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

The hybrid parallelization of TFETI-1 method

Hybridní paralelizace TFETI-1 metody

2015 Radim Sojka

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

Zadání diplomové práce

Student: **Bc. Radim Sojka**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 1103T031 Výpočetní matematika

Téma: Hybridní paralelizace TFETI-1 metody
The hybrid parallelization of TFETI-1 method

Zásady pro vypracování:

Metody FETI typu jsou velmi úspěšné k řešení obrovských inženýrských úloh. Důvodem je to, že dualita redukuje velkou dekomponovanou primární úlohu na menší duální relativně dobře podmíněnou úlohu řešenou iteračně např. metodou sdružených gradientů. Total-FETI-1 (TFETI-1) [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 nejen vývojem škálovatelných algoritmů, které kombinují klasický FETI přístup s efektivními iteračními a přímými řešiči, resp. jejich kombinacemi, ale i jejich paralelní implementací a testováním škálovatelnosti na různých architekturách. Řada metod a potřebných komponent je již implementována v knihovně FLLOP (FETI Light Layer On top of PETSc) [2], [3], vyvíjené na Katedře aplikované matematiky a využívající k paralelizaci MPI.
V současné době je trendem zvětšování počtu výpočetních jader na procesoru. Takováto jednotka se pak chová jako systém se sdílenou pamětí. Je zřejmé, že MPI paralelizace v rámci jednoho procesoru nebude tou optimální. Nabízí se tedy možnost tzv. hybridní paralelizace kombinující MPI paralelizaci mezi procesory s využitím vláken např. OpenMP uvnitř procesoru v rámci jedné sdílené paměti. Samotné PETSc tuto kombinaci ve své developerské verzi již podporuje.
Cílem práce je analýza dostupných nástrojů použitelných k hybridní paralelizaci metody TFETI-1 implementované v knihovně FLLOP, následná vlastní implementace a systematické porovnání škálovatelností demonstrující přínos této kombinace vůči implementaci využívající pouze MPI na obrovských inženýrských úlohách řešených na desetitisících výpočetních jader.

FETI methods are very successful for the solution of large scale engineering problems. The reason is that duality reduces large primal decomposed problem to smaller dual, relatively well conditioned problem, which is solved iteratively using e.g. conjugate gradient method. Total-FETI-1 (TFETI-1) [1] simplifies the inversion of stiffness matrices of subdomains by using Lagrange multipliers not only for gluing the subdomains along the auxiliary interfaces, but also for the implementation of the Dirichlet boundary conditions. This method may be even more efficient than the original FETI-1.
Our research concerns not only the development of the scalable algorithms combining standard FETI approach with efficient iterative and direct solvers or their combinations respectively, but also their parallel implementation and scalability testing on various architectures. A lot of methods and required components are implemented in FLLOP library (FETI Light Layer On top of PETSc) [2], [3] being developed at the Department of applied mathematics and using MPI for the parallelization.
In this time, there is a trend of increasing number of cores per one processor. Such a computational unit behaves then as the shared memory system. It is obvious that MPI parallelization in the scope of this processor is not optimal. Thus there is possible so called hybrid parallelization combining the MPI parallelization among the processors with the usage of pthreads e.g. OpenMP inside one processor with

shared memory. PETSc itself supports this combination in its developer branch.
The aim of the thesis is the analysis of the available tools being suitable for the hybrid parallelization of TFETI-1 method implemented in FLLOP library, the subsequent real implementation and the systematic scalability comparison demonstrating the benefit of this combination in relation to the implementation using MPI only for the large-scale engineering problems solved by means of tens thousand cores.

Seznam doporučené odborné literatury:

[1] Z. Dostál, D. Horák, 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.
[2] http://www.mcs.anl.gov/petsc/
[3] http://industry.it4i.cz/produkty/fllop/

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

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

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA 1

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

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

I declare that I have worked up this thesis by myself. I have referenced all the sources and publications I have used.

Ostrava, May 7, 2015 .....

I would like to thank to supervisor of my thesis, Ing. David Horák, Ph.D. for his help, advices and suggestions.

Further I thank to Ing. Václav Hapla especially for the help with an implementation in PermonFLLOP library.

My gratitude also belongs to Ing. Alexandros Markopoulos, Ph.D. for help with tool PermonCube and to other members of PERMON team for their advices and support.

Finally, I would like to thank to my family and friends for their support during the creation of this thesis.

This thesis was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

Abstrakt

Práce se zabývá hybridní paralelizací TFETI-1 metody implementované v knihovně PermonFLLOP. Nejdříve představuje různé paralelní programovací modely a způsoby hybridní paralelizace. Poté uvádí možné výhody hybridní paralelizace v porovnání s čistou MPI paralelizací. Dále pak tato práce analyzuje balíčky poskytující přímé řešiče, které hrají klíčovou roli v implementaci TFETI-1, z hlediska vhodnosti pro hybridní paralelizaci. Nová implementace TFETI-1 metody rozšiřuje stávající implementaci paralelizovanou čistě pomocí MPI. Díky ní je možné držet data více podoblastí na jednom MPI procesu. To umožnuje využít dobrých vlastností numerické škálovatelnosti FETI metod. V numerických experimentech je pak testována numerická škálovatelnost a vliv hybridní paralelizace.

Klíčová slova: hybridní paralelizace, TFETI-1, PermonFLLOP, PETSc, škálovatelnost, přímé řešiče, OpenMP, MPI

Abstract

The thesis deals with the hybrid parallelization of the TFETI-1 method which is implemented in the library PermonFLLOP. At first, the thesis presents various parallel programming models and ways of the hybrid parallelization. There are listed the possible benefits of the hybrid parallelization compared with the pure MPI parallelization. Then the thesis analyzes the packages providing direct solvers, which have important role in the TFETI-1 implementation, in terms of suitability for hybrid parallelization. New implementation extends the existing pure MPI implementation. With this extension, data of more subdomains can be now stored per one MPI process. This allows the use of good properties of the numerical scalability of the FETI methods. In numerical experiments, the numerical scalability and the impact of hybrid parallelization are tested.

Keywords: hybrid parallelization, TFETI-1, PermonFLLOP, PETSc, scalability, direct solvers, OpenMP, MPI

List of used abbreviations and symbols

TFETI	–	Total finite element tearing and interconnecting
HPC	–	High performance computing
PCGP	–	Projected preconditioned conjugated gradient
PDE	–	Partial differential equation
Im	–	Image space
Ker	–	Null space
QP	–	Quadratic programming

List of tables

List of figures

1 Introduction

The FETI methods are very successful for the solution of large engineering problems. These methods enable massive parallelization and they can be efficiently used on HPC systems. The FETI methods are based on the decomposition of the original body into several subdomains. So, the problem can be solved in parallel for each subdomain. The continuity between subdomains is enforced using the Lagrange multipliers. The FETI methods use duality of the quadratic programming. Large decomposed primal problem is transformed to smaller and better conditioned dual problem. The FETI methods use combination of direct and iterative solvers.

The research at the Department of Applied Mathematics and at IT4Innovations, National Supercomputing Center concerns efficient implementation of the FETI methods. Permon is one of the libraries which implements these methods. This library is based on PETSc and uses MPI standard for parallelization.

The most powerful supercomputers consist of a large number of nodes. Each node has a few multi-core processors. The number of cores per node is increasing. All computational cores on one node share the same memory address space. Architecture of supercomputers encourages hybrid parallelization. It uses threads for parallelization inside shared memory node and message passing for communication between nodes. The hybrid parallelization can lead to more efficient algorithms than pure MPI approach.

The aims of the thesis are to analyze the available libraries and tools for hybrid paralelization and to implement real hybrid paralalelization. This implementation is based on the current implementation in PermonFLLOP. Consequently benefits of the new method of parallelization are demonstrated on numerical experiments.

The thesis is divided as follows. In section 2, TFETI-1 method is explained. There is desribed decomposition and assembly of the primal FETI objects, the transformation to the dual problem. Then PCGP algorithm and the bound of condition number of the system are shown.

Section 3 presents the main parallel programming models. Especially it is focused on OpenMP and MPI and their combined use in the hybrid parallelism.

In next section 4, libraries which are related to the thesis are introduced. One subsection focuses on direct solver packages which have an important role in the TFETI-1 implementation.

Section 5 shows practical implementation of TFETI-1 method. At first, original implementation of TFETI-1 in PermonFLLOP is briefly described. My own implementation enabling the hybrid parallelization is then introduced.

Finally, the benefits of the new implementation are demonstrated on numerical experiments, which were performed on supercomputer Anselm, in Section 6 .

2 FETI methods

The FETI-1 (Finite Element Tearing and Interconnecting) method proposed by Farhat and Roux in [1] is domain decomposition (DD) method which enables solving problems described by elliptic PDE in parallel. In FETI methods, the original domain is decomposed to smaller non-overlapping subdomains. Continuity between these subdomains is enforced by the Lagrange multipliers. This thesis is focused on one of the FETI methods - Total FETI-1 (TFETI-1) proposed by Dostál, Horák and Kučera in [2]. TFETI-1 is described in more detail in this section.

