jiří Bouchala, Ph.D.
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

I hereby declare that this bachelor's thesis was written by myself. I have referenced all the sources and publications I have drawn upon.

Ostrava, 29th April 2016 .....

I would like to express my thanks to the supervisor of my thesis Ing. David Horák, Ph.D. for his invaluable help, explanations, advices and suggestions that certainly made this thesis much better.

I am very grateful to Ing. Václav Hapla for all of his helpful advices and explanations.

My thanks also belongs to the entire PERMON team for all their help, and for creating a working environment that is a joy to be part of.

For the explanation of certain geometrical and physical aspects of the FETI methods, my thanks belongs to Ing. Alexandros Markopoulos, Ph.D.

Heartfelt thanks goes to my family and friends for their continued support.

I would also like to thank my employer IT4Innovations national supercomputing center, VSB – Technical University of Ostrava for providing me with the opportunities to learn and to work on exciting projects.

This work made use of the facilities of ARCHER, the UK's national high-performance computing service, provided by The Engineering and Physical Sciences Research Council (EPSRC), The Natural Environment Research Council (NERC), EPCC, Cray Inc. and The University of Edinburgh.

Abstrakt

Metody založené na FETI, používané pro řešení eliptických parcialních diferencialních rovnic, představují velmi úspěšnou třídu metod dekompozice oblasti, které se používají pro paralelizaci dobře známých metod konečných prvků. Původní problém ve FETI methodách je rozdělen na menší problémy definované na podoblastech. Díky tomu, že se podoblasti nepřekrývají, můžeme menší problémy nezávisle na sobě řešit paralelně. Počet podoblastí cheme zvyšovat tak, aby se menší problémy řešily rychleji. To ale zároveň vede k růstu velikosti hrubého problému. Pro složité problémy je navíc potřeba řešit hrubý problém mnohokrát. Díky tomu je potřeba najít řešení hrubého problému co nejefektivněji. Tato práce se zabývá paralelnímy strategiemi řešení hrubého problému TFETI–1 metody.

Klíčová slova: TFETI–1, FETI, hrubý problém, paralelizace, PERMON, PETSc, kvadratické programování, sdružené gradienty, přímé řešiče, communication avoiding, communication hiding

Abstract

The FETI based methods, used for the solution of elliptical partial differential equations, form a highly successful class of domain decomposition methods used for parallelization of well known finite element methods. In the FETI methods we partition the original problem into smaller problems defined on subdomains. Since the subdomains are non-overlapping we can naturally solve the smaller problems independently in parallel. We want to increase the number of subdomains so that the smaller problems are solved faster. This however leads to the increase in the size of the coarse problem. Moreover, for complex problems, the number of coarse problem solutions needed can be very high. Therefore, it is important to find the solution of the coarse problem efficiently. This thesis deals with parallelization strategies of the TFETI–1 coarse problem.

Keywords: TFETI–1, FETI, coarse problem, parallelization, PERMON, PETSc, quadratic programming, conjugate gradients, direct solver, communication avoiding, communication hiding

Contents

List of Symbols and Abbreviations

CG	– Conjugate Gradient
CP	– Coarse Problem
DDM	– Domain Decomposition Method
FEM	– Finite Element Method
FETI	– Finite Element Tearing and Interconnecting
HPC	– High Performance Computing
KKT	– Karush–Kuhn–Tucker
PCG	– Preconditioned Conjugate Gradient
PDE	– Partial Differential Equation
PipePCG	– Pipelined Preconditioned Conjugate Gradient
QP	– Quadratic Programming
SPD	– Symmetric Positive Definite
TFETI	– Total FETI – Total Finite Element Tearing and Interconnecting
∇	– Gradient
$\boldsymbol{I}$	– Identity matrix
$\mathcal{K}$	– Krylov subspace
Im	– Matrix image
Ker	– Matrix kernel (nullspace)
$\boldsymbol{O}$	– Null matrix
$\boldsymbol{o}$	– Null vector
$\mathbb{R}^n$	– Real coordinate space of n dimensions

List of Figures

List of Tables

List of Algorithms

1 Introduction

Many problems in physics can be formulated in terms of partial differential equations (PDEs) with boundary conditions, commonly known as boundary value problems. For solution of these, a group of methods known as the finite element methods (FEMs) was developed. For the parallelization of FEMs, domain decomposition methods (DDMs) were introduced. One of these methods is known as FETI–1 (or equivalently FETI).

The finite element tearing and interconnecting (FETI) method was introduced in 1991 by Fahrat and Roux [1] as a method for parallel solution of elliptical PDEs. Main idea of the FETI based methods is that a discretized domain is partitioned (teared) into several non-overlapping subdomains and continuity of solution across subdomain interfaces is enforced by the Lagrange multipliers (interconnecting). Then the problem on each subdomain can be solved independently in parallel.

The total finite element tearing and interconnecting (TFETI–1, TFETI) method [2] introduced by Dostal, Horak and Kucera is an extension of the original FETI method. Main idea of TFETI is to make all subdomains floating, that is achieved by tearing away the Dirichlet boundary conditions from the subdomains, and then interconnecting them again with its respective subdomains using the Lagrange multipliers. This change makes all subdomains floating, and so we know the kernel of the stiffness matrix a priory. This change leads to an easier implementation of the method.

The main bottleneck of the TFETI method is solution of the coarse problem (CP). This thesis examines strategies for solution of CP.

This thesis is divided into following sections. In Section 2 we derive the TFETI method. Using the dualization theory we transform the original (primal) problem into a dual one, that has the same structure but is more easily solvable. Then we introduce a projector onto the natural coarse space to further improve our formulation.

Section 3 explains what is CP. It shows from what objects it is composed and how these should be stored. In this section we also suggest several strategies for CP solution.

In Section 4 we introduce algorithms employed for solution of CP. We derive the LU and Cholesky factorizations and the conjugate gradient (CG) method. For the CGs we also briefly discuss a communication hiding technique known as pipelining.

Libraries and the high performance computing (HPC) infrastructure used for numerical experiments are overviewed in Section 5.

Finally, in Section 6 we show how the strategies and algorithms for solution of CP perform.

2 The TFETI Method

In this section we will follow classical descriptions of the TFETI method as seen in [2] and followed by others [3, 4].

After finite element discretization (mesh creation) of the given problem, we decompose (tear) the discretized domain Ω into N_s non-overlapping subdomains Ω_s, $s \in \{1, \ldots, N_s\}$. Then we introduce the Lagrange multipliers (see Section 2.1), these works as gluing forces – enforcing continuity across the subdomains (interconnecting). Unlike the original FETI method, we tear away the Dirichlet boundary conditions as well and enforce them again by means of the Lagrange multipliers. This makes all the subdomains floating. See Figure 1 for illustration.

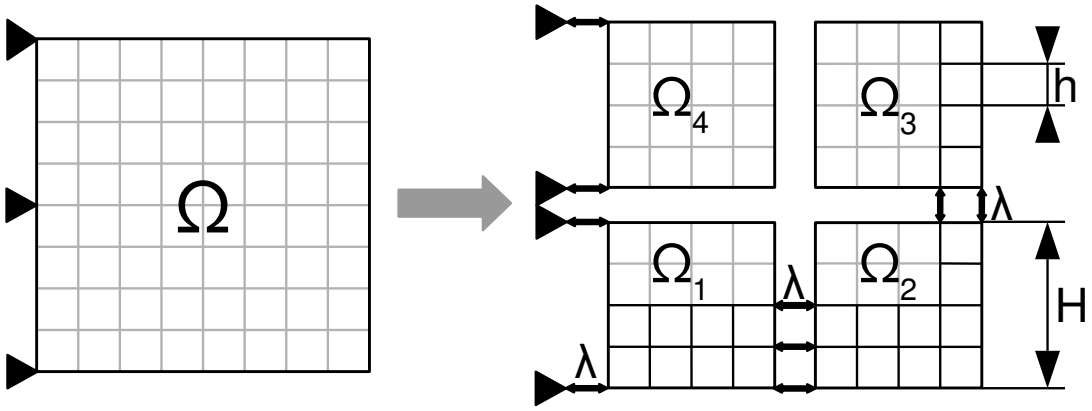

Figure 1: TFETI domain decomposition with outlined discretization

Let us denote for each subdomain Ω_s a stiffness matrix $\boldsymbol{K}_s$, prescribed nodal load vector $\boldsymbol{f}_s$, displacement vector $\boldsymbol{u}_s$ and matrix $\boldsymbol{B}_s$ with entries -1,0,1 describing the subdomain interconnectivity.

The problem of finding the reaction (displacement) vector $\boldsymbol{u}$ caused by the force $\boldsymbol{f}$ exerted on the domain which is represented by its stiffness matrix $\boldsymbol{K}$ can be described by

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \qquad \text{s.t. } \boldsymbol{B}\boldsymbol{u} = \boldsymbol{c}, \tag{2.1}$$

which can be equivalently written in its energetic formulation as

$$\min_{\boldsymbol{u}} \frac{1}{2}\boldsymbol{u}^T\boldsymbol{K}\boldsymbol{u} - \boldsymbol{u}^T\boldsymbol{f} \qquad \text{s.t. } \boldsymbol{B}\boldsymbol{u} = \boldsymbol{c}, \tag{2.2}$$

where

$$\boldsymbol{K} = \begin{pmatrix} \boldsymbol{K}_1 & & \\ & \ddots & \\ & & \boldsymbol{K}_{N_s} \end{pmatrix}, \quad \boldsymbol{f} = \begin{pmatrix} \boldsymbol{f}_1 \\ \vdots \\ \boldsymbol{f}_{N_s} \end{pmatrix}, \quad \boldsymbol{u} = \begin{pmatrix} \boldsymbol{u}_1 \\ \vdots \\ \boldsymbol{u}_{N_s} \end{pmatrix}, \quad \boldsymbol{B} = \begin{pmatrix} \boldsymbol{B}_1 & \cdots & \boldsymbol{B}_{N_s} \end{pmatrix},$$

and the vector $\boldsymbol{c}$ prescribes the Dirichlet boundary conditions. The matrix $\boldsymbol{K}$ is of order N_p (primal dimension), and the matrix $\boldsymbol{B}$ dimension is $N_d \times N_p$ (where N_d is the number of Lagrange multipliers – dual dimension, see Section 2.1).

Note that (2.2) represents a quadratic programming problem (QP) where we minimize the quadratic cost function with respect to the equality constraint.

2.1 Dualization

Let us introduce the Lagrangian function associated with the problem (2.2)

$$L(\boldsymbol{u},\boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{u}^T\boldsymbol{K}\boldsymbol{u} - \boldsymbol{u}^T\boldsymbol{f} + (\boldsymbol{B}\boldsymbol{u}-\boldsymbol{c})^T\boldsymbol{\lambda}.$$

Notice that maximizing $\boldsymbol{\lambda}$ while minimizing $\boldsymbol{u}$ we are enforcing the equality constraint and simultaneously solving the minimization part of the problem (2.2). Therefore at the cost of introducing another variable we can reformulate our problem as

$$\text{find } (\bar{\boldsymbol{u}},\bar{\boldsymbol{\lambda}}) \text{ so that } L(\bar{\boldsymbol{u}},\bar{\boldsymbol{\lambda}}) = \sup_{\boldsymbol{\lambda}}\inf_{\boldsymbol{u}} L(\boldsymbol{u},\boldsymbol{\lambda}). \tag{2.3}$$

See [5, 6] for a proof and an additional discussion of duality.

The necessary conditions for existence of the saddle-point $(\bar{\boldsymbol{u}},\bar{\boldsymbol{\lambda}})$ is the equivalence of the gradients to the null vector

$$\nabla_{\boldsymbol{u}} L(\boldsymbol{u},\boldsymbol{\lambda}) = \boldsymbol{K}\boldsymbol{u} - \boldsymbol{f} + \boldsymbol{B}^T\boldsymbol{\lambda} = \boldsymbol{o}, \tag{2.4}$$
$$\nabla_{\boldsymbol{\lambda}} L(\boldsymbol{u},\boldsymbol{\lambda}) = \boldsymbol{B}\boldsymbol{u} - \boldsymbol{c} = \boldsymbol{o}, \tag{2.5}$$

this is known as the Karush–Kuhn–Tucker (KKT) system, often written in the matrix form as

$$\begin{pmatrix} \boldsymbol{K} & \boldsymbol{B}^T \\ \boldsymbol{B} & \boldsymbol{O} \end{pmatrix} \begin{pmatrix} \boldsymbol{u} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{c} \end{pmatrix}. \tag{2.6}$$

The dual formulation (2.3) is therefore equivalent to finding $(\bar{\boldsymbol{u}},\bar{\boldsymbol{\lambda}})$ satisfying the KKT system.

Equation (2.4) has solution if and only if

$$(\boldsymbol{f} - \boldsymbol{B}^T\boldsymbol{\lambda}) \in Im\boldsymbol{K}. \tag{2.7}$$

Denoting matrix $\boldsymbol{R}$ with columns spanning the null space of $\boldsymbol{K}$, and observing that $Ker\boldsymbol{K}^T = Im\boldsymbol{R}$ is orthogonal to $Im\boldsymbol{K}$, we can express the previous statement as

$$\boldsymbol{R}^T(\boldsymbol{f} - \boldsymbol{B}^T\boldsymbol{\lambda}) = \boldsymbol{o}. \tag{2.8}$$

Let us mention that, since all the subdomains are floating, a basis of $\boldsymbol{R}$ can be formed directly using the subdomain rigid body modes. Assuming a subdomain Ω_s is discretized by n_s nodes with coordinates (x_i, y_i) in 2D or (x_i, y_i, z_i) in 3D, we denote

$$\text{for } i = \{1,\dots,n_s\} \begin{cases} \text{in 2D: } \boldsymbol{R}_s^i &= \begin{pmatrix} 1 & 0 & -y_i \\ 0 & 1 & x_i \end{pmatrix} \\ \text{in 3D: } \boldsymbol{R}_s^i &= \begin{pmatrix} 1 & 0 & 0 & 0 & -z_i & y_i \\ 0 & 1 & 0 & z_i & 0 & -x_i \\ 0 & 0 & 1 & -y_i & x_i & 0 \end{pmatrix} \end{cases}$$

$$\boldsymbol{R}_s = \begin{pmatrix} \boldsymbol{R}_s^1 & \cdots & \boldsymbol{R}_s^{n_s} \end{pmatrix}^T, \quad \boldsymbol{R} = \begin{pmatrix} \boldsymbol{R}_1 & & \\ & \ddots & \\ & & \boldsymbol{R}_{N_s} \end{pmatrix} \in \mathbb{R}^{N_p \times N_k}, \quad N_k = N_p - rank(K).$$

Note that the null space dimension $N_k = 3N_s$ in 2D, respectively $N_k = 6N_s$ in 3D.