2.1 TFETI-1

Let us consider the elastic body represented by domain $\Omega \subset \mathbb{R}^d$ where $d = 2$ or $d = 3$. Let us suppose that the boundary Γ is decomposed into two disjoint parts Γ_U and Γ_F, $\Gamma = \overline{\Gamma}_U \cup \overline{\Gamma}_F$. There are prescribed Dirichlet boundary conditions on Γ_U and Neumann boundary conditions on Γ_F. We denote by $\mathbf{K_g}$ global stiffness matrix, $\mathbf{f_g}$ the load vector and $\mathbf{u_g}$ vector of displacements. The problem to be solved is

$$\mathbf{K_g u_g} = \mathbf{f_g}.$$

2.1.1 Decomposition and primal QP problem

The first step of TFETI-1 method is decomposition of Ω into N non-overlapping subdomains Ω_s as in Figure 1 After decomposition, the boundary of each subdomain Γ_s consists of three disjoint parts Γ_U^s, Γ_F^s and Γ_G^s, $\Gamma^s = \overline{\Gamma}_U^s \cup \overline{\Gamma}_F^s \cup \overline{\Gamma}_G^s$. There are prescribed Dirichlet boundary conditions on Γ_U^s and Neummann boundary conditions on Γ_N^s, which are inherited from original boundary conditions prescribed on Γ_U and Γ_F. Γ_G^s is part of boundary which is glued to other subdomains. We introduce "gluing" conditions on this artificial boundary which enforce continuity of the displacement between subdomains. Prescribed displacements on Γ_U^s and continuity of displacements on Γ_G^s are enforced by Lagrange multipliers in TFETI-1 method.

Decomposition and finite element discretization lead to the quadratic programming (QP) problem

$$\min_{\mathbf{u}} \frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathbf{K}\mathbf{u} - \mathbf{u}^{\mathrm{T}}\mathbf{f} \text{ s. t. } \mathbf{B}\mathbf{u} = \mathbf{c}, \tag{2.1}$$

where $\mathbf{K} = \mathrm{diag}(\mathbf{K_1}, \ldots, \mathbf{K_N})$ is a symmetric positive semidefinite block-diagonal stiffness matrix of order n, $\mathbf{f} = (\mathbf{f_1}, \ldots, \mathbf{f_N})^{\mathrm{T}} \in \mathbb{R}^n$ is load vector and $\mathbf{u} = (\mathbf{u_1}, \ldots, \mathbf{u_N})^{\mathrm{T}} \in \mathbb{R}^n$

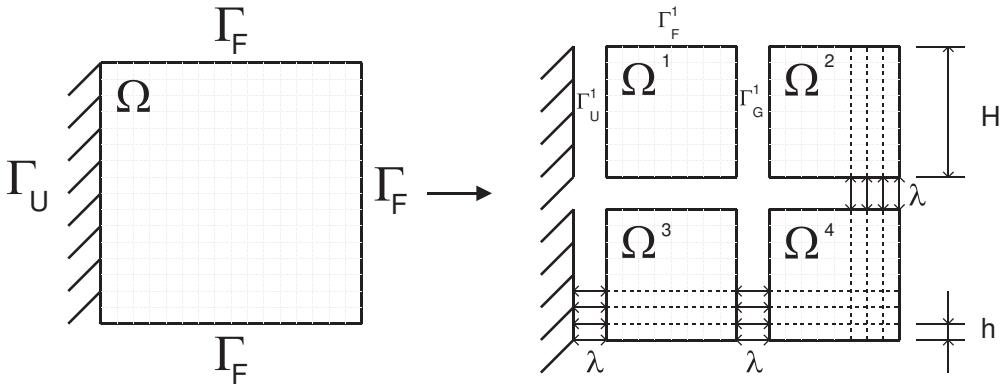

Figure 1: Decomposition and discretization of Ω

is a vector of displacement. $\mathbf{B}$ denotes a constraint matrix of order m by n and $\mathbf{c} \in \mathbb{R}^m$ is a constraint vector.

The matrix $\mathbf{B}$ with the rows $\mathbf{b_i}$ and the vector $\mathbf{c}$ with the entries $\mathbf{c_i}$ enforce the continuity of the displacement on the artificial boundary and the prescribed displacement on the boundary with imposed Dirichlet conditions. The continuity of the displacement in node which is shared by two subdomains is realized by the row $\mathbf{b_i}$ with zero entries except -1 and 1 at appropriate positions corresponding to this node and $\mathbf{c_i} = 0$. The Dirichlet boundary condition $\mathbf{u_j} = u_D(j)$ is enforced by the row with zero entries except 1 at the position j and $\mathbf{c_i} = u_D(j)$.

Because Dirichlet boundary conditions are enforced by constraints, all subdomains stiffness matrices have known and the same defect. The kernel $\mathbf{R_s}$ of the local stiffness matrix $\mathbf{K_s}$ can be formed directly.

The primal formulation (2.1) is not suitable for numerical solution, because $\mathbf{K}$ is typically ill-conditioned, singular and very large matrix. These complications may be reduced by applying the duality theory of convex programming.

2.1.2 Dual problem

All the constraints are enforced by the Lagrange multipliers $\boldsymbol{\lambda}$. Lagrangian associated with problem (2.1) is

$$L(\mathbf{u}, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathbf{K}\mathbf{u} - \mathbf{u}^{\mathrm{T}}\mathbf{f} + \boldsymbol{\lambda}^{\mathrm{T}}(\mathbf{B}\mathbf{u} - \mathbf{c}). \tag{2.2}$$

It is known, that (2.1) is equivalent to the saddle point problem

$$\text{Find } (\overline{\mathbf{u}}, \overline{\boldsymbol{\lambda}}) \text{ so that } L(\overline{\mathbf{u}}, \overline{\boldsymbol{\lambda}}) = \sup_{\boldsymbol{\lambda}} \inf_{\mathbf{u}} L(\mathbf{u}, \boldsymbol{\lambda}) \tag{2.3}$$

or equivalently find $(\overline{\mathbf{u}}, \overline{\boldsymbol{\lambda}})$ satisfying Karush–Kuhn–Tucker (KKT) conditions

$$\begin{bmatrix} \mathbf{K} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{B} & \mathbf{O} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{c} \end{bmatrix}. \tag{2.4}$$

We suppose that (2.4) is uniquely solvable and it is guaranteed by following necessary and sufficient conditions:

$$Ker\mathbf{B}^{\mathrm{T}} = \mathbf{o} \tag{2.5}$$

$$Ker\mathbf{K} \cap Ker\mathbf{B} = \mathbf{o}. \tag{2.6}$$

The first equation in (2.4) has a solution if and only if

$$\mathbf{f} - \mathbf{B}^{\mathrm{T}}\bar{\boldsymbol{\lambda}} \in \mathrm{Im}\mathbf{K}. \tag{2.7}$$

It can be expressed by means of a matrix $\mathbf{R} = \mathrm{diag}(\mathbf{R_1}, \ldots, \mathbf{R_N})$ whose columns span is the null space of $\mathbf{K}$:

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

As mentioned above we can easily assemble the matrix $\mathbf{R}$. For subdomain $\mathbf{\Omega^i} \subset \mathbb{R}^2$ discretized by means of n_i nodes with the coordinates $(x_j, y_j),\ j = 1, \ldots, n_i$

$$\mathbf{R}_i = \begin{bmatrix} \mathbf{R}_i^1 \\ \vdots \\ \mathbf{R}_i^{n_i} \end{bmatrix}, \qquad \mathbf{R}_i^1 = \begin{bmatrix} 1 & 0 & -y_j \\ 0 & 1 & x_j \end{bmatrix}. \tag{2.9}$$

Let us assume that $\bar{\boldsymbol{\lambda}}$ satisfies (2.7) and denote by $\mathbf{K}^+$ arbitrary generalized inverse of $\mathbf{K}$ that satisfies $\mathbf{K}\mathbf{K}^+\mathbf{K} = \mathbf{K}$.

If $\mathbf{u}$ solves the first equation in (2.4), then there is a vector $\boldsymbol{\alpha} \in \mathbb{R}^m$ such that

$$\bar{\mathbf{u}} = \mathbf{K}^+(\mathbf{f} - \mathbf{B}^{\mathrm{T}}\bar{\boldsymbol{\lambda}}) + \mathbf{R}\bar{\boldsymbol{\alpha}}. \tag{2.10}$$

After substituting expression (2.10) into second equation in (2.4) we get

$$\mathbf{B}\mathbf{K}^+\mathbf{B}^{\mathrm{T}}\bar{\boldsymbol{\lambda}} - \mathbf{B}\mathbf{R}\bar{\boldsymbol{\alpha}} = -\mathbf{c} + \mathbf{B}\mathbf{K}^+\mathbf{f}. \tag{2.11}$$

Summarizing (2.8) and (2.11) we obtain new problem to find $(\overline{\boldsymbol{\lambda}}, \overline{\boldsymbol{\alpha}})$ satisfying

$$\begin{bmatrix} \mathbf{B}\mathbf{K}^+\mathbf{B}^{\mathrm{T}} & -\mathbf{B}\mathbf{R} \\ -\mathbf{R}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}} & \mathbf{O} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{B}\mathbf{K}^+\mathbf{f} - \mathbf{c} \\ -\mathbf{R}^{\mathrm{T}}\mathbf{f} \end{bmatrix}. \tag{2.12}$$

Let us denote

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

We can write problem (2.12) as

$$\begin{bmatrix} \mathbf{F} & \mathbf{G}^{\mathrm{T}} \\ \mathbf{G} & \mathbf{O} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{d} \\ \mathbf{e} \end{bmatrix}. \tag{2.13}$$

Problem (2.13) has the same saddle-point structure as (2.4), however, its size is considerably smaller and $\mathbf{F}$ is much better conditioned than $\mathbf{K}$.