Now we try to eliminate the primal variable $\boldsymbol{u}$ from (2.4), assuming $\boldsymbol{\lambda}$ satisfies (2.7) and denoting a generalized inverse by $\boldsymbol{K}^+$, such that $\boldsymbol{K}\boldsymbol{K}^+\boldsymbol{K} = \boldsymbol{K}$. Notice that $\forall \boldsymbol{\alpha} \in \mathbb{R}^{N_k} : \boldsymbol{R}\boldsymbol{\alpha} \in Ker\boldsymbol{K}$ and therefore $\boldsymbol{K}\boldsymbol{R}\boldsymbol{\alpha} = \boldsymbol{o}$. Using this, we can rewrite (2.4) as

$$\boldsymbol{K}\boldsymbol{u} - \boldsymbol{f} + \boldsymbol{B}^T\boldsymbol{\lambda} = \boldsymbol{K}\boldsymbol{R}\boldsymbol{\alpha}.$$

Rearranging and multiplying this equation by $\boldsymbol{K}^+$, we get

$$\boldsymbol{u} = \boldsymbol{K}^+(\boldsymbol{f} - \boldsymbol{B}^T\boldsymbol{\lambda}) + \boldsymbol{R}\boldsymbol{\alpha} \tag{2.9}$$

and by substituting into (2.5) we arrive at

$$\boldsymbol{B}\boldsymbol{K}^+\boldsymbol{B}^T\boldsymbol{\lambda} - \boldsymbol{B}\boldsymbol{R}\boldsymbol{\alpha} = \boldsymbol{B}\boldsymbol{K}^+\boldsymbol{f} - \boldsymbol{c}. \tag{2.10}$$

By combining (2.8) and (2.10) we obtain linear system for the unknowns $\boldsymbol{\lambda}$ and $\boldsymbol{\alpha}$

$$\begin{pmatrix} \boldsymbol{B}\boldsymbol{K}^+\boldsymbol{B}^T & -\boldsymbol{B}\boldsymbol{R} \\ -\boldsymbol{R}^T\boldsymbol{B}^T & \boldsymbol{O} \end{pmatrix} \begin{pmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\alpha} \end{pmatrix} = \begin{pmatrix} \boldsymbol{B}\boldsymbol{K}^+\boldsymbol{f} - \boldsymbol{c} \\ -\boldsymbol{R}^T\boldsymbol{f} \end{pmatrix}.$$

Denoting

$$\begin{aligned} \boldsymbol{F} &= \boldsymbol{B}\boldsymbol{K}^+\boldsymbol{B}^T, & \boldsymbol{G} &= -\boldsymbol{R}^T\boldsymbol{B}^T, \\ \boldsymbol{d} &= \boldsymbol{B}\boldsymbol{K}^+\boldsymbol{f} - \boldsymbol{c}, & \boldsymbol{e} &= -\boldsymbol{R}^T\boldsymbol{f}, \end{aligned}$$

we have

$$\begin{pmatrix} \boldsymbol{F} & \boldsymbol{G}^T \\ \boldsymbol{G} & \boldsymbol{O} \end{pmatrix} \begin{pmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\alpha} \end{pmatrix} = \begin{pmatrix} \boldsymbol{d} \\ \boldsymbol{e} \end{pmatrix}. \tag{2.11}$$

Now we cloud find the solution $(\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\alpha}}) \in \mathbb{R}^{N_d} \times \mathbb{R}^{N_k}$ of the problem (2.11). Formally (2.11) is the same problem as (2.6), however $\boldsymbol{F}$ is much better conditioned than $\boldsymbol{K}$ and also the size of the problem is significantly smaller [3]. Note that our solution $\bar{\boldsymbol{u}}$ can be easily computed using (2.9).

2.2 Natural Coarse Grid

To further improve the problem formulation (2.11), let us introduce a projector onto the natural coarse space [7]. This is a projection onto the subspace orthogonal to a space spanned by the columns of $\boldsymbol{G}^T$, i.e. onto $Ker\boldsymbol{G}$ and is defined by

$$\boldsymbol{P} = \boldsymbol{P}^T = \boldsymbol{I} - \boldsymbol{G}^T(\boldsymbol{G}\boldsymbol{G}^T)^{-1}\boldsymbol{G}.$$

Applying $\boldsymbol{P}$ on the first equation of (2.11) and using the orthogonality of $\boldsymbol{P}$ and $\boldsymbol{G}^T$ we obtain

$$\boldsymbol{P}\boldsymbol{F}\boldsymbol{\lambda} = \boldsymbol{P}\boldsymbol{d}. \tag{2.12}$$

We split $\boldsymbol{\lambda}$ into $\boldsymbol{\lambda}_{Im} \in Im\boldsymbol{G}^T$ and $\boldsymbol{\lambda}_{Ker} \in Ker\boldsymbol{G}$ so that

$$\boldsymbol{\lambda} = \boldsymbol{\lambda}_{Im} + \boldsymbol{\lambda}_{Ker}. \tag{2.13}$$

Substituting into the second equation of (2.11) we have

$$\boldsymbol{G\lambda} = \boldsymbol{G}(\boldsymbol{\lambda}_{Im} + \boldsymbol{\lambda}_{Ker}) = \boldsymbol{G\lambda}_{Im} + \boldsymbol{G\lambda}_{Ker} = \boldsymbol{G\lambda}_{Im} = \boldsymbol{e}.$$

It can be verified directly that we can obtain $\boldsymbol{\lambda}_{Im}$ as

$$\boldsymbol{\lambda}_{Im} = \boldsymbol{G}^T(\boldsymbol{GG}^T)^{-1}\boldsymbol{e}.$$

Note that by choosing this particular solution $\boldsymbol{\lambda}_{Im}$ we have effectively homogenized the equality constraint in the KKT system (2.11) [6]. Now we just have to identify $\boldsymbol{\lambda}_{Ker}$. Substituting (2.13) into (2.12) we get

$$\begin{aligned} \boldsymbol{PF}(\boldsymbol{\lambda}_{Im} + \boldsymbol{\lambda}_{Ker}) &= \boldsymbol{Pd} \\ \boldsymbol{PF\lambda}_{Im} + \boldsymbol{PF\lambda}_{Ker} &= \boldsymbol{Pd} \\ \boldsymbol{PF\lambda}_{Ker} &= \boldsymbol{P}(\boldsymbol{d} - \boldsymbol{F\lambda}_{Im}). \end{aligned} \tag{2.14}$$

The equation (2.14) can be solved by the CG method, which is described in Section 4.3. After obtaining $\boldsymbol{\lambda}$ we find $\boldsymbol{\alpha}$ from the first equation of (2.11) as

$$\begin{aligned} \boldsymbol{F\lambda} + \boldsymbol{G}^T\boldsymbol{\alpha} &= \boldsymbol{d} \qquad /(\boldsymbol{GG}^T)^{-1}\boldsymbol{G}\cdot \\ (\boldsymbol{GG}^T)^{-1}\boldsymbol{GF\lambda} + (\boldsymbol{GG}^T)^{-1}\boldsymbol{GG}^T\boldsymbol{\alpha} &= (\boldsymbol{GG}^T)^{-1}\boldsymbol{Gd} \\ \boldsymbol{\alpha} &= (\boldsymbol{GG}^T)^{-1}\boldsymbol{G}(\boldsymbol{d} - \boldsymbol{F\lambda}). \end{aligned} \tag{2.15}$$

Suppose that the domain Ω is a regular square or cube with the subdomain size H and has a discretization parameter h (as in Figure 1). Then the following estimate of FETI condition number proved in [7]

$$\kappa\left(\boldsymbol{PF}|Im\boldsymbol{P}\right) \leq const\frac{H}{h}, \tag{2.16}$$

remains also valid for the TFETI method [2]. Note that keeping the mesh size fixed and increasing the number of subdomains (decreasing size of the subdomains – H parameter) decreases this condition number.

2.3 Algorithm

We can sum up the previous observations in the algorithmic scheme 1, as seen in [3].

Algorithm 1: TFETI algorithmic scheme

1. Set $\boldsymbol{G} = -\boldsymbol{R}^T\boldsymbol{B}^T$, $\boldsymbol{d} = \boldsymbol{BK}^+\boldsymbol{f} - \boldsymbol{c}$, and $\boldsymbol{e} = -\boldsymbol{R}^T\boldsymbol{f}$
2. Compute $\boldsymbol{\lambda}_{Im} = \boldsymbol{G}^T\left(\boldsymbol{GG}^T\right)^{-1}\boldsymbol{e}$
3. Assemble $\bar{\boldsymbol{d}} = \boldsymbol{d} - \boldsymbol{F\lambda}_{Im}$
4. Compute $\boldsymbol{\lambda}_{Ker}$ by solving $\boldsymbol{PF\lambda}_{Ker} = \boldsymbol{P}\bar{\boldsymbol{d}}$ on $Ker\boldsymbol{G}$
5. Set $\boldsymbol{\lambda} = \boldsymbol{\lambda}_{Im} + \boldsymbol{\lambda}_{Ker}$
6. Compute $\boldsymbol{\alpha} = \left(\boldsymbol{GG}^T\right)^{-1}\boldsymbol{G}\left(\boldsymbol{d} - \boldsymbol{F\lambda}\right)$
7. Compute $\boldsymbol{u} = \boldsymbol{K}^+\left(\boldsymbol{f} - \boldsymbol{B}^T\boldsymbol{\lambda}\right) + \boldsymbol{R\alpha}$

3 The TFETI Coarse Problem

CP solution defined as

$$\boldsymbol{G}\boldsymbol{G}^T\boldsymbol{x} = \boldsymbol{y} \tag{3.1}$$

or, using the explicit inverse, as

$$\boldsymbol{x} = \left(\boldsymbol{G}\boldsymbol{G}^T\right)^{-1}\boldsymbol{y} \tag{3.2}$$

is important to obtain efficiently as it features in every iteration of the CG method used for the solution of (2.14). This is because $\boldsymbol{P}\boldsymbol{F}$ is kept unassembled.

Increasing the number of subdomains for fixed problem size decreases the size of the subdomain stiffness matrix. This leads to the acceleration of the subdomain stiffness matrix factorization and the subsequent pseudoinverse applications. Moreover, it improves the condition number (for regular domains) as shown in (2.16). On the other hand, the dual (N_d) and null space dimensions (N_k) are increased. This leads to the deceleration of CP solution [4].

3.1 The Storage and Assembly of CP Matrices

Assembly of $\boldsymbol{G} = -\boldsymbol{R}^T\boldsymbol{B}^T \in \mathbb{R}^{N_k \times N_d}$ is very cheap, because the application of the interconnecting matrix $\boldsymbol{B}$ should be thought of as an extraction process rather than multiplication [8]. Moreover, the matrices $\boldsymbol{R}$ and $\boldsymbol{B}$ are naturally divided into sparse sequential blocks $\boldsymbol{R}_{[loc]}$ and $\boldsymbol{B}_{[loc]}$. That is way no communication is needed for the assembly of $\boldsymbol{G}_{[loc]} = -\boldsymbol{R}^T_{[loc]}\boldsymbol{B}^T_{[loc]}$ [9]. See Figure 2 for illustration.

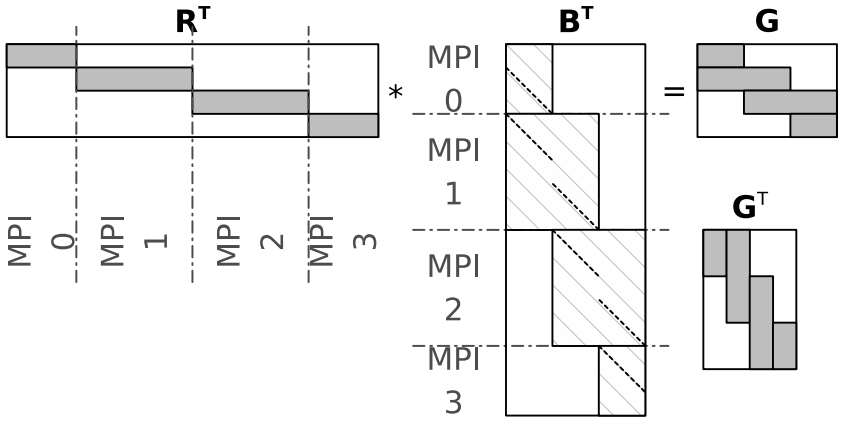

Figure 2: Illustration of the matrix $\boldsymbol{G}$ assembly [10]

It is important to consider the distribution of the matrix $\boldsymbol{G}$, because CP is composed of the matrices $\boldsymbol{G}$ and $\boldsymbol{G}^T$. The matrix $\boldsymbol{G}$ can be distributed into either horizontal or vertical blocks. It is better to store the horizontal blocks $\boldsymbol{G}^T$, because then we are operating on vectors with size of N_k instead of N_d, and the N_d vectors and operations on them can be naturally parallelized [11].

Our own numerical tests had shown that it is better to assemble $\boldsymbol{G}\boldsymbol{G}^T \in \mathbb{R}^{N_k \times N_k}$ (illustrated in Figure 3). This has a drawback that more memory is needed. The advantage is the possibility to use direct solvers for CP, and it turns out, due to efficient implementation of the matrix type *MPIAIJ* in PETSc (see Section 5.1), that the assembly process is fairly cheap.

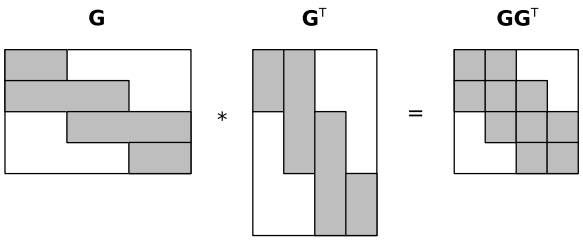

Figure 3: $\boldsymbol{GG}^T$ assembly illustration [10]

3.2 CP Solution Strategies

Several methods for solving CP were proposed and tested [3, 4, 9–11]. It is well established that the fully parallel approach is not justifiable, because the CP dimension is fairly small. Generally, the best approach is to increase locally owned portion of $\boldsymbol{GG}^T$ per processor. This leads to decrease in the number of messages needed, while keeping the size of the messages for dot-products constant. For sparse matrix-vector products (neighbour wise communication) it increases the messages size, but it keeps the amount of data transfered constant, meaning that less messages are sent. In our current implementation of PermonFLLOP (see Section 5.2.2) this is achieved by splitting the global communicator into subcommunicators. For this purpose we use PETSc's built-in *PCREDUNDANT*, that creates the subcommunicators and copies and distributes the whole matrix $\boldsymbol{GG}^T$ onto ranks residing on the subcommunicators. Computational cores are assigned to the subcommunicators contiguously, exploiting data locality – minimizing distances between ranks and limiting extranode communication as much as possible. In fact, our employment of the subcommunicators can be viewed as a communication avoiding technique. Let us denote by N_r the number of subcommunicators and by N_c the number of cores. Then in each subcommunicator is approximately N_c/N_r cores.

For the strategies introduced bellow we are using both direct (see Sections 4.1.3 and 4.2.1) and iterative (see Section 4.3.6) parallel solvers.

In this work we are exploring two strategies, namely:

Strategy S1: Obtain a solution of CP by solving the system (3.1). For the direct solvers this strategy consists of a preprocessing phase where $\boldsymbol{GG}^T$ is factorized. Every application of $\left(\boldsymbol{GG}^T\right)^{-1}$ then consists of the backward and forward substitution, as described in Section 4.1.2. Note that the factorization and each backward and forward substitution are done on each subcommunicator independently, the solution vector is then scattered onto the appropriate ranks in the global communicator (this is again achieved by communicating only in the subcommunicator). This strategy can be used even with iterative solvers, however some form of acceleration (for CG based method some form of deflation) has to be employed to improve convergence of the iterative solvers for multiple right hand sides.

Strategy S2: Explicitly compute the inverse $\left(\boldsymbol{GG}^T\right)^{-1}$ as in the equation (3.2). This is done by assigning a contiguous portion of columns of the identity matrix to each subcommunicator. Each subcommunicator has assigned approximately N_k/N_r columns. A parallel solver is used to compute a columns of $\left(\boldsymbol{GG}^T\right)^{-1}$. Using the symmetry of $\boldsymbol{GG}^T$, these columns are at the same time the rows of its inverse. These rows are then logically (i.e. without any data movement, communication or computation involved) reassigned from the subcommunicators into the

global communicator. The $\left(\boldsymbol{G}\boldsymbol{G}^T\right)^{-1}$ is a dense matrix, and so its application consists of a parallel dense matrix-vector product.

Note that in the strategy S1, the computation is done redundantly. In the current implementation of *PCREDUNDANT* the subcommunicators are doing same work in parallel. This is inefficient from the energy consumption standpoint, but it allows us to scatter the solution vector to the appropriate ranks in the global communicator using communication only in the subcommunicators. After the solution vector is scattered, the CG algorithm runs fully in parallel again.

On the other hand, in the strategy S2, the computation is always fully parallel, because each portion of the identity matrix (right hand sides) is solved in parallel on the subcommunicator level, and a parallel solver is used inside the subcommunicators as well.

4 Algorithms for Solution of Coarse Problem

In this section, we shall introduce algorithms that were applied as solvers for the CP solution strategies shown in the previous section.

For the whole section we assume that we have a square matrix $\boldsymbol{A}$, right hand side vector $\boldsymbol{b}$ and vector of variables $\boldsymbol{x}$, all real and of the dimension n. Our goal is to find the solution of the system of linear equations denoted as

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}. \tag{4.1}$$

4.1 LU Factorization

LU factorization can be viewed as Gaussian elimination described by a transformation matrix. Gaussian elimination transforms the matrix $\boldsymbol{A}$ into an upper triangular matrix. The idea behind the LU factorization is to decompose (hence the other commonly used name – LU decomposition) the matrix $\boldsymbol{A}$ into a product of the lower and upper triangular matrices, i.e.

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}.$$

We will now describe the Gauss's zeroing process by a transformation matrix. For a vector $\boldsymbol{a} \in \mathbb{R}^n$ where $a_k \neq 0$ we can zero each a_i, $i \in \{k+1, \cdots, n\}$ by multiplying the vector $\boldsymbol{a}$ by the matrix $\boldsymbol{L}_k$ defined as

$$\boldsymbol{L}_k\boldsymbol{a} = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -l_{k+1,k} & 1 & \\ & & \vdots & & \ddots \\ & & -l_{n,k} & & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} a_1 \\ \vdots \\ a_k \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

where $l_{i,k} = \frac{a_i}{a_k}$, $i \in \{k+1, \cdots, n\}$ are called multipliers. This can be rewritten as

$$\boldsymbol{L}_k = \boldsymbol{I} - \boldsymbol{l_k}\boldsymbol{e}_k^T, \quad \boldsymbol{l_k}^T = \begin{pmatrix} 0 & \cdots & 0 & l_{k+1,k} & \cdots & l_{n,k} \end{pmatrix}.$$

The vector $\boldsymbol{l_k}$ is called a Gauss vector. The matrix $\boldsymbol{L}_k$ is referred to as a Gauss transformation matrix [12].

Applying the Gauss transformations to the matrix $\boldsymbol{A}$ we get the upper triangular matrix $\boldsymbol{U}$ as

$$\boldsymbol{L}_{n-1}\boldsymbol{L}_{n-2}\cdots\boldsymbol{L}_1\boldsymbol{A} = \boldsymbol{L}_{n-1}\boldsymbol{L}_{n-2}\cdots\boldsymbol{L}_2\boldsymbol{A}^{(1)} = \boldsymbol{U}, \tag{4.2}$$

assuming that after the application of each transformation matrix $\boldsymbol{L}_k$, the diagonal component of the resulting matrix $a^{(k)}_{k+1,k+1} \neq 0$.

It is easy to show [13] that

$$\boldsymbol{L}_k^{-1} = \boldsymbol{I} + \boldsymbol{l_k}\boldsymbol{e}_k^T,$$

and that we can define

$$\boldsymbol{L} = \boldsymbol{L}_1^{-1}\boldsymbol{L}_2^{-1}\cdots\boldsymbol{L}_{n-1}^{-1} = \begin{pmatrix} 1 & & & & \\ l_{2,1} & 1 & & & \\ l_{3,1} & l_{3,2} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n-1} & 1 \end{pmatrix}.$$

Finally, multiplying (4.2) by $\boldsymbol{L}$ we obtain

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}.$$

4.1.1 Pivoting

In the previous section in the derivation of the zeroing process, we required that after each application of the transformation matrix $\boldsymbol{L}_k$, the diagonal component of the resulting matrix $a^{(k)}_{k+1,k+1} \neq 0$. This requirement is necessary, because otherwise we would divide by zero in the next step. Therefore, we could not apply the LU decomposition on some matrices, nor we would know beforehand whether it is possible to compute the LU factorization, as defined by the equation (4.2). This problem is enhanced in floating point arithmetics due to the limited precision and our algorithm becomes unstable. For an in-depth discussion with examples see [12, 13].

To counter the instability, we define a pivoting process. Reordering rows of an identity matrix we obtain a permutation matrix $\boldsymbol{P}$. Multiplying the matrix $\boldsymbol{A}$ from left by the permutation matrix we reorder the rows of $\boldsymbol{A}$ in the same way as are reordered the rows of $\boldsymbol{P}$ in respect to the $\boldsymbol{I}$. For example if $n = 3$ and

$$\boldsymbol{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

then $\boldsymbol{P}\boldsymbol{A}$ is equivalent to $\boldsymbol{A}$ with the second and third rows swapped.

One of the pivoting methods, called partial pivoting, in the k^{th} iteration (i.e. zeroing the components of k^{th} column bellow the diagonal of matrix $\boldsymbol{A}^{(k-1)}$) finds the largest component of the k^{th} column bellow and on the diagonal ($\max \left|a^{(k-1)}_{i,k}\right|$, $i \geq k$) and then creates a permutation matrix $\boldsymbol{P}_k$, applying which swaps rows so that the largest component is on the diagonal [12].

This process modifies (4.2) into

$$\boldsymbol{L}_{n-1}\boldsymbol{P}_{n-1}\boldsymbol{L}_{n-2}\boldsymbol{P}_{n-2}\cdots\boldsymbol{L}_1\boldsymbol{P}_1\boldsymbol{A} = \boldsymbol{L}_{n-1}\boldsymbol{P}_{n-1}\boldsymbol{L}_{n-2}\boldsymbol{P}_{n-2}\cdots\boldsymbol{L}_2\boldsymbol{P}_2\boldsymbol{A}^{(1)} = \boldsymbol{U}. \tag{4.3}$$

Note that for an arbitrary permutation matrix $\boldsymbol{P}$ we have $\boldsymbol{P}\boldsymbol{P}^T = \boldsymbol{I}$. Therefore the previous equation can be rewritten as

$$\bar{\boldsymbol{L}}_{n-1}\bar{\boldsymbol{L}}_{n-2}\cdots\bar{\boldsymbol{L}}_1\boldsymbol{P}\boldsymbol{A} = \boldsymbol{U},$$

where

$$\bar{\boldsymbol{L}}_k = (\boldsymbol{P}_{n-1}\cdots\boldsymbol{P}_{k+1})\,\boldsymbol{L}_k\,(\boldsymbol{P}_{k+1}\cdots\boldsymbol{P}_{n-1})\,.$$

It follows that

$$\bar{\boldsymbol{L}}_k = (\boldsymbol{P}_{n-1}\cdots\boldsymbol{P}_{k+1})\left(\boldsymbol{I} - \boldsymbol{l}_{\boldsymbol{k}}\boldsymbol{e}_k^T\right)(\boldsymbol{P}_{k+1}\cdots\boldsymbol{P}_{n-1}) = \boldsymbol{I} - (\boldsymbol{P}_{n-1}\cdots\boldsymbol{P}_{k+1})\,\boldsymbol{l}_{\boldsymbol{k}}\boldsymbol{e}_k^T = \boldsymbol{I} - \bar{\boldsymbol{l}}_{\boldsymbol{k}}\boldsymbol{e}_k^T.$$

This shows that $\bar{\boldsymbol{L}}_k$ is a Gauss transformation matrix and that the pivoting process is easily implementable.

4.1.2 Solution of Triangular Systems

We want to find the solution of (4.1). Since we have $\boldsymbol{A} = \boldsymbol{LU}$, respectively $\boldsymbol{PA} = \boldsymbol{LU}$ it follows that

$$\begin{aligned} \boldsymbol{Ax} &= \boldsymbol{b} \\ \boldsymbol{PAx} &= \boldsymbol{Pb} \\ \boldsymbol{LUx} &= \boldsymbol{Pb}. \end{aligned}$$

Denoting $\boldsymbol{y} = \boldsymbol{Ux}$ we can solve $\boldsymbol{Ly} = \boldsymbol{Pb}$ and then obtain $\boldsymbol{x}$ as the solution of $\boldsymbol{Ux} = \boldsymbol{y}$. This is known as the forward and backward substitution [13].

Note that since $\boldsymbol{L}$ and $\boldsymbol{U}$ are respectively lower and upper triangular, the solutions of the above systems are obtained very cheaply. Moreover, the LU factorization is advantageous for solving a system with several right hand sides. This is heavily utilized in the Section 3.2.

4.1.3 Implementation

We used SuperLU_DIST [14] for our numerical tests. SuperLU_DIST is an effective implementation of the above described algorithm, with modifications that ensures effectiveness on distributed-memory parallel machines. The algorithm used in this solver is called GESP (Gaussian elimination with static pivoting). The difference from our description is a diagonal scaling that improves the effect of pivoting (that is chosen by a graph theory algorithm), then another permutation matrix is applied to preserve the sparsity of $\boldsymbol{A}$. These steps are called static pivoting, since they are done only once before factorization. After these additional steps are done the LU decomposition, broadly the same as described above, takes place. For detailed information about SuperLU and the GESP algorithm please refer to [15].

4.2 Cholesky Factorization

For a SPD matrix $\boldsymbol{A}$ it is possible to improve the LU factorization by exploiting the symmetry of $\boldsymbol{A}$. It turns out that it is possible to create the following decomposition of $\boldsymbol{A}$

$$\boldsymbol{A} = \boldsymbol{LL}^T,$$

where $\boldsymbol{L}$ is a lower triangular matrix.

Let us now describe how we can obtain the matrix $\boldsymbol{L}$. Applying Gaussian elimination to zero the first column of the matrix $\boldsymbol{A}$ we have

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & \boldsymbol{u}^T \\ \boldsymbol{u} & \boldsymbol{K} \end{pmatrix} = \begin{pmatrix} \sqrt{a_{11}} & \boldsymbol{o}^T \\ \boldsymbol{u}/\sqrt{a_{11}} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} 1 & \boldsymbol{u}^T \\ \boldsymbol{o} & \boldsymbol{K} - \boldsymbol{uu}^T/\sqrt{a_{11}} \end{pmatrix}.$$

We can zero the first row

$$\boldsymbol{A} = \begin{pmatrix} \sqrt{a_{11}} & \boldsymbol{o}^T \\ \boldsymbol{u}/\sqrt{a_{11}} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} 1 & \boldsymbol{o}^T \\ \boldsymbol{o} & \boldsymbol{K} - \boldsymbol{uu}^T/a_{11} \end{pmatrix} \begin{pmatrix} \sqrt{a_{11}} & \boldsymbol{u}^T/\sqrt{a_{11}} \\ \boldsymbol{o} & \boldsymbol{I} \end{pmatrix} = \boldsymbol{L}_1 \boldsymbol{A}^{(1)} \boldsymbol{L}_1^T,$$

and so the symmetry is maintained.

Now we can repeat the process, zeroing the successive columns and rows. After n steps the $\boldsymbol{A}^{(n)} = \boldsymbol{I}$, therefore we have

$$\boldsymbol{A} = \underbrace{\boldsymbol{L}_1\boldsymbol{L}_2\cdots\boldsymbol{L}_n}_{\boldsymbol{L}}\underbrace{\boldsymbol{L}_n^T\cdots\boldsymbol{L}_2^T\boldsymbol{L}_1^T}_{\boldsymbol{L}^T}.$$

Once we obtain the Cholesky decomposition of $\boldsymbol{A}$, the system (4.1) can be solved easily in the same way as described in Section 4.1.2. Note that from the positive definiteness of $\boldsymbol{A}$ it follows that the algorithm is stable and therefore no pivoting is needed [13]. The number of floating point operations required by the Cholesky decomposition ($\approx \frac{n^3}{3}$) is reduced by about a half compared to LU ($\approx \frac{2n^3}{3}$) [12, 13].

4.2.1 Implementation

MUMPS [16, 17] was used as an efficient implementation of the Cholesky factorization, optimized for large distributed-memory systems. Similar techniques for the preservation of the sparsity as described in Section 4.1.3 are used in MUMPS as well. While the Cholesky factorization algorithm is cheaper then LU and therefore generally MUMPS is faster then SuperLU_DIST, SuperLU is considered to be better scalable [18].

4.3 The Conjugate Gradient Method

The conjugate gradient (CG) method is one of the basic, but the most successful methods for solving systems described by a SPD matrix. It was developed independently in the early 1950s by Hestenes and Stiefel [19].

The equation (4.1) can be equivalently written as a problem of an unconstrained quadratic minimization

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}), \quad \text{where } f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b}. \tag{4.4}$$

This is easy to see, because from the necessary condition for extremum we have that $\boldsymbol{x}$ is minimizer of $f(\boldsymbol{x})$ if

$$\nabla f(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = \boldsymbol{o}.$$

The quadratic form $f(\boldsymbol{x})$ for a SPD matrix $\boldsymbol{A}$ of dimension $n = 2$ is depicted in Figure 4. By (4.4), the solution of (4.1) is the minimum of this elliptic paraboloid. This holds true, with the appropriate generalization, for higher dimensions as well.

4.3.1 The Method of Steepest Descent

We shall first introduce an easier method of steepest descent. This method is based on the line search procedure that minimizes $f(\boldsymbol{x})$ along a search direction with a suitable step, giving the distance along the search direction [20]. Given the point $\boldsymbol{x}_i$ approximating solution of (4.4) we will generate a better approximation by

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\boldsymbol{v}_i, \tag{4.5}$$

where $\boldsymbol{v_i}$ is the direction along which we minimize $f(\boldsymbol{x}_i)$ and α_i is the step length. We will stop refining our approximations when the error $\boldsymbol{e}_i = \boldsymbol{x}_i - \boldsymbol{x}$ is sufficiently small.