Solution $\bar{\boldsymbol{\lambda}}$ can be decomposed into $\bar{\boldsymbol{\lambda}}_{Im} \in Im\mathbf{G}^{\mathrm{T}}$ and $\bar{\boldsymbol{\lambda}}_{Ker} \in Ker\mathbf{G}$ as

$$\bar{\boldsymbol{\lambda}} = \bar{\boldsymbol{\lambda}}_{Im} + \bar{\boldsymbol{\lambda}}_{Ker}. \tag{2.14}$$

Let us introduce

$$\mathbf{Q} = \mathbf{G}^{\mathrm{T}}(\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-\mathbf{1}}\mathbf{G} \tag{2.15}$$

$$\mathbf{P} = \mathbf{I} - \mathbf{Q} \tag{2.16}$$

projectors on the image space of $\mathbf{G}^{\mathrm{T}}$ and on the kernel of $\mathbf{G}$, respectively.

Using particular solution

$$\bar{\boldsymbol{\lambda}}_{Im} = \mathbf{G}^{\mathrm{T}}(\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-\mathbf{1}}\mathbf{e}, \tag{2.17}$$

we can homogenize constraint equation

$$\mathbf{G}(\bar{\boldsymbol{\lambda}}_{Ker} + \bar{\boldsymbol{\lambda}}_{Im}) = \mathbf{e} \tag{2.18}$$

$$\mathbf{G}\bar{\boldsymbol{\lambda}}_{Ker} = \mathbf{e} - \mathbf{G}\mathbf{G}^{\mathrm{T}}(\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-1}\mathbf{e} = \mathbf{o} \tag{2.19}$$

After homogenization we obtain from (2.12)

$$\begin{bmatrix} \mathbf{F} & \mathbf{G}^{\mathrm{T}} \\ \mathbf{G} & \mathbf{O} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda}_{Ker} \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{d} - \mathbf{F}\bar{\boldsymbol{\lambda}}_{Im} \\ \mathbf{o} \end{bmatrix}. \tag{2.20}$$

Let us denote

$$\tilde{\mathbf{d}} = \mathbf{d} - \mathbf{F}\bar{\boldsymbol{\lambda}}_{Im}.$$

Now we eliminate $\boldsymbol{\alpha}$ by applying projector $\mathbf{P}$ on the first equation of (2.20). Solution $\bar{\boldsymbol{\lambda}}_{Ker}$ can be obtained from equation

$$\mathbf{P}\mathbf{F}\boldsymbol{\lambda}_{Ker} = \mathbf{P}\tilde{\mathbf{d}} \tag{2.21}$$

If $\bar{\boldsymbol{\lambda}}$ is known, we can obtain $\bar{\boldsymbol{\alpha}}$ from the first equation in (2.13)

$$\bar{\boldsymbol{\alpha}} = (\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-\mathbf{1}}\mathbf{G}(\mathbf{d} - \mathbf{F}\bar{\boldsymbol{\lambda}}) \tag{2.22}$$

and final solution $\bar{\mathbf{u}}$ from (2.10).

Diagonal blocks $\mathbf{K}^{\mathbf{s}}$ are in TFETI-1 method sparse and singular matrices with known kernels $\mathbf{R}^{\mathbf{s}}$ so we can regularize them efficiently. This enables the use of the Cholesky factorization for nonsingular matrices which is important for evaluating $\mathbf{K}^{+}\mathbf{x}$ for any vector $\mathbf{x}$. This is described in more detail in [3].

2.1.3 PCGP algorithm and TFETI convergence

The preconditioned conjugate projected gradient algorithm (PCGP) proposed in [4] can be used for solving (2.21). The PCGP algorithm is identical to the standard PCG

Algorithm 1 PCGP algorithm

Require: $\lambda_{Ker}^0 = 0$
$r^0 = d$

for k = 1, 2, ... until convergence **do**
Project: $w^{k-1} = Pr^{k-1}$;
Precondition: $z^{k-1} = \bar{F}^{-1} w^{k-1}$;
Re-project: $y^{k-1} = Pz^{k-1}$;
$\beta^k = (y^{k-1})^{\mathrm{T}} w^{k-1} / (y^{k-2})^{\mathrm{T}} w^{k-2}$; $(\beta^1 = 0)$
$p^k = y^{k-1} + \beta^k p^{k-1}$; $(p^1 = y^0)$
$\alpha^k = (y^{k-1})^{\mathrm{T}} w^{k-1} / (p^k)^{\mathrm{T}} F p^k$;
$\lambda_{Ker}^k = \lambda_{Ker}^{k-1} + \alpha^k p^k$;
$r_{Ker}^k = r^{k-1} - \alpha^k F p^k$;
end for

algorithm but applied to the system operator $\mathbf{PF}$ and the preconditioner $\mathbf{P\bar{F}^{-1}}$ so it is well known that error reduction bound in k-th iteration is

$$||e^k||_{\mathbf{PF}} \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k ||e^0||_{\mathbf{PF}}, \tag{2.23}$$

where κ is condition number of matrix $\mathbf{P\bar{F}^{-1}PF}$.

Now let us suppose that we have square or cube subdomains with the regular size H and assume that finite element discretization has the regular mesh size h. It was shown in [5] that for unpreconditioned case ($\bar{\mathbf{F}}^{-1} = \mathbf{I}$) the condition number of the FETI method applied to the Poisson or elasticity problem is bounded by:

$$\kappa(\mathbf{PF}|Im\mathbf{P}) \leq C \frac{H}{h}. \tag{2.24}$$

From bound (2.24) we can see that if the mesh size is fixed and the number of subdomains increases, i.e. subdomain size decreases, then the condition number of $\mathbf{PF}$ decreases. In many cases constant C in (2.24) is smaller for TFETI-1 than for classical FETI but dimension of the coarse problem in TFETI-1 is larger, so that it may happen that the number of the iterations may be slightly larger than for FETI [2].

For FETI methods two preconditioners were introduced, the first one called Dirichlet

$$\mathbf{D}^{-1} = \mathbf{B}^{\mathrm{T}}\mathbf{S}\mathbf{B}, \tag{2.25}$$

where $\mathbf{S} = \mathrm{diag}(\mathbf{S_1}, \ldots, \mathbf{S_N})$. With $\mathbf{S_i}$ denoting Schur complements of the subdomain stiffness matrices obtained by the elimination of the interior degrees of freedom. The second one is called Lumped

$$\mathbf{L}^{-1} = \mathbf{B}\mathbf{K}\mathbf{B}^{\mathrm{T}}. \tag{2.26}$$

The convergence of FETI methods using these preconditioners were numerically shown in [5] and the bound of the condition number for a problem with Dirichlet preconditioner was proven in [6]:

$$\kappa(\mathbf{P}\mathbf{D}^{-1}\mathbf{P}\mathbf{F}|Im\mathbf{P}) \leq C\left(1 + \log\frac{H}{h}\right)^{\gamma}, \tag{2.27}$$

with $\gamma = 3$, and $\gamma = 2$ in the special cases listed in Lemma 3.8. in [6].

3 Parallel computing and programming models

3.1 Parallel programming models

Depending on memory architecture, we can distinguish multiprocessors - computers with a shared memory and multicomputers (or clusters) - computers with a distributed memory architecture. The largest and the fastest computers in the world today use a hybrid distributed-shared memory.

Parallel programming model is an abstraction above hardware. In next sections, three main programming models are introduced. Although these models correspond to the mentioned memory architectures of computers, it is not necessary to use programming model on corresponding computer architecture. For example, distributed memory model (message passing) can be implemented on computers with shared, distributed and hybrid memory architecture.

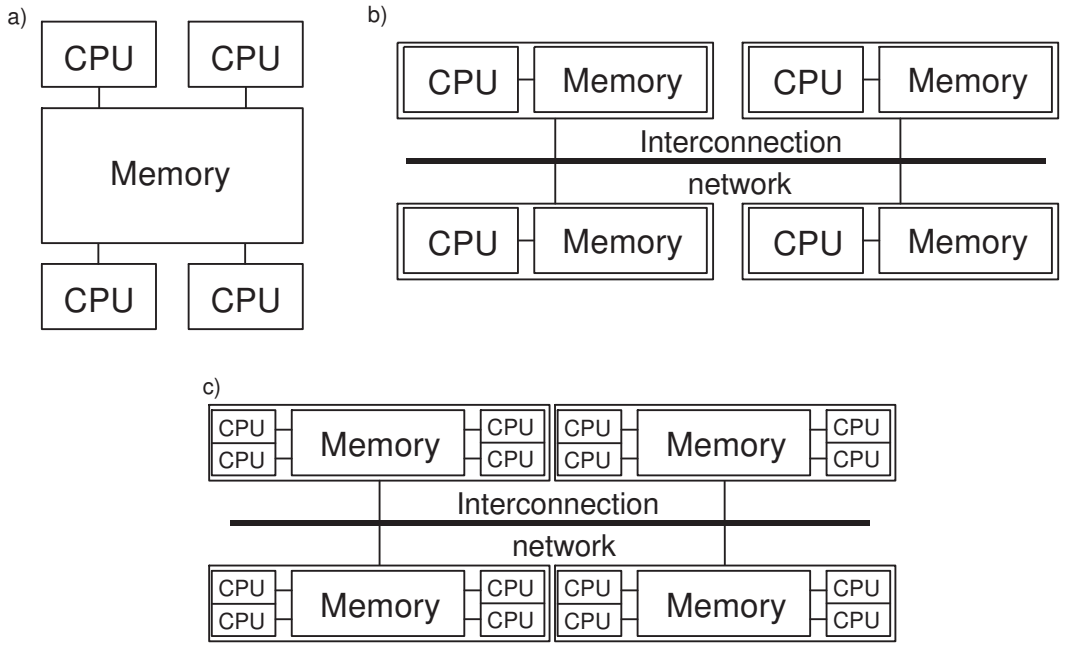

Figure 2: Memory architectures of parallel computers, a) shared memory computer, b) distributed memory computer, c) distributed-shared memory computer.