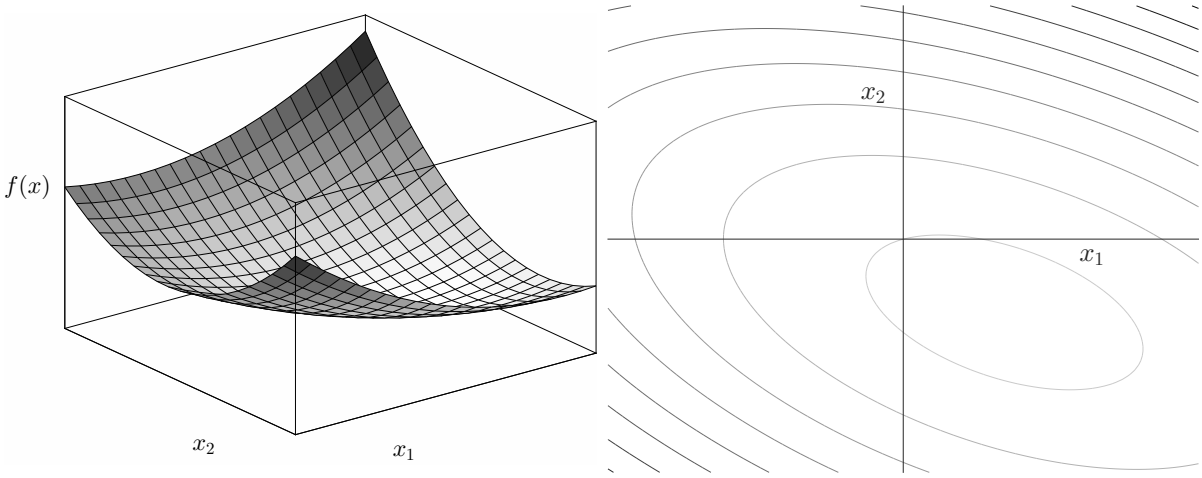

Figure 4: Surface and contour plot of $f(\boldsymbol{x})$ for a SPD matrix of dimension $n = 2$

As the name suggests, the steepest descent method uses the direction opposite to the gradient of $f(\boldsymbol{x})$, because at any point of $f(\boldsymbol{x})$ the gradient points in the direction of the greatest rate of increase of the given function. Therefore, we have

$$\boldsymbol{v_i} = -\nabla f(\boldsymbol{x}_i) = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i.$$

Let us define a residual $\boldsymbol{r}_i = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i$, representing the distance from the solution of (4.4). Note that from the previous equation it follows that $\boldsymbol{v}_i = \boldsymbol{r}_i$. Also notice that $\boldsymbol{r}_i = -\boldsymbol{A}\boldsymbol{e}_i$ and so we can use the norm of the residual as a stopping criteria.

We can rewrite (4.5) as

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{r}_i.$$

Multiplying the previous equation by $-\boldsymbol{A}$ and adding $\boldsymbol{b}$, we can write the residuals in terms of the following recurrence

$$\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i \boldsymbol{A}\boldsymbol{r}_i.$$

Now that we know the minimization direction, we only need to find out how far along it we should move before the value of $f(\boldsymbol{x}_i)$ starts increasing. Again, using the necessary condition for extremum, we find a minimum of $f(\boldsymbol{x}_i)$, where the only variable in that function is α_i and so we have

$$\frac{d}{d\alpha_i} f(\boldsymbol{x}_{i+1}) = (\nabla f(\boldsymbol{x}_{i+1}))^T \frac{d}{d\alpha_i} \boldsymbol{x}_{i+1} = (\nabla f(\boldsymbol{x}_{i+1}))^T \boldsymbol{r}_i = -\boldsymbol{r}_{i+1}^T \boldsymbol{r}_i = 0. \tag{4.6}$$

From the previous equation we find α_i at a point, where the descent direction $\boldsymbol{r}_i$ is orthogonal to the gradient $\nabla f(\boldsymbol{x}_{i+1}) = -\boldsymbol{r}_{i+1}$. This intuitively makes sense, see [20] for illustration. Using

this information, we may find the value of α_i as follows

$$\begin{aligned} \boldsymbol{r}_{i+1}^T \boldsymbol{r}_i &= 0 \\ \left(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{i+1}\right)^T \boldsymbol{r}_i &= 0 \\ \left(\boldsymbol{b} - \boldsymbol{A}\left(\boldsymbol{x}_i + \alpha_i \boldsymbol{r}_i\right)\right)^T \boldsymbol{r}_i &= 0 \\ \left(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i\right)^T \boldsymbol{r}_i - \alpha \left(\boldsymbol{A}\boldsymbol{r}_i\right)^T \boldsymbol{r}_i &= 0 \\ \boldsymbol{r}_i^T \boldsymbol{r}_i &= \alpha \left(\boldsymbol{A}\boldsymbol{r}_i\right)^T \boldsymbol{r}_i \\ \alpha_i &= \frac{\boldsymbol{r}_i^T \boldsymbol{r}_i}{\left(\boldsymbol{A}\boldsymbol{r}_i\right)^T \boldsymbol{r}_i} = \frac{\boldsymbol{r}_i^T \boldsymbol{r}_i}{\boldsymbol{r}_i^T \boldsymbol{A}\boldsymbol{r}_i}. \end{aligned} \tag{4.7}$$

We can sum up the previous observations in the Algorithm 2. Notice that by defining an additional vector $\boldsymbol{s} = \boldsymbol{A}\boldsymbol{r}_i$, we need only a single matrix-vector multiplication per iteration.

Algorithm 2: Steepest descent method

Input: $\boldsymbol{A}$, $\boldsymbol{x}_0$, $\boldsymbol{b}$, $\epsilon > 0$

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$

$i = 0$

while $||\boldsymbol{r}_i|| \geq \epsilon ||\boldsymbol{b}||$**:**

    $\boldsymbol{s} = \boldsymbol{A}\boldsymbol{r}_i$

    $\alpha_i = \left(\boldsymbol{r}_i^T \boldsymbol{r}_i\right) / \left(\boldsymbol{s}^T \boldsymbol{r}_i\right)$

    $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{r}_i$

    $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i \boldsymbol{s}$

    $i = i + 1$

Output: $\boldsymbol{x}_i$

4.3.2 Conjugate Directions

It turns out that there exist better directions than those of the steepest descent. These directions are conjugate or $\boldsymbol{A}$-orthogonal directions [20]. Two vectors $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ are conjugate, if

$$\boldsymbol{p}_i^T \boldsymbol{A}\boldsymbol{p}_j = 0, \quad \left(\boldsymbol{p}_i \neq \boldsymbol{p}_j\right).$$

It is easy to prove that a set of n of these directions $\{\boldsymbol{p}_0, \boldsymbol{p}_1, \cdots, \boldsymbol{p}_{n-1}\}$ is a basis of $\mathbb{R}^n$ [6]. Therefore, the solution vector $\boldsymbol{x}$ can be found as a linear combination

$$\boldsymbol{x} = \alpha_0 \boldsymbol{p}_0 + \alpha_1 \boldsymbol{p}_1 + \cdots + \alpha_{n-1} \boldsymbol{p}_{n-1}. \tag{4.8}$$

As in the steepest descent method (equation (4.5)) we will improve the approximation of $\boldsymbol{x}_i$ by

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{p}_i. \tag{4.9}$$

Again, using the definition of the error term and residual from the steepest descent, and similarly to (4.6), we have

$$\frac{d}{d\alpha_i} f(\boldsymbol{x}_{i+1}) = \left(\nabla f(\boldsymbol{x}_{i+1})\right)^T \frac{d}{d\alpha_i} \boldsymbol{x}_{i+1} = -\boldsymbol{r}_{i+1}^T \boldsymbol{p}_i = \boldsymbol{p}_i^T \boldsymbol{A}\boldsymbol{e}_{i+1} = 0.$$

We derive α_i like in (4.7), so that by using (4.9) and (4.8) we arrive at

$$\alpha_i = -\frac{\boldsymbol{p}_i^T \boldsymbol{A}\boldsymbol{e}_i}{\boldsymbol{p}_i^T \boldsymbol{A}\boldsymbol{p}_i} = \frac{\boldsymbol{p}_i^T \boldsymbol{r}_i}{\boldsymbol{p}_i^T \boldsymbol{A}\boldsymbol{p}_i}.$$

For the search direction vectors $\boldsymbol{p}_i$ and their respective step lengths α_i it is easily proved [20] that $\boldsymbol{x}_i$ defined by the formula (4.9) converges to the solution vector $\boldsymbol{x}$, and does so in n steps. Note that this is true only in the exact arithmetic. Therefore, as in the steepest descent method, we will stop iterating when we get a sufficiently good approximation of the solution.

4.3.3 Gram-Schmidt A-orthogonalization Process

The only thing left to show is how to get the conjugate search directions $\boldsymbol{p}_i$. As it turns out, it is possible to use the well known Gram-Schmidt orthogonalization process [6], which we will briefly describe bellow.

Let us assume that we have $\boldsymbol{p}_0, \cdots, \boldsymbol{p}_i$ nonzero conjugate directions and a vector $\boldsymbol{h}_i \notin span\{\boldsymbol{p}_0, \cdots, \boldsymbol{p}_i\}$. We can generate a new conjugate direction in the form

$$\boldsymbol{p}_{i+1} = \boldsymbol{h}_i + \beta_i^0 \boldsymbol{p}_0 + \cdots + \beta_i^i \boldsymbol{p}_i$$

Using the conjugacy, we have

$$(\boldsymbol{p}_j)^T \boldsymbol{A}\boldsymbol{p}_{i+1} = (\boldsymbol{p}_j)^T \boldsymbol{A}\boldsymbol{h}_i + \beta_i^j (\boldsymbol{p}_j)^T \boldsymbol{A}\boldsymbol{p}_j = 0, \quad j = 0, \cdots, i.$$

It follows that

$$\beta_i^j = -\frac{(\boldsymbol{p}_j)^T \boldsymbol{A}\boldsymbol{h}_i}{(\boldsymbol{p}_j)^T \boldsymbol{A}\boldsymbol{p}_j}, \quad j = 0, \cdots, i.$$

Now we could, setting $\boldsymbol{h}_0 = \boldsymbol{p}_0$, construct the set of mutually $\boldsymbol{A}$-conjugate directions $\boldsymbol{p}_0 \cdots \boldsymbol{p}_i$. However this would be expensive to compute since the number of βs needed to be computed grows with the number of vectors against which we orthogonalize. Fortunately we can adapt the procedure so that it very efficiently generates a conjugate basis of the Krylov subspace [6]

$$\mathcal{K}_i = \mathcal{K}_i(\boldsymbol{A}, \boldsymbol{r}_o) = span\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \cdots \boldsymbol{A}^{i-1}\boldsymbol{r}_0\} = span\{\boldsymbol{r}_0, \boldsymbol{r}_1, \cdots \boldsymbol{r}_{i-1}\}, \quad i = 1 \cdots n.$$

Note that we can find the residual using following recurrence

$$\boldsymbol{r}_{i+1} = \boldsymbol{A}\boldsymbol{x}_{i+1} - \boldsymbol{b} = \boldsymbol{A}(\boldsymbol{x}_i - \alpha_i \boldsymbol{p}_i) - \boldsymbol{b} = (\boldsymbol{A}\boldsymbol{x}_i - \boldsymbol{b}) - \alpha_i \boldsymbol{A}\boldsymbol{p}_i = \boldsymbol{r}_i - \alpha_i \boldsymbol{A}\boldsymbol{p}_i.$$

.

Since $\boldsymbol{r}_i^T \boldsymbol{x} = 0$ for any $\boldsymbol{x} \in \mathcal{K}_i$, by using the previous equation and assuming $\boldsymbol{p}_i = \boldsymbol{r}_i$ it turns out [20] that

$$\beta_i^j = 0, \quad j = 0, \cdots, i-1.$$

And so that

$$\beta_i = \beta_i^i = \frac{\boldsymbol{r}_i^T \boldsymbol{r}_i}{\boldsymbol{r}_{i-1}^T \boldsymbol{r}_{i-1}}.$$

4.3.4 Conjugate Gradient Algorithm

Putting together the observations from Sections 4.3.2 and 4.3.3 we get Algorithm 3.

The speed of convergence depends on the spectral properties of $\boldsymbol{A}$ [6, 13]. To improve the convergence of the CG method we can apply preconditioning [20]. This consists of applying a preconditioner in form of a SPD matrix $\boldsymbol{M}$ that approximates $\boldsymbol{A}$ as follows

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}.$$

Note that if $\boldsymbol{M}^{-1} = \boldsymbol{A}^{-1}$ then $\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}$ and we have found the solution. Finding the inverse of $\boldsymbol{A}$ is however costly and so we want to use an approximation that is easy to compute and apply, and that will improve the spectral properties of $\boldsymbol{A}$. For an overview of the most widely used preconditioners see [12].

How to apply the preconditioning, described bellow, was closely inspired by introduction to the CG methods by Shewchuk [20].

The problem with preconditioner $\boldsymbol{M}$ is that $\boldsymbol{M}^{-1}\boldsymbol{A}$ is not generally a SPD matrix. This problem can be avoided because for every SPD matrix $\boldsymbol{M}$ there exists a SPD matrix $\boldsymbol{E}$ such that $\boldsymbol{M} = \boldsymbol{E}\boldsymbol{E}^T$. Moreover, $\boldsymbol{E}^{-1}\boldsymbol{A}\boldsymbol{E}^{-T}$ is SPD and has the same eigenvalues as $\boldsymbol{M}^{-1}\boldsymbol{A}$.

We can transform the system (4.1) into

$$\boldsymbol{E}^{-1}\boldsymbol{A}\boldsymbol{E}^{-T}\widehat{\boldsymbol{x}} = \boldsymbol{E}^{-1}\boldsymbol{b}, \quad \widehat{\boldsymbol{x}} = \boldsymbol{E}^T\boldsymbol{x},$$

where we use the CG method to find $\widehat{\boldsymbol{x}}$, and then from the equation above we get $\boldsymbol{x}$. This approach, however, has a drawback that $\boldsymbol{E}$ must be computed. Applying the CG method on the transformed system above, and denoting all the vectors in Algorithm 3 by a hat symbol, we can with a few substitutions rectify this problem. Then, by setting $\widehat{\boldsymbol{r}}_i = \boldsymbol{E}^{-1}\boldsymbol{r}_i$ and $\widehat{\boldsymbol{p}}_i = \boldsymbol{E}^T\boldsymbol{p}_i$, and using the identities $\widehat{\boldsymbol{x}} = \boldsymbol{E}^T\boldsymbol{x}$ and $\boldsymbol{E}^{-T}\boldsymbol{E}^{-1} = \boldsymbol{M}^{-1}$, we get the preconditioned conjugate gradient (PCG) method shown in Algorithm 4.

Algorithm 3: CG

Input: $\boldsymbol{A}$, $\boldsymbol{x}_0$, $\boldsymbol{b}$, $\epsilon > 0$

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$

$\boldsymbol{p}_0 = \boldsymbol{r}_0$

$i = 0$

while $||\boldsymbol{r}_i|| \geq \epsilon||\boldsymbol{b}||$**:**

  $\boldsymbol{s} = \boldsymbol{A}\boldsymbol{p}_i$

  $\alpha_i = \left(\boldsymbol{r}_i^T\boldsymbol{r}_i\right) / \left(\boldsymbol{s}^T\boldsymbol{p}_i\right)$

  $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\boldsymbol{p}_i$

  $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i\boldsymbol{s}$

  $\beta_{i+1} = \left(\boldsymbol{r}_{i+1}^T\boldsymbol{r}_{i+1}\right) / \left(\boldsymbol{r}_i^T\boldsymbol{r}_i\right)$

  $\boldsymbol{p}_{i+1} = \boldsymbol{r}_{i+1} + \beta_{i+1}\boldsymbol{p}_i$

  $i = i + 1$

Output: $\boldsymbol{x}_i$

Algorithm 4: PCG

Input: $\boldsymbol{A}$, $\boldsymbol{M}^{-1}$, $\boldsymbol{x}_0$, $\boldsymbol{b}$, $\epsilon > 0$

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$

$\boldsymbol{u}_0 = \boldsymbol{M}^{-1}\boldsymbol{r}_0$

$\boldsymbol{p}_0 = \boldsymbol{u}_0$

$i = 0$

while $||\boldsymbol{r}_i|| \geq \epsilon||\boldsymbol{b}||$**:**

  $\boldsymbol{s} = \boldsymbol{A}\boldsymbol{p}_i$

  $\alpha_i = \left(\boldsymbol{r}_i^T\boldsymbol{u}_i\right) / \left(\boldsymbol{s}^T\boldsymbol{p}_i\right)$

  $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\boldsymbol{p}_i$

  $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i\boldsymbol{s}$

  $\boldsymbol{u}_{i+1} = \boldsymbol{M}^{-1}\boldsymbol{r}_{i+1}$

  $\beta_{i+1} = \left(\boldsymbol{r}_{i+1}^T\boldsymbol{u}_{i+1}\right) / \left(\boldsymbol{r}_i^T\boldsymbol{u}_i\right)$

  $\boldsymbol{p}_{i+1} = \boldsymbol{u}_{i+1} + \beta_{i+1}\boldsymbol{p}_i$

  $i = i + 1$

Output: $\boldsymbol{x}_i$

Additionally, in both algorithms $\boldsymbol{s} = \boldsymbol{A}\boldsymbol{p}_i$, and $\boldsymbol{u}_i = \boldsymbol{M}^{-1}\boldsymbol{r}_i$ in PCG are computed in order to apply these matrix-vector multiplications only once per iteration.

Note that we have derived the CG method in terms of residuals, but $\boldsymbol{r}_i = -\nabla f(\boldsymbol{x}_i)$, as shown in Section 4.3.1. Therefore we could have just as easily derived the CG algorithm in the terms of gradients – this is where the method name comes from. The residual notation is widely used for historic reasons, as it was used in the original paper by Hestenes and Stiefel [19].

4.3.5 Communication Hiding in the CG Method

Recently Ghysels and Vanroose introduced the pipelined conjugate gradient method [21], that hides the global synchronization latency in the PCG algorithm. This section is based on their article. Main idea here is that operations requiring global reductions are overlapped with a local computation.

They examined the PCG method (Algorithm 4) and identified the communication patterns. Communication wise are important following parts of PCG: the sparse matrix-vector product (SpMV) $\boldsymbol{A}\boldsymbol{p}_i$ in line 6, the application of the preconditioner $\boldsymbol{M}^{-1}\boldsymbol{r}_{i+1}$ in line 10, and the dot-products $\boldsymbol{s}^T\boldsymbol{p}_i$ and $\boldsymbol{r}_{i+1}^T\boldsymbol{u}_{i+1}$ in lines 7 and 11. The SpMV and preconditioner application requires communication only between neighbouring nodes, and as such they scale well. On the other hand, the two dot-products are causing two global synchronizations per iteration. Other operations in the PCG algorithm are purely local – requiring no communication.

In the illustrations, the dot-products are shown in red, the SpMV and the application of the preconditioner are in orange, and finally the vector plus constant times vector (AXPY) operations are shown in green.

The preconditioned pipelined CG method (PipePCG), can be found in Algorithm 5. It is derived from PCG, through Chronopoulos/Gear CG [22] respectively, by some reordering and introduction of additional AXPY operations.

Algorithms 4 and 5 are mathematically equivalent. The main advantage of the PipePCG method is that the reductions for the dot-products (lines 6 and 7) are grouped into a single non-blocking reduction, and can be overlapped by the preconditioner application (line 8) and the SpMV (line 9). This however comes at the cost of five additional AXPY operations and vectors that need to be kept in memory. The time spent in the global all-reduce communication (G), matrix vector product (SpMV) and preconditioner application (PC) is 2G + SpMV + PC for PCG, max(G, SpMV + PC) for PipePCG.

A chart comparing PCG and PipePCG with respect to both the computational and communication aspects can be found in Figures 5 and 6.

4.3.6 Implementation

Both the PCG and PipePCG implementations are from PETSc (see Section 5.1). They are *KSPPCG*, *KSPPIPECG* respectively. In *KSPCG*, an option performing a single reduction for the dot-products is not used, which is the default behaviour.

Algorithm 5: PipePCG

Input: $\boldsymbol{A}$, $\boldsymbol{M}^{-1}$, $\boldsymbol{x}_0$, $\boldsymbol{b}$, $\epsilon > 0$

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
$\boldsymbol{u}_0 = \boldsymbol{M}^{-1}\boldsymbol{r}_0$
$\boldsymbol{w}_0 = \boldsymbol{A}\boldsymbol{u}_0$
$i = 0$
while $||\boldsymbol{r}_i|| \geq \epsilon||\boldsymbol{b}||$**:**
    $\gamma_i = \boldsymbol{r}_i^T\boldsymbol{u}_i$
    $\delta = \boldsymbol{w}_i^T\boldsymbol{u}_i$
    $\boldsymbol{m}_i = \boldsymbol{M}^{-1}\boldsymbol{w}_i$
    $\boldsymbol{n}_i = \boldsymbol{A}\boldsymbol{m}_i$
    if $i > 0$**:**
        $\beta_i = \gamma_i/\gamma_{i-1}$
        $\alpha_i = \gamma_i/\left(\delta - \beta_i\gamma_i/\alpha_{i-1}\right)$
    else:
        $\beta_i = 0$
        $\alpha_i = \gamma_i/\delta$
    $\boldsymbol{z}_i = \boldsymbol{n}_i + \beta_i\boldsymbol{z}_{i-1}$
    $\boldsymbol{q}_i = \boldsymbol{m}_i + \beta_i\boldsymbol{q}_{i-1}$
    $\boldsymbol{s}_i = \boldsymbol{w}_i + \beta_i\boldsymbol{s}_{i-1}$
    $\boldsymbol{p}_i = \boldsymbol{u}_{i+1} + \beta_{i+1}\boldsymbol{p}_{i-1}$
    $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\boldsymbol{p}_i$
    $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i\boldsymbol{s}_i$
    $\boldsymbol{u}_{i+1} = \boldsymbol{u}_i - \alpha_i\boldsymbol{q}_i$
    $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i + \alpha_i\boldsymbol{z}_i$
    $i = i + 1$

Output: $\boldsymbol{x}_i$

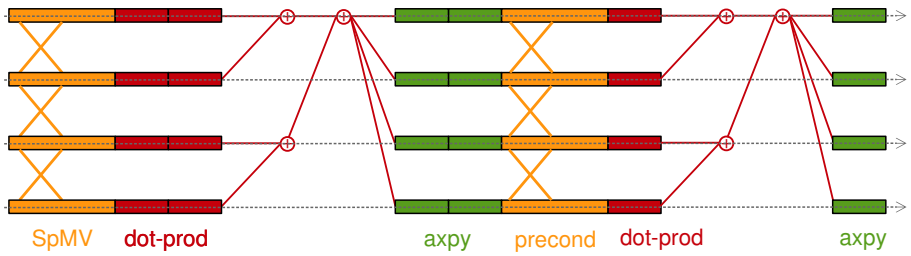

Figure 5: PCG

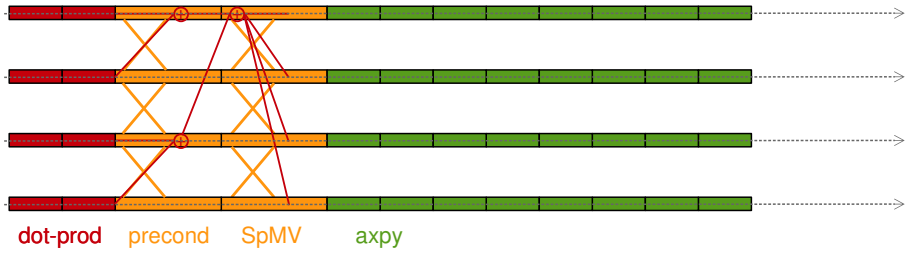

Figure 6: PipePCG

5 Libraries and HPC environment

5.1 PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) [23, 24] is a suite of data structures and routines that provides building blocks for the implementation of large-scale application codes on parallel computers. Written in C, PETSc uses the MPI standard for all message-passing communication. The bulding blocks that PETSc provides are for example parallel vectors and matrices or number of parallel linear and nonlinear solvers. PETSc also has many nice features like automatic profiling or a possibility to change applications behaviour by command line options.

The version 3.5.4 of PETSc was used for all the numerical experiments that can be found in Section 6.

5.2 PERMON

PERMON [25] is set of tools for the Parallel, Efficient, Robust, Modular, Object-oriented, Numerical simulations. It provides an unique combination of DDM and QP algorithms. Its applications include elasto-plasticity problems, medical imaging, or modelling of ice-sheet melting. PERMON is based on/extends PETSc in a way similar to SLEPc or TAO.

5.2.1 PermonQP

PermonQP is the main PERMON module providing an easy to use, massively parallel framework for the solution of QP problems. After specification of a QP problem we can optionally apply QP transformations. A QP transformation creates a new QP problem, that should be easier to solve, and saves it as a node of a doubly linked list. A QP transformation function also inserts a reconstruction function, so that we can transform the solution of the new QP problem back into the the solution of the original one. When the QP problem is simple enough, we solve it using a solver selected automatically or manually. This workflow is illustrated in Figure 7.

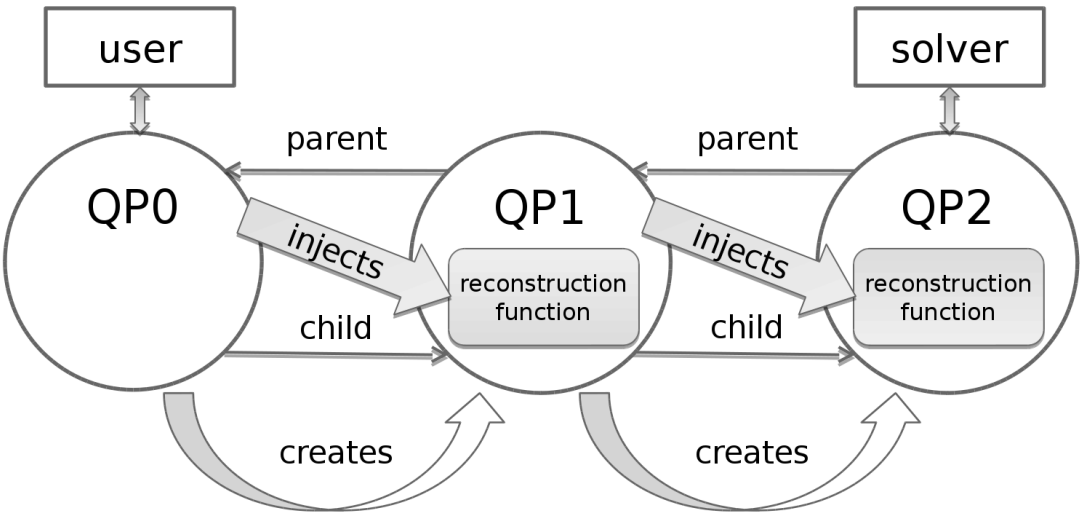

Figure 7: QP transformation chain: illustration of the PermonQP workflow [10]

5.2.2 PermonFLLOP

PermonFLLOP provides an implementation of the TFETI method. User provides a stiffness matrix, load vector and domain mapping. PermonFLLOP then computes the kernel $\boldsymbol{R}$ of the stiffness matrix and the gluing matrix $\boldsymbol{B}$. Afterwards a series of QP transformations is executed, most importantly the dualization, homogenization of the equality constraint and its enforcement by means of the projector onto the natural coarse space.

5.2.3 PermonCube

PermonCube is a benchmark that creates a mesh up to billions of unknowns over a cubical domain. It assembles the FEM objects associated with the problem. Then it uses PermonFLLOP to find the problem solution.

5.3 ARCHER

The numerical tests were run on the ARCHER supercomputer [26]. The ARCHER is a Cray XC30 based supercomputer. It consists of 4920 compute nodes. Each compute node contains two 2.7 GHz, 12-core Intel E5-2697 v2 (Ivy Bridge) processors and at least 64 GB of memory. Compute nodes are interconnected by the Aries interconnect using a Dragonfly topology. According to the current (November 2015) TOP500 list [27], the ARCHER is the 41st most powerful supercomuter with Rmax of 1642.5 TFlop/s in the Linpack benchmark.

As a compiler the cce/8.4.1 (Cray Compiler Environment) was used. A MPI library was provided by the cray-mpich/7.2.6 module. The libsci/13.2.0 was used as an implementation of the BLAS, LAPACK and Scalapack routines.

6 Numerical Experiments

6.1 Problem Description

The PermonCube (see Section 5.2.3) benchmark was used for the numerical experiments. The problem generated in each test was a 3D linear elastic cube with the bottom face fixed, and the top face loaded with a surface force. The loading was $f = 465N/mm^2$, side length $1mm$, Young's modulus $E = 2 \cdot 10^5\ MPa$ and the Poisson's ratio $\mu = 0.33$. There were always 64 elements per subdomain, and the number of subdomains was variable for different test cases. Note that our results for CP solution do not depend on the number of elements in the subdomains.

6.2 Results

After the initial tests for the 216 subdomains, MUMPS was disregarded as a solver, because even on a small problem it performed worse (in the strategy S1 by 95%, in the S2 by 12%) than SuperLU. Moreover, SuperLU should scale better (see Section 4.2.1). It also turned out that the preconditioner for the CG methods is too expensive to apply. Therefore we denote unpreconditioned PCG as CG, and PipePCG as PipeCG respectively.

In both strategies, we denote by GGtinv a preprocesing phase that includes the time needed for the creation of the subcommunicators and copying of the $\boldsymbol{GG}^T$ matrix into the subcommunicators. Additionally for the strategy S1, it includes the time needed for the factorization of $\boldsymbol{GG}^T$, and for the strategy S2 the $\left(\boldsymbol{GG}^T\right)^{-1}$ computational time is included. Let us reiterate that N_r represents the number of subcommunicators epmloyed, and N_c is the number of cores. Note that number of cores is the same as the number of subdomains ($N_c = N_s$).

In the following text, you can find tables with a performance given for several numbers of the subdomains (sizes of CP). Each table shows the time of CP solution and its dependence upon the number of subcommunicators. Graphs representing these data accompany each table.

Note that for the 13824 subdomains the CG method was disregarded, as was PipeCG for the 27000 subdomains, because they did not performed as well as SuperLU.

					GGtinv + X CP appl.		
N_r (N_c/N_r)	Solver	Strategy	GGtinv	CP appl.	$X = 100$	$X = 100$	$X = 1000$
5 (200)	SuperLU	S1	0.0191	0.0072	0.7394	7.2219	72.046
10 (100)	SuperLU	S1	0.0212	0.0043	**0.4528**	4.3374	43.183
50 (20)	SuperLU	S1	0.1314	0.0049	0.6231	5.0483	49.300
10 (100)	SuperLU	S2	3.4010	0.0008	3.4773	4.1642	11.033
50 (20)	SuperLU	S2	0.7147	0.0008	0.7910	**1.4778**	**8.3464**
100 (10)	SuperLU	S2	3.1202	0.0008	3.1965	3.8834	10.752
10 (100)	CG	S2	11.130	0.0008	11.206	11.893	18.762
50 (20)	CG	S2	2.0988	0.0008	2.1751	2.8620	9.7305
100 (10)	CG	S2	2.1599	0.0008	2.2362	2.9231	9.7916
10 (100)	PipeCG	S2	5.8369	0.0008	5.9132	6.6001	13.469
50 (20)	PipeCG	S2	1.3949	0.0008	1.4712	2.1581	9.0266
100 (10)	PipeCG	S2	2.0276	0.0008	2.1039	2.7908	9.6593

Table 1: 1000 subdomains: Time in seconds for different settings. The best time for the GGtinv + a number of CP applications is given in bold.