3.1.1 Shared memory model - threads

In this programming model, parallel tasks share a global address space, which they read and write to asynchronously. Threaded programming is one of the shared memory

models. Thread indicates sequence of instructions. In threads model, the process consists of global address space and threads which share it. Several mechanisms may be used for safe access to shared memory (locks, barriers, ...) in this model. One of the standards for the programming with threads is OpenMP.

The following terms related to the thread programming will be used in this thesis.

Data race - This is the name for conflict of threads during memory access. A data race occurs when two or more threads access to the same memory location and at least one performs writing. Data races are dangerous because they may not cause an error in each code execution so this is not obvious that application code is wrong. It is good to use appropriate debugging tools for detection all possible thread conflicts, for example Valgrind tools Helgrind or DRD, Allinea or Intel VTune Amplifier.

Thread safety - The code is thread-safe if it can be called from multiple threads without generating data races.

False sharing - Modern processors use for fast access to data local caches. Data is stored in units called cache lines. Copies of one cache line can be stored in the main memory and each local cache at the same time. Cache coherency ensures that each modification of data in cache line is propagated throughout the entire memory. False sharing is a situation when more processors update different elements of the same cache line at one time frequently. If one processor updates an element of cache line, whole line is marked as invalid. Other processors which use different element in the same cache line must wait for the update of the whole cache line. So false sharing has big impact on the application performance.

OpenMP

OpenMP (Open Multi-Processing) is a collection of compiler directives, library routines and environmental variables, which enables to parallelize sequential code written in languages Fortran, C or C++, on most processor architectures and operating systems. OpenMP is managed by the nonprofit technology consortium OpenMP ARB. More informations about standard OpenMP can be found at [8].

The programming with OpenMP is based on a fork-join model (see Figure 8). OpenMP program is executed sequentially in one master thread until the first directive which creates a parallel region. Then master thread creates group of parallel threads, where each one has unique identifier (thread ID). In this parallel region, threads can access to the shared memory but also use private variables, which cannot be accessed by another

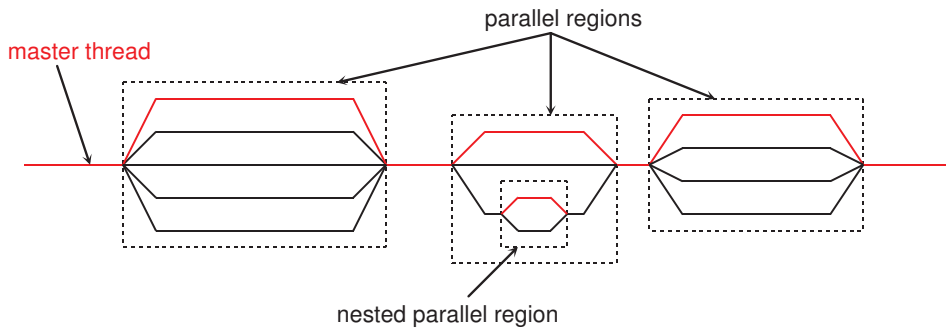

Figure 3: Fork-join model of OpenMP

thread. At the end of parallel regions, the implicit synchronization is performed and all threads except the master are terminated.

OpenMP directives have following syntax:

```
#pragma omp directive-name [clause, ... ]
```

So for example, to execute block of code with private variable `i` in 4 threads, we use

```
#pragma omp parallel num_threads(4) default(shared) private(i)
{
    threaded code
}
```

OpenMP also contains several runtime routines. Examples of the most important are

```
void omp_set_num_threads (int num_threads)
```

which sets number of threads in next parallel regions or

```
int omp_get_thread_num (void)
```

to obtain unique thread ID. A list of OpenMP directives, clauses, routines or environment variables and their description for version 4.0 can be found in [9].

3.1.2 Distributed memory model - message passing

Message passing model is based on communications between processes. Each process has a local memory address space, which can not be directly accessed by other processes. There is no global memory. The data exchange is performed by sending and receiving messages.

MPI

Message Passing Interface (MPI) is a message-passing library interface specification. The MPI standard is defined and maintained by MPI Forum [10], which is an open group which has many participants including vendors, researchers, software library developers etc. Last version of MPI standard MPI-3.0 has been released in September 2012 and its documentation is available from [11].

The goal of MPI is to establish a practical, portable, efficient and flexible standard for message-passing. Interface specifications allow language bindings for C, C++ and Fortran77. There are many of libraries implementing the MPI standard. Most used ones are freely available libraries MPICH, LAM/MPI and Open MPI. MPI is the most popular model used in high-performance computing today.

As mentioned above, the MPI program is formed by a collection of processes, which can exchange messages. For MPI-1 standard, static process model is used. It means that number of processes is set at beginning of program. Standard MPI-2 enables to change number of processes during program execution. Launching an MPI program is usually done using command mpirun or mpiexec with parameter `-n N`. This ensures start of MPI program with `N` processes which are distributed on available processors (or cores). These processes can be clustered into so called communicators.

MPI operations can be divided into several groups. There is environment management routines, for example function `MPI_Init` and `MPI_Finalize` for initialization and termination of MPI execution environment or `MPI_Comm_size` and `MPI_Comm_rank` for obtaining size of communicator and rank of process.

Exchange of messages is provided by point-to-point or collective communication. MPI point-to-point operations ensure message passsing only between two processes. One process is performing send operation and the other one is performing a matching receive operation. Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode. The difference is that program does not wait for completion of the operation. Typical representatives of the point-to-point operations are functions `MPI_Recv`, `MPI_Send` for receiving and sending message.

Collective operations involve all or a subset of the processes of parallel program. Routine `MPI_Bcast` enables to broadcast of a message from the "root" process to all other processes, operations `MPI_Scatter`, `MPI_Gather` distributes messages from one process to all processes in the communicator and gathers messages from each communicator to one root process, respectively. Collective computation operation `MPI_Reduce` applies a reduction on all processes in communicator and places the result in the root process. For the synchronizations of groups of processes the MPI operation `MPI_Barrier` is used.

3.1.3 Hybrid model

A hybrid model combines more than one programming model. A typical example of a hybrid model is combination of models which were described above - message passing model with threads model. Currently also hybrid model with MPI + GPUs is increasingly used.

Implementation of distributed/shared memory model is usually performed using MPI and OpenMP. Since the release of standard MPI 3.0, which supports shared memory paralellization, it is possible to use hybrid MPI+MPI model (MPI is used for inter-node communication and MPI 3.0 for shared memory programming) [12].

MPI + OpenMP model

In this programming model, MPI performs communication between nodes and each MPI process is parallelized using OpenMP. There are several methods how to implement this model. These methods differ dividing computations and communications between threads.

We use categorization specified in [13]:

1. **Hybrid masteronly model** - without overlapping MPI communications and computations. MPI communication routines are called only outside parallel regions by master thread in this model. Disadvantages of this model are
   - All other threads are idle during communication of the master thread. It can have major impact on performance.
   - The full inter-node MPI bandwidth might not be saturated by using a single communication thread.

```
#pragma omp parallel
{
    /* Some parallelized computations */
}
/* Only master thread − sequential */
MPI_Send(...)
MPI_Recv(...)
#pragma omp parallel
{
    /* Some parallelized computations */
}
```

Code 1: Example of Hybrid masteronly model

2. **Hybrid with overlap** - while one thread performs or more threads perform communication, other threads execute computations so communications and computations overlap communication.

   - Compute threads do not idle during communication.
   - If we use more threads for communication, they can saturate full MPI bandwidth. However, the fewer threads are reserved for the computation.
   - It is more difficult to implement this model.

```
#pragma omp parallel
{
    if (thread_ID < n){
        /* communications thread */
        MPI_Send(...)
        MPI_Recv(...)
    } else {
    /* compute threads */
    /* Some parallelized computations */
    }
    /* all threads */
    /* Some parallelized computations which needs data*/
}
```

Code 2: Example of hybrid model with overlap with n threads reserved for communications

Thread safety of MPI

Standard MPI defines four levels of thread safety:

`MPI_THREAD_SINGLE` - A process is executed only with one thread.

`MPI_THREAD_FUNNELED` - Support for multithreading but only thread that initialized of MPI can call MPI routines.

`MPI_THREAD_SERIALIZED` - Support for multithreading but only one thread can call MPI routines at one time.