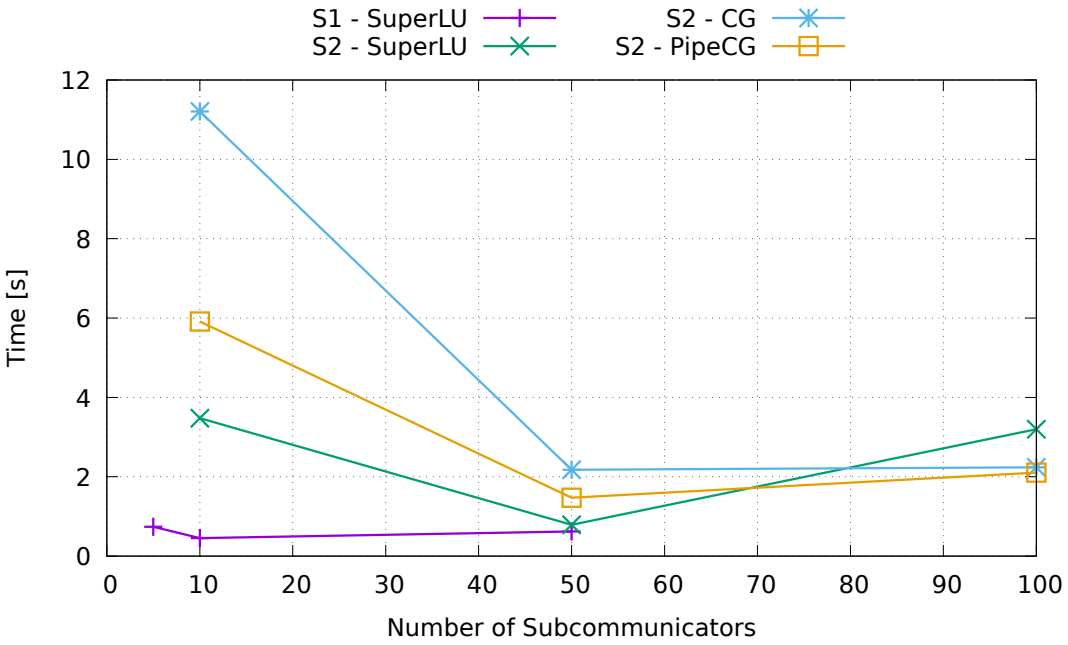

Figure 8: 1000 subdomains: GGtinv + 100 CP applications

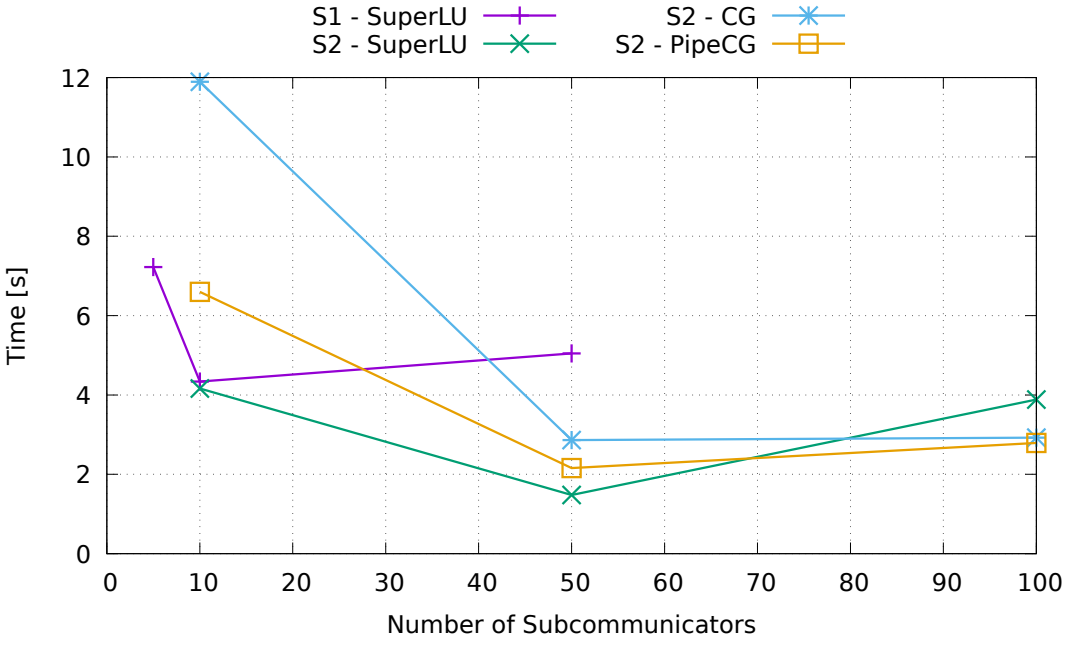

Figure 9: 1000 subdomains: GGtinv + 1000 CP applications

N_r (N_c/N_r)	Solver	Strategy	GGtinv	CP appl.	GGtinv + X CP appl.		
					$X = 100$	$X = 100$	$X = 1000$
32 (128)	SuperLU	S1	0.9911	0.0095	1.9371	10.450	95.584
64 (64)	SuperLU	S1	0.1033	0.0110	**1.2065**	11.136	110.43
128 (32)	SuperLU	S1	1.3202	0.0115	2.4716	12.835	116.47
192 (21.3)	SuperLU	S1	2.1806	0.0128	3.4626	15.001	130.38
256 (16)	SuperLU	S1	3.3994	0.0169	5.0882	20.287	172.28
64 (64)	SuperLU	S2	5.1644	0.0013	5.2931	6.4514	18.034
128 (32)	SuperLU	S2	4.0476	0.0013	4.1763	**5.3346**	**16.917**
192 (21.3)	SuperLU	S2	4.1978	0.0013	4.3265	5.4848	17.068
256 (16)	SuperLU	S2	5.9242	0.0013	6.0529	7.2112	18.794
64 (64)	CG	S2	13.914	0.0013	14.043	15.201	26.784
128 (32)	CG	S2	8.6927	0.0013	8.8214	9.9797	21.563
192 (21.3)	CG	S2	7.1820	0.0013	7.3107	8.4690	20.052
256 (16)	CG	S2	8.3792	0.0013	8.5079	9.6662	21.250
64 (64)	PipeCG	S2	9.6049	0.0013	9.7336	10.892	22.475
128 (32)	PipeCG	S2	7.4454	0.0013	7.5741	8.7324	20.316
192 (21.3)	PipeCG	S2	6.7212	0.0013	6.8499	8.0082	19.592
256 (16)	PipeCG	S2	7.9968	0.0013	8.1255	9.2838	20.867

Table 2: 4096 subdomains: Time in seconds for different settings. The best time for the GGtinv + a number of CP applications is given in bold.

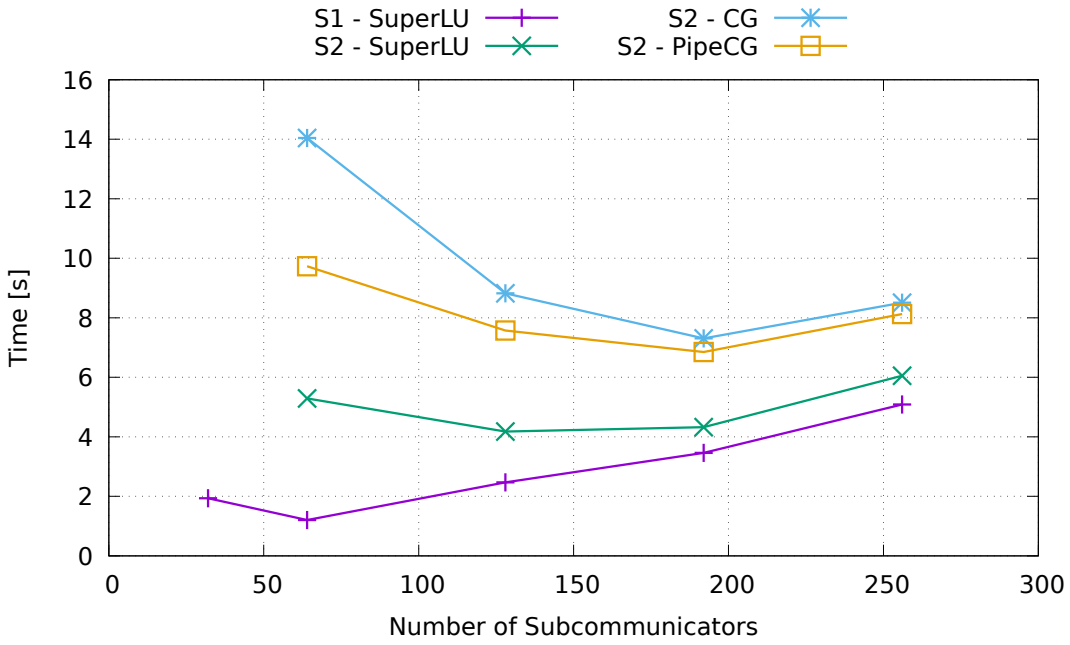

Figure 10: 4096 subdomains: GGtinv + 100 CP applications

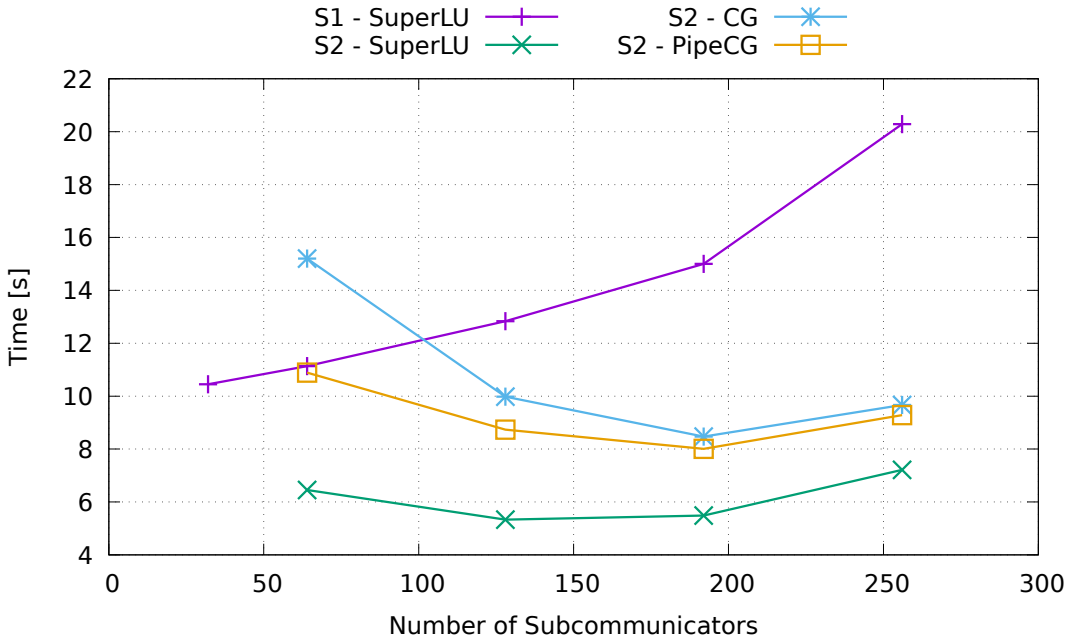

Figure 11: 4096 subdomains: GGtinv + 1000 CP applications

N_r (N_c/N_r)	Solver	Strategy	GGtinv	CP appl.	GGtinv + X CP appl.		
					$X = 100$	$X = 100$	$X = 1000$
20 (400)	SuperLU	S1	0.0497	0.0264	2.6868	26.420	263.76
50 (160)	SuperLU	S1	0.1222	0.0173	**1.8526**	17.426	173.16
80 (100)	SuperLU	S1	0.2038	0.0249	2.6986	25.152	249.69
125 (64)	SuperLU	S1	0.5049	0.0242	2.9291	24.747	242.93
250 (32)	SuperLU	S1	2.2229	0.0273	4.9519	29.513	275.12
125 (64)	SuperLU	S2	11.700	0.0025	11.954	14.243	37.129
250 (32)	SuperLU	S2	9.2003	0.0025	9.4546	**11.743**	**34.630**
320 (25)	SuperLU	S2	11.106	0.0025	11.360	13.649	36.535
125 (64)	CG	S2	23.044	0.0025	23.298	25.587	48.473
250 (32)	CG	S2	20.851	0.0025	21.105	23.394	46.280
320 (25)	CG	S2	17.978	0.0025	18.232	20.521	43.407
125 (64)	PipeCG	S2	18.277	0.0025	18.531	20.820	43.706
250 (32)	PipeCG	S2	16.557	0.0025	16.811	19.100	41.986
320 (25)	PipeCG	S2	19.443	0.0025	19.697	21.986	44.872

Table 3: 8000 subdomains: Time in seconds for different settings. The best time for the GGtinv + a number of CP applications is given in bold.