`MPI_THREAD_MULTIPLE` - Support for multithreading and multiple threads can call MPI routines at any time.

There is need to call MPI initialization function

```
int MPI_Init_thread( int *argc, char ***argv, int required,
                     int *provided )
```

instead of `MPI_Init` to determinate level of thread safety. Input parameter `required` is level of desired thread support. Output parameter `provided` determines level of provided thread support.

Threading support of MPI implementation must be sometimes set before the compilation. For example the configuration of Open MPI with option

```
configure --enable-mpi-threads
```

sets support for full `MPI_THREAD_MULTIPLE`.

In numerical experiment in section 6, the Intel MPI implementation is used. Thread safe version of Intel MPI Library is set by using the compiler driver option

```
-mt_mpi
```

Detailed description and analysis of MPI thread safety is in [16].

3.2 Pure MPI vs. MPI+OpenMP on HPC systems

As already mentioned, the most powerful HPC systems in the world are based on distributed-shared memory architecture. In most cases, the shared memory nodes, which are connected via network infrastructure, consist of a couple of multi-core processors. For example, supercomputer Anselm has two 8-core processors per nodes [14].

Following approaches for parallelization are typically used for HPC systems having this architecture.

- **Pure MPI** - MPI process is created for each core.
- **Fully hybrid MPI/OpenMP** - OpenMP is used for parallelization inside memory shared node and communication between nodes is realized using MPI.
- **Mixed hybrid MPI/OpenMP** - There is more than one MPI process per node. For example, one MPI process is created per multi-core CPU and OpenMP parallelizes code within CPU.

Although hybrid model seems to be more suitable for current HPC systems, often pure MPI codes can overcome the hybrid code.

The possible advantages of a hybrid approach compared to pure MPI are

- Additional level of parallelism. It can help in some applications where performance of MPI parallelism is limited to a certain number of processes. Then nested OpenMP parallelism can improve performance by using more cores.
- Better load balancing. It is very difficult to implement dynamic load balancing using MPI. However, OpenMP offers dynamic loop scheduling.
- Reducing memory usage. OpenMP parallelization inside node eliminates data duplication. Moreover, threads use less memory than process, which allocates extra memory to manage communication and MPI environment.
- Reducing additional communication. The number of sent and received messages per node decreases by using hybrid parallelization. It reduces adverse effect of MPI latency.

The possible reasons, why hybrid code is slower than pure MPI code:

- All threads except master sleep during MPI communication in masteronly model.

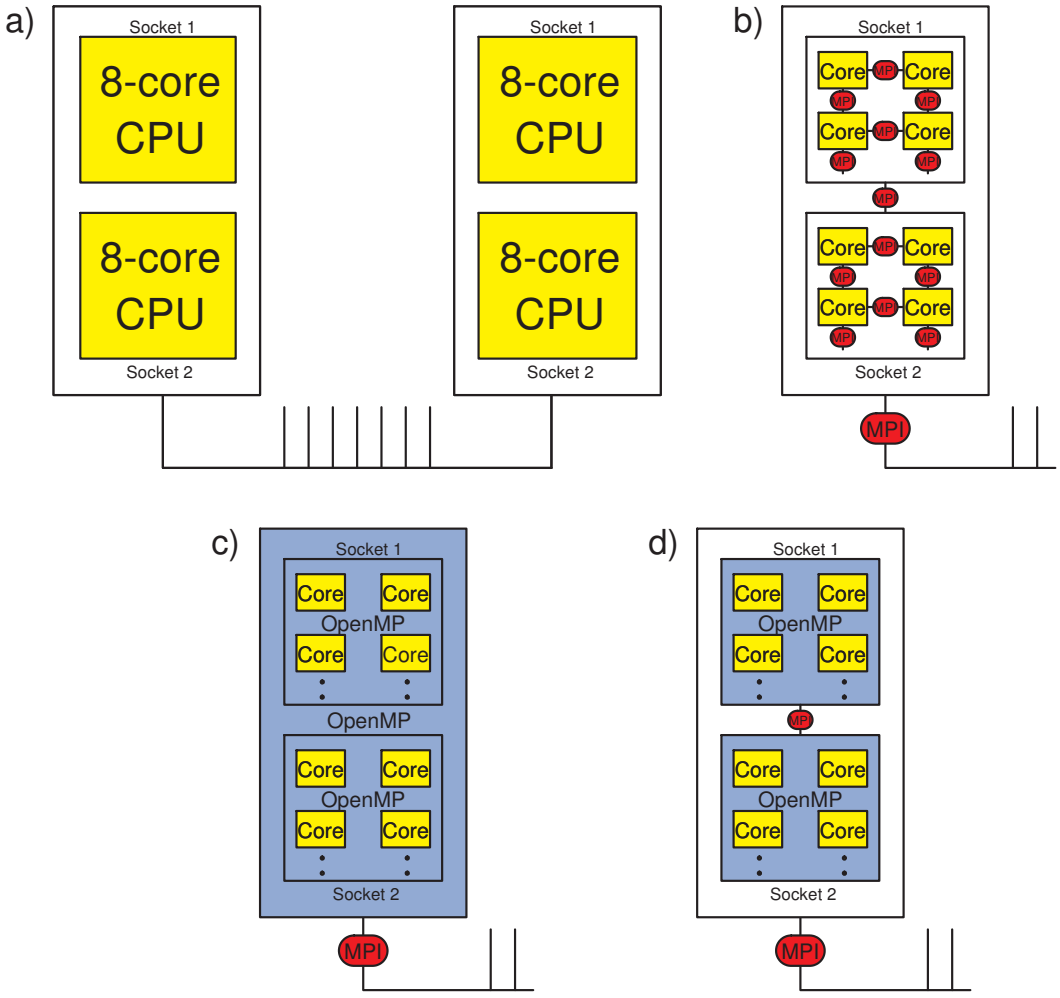

Figure 4: Typical methods for parallelization on multisocket HPC systems-a): b) Pure MPI model, c) fully hybrid model, d) mixed hybrid model.

- Overhead of thread creation.
- Cache coherency and false sharing issues.

More informations about hybrid parallel programming and comparison with pure MPI approach can be found in [13], [15].

4 Parallel numerical libraries

4.1 PERMON Toolbox

The aim of this thesis was to implement the hybrid parallelization of the TFETI-1 method in the PermonFLLOP package [18] which has been originally developed by Václav Hapla and David Horák.

The PermonFLLOP is one of the modules included in PERMON (Parallel, Efficient, Robust, Modular, Object-oriented, Numerical) toolbox [17]. It is the toolbox developed at IT4Innovations and it is designed to solve large scale problems from engineering (linear elasticity, contact problems, elasto-plasticity, shape optimization and other) but also problems of medical imaging, climate changes etc. The PERMON packages are written in C/C++ language and they are based on PETSc library.

Another module of PERMON is PermonQP which provides a base for the solution of quadratic programming problems. The main operation of PermonQP is performing QP transformations that reformulate an original QP problem, create chain of these QP problems and the last one is solved by QP solver (QPS). PermonQP allows solving unconstrained QP problems and equality constrained ones.

The PermonFLLOP package is an extension of the PermonQP, adding domain decomposition methods of the FETI type. It provides support for assembling of the FETI-specific objects. Then a special combination of QP transformations from PermonQP is called (dualization, homogenize equality constraint and enforcing equality constraint by projector) and the last QP problem is solved by combination of direct solvers (for stiffnes matrix pseudoinverse action and the coarse problem solution) and iterative solver (main FETI loop).

PERMON tool PermonCube [19] serves to generating benchmarks for PermonFLLOP. The typical benchmark is a cantilever beam loaded on the top and fixed on the bottom. Number of subdomains N_x, N_y, N_z (in x, y and z directions) and number of elements per subdomains n_x, n_y, n_z can be set from command line. Essential data provided by PermonCube are the stiffness matrix $\mathbf{K}$ and the load vector $\mathbf{f}$, both assembled in purely parallel way on each MPI process. PermonCube is able to assemble also linear equality constraint matrix $\mathbf{B}$ for both gluing and Dirichlet boundary conditions.

PermonFLLOP can be used by general pure C array interface PermonAIF. It can be useful if user does not use object-oriented PETSc API. PermonAIF manages just one instance of QP, created implicitly by an initialization routine. The user can provide necessary data using PermonAIF API. It is the subdomain stiffness matrices in CSR format,

the load vector and the initial solution vector as a double array. The solution vector is overwritten by the computed solution by PermonFLLOP. Further the constraint matrix and vector and kernels of subdomain stiffness matrices can be supplied. PermonFLLOP solvers can be called as shown below:

```
FllopAIFInitialize(comm,argc,args);
...
FllopAIFSetArrayBase(1);
FllopAIFSetFETIOperator(n, i, j, symmetry, v, name);
FllopAIFSetRhs(n, b, name);
FllopAIFSetSolutionVector(n, x, name);
FllopAIFSolve();
...
FllopAIFFinalize();
```

4.2 PETSc

PETSc (The Portable, Extensible Toolkit for Scientific Computation) is a parallel numerical library developed by groups of programmers from Argone National Laboratory [20].

This library can be used for the numerical solution of partial differential equations and related problems on high-performance computing. PETSc is a suite of data structures and routines that provide the building blocks for the implementation of large-scale application codes on parallel computers.