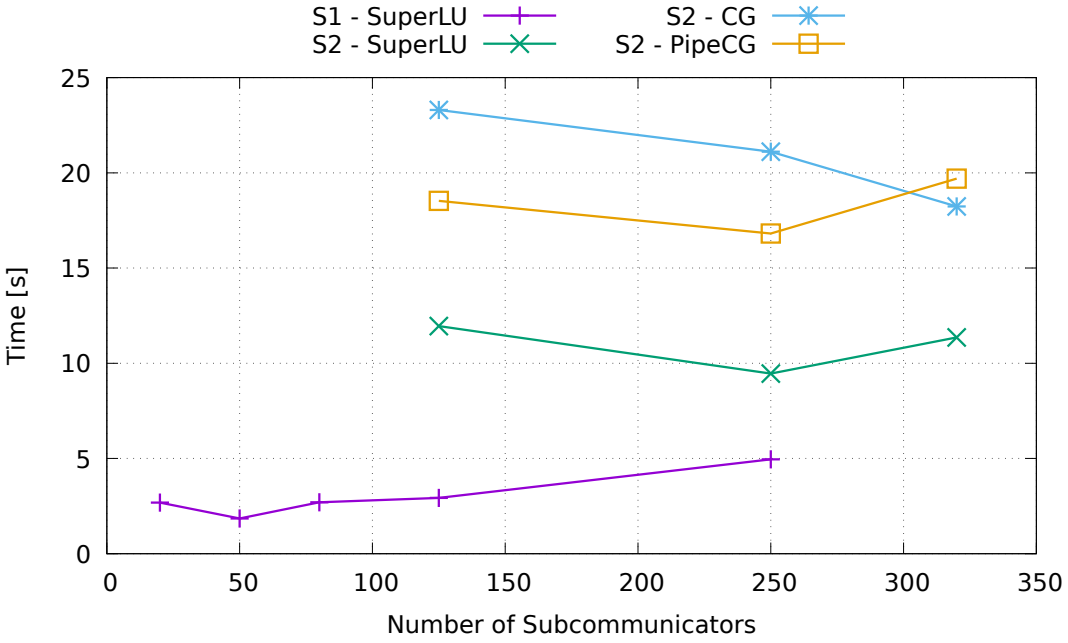

Figure 12: 8000 subdomains: GGtinv + 100 CP applications

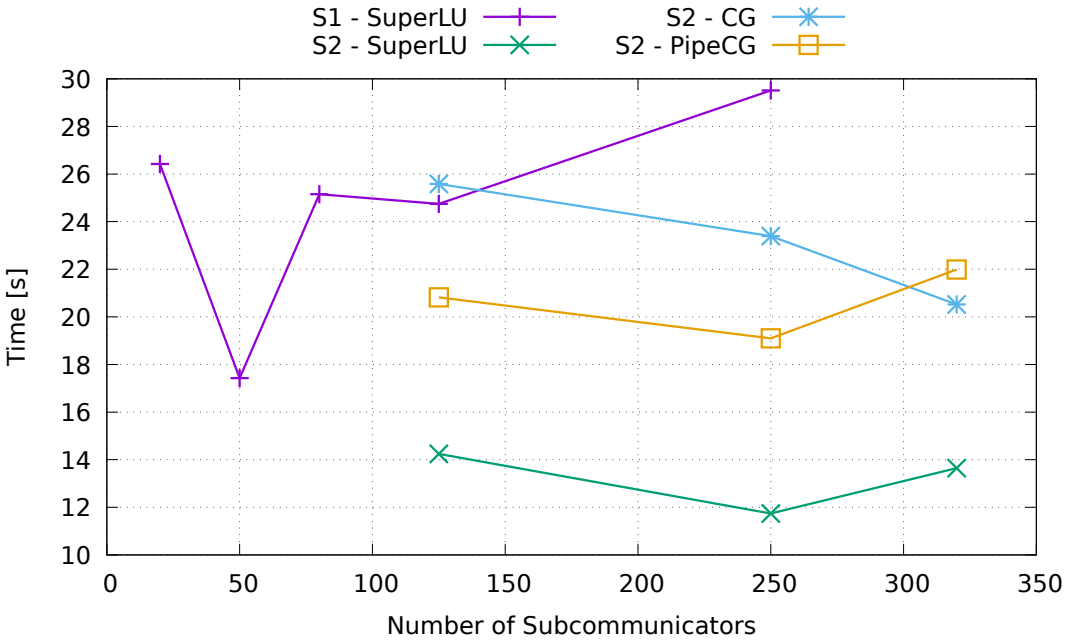

Figure 13: 8000 subdomains: GGtinv + 1000 CP applications

N_r (N_c/N_r)	Solver	Strategy	GGtinv	CP appl.	GGtinv + X CP appl.		
					$X = 100$	$X = 100$	$X = 1000$
32 (432)	SuperLU	S1	0.1073	0.0282	2.9244	28.278	281.82
48 (288)	SuperLU	S1	0.2006	0.0244	**2.6373**	24.568	243.87
72 (192)	SuperLU	S1	0.2920	0.0264	2.9301	26.674	264.11
144 (96)	SuperLU	S1	1.2215	0.0310	4.3248	32.255	311.56
216 (64)	SuperLU	S2	19.400	0.0049	22.467	26.873	70.936
432 (32)	SuperLU	S2	19.274	0.0049	19.764	**24.170**	**68.234**
576 (24)	SuperLU	S2	21.977	0.0049	19.890	24.296	68.359
144 (96)	PipeCG	S2	37.776	0.0049	38.266	42.672	86.735
216 (64)	PipeCG	S2	37.408	0.0049	37.898	42.304	86.367
432 (32)	PipeCG	S2	61.530	0.0049	62.020	66.426	110.489
576 (24)	PipeCG	S2	67.763	0.0049	68.253	72.659	116.722

Table 4: 13824 subdomains: Time in seconds for different settings. The best time for the GGtinv + a number of CP applications is given in bold.

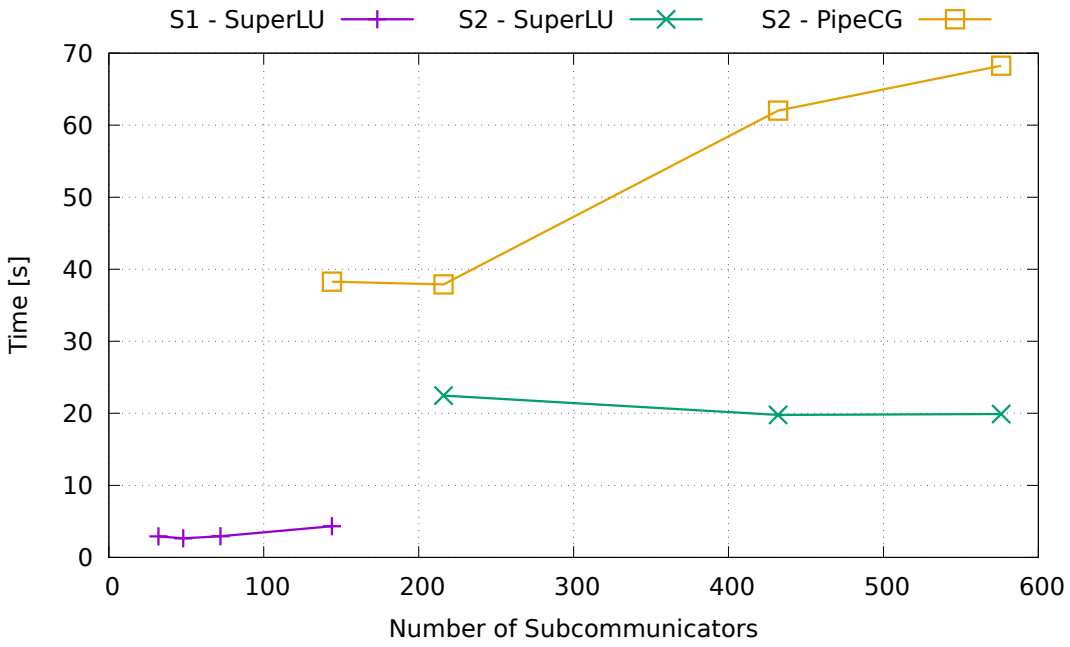

Figure 14: 13824 subdomains: GGtinv + 100 CP applications

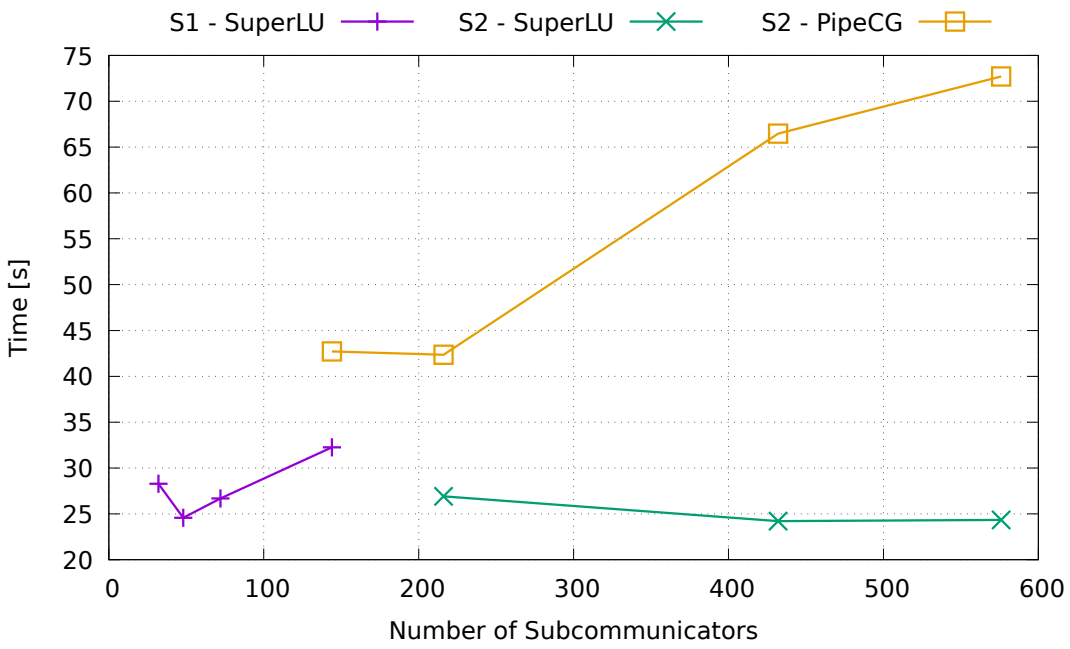

Figure 15: 13824 subdomains: GGtinv + 1000 CP applications

N_r (N_c/N_r)	Solver	Strategy	GGtinv	CP appl.	GGtinv + X CP appl.		
					$X = 100$	$X = 100$	$X = 1000$
25 (1080)	SuperLU	S1	0.1165	0.0516	5.2758	51.710	516.05
50 (540)	SuperLU	S1	0.2089	0.0473	**4.9427**	**47.547**	473.59
72 (375)	SuperLU	S1	0.2823	0.0478	5.0613	48.072	478.18
125 (64)	SuperLU	S2	48.504	0.0085	49.356	57.025	133.71
250 (32)	SuperLU	S2	41.063	0.0085	41.915	49.584	**126.27**
320 (25)	SuperLU	S2	42.568	0.0085	43.420	51.089	127.77

Table 5: 27000 subdomains: Time in seconds for different settings. The best time for the GGtinv + a number of CP applications is given in bold.

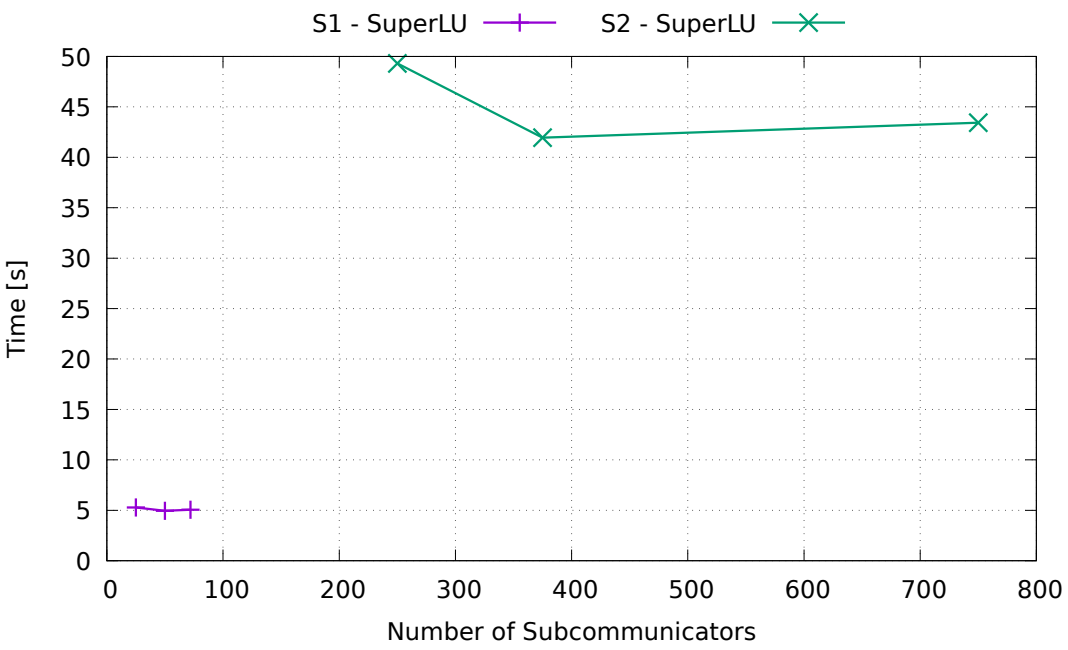

Figure 16: 27000 subdomains: GGtinv + 100 CP applications

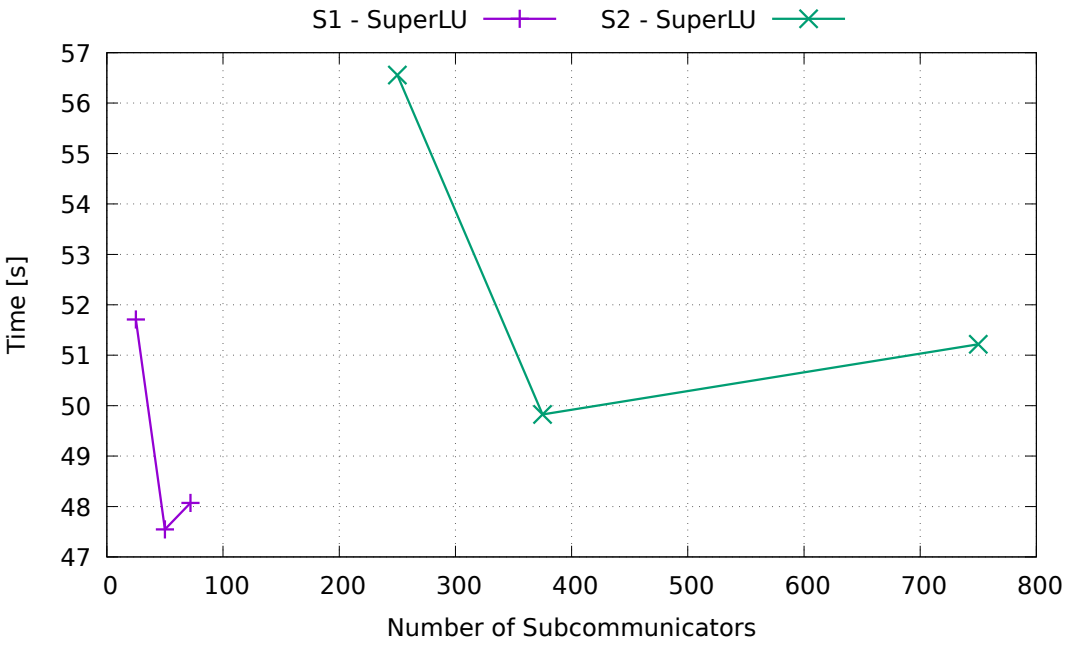

Figure 17: 27000 subdomains: GGtinv + 1000 CP applications

6.3 Evaluation of CP Strategies

From the previous section, it is clear that the direct solver SuperLU is best suited for CP solution. This is true regardless of the strategy chosen.

For the following discussion about apparent patterns in each strategy, we will concern ourselves with the number of subdomains of 4096 and higher. This is done because in the smaller cases the results are biased by several influences (like caches), and so they are highly impractical for the identification of patterns.

The strategy S1 seems to perform optimally with about 50 subcommunicators. Keeping the number of subcommunicators approximately constant leads to the increase in the number of cores in the subcommunicators by the same factor as the increase in the number of subdomains. The time for factorization scales well, but since a single CP application does not scale linearly, the strategy scales badly because a number of CP applications is needed.

For the strategy S2 it appears that up to the 13824 subdomains it is best to keep the number of cores per subcommunicator constant at about 32 cores per subcommunicator. This however does not hold true for the 27000 subdomains case, where we see a shift in favor of the employment of more cores per the subcommunicators. Increase in the number of subdomains by a certain factor leads to the increase of a locally owned portion of $\boldsymbol{GG}^T$ and also of the number of columns of the identity matrix (right hand sides) by the same factor. We can see that the time for both the GGtinv and the application of CP also grows approximately by this factor too. This again leads to poor weak scalability.

The scalability of both strategies is shown in Figures 18 to 20 for the different numbers of applications of CP. These graphs nicely illustrate that scalability of the strategy S1 depends on scalability of CP application, while scalability of the strategy S2 depends on scalability of the GGtinv.

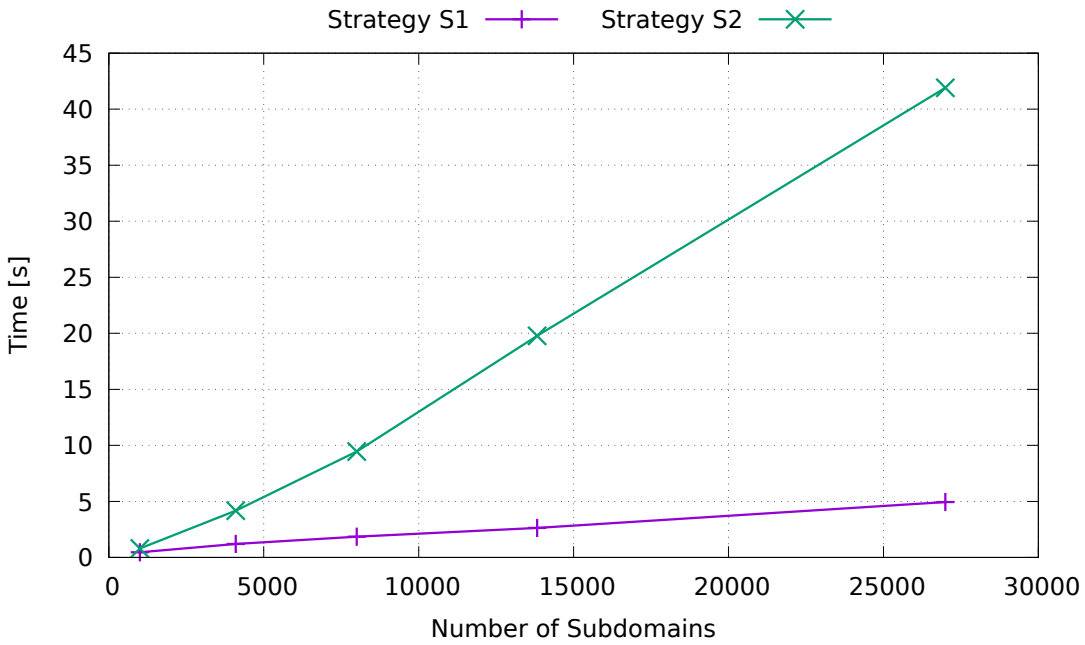

Figure 18: Weak scaling: GGtinv + 100 CP applications

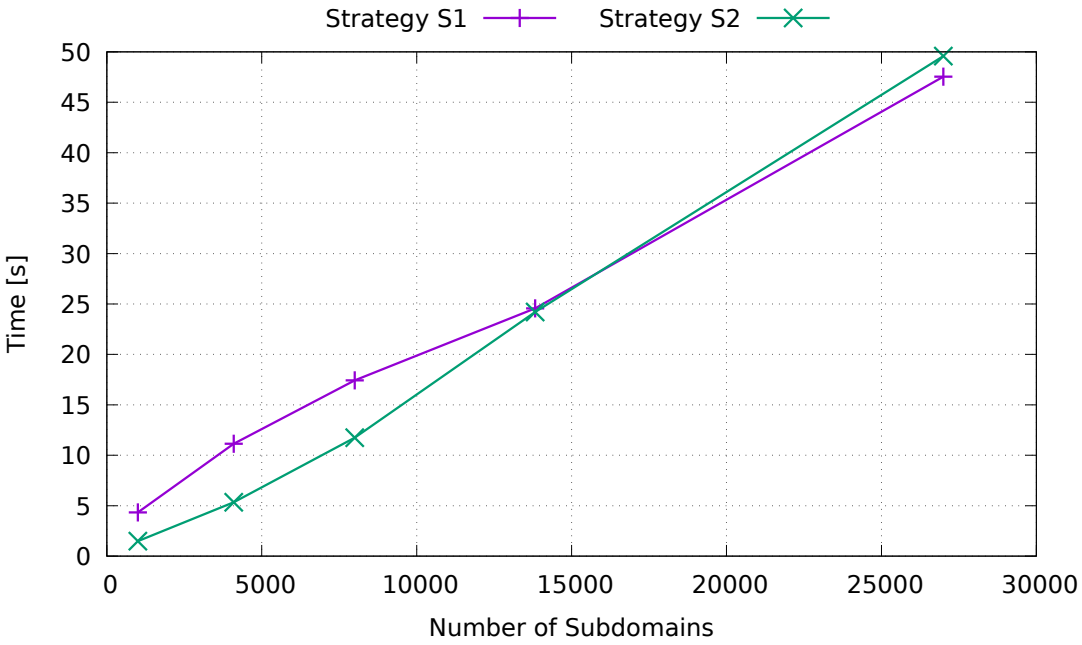

Figure 19: Weak scaling: GGtinv + 1000 CP applications

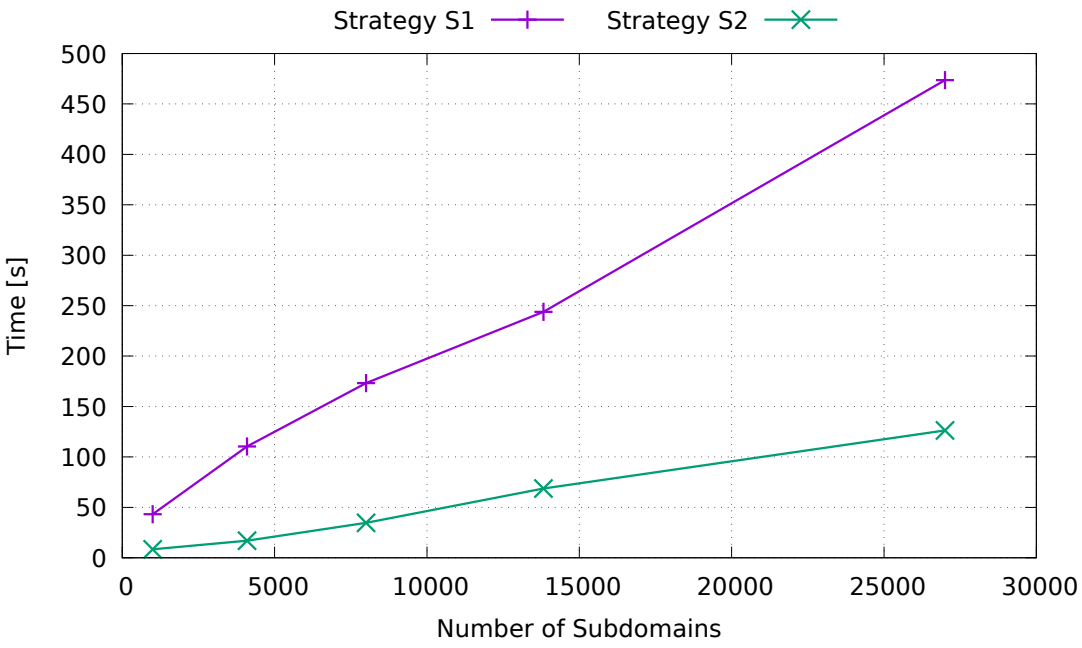

Figure 20: Weak scaling: GGtinv + 10000 CP applications

The GGtinv phase of the strategy S1 is very cheap in comparison with the strategy S2. On the other hand, the strategy S1 has much more expensive CP applications compared with the strategy S2. So it is no surprise, that for problems with higher number of expected CP applications, it is better to use the strategy S2. We could already notice that from the scalability

graphs in Figures 18 to 20. This is further illustrated in Figure 21, where combinations of the number of subdomains and expected iterations on the left of (or above) the line performs better when the strategy S2 is employed, and vice versa the strategy S1 is better on the right of (or below) the dividing line.

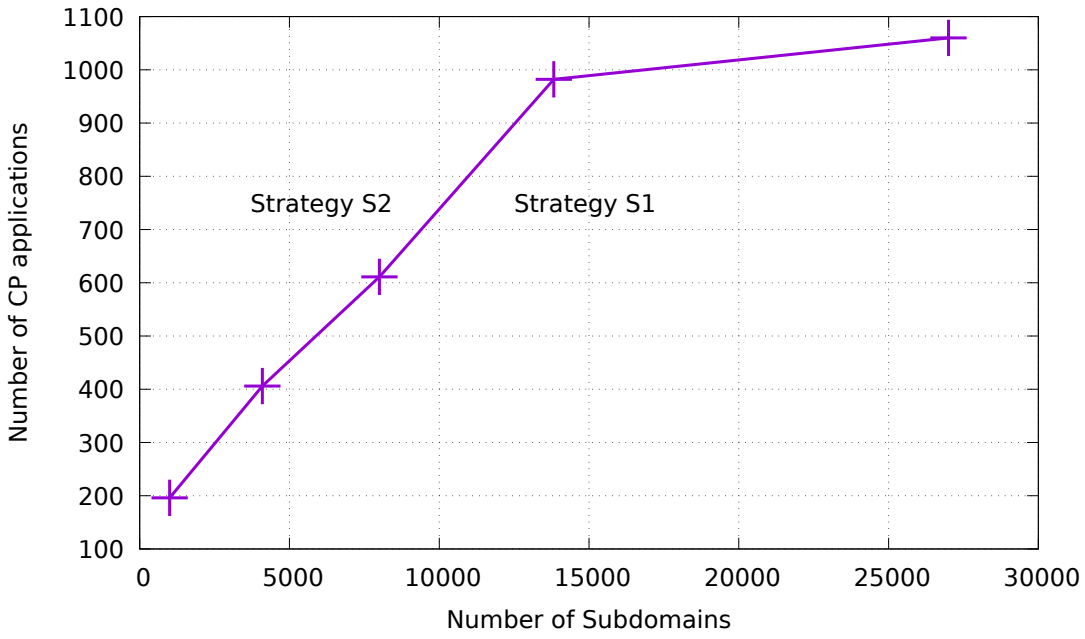

Figure 21: Choice of a strategy depends on the number of expected CG iterations and the number of subdomains

7 Conclusion

In this thesis, the most promising strategies for solution of the TFETI coarse problem were analysed. The analysis dealt with all the important ingredients, including data distribution and communication, for a massively parallel implementation. The performance measurements were done from the 1000 up to 27000 subdomains, and a further discussion of the performance trends was provided.

All programming work is now part of the PERMON library. I have improved the implementation of the strategy S1 by circumventing a PETSc bug (that is now fixed in the current development branch), gaining a speedup of 4 to 7 on the ARCHER supercomputer. I have also rewritten a portion of a PERMON code related to the explicit inverse (strategy S2), so that it works with newer versions of PETSc and is more efficient.

While these changes lead to a significant improvement in the performance of CP solution phase, they could not eliminate the bottleneck that the CP is. This bottleneck is an intrinsic part of the TFETI method. Some methods (like HTFETI [28]) attempts to reduce this problem. However, to eliminate the bottleneck of CP altogether a novel approach is required.

In future work, I would like to focus on a further research into (and development of) the FETI based methods. Also, I would like to further participate on the development of the PERMON toolbox, as the QP approach is very intriguing. In the course of this work I have become interested in the Krylov subspace methods and a communication avoiding and hiding techniques as well.

Jakub Kružík

References

[1] C. Farhat and F.-X. Roux, "A method of finite element tearing and interconnecting and its parallel solution algorithm", *International Journal for Numerical Methods in Engineering*, vol. 32, no. 6, pp. 1205–1227, 1991, ISSN: 1097-0207. DOI: 10.1002/nme.1620320604.

[2] Z. Dostál, D. Horák and R. Kučera, "Total FETI – an easier implementable variant of the FETI method for numerical solution of elliptic PDE", *Communications in Numerical Methods in Engineering*, vol. 22, no. 12, pp. 1155–1162, 2006, ISSN: 1099-0887. DOI: 10.1002/cnm.881.

[3] T. Kozubek, V. Vondrak, M. Mensik, D. Horak, Z. Dostal, V. Hapla, P. Kabelikova and M. Cermak, "Total FETI domain decomposition method and its massively parallel implementation", *Advances in Engineering Software*, vol. 60-61, pp. 14–22, Jun. 2013, ISSN: 09659978. DOI: 10.1016/j.advengsoft.2013.04.001.

[4] V. Hapla and D. Horak, "A comparison of FETI natural coarse space projector implementation strategies", in *Proceedings of the Third International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, UK: Civil-Comp Press, 2013, ISBN: 9781905088560. DOI: 10.4203/ccp.101.6.

[5] D. P. Bertsekas, *Nonlinear Programming*, 2nd. Athena Scientific, Sep. 1999, ISBN: 1886529000.

[6] Z. Dostál, *Optimal Quadratic Programming Algorithms: With Applications to Variational Inequalities*, ser. Springer Optimization and Its Applications. Springer US, 2009, vol. 23, ISBN: 9780387848068.

[7] C. Farhat, J. Mandel and F. X. Roux, "Optimal convergence properties of the FETI domain decomposition method", *Computer Methods in Applied Mechanics and Engineering*, vol. 115, no. 3–4, pp. 365–385, 1994, ISSN: 0045-7825. DOI: 10.1016/0045-7825(94)90068-X.

[8] C. Farhat and F.-X. Roux, "An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems", *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 1, pp. 379–396, Jan. 1992, ISSN: 0196-5204. DOI: 10.1137/0913020.

[9] T. Kozubek, D. Horak and V. Hapla, "FETI coarse problem parallelization strategies and their comparison", PRACE, White Paper. [Online]. Available: http://www.prace-ri.eu/IMG/pdf/feticoarseproblemparallelization.pdf (visited on 07/04/2016).

[10] V. Hapla, "Massively parallel quadratic programming solvers with applications in mechanics", PhD thesis, VSB – Technical University of Ostrava, 2016.

[11] V. Hapla and D. Horák, "TFETI coarse space projectors parallelization strategies", in *Parallel Processing and Applied Mathematics - 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part I*, 2011, pp. 152–162. DOI: 10.1007/978-3-642-31464-3_16.

[12] G. H. Golub and C. F. van Loan, *Matrix Computations*, Fourth. JHU Press, 2013, ISBN: 1421407949 9781421407944.

[13] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM, 1997, ISBN: 0898713617.

[14] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface", vol. 31, no. 3, pp. 302–325, Sep. 2005.

[15] X. S. Li and J. W. Demmel, "Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems", *ACM Trans. Mathematical Software*, vol. 29, no. 2, pp. 110–140, Jun. 2003.

[16] P. R. Amestoy, I. S. Duff, J. Koster and J.-Y. L'Excellent, "A fully asynchronous multifrontal solver using distributed dynamic scheduling", *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2001.

[17] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent and S. Pralet, "Hybrid scheduling for the parallel solution of linear systems", *Parallel Computing*, vol. 32, no. 2, pp. 136–156, 2006.

[18] P. R. Amestoy, I. S. Duff, J.-Y. L'excellent and X. S. Li, "Analysis and comparison of two general sparse solvers for distributed memory computers", *ACM Trans. Math. Softw.*, vol. 27, no. 4, pp. 388–421, Dec. 2001, ISSN: 0098-3500. DOI: 10.1145/504210.504212.

[19] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems", *Journal of research of the National Bureau of Standards*, vol. 49, pp. 409–436, 1952.

[20] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain", Pittsburgh, PA, USA, Tech. Rep., 1994.

[21] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm", *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014, 7th Workshop on Parallel Matrix Algorithms and Applications, ISSN: 0167-8191. DOI: 10.1016/j.parco.2013.06.001.

[22] A. Chronopoulos and C. Gear, "S-step iterative methods for symmetric linear systems", *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153–168, 1989, ISSN: 0377-0427. DOI: 10.1016/0377-0427(89)90045-9.

[23] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith and H. Zhang, "PETSc users manual", Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.5, 2014.

[24] S. Balay, W. D. Gropp, L. C. McInnes and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries", in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset and H. P. Langtangen, Eds., Birkhäuser Press, 1997, pp. 163–202.

[25] PERMON web page, [Online]. Available: http://permon.it4i.eu (visited on 21/04/2016).

[26] ARCHER web page, [Online]. Available: http://www.archer.ac.uk/ (visited on 22/04/2016).

[27] TOP500 November 2015 list, [Online]. Available: http://top500.org/list/2015/11/ (visited on 22/04/2016).

[28] T. Brzobohatý, M. Jarošová, T. Kozubek, M. Menšík and A. Markopoulos, "The Hybrid Total FETI method", in *Proceedings of the Third International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, UK: Civil-Comp Press, 2013, ISBN: 9781905088560. DOI: 10.4203/ccp.101.2.