PETSc includes number of parallel linear and nonlinear solvers and provides many of the mechanisms needed within parallel application codes. The library is organized hierarchically, enabling users to employ the level of abstraction that is most appropriate for a particular problem (see Fig. 5). Application codes may be written in Fortran, C, C++, Python or MATLAB.

PETSc uses MPI standard for inter-process communications and supports shared memory (pthreads, OpenMP), hybrid (MPI + OpenMP or pthreads) and GPUs (CUDA or OpenCL) parallelism (more in Subsection 4.2.1). PETSc also uses routines from BLAS, LAPACK, LINPACK etc. and interfaces to many external software, for example MATLAB, MUMPS, SuperLU.

PETSc consists of a variety of libraries (similiar to classes in C++). Each library manipulates with a particular family of objects and the operations. Some of the PETSc modules deal with

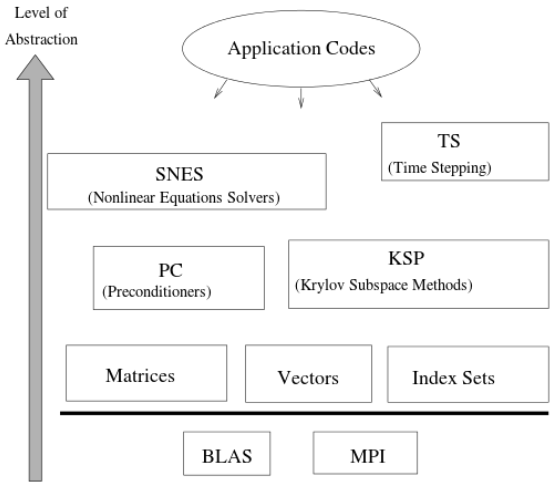

Figure 5: Organization of the PETSc library.

- index sets (IS)
- vectors (Vec)
- matrices (Mat)
- data management - managing interactions between mesh data structures and vectors and matrices (DM)
- Krylov subspace methods (KSP)
- preconditioners including sparse direct solvers (PC)
- nonlinear solvers (SNES)
- timesteppers for solving time-dependent PDEs (TS)

4.2.1 PETSc and threads

Currently PETSc supports shared memory parallelism with OpenMP and pthreads. In combination with MPI it enables to run hybrid (MPI + OpenMP) parallel code.

However, there are parallelized only some of methods at the level matrix and vectors operations. User are free also to use threads for OpenMP in their code so long as that code does not make PETSc calls. A more complicated model of PETSc with threads, which would allow users to write threaded code that made PETSc calls, is not supported

because PETSc is not currently thread-safe. Authors of PETSc state that there are few reasons, why PETSc is not thread safe. First PETSc uses a few miscellaneous global variables. This is not big problem and it may be fixed in next versions of PETSc. Second reason is a variety of global data structures for profiling and error handling. This cannot be easily modified because some profiling data structures are constantly updated by the running code and simply putting locks around all of these data accesses would have major impact on performance of library. Another problem is that all the PETSc objects created during a execution do not have locks associated with them and the reasons is again performance.

It is recommended to work with developers versions of PETSc if user wants to use threads. For compilations with threads and OpenMP, the following modification of compilation script is necessary:

```
--with-threadcomm

--with-openmp
```

eventually for pthreads support:

```
--with-pthreadclasses.
```

For execution of threaded code, three runtime options are important:

`-threadcomm_nthreads` <nthreads>: to set the number of threads

`-threadcomm_affinities` <list_of_affinities>: to set the core affinities of threads

`-threadcomm_type` <nothread,pthread,openmp>: to set threading model

Threads are managed by the object `PetscThreadComm` (this is similar to the MPI communicator). For each function which supports threading there exists the kernel function. For example, if we call PETSc routine for dot product `VecDot`, there is called sequence of functions which execute kernel `VecDot_kernel` in parallel. Each thread works with a certain part of vector. This range is defined by array `trstarts` held in `PetscLayout` object for each vector (or matrix).

Threaded PETSc calls MPI routine `MPI_Init_thread` with the level of thread support `MPI_THREAD_FUNNELED` in own initialization function `PetscInitialize`.

4.3 Direct solver packages

This section focuses on parallel linear direct solver packages. Direct solvers have an important role in TFETI-1. Packages listed here are available from PETSc. Key properties for the use in the TFETI-1 hybrid implementation are the support of the shared memory/hybrid parallelism and threadsafety.

MUMPS

Package MUMPS (MUltifrontal Massively Parallel Solver) [22] is parallel sparse direct solver for systems of linear equations with unsymmetric, symmetric positive definite, or general symmetric matrices. The last version MUMPS 5.0.0 released in February 2015 was developed by employees of CERFACS, ENS Lyon, INPT(ENSEEIHT)-IRIT, Intria and University of Bordeuax. MUMPS is based on multifrontal method and provides LU, Cholesky and LDLT factorizations. MUMPS is not thread-safe but it supports hybrid parallelization. It is necessary to use threaded BLAS library for these purposes. The code of the last version includes preliminary experimental OpenMP directives to additional gains of multithreading. Number of threads is controlled by the OpenMP environment variable `OMP_NUM_THREADS`.

SuperLU

SuperLU library [23] exists in three different variants

- SuperLU - sequential variant
- SuperLU_MT - shared memory parallel variant
- SuperLU_DIST - distributed memory parallel variant

SuperLU is based on supernodal method and uses LU factorization for solving linear systems. SuperLU was developed at the UC Berkeley Computer Science Division and at NERSC. Sequential variant is declared as threadsafe since release 4.0. SuperLU_MT is not available from PETSc. Originally distributed SuperLU_DIST since the last version 4.0.0 released in November 2014 supports multithreading with OpenMP and GPU parallelism with CUDA. However, PETSc 3.5.3 currently supports only version SuperLU_DIST 3.3.

PaStiX

PaStiX [24] is parallelized library developed by BACCHUS team from INRIA. PaStiX uses supernodal method. It provides solving linear systems using LU and Cholesky factorization. Default version of PaStiX uses threads and MPI. The default thread library used by PaStiX is the POSIX one. PaStiX is available from PETSc and number of threads can be set by option:

```
-mat_pastix_threadnbr nthreads
```

PaStiX offers versions for both MPI implementation `MPI_THREAD_MULTIPLE` and `MPI_THREAD_FUNNELED`. So funneled version must be used in application where PETSc initializes MPI.

Intel MKL Pardiso

Intel MKL Pardiso [25] is shared-memory multiprocessing parallel direct sparse solver. It is supported in PETSc since version 3.5. MKL Pardiso supports symmetric, structurally symmetric and nonsymmetric matrices and provides LU, LDLT and Cholesky factorization. Number of threads in MKL library can be controlled by environment variable `MKL_NUM_THREADS`.

This solver was used out of the PETSc functions in our implementation for the reasons mentioned in the Section 5.2. Therefore, the more detailed description of MKL Pardiso usage is presented here.

First it is necessary to initialize Pardiso by function

```
void pardisoinit(_MKL_DSS_HANDLE_t pt, MKL_INT *mtype,
                 MKL_INT *iparm).
```

Values of input parameter `mtype` specify a matrix type. Based on this value `pardisoinit` sets default values for the output parameter `iparm`. The elements of array `iparm` keep various parameters to Intel MKL Pardiso and some useful information after execution of the solver. Parameter `pt` is pointer to array which holds internal data of solvers. Initializing function sets all values of `pt` to zeros. After the first call the pointer should never be modified, because it could cause a serious memory leak or a crash.

Computation operations of the package Intel MKL Pardiso are accessible by calling the function

```
void pardiso(_MKL_DSS_HANDLE_t pt, MKL_INT *maxfct,
             MKL_INT *mnum, MKL_INT *mtype, MKL_INT *phase,
             MKL_INT *n, void *a, MKL_INT *ia, MKL_INT *ja,
             MKL_INT *perm, MKL_INT *nrhs, MKL_INT *iparm,
             MKL_INT *msglvl, void *b, void *x,
             MKL_INT *error).
```

The function calculates the solution of $AX = B$ with single or multiple right-hand sides. Execution step of solver is controlled by parameter `phase`. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. MKL Pardiso has following stages

Phase 1: Fill-reduction analysis and symbolic factorization.

Phase 2: Numerical factorization.

Phase 3: Forward and Backward substitutions including optional iterative refinement. Forward, backward and diagonal substitutions can be separated.

For example, if `phase=12`, function `pardiso` executes symbolic and numerical factorizations. Matrix A is passed to the function by parameters `a`, `ia` and `ij` in CRS format. Number of right hand side vectors is set by parameter `rhs` and their values by parameter `b`. Output parameter `x` holds solution of linear system. For more detailed information see Intel website [25].

Since version 11.2 Intel MKL library contains also direct sparse solver for cluster. This solver supports hybrid parallelism.

CHOLMOD

CHOLMOD [26] is a set of ANSI C routines for solving systems of linear equations with sparse and symmetric positive definite matrix. It is based on supernodal factorization. CHOLMOD is part of the SuiteSparse linear algebra package authored by Prof. Tim Davis from Texas A&M University. CHOLMOD has supported GPU acceleration since 2012 with version 4.0.0.

5 Implementation of TFETI-1

5.1 Pure MPI implementation

Pure MPI implementation from package PermonFLLOP was the basis for hybrid parallizetion of TFETI-1 method described in Section 5.2. In pure MPI implementation, each process holds data of one subdomain, i.e. number of used cores equals to number of subdomains. Structure of primal data and their distribution is shown in Figure 6.

Matrices $\mathbf{K}$ and $\mathbf{R}$ have nice block-diagonal structure. For an efficient work with these matrices, special parallel composite matrix type `MATBLOCKDIAG` was implemented in PermonFLLOP. Diagonal blocks are represented by sequential PETSc matrix types (`MATSEQAIJ`, `MATSEQDENSE`).

In QP transformation Dualization, first the singular stiffness matrix $\mathbf{K}$ is regularized. So modified matrix is non-singular and its inverse is a pseudoinverse of the original matrix $\mathbf{K}$. Then an implicit inverse of the matrix $\mathbf{K}$ is created. For this purpose another PermonFLLOP matrix type `MATINV` is used, which wraps PETSc function `KSPSolve` in `MatMult`. So matrix vector multiplication $\mathbf{K}^{+}\mathbf{x} = \mathbf{y}$ is solved by direct sparse solver. Cholesky or LU factorization of the regularized $\mathbf{K}$ is made in setup phase. The regularized matrix can be released from memory. One backward and one forward substitution are performed in each FETI iteration. For factorization and solving, the sparse direct solver libraries MUMPS, SUPERLU, SUPERLU_DIST and other are used. The regularization, factorization and $\mathbf{K}^{+}$ action do not need any MPI communication.

The critical part of the TFETI-1 is the application of the projector $\mathbf{Q}$ or the matrix $(\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-1}$, respectively. This matrix is again represented by the matrix type `MATINV` in PermonFLLOP implementation. Currently there are implemented three approaches to solve the coarse problem in the PermonFLLOP:

1. applying the explicit inverse of $\mathbf{G}\mathbf{G}^{\mathrm{T}}$
2. using direct solver (Cholesky factorization)
3. orthonormalization of columns of $\mathbf{G}^{\mathrm{T}}$, so the coarse problem is eliminated

Experiments show that it is not suitable to solve the coarse problem fully parallel for cases 1 and 2 [27]. PermonFLLOP allows to use for it subset of processes to reduce the communication overhead. Dividing all the processes of the global `PETSC_COMM_WORLD` communicator into the subcommunicators is realized by using PETSc built-in "pseudo-preconditioner" `PCREDUNDANT`. Both these variants use again direct solver packages.

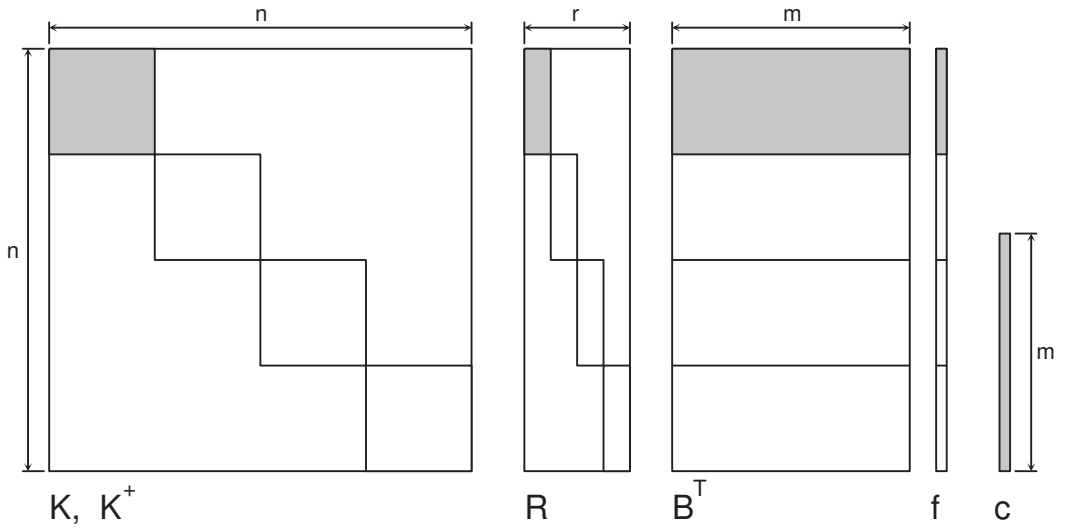

Figure 6: Primal data and their distribution (coloured parts are stored on single core). Letters n, m and r denote dimension of primal problem, dimension of dual problem and defect of matrix K.

The orthonormalization in case 3 is performed by iterative classical version of Gram-Schmidt process. If matrix $\mathbf{G}^{\mathrm{T}}$ has orthonormal columns then the coarse problem is eliminated because $(\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-1} = \mathbf{I}$. Comparison of these variants can be found in [7].

5.2 Hybrid implementation of TFETI-1

5.2.1 Sequential extension of PermonFLLOP

It was necessary to extend the PermonFLLOP implementation by adding new matrix type called `MATBLOCKDIAGSEQ` before the hybrid parallelization. This matrix type is sequential equivalent of distributed matrix type `MATBLOCKDIAG` which was mentioned in the previous section. Matrix of type `MATBLOCKDIAGSEQ` holds array of diagonal blocks, which are again represented by sequential matrices.

In practice, matrices $\mathbf{K}$, $\mathbf{K}^{+}$ and $\mathbf{R}$ are represented by distributed matrices `MATBLOCK-DIAGSEQ` whose diagonal blocks are sequential matrices `MATBLOCKDIAGSEQ`. This matrix contains an array of matrices which correspond to the subdomains and are represented by some PETSc matrix type.

It is possible to hold an arbitrary number of subdomains to one MPI process with this extension. So domain Ω can be divided to any number of subdomains regardless of

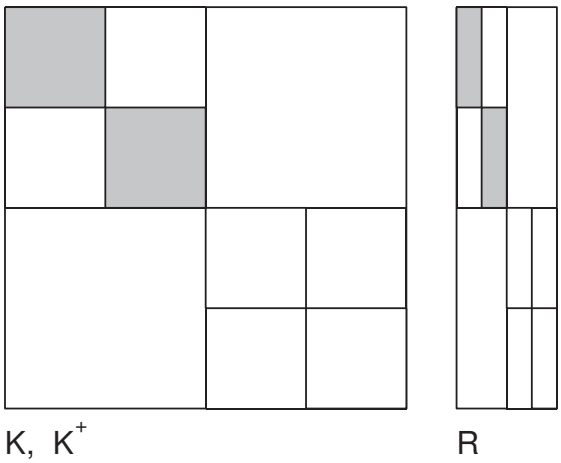

Figure 7: New distribution of matrices K and R.

number of available cores. As mentioned in Section 2.1.3, splitting of original subdomain into more smaller subdomains has positive effect on the conditioning number and the number of iterations decreases. Moreover dimensions of matrices $\mathbf{K}_i$ are smaller so their factorizations are faster. The negative effect is the increasing coarse problem size.

5.2.2 Parallelization

The hybrid parallelization of TFETI-1 can be divided into three areas

1. Parallelization of the operation with stiffness matrix $\mathbf{K}$
2. Parallelization of the coarse problem
3. Parallelization of other computations in TFETI-1 method

Whole parallelization of matrix $\mathbf{K}$ with threads is managed in functions of matrix type `MATBLOCKDIAGSEQ`. Dominant operation is factorization of $\mathbf{K}$. Factorization is performed during setup phase. In each FETI iteration, the linear system $\mathbf{Kx} = \mathbf{b}$ is solved by forward and backward substitutions. Parallelization of these operations can be done in two ways.

Outer parallelization - It is similar to pure MPI parallelization. Team of threads is created and each one provides factorization (or forward and backward substitution) of another diagonal block of $\mathbf{K}$. It is necessary to use threadsafe package for direct solver in this case. If we use PETSc, it means calling the function `PCSetUP` for all PC objects

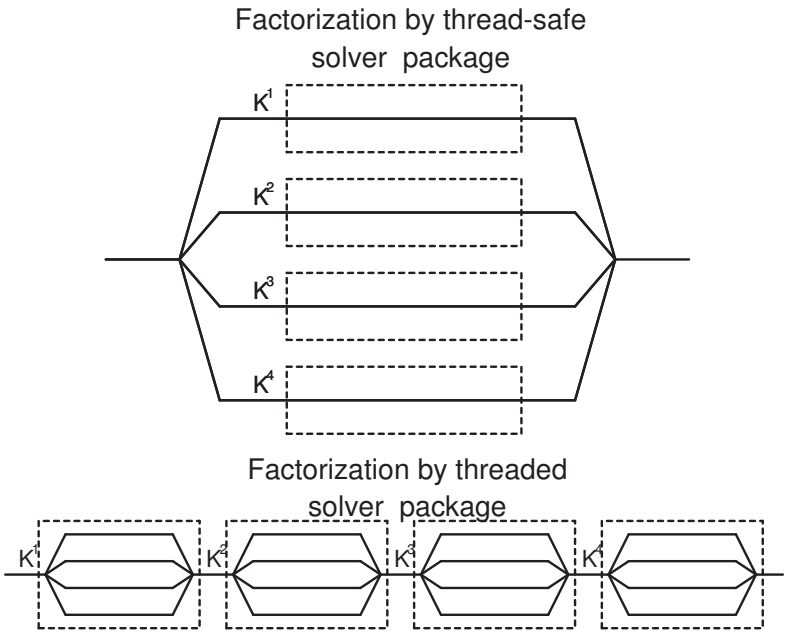

Figure 8: Two possibilities how to parallelize the factorization of K.

associated with diagonal blocks in parallel region. This is a problem, because PETSc is not threadsafe and calling PETSc function in threads can cause data races. For this reasons direct solver is called directly without using PETSc objects `PC` and `KSP` in our implementation. For this purpose Intel MKL Pardiso was used.

Inner parallelization - Factorization and solving linear system for each block is performed consecutively but multi-threaded. So there is a need to use a package supporting parallelization in shared memory.

If we have one subdomain per core (as in pure MPI implementation), it is not advantageous to use inner parallelization. Direct solvers can not achieve optimal speedup. So it is better to use the first approach, because operations with blocks are independent. The first approach would be more favorable if subdomains can be equally divided between computing cores. Load imbalance problem arises in cases where the number of blocks cannot be equally distributed among the cores or in case of significant differences between dimensions of blocks. Then using the inner parallelization approach can be preferable.

So far only cases have been considered where number of subdomains is greater than or equal to the number of available cores. There is an effort to divide original domain into large number of subdomains in FETI methods. Another strategy is to have one sub-

domain shared by more computing cores. The advantage is the reduction of the coarse problem dimension. Inner parallelization has to be used in this approach.

As mentioned in previous section, three approaches to solve the coarse problem are implemented in PermonFLLOP. Approach with the usage of direct solver was used and tested for hybrid implementation. The coarse problem can be again solved fully parallel or on smaller number of processes or on single process. So there can be used the direct solver using shared memory parallelism on single process or the direct solver supporting hybrid parallelism on more processes. For solving the coarse problem, the grater benefit of hybrid parallelization can be expected. Hybrid parallelization of TFETI-1 enables solving the coarse problem on single node in parallel and without communication. It also reduces MPI overhead if coarse problem is solved on more processes.

Parallelization of other TFETI-1 computations (for example assembling dual objects $\mathbf{G} = \mathbf{R}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}$, $\mathbf{d} = \mathbf{B}\mathbf{K}^{+}\mathbf{f} - \mathbf{c}$, $\mathbf{e} = \mathbf{R}^{\mathrm{T}}\mathbf{f}$ or matrix vector multiplication $\mathbf{B}^{\mathrm{T}}\boldsymbol{\lambda}$, $\mathbf{B}\mathbf{x}$ in each iteration of CG) is performed by the hybrid parallelization of PETSc, which was described in Section 4.2.1.

5.2.3 Extension of functionality of PermonAIF

It was necessary to extend the functionality of the module PermonAIF. New functions are similar to those which were presented in the Section 4.1.

```
FllopAIFSetFETIOperatorArray()
FllopAIFSetFETIOperatorNullspaceArray()
FllopAIFSetRhsArray()
FllopAIFSetEqArray_COO()
FllopAIFSetSolutionVectorArray();
```

Parameters of these functions are array of vectors or matrices. So they enable to pass to the PermonFLLOP data for an arbitrary number of subdomains.

6 Numerical experiments

All following experiments were run on the Anselm supercomputer [14] located at IT4Innovations, VŠB-TU Ostrava. Anselm is cluster of x86-64 Intel based nodes. The cluster contains four types of compute nodes - 180 regular nodes without accelerator, 23 nodes with GPU accelerator, 4 nodes with MIC accelerator and 2 fat compute nodes with large amount of memory. The regular node is equipped with two Intel Sandy Bridge, 8-core, 2.4 GHz processors, 64 GB of physical memory and 500 GB HDD. Computes nodes are interlinked by high speed fat-tree InfiniBand and Ethernet networks. Totally Anselm cluster contains 209 compute nodes with 3,344 cores, 15 TB RAM and its total theoretical peak performance is 94 Tflop/s.

All tests were performed on cantilever beam (see Figure 9) generated by tool PermonCube. Decomposition and discretization of beam are given by nine parameters. The first three parameters C_x, C_y, C_z determine number of clusters in each direction. Each cluster is mapped to one MPI process. Number of subdomains per cluster is determined by parameters N_x, N_y, N_z and number of elements per subdomains by parameters n_x, n_y, n_z.

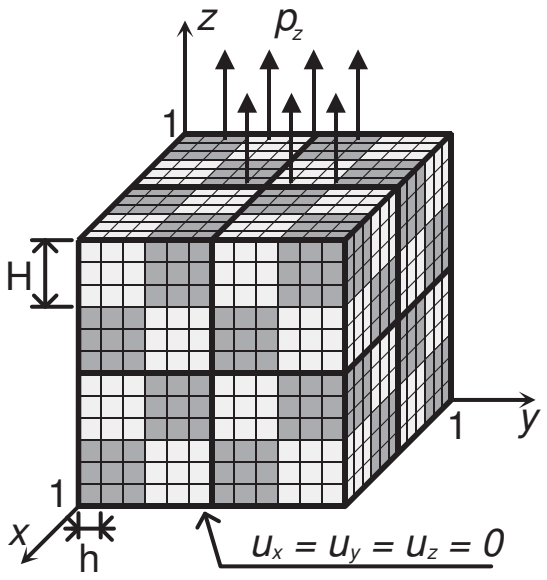

Figure 9: Model cube benchmark decomposed into 8 clusters (for 8 MPI processes). Each cluster is decomposed into 8 subdomains (distinguished by colour).

6.1 Numerical scalability

The first test shows the numerical scalability of the FETI method within PERMON. In this mode, subdomains are processed purely consecutively. Thus this test shows the effect of decomposition without parallelization. Number of elements is fixed, $h = {}^1/_{32}$. Number of DOFs of undecomposed problem is 107,811. To demonstrate numerical scalability, size of subdomains H decreases.

Number of subdomains	1	8	64	512
H/h	32	16	8	4
Primal DOFs	107,811	117,912	139,968	192,000
Preprocessing time	15.371	3.879	2.830	22.145
Factorization of $\mathbf{K}$	15.028	3.591	1.747	2.024
$\mathbf{G}\mathbf{G}^{\mathrm{T}}$ preprocessing	0.002	0.011	0.132	1.940
Solution time	3.815	4.479	3.533	2.103
$\mathbf{K}^{+}$ action	0.151	0.088	0.069	0.048
Total time	19.186	8.358	6.363	24.248
Number of PCG iterations	20	45	44	34

Table 1: Numerical scalability with fixed discretization size $h = {}^1/_{32}$ and varying decomposition

Decomposition into large number of subdomains favorably affects the time of factorization of $\mathbf{K}$ and $\mathbf{K}^{+}$ action. Therefore the total time decreases. The large increase in preprocessing time for 512 subdomains is caused by operation $\mathbf{G}^{\mathrm{T}} = \mathbf{B}\mathbf{R}$. Currently the function `MatMatMultByColumns`, which multiplies matrix $\mathbf{B}$ with columns of $\mathbf{R}$ is not optimized for sequential block-diagonal matrix. It is expectable that after the optimization the time of matrix by matrix multiplication will not play such significant role.

The number of iterations significantly increases when there are more than one subdomains and thus gluing conditions are added to the matrix B. However, we can see small decrease with increasing number of subdomains. This is in accordance with the theoretical estimate. The number of iterations is influenced by generating redundant constraints by PermonCube because they artificially inflate residuum in each iteration of PCGP. In table 2 number of iterations is shown for the same benchmark generated in the library MatSol with orthonormal rows of matrix $\mathbf{B}$.

Number of subdomains	1	8	64	512
Number of PCGP iterations	19	27	27	23

Table 2: Number of iterations with orthonormal $\mathbf{B}$.

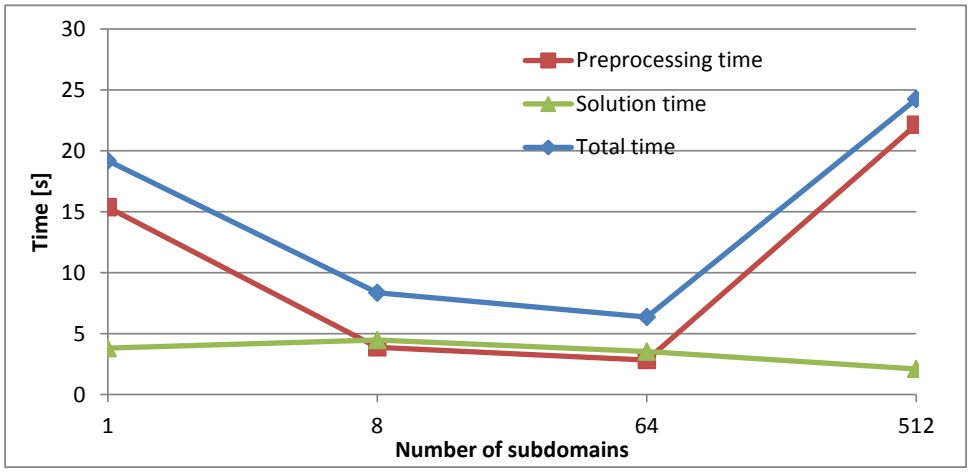

Figure 10: Numerical scalability, preprocessing, solution and total time on one core.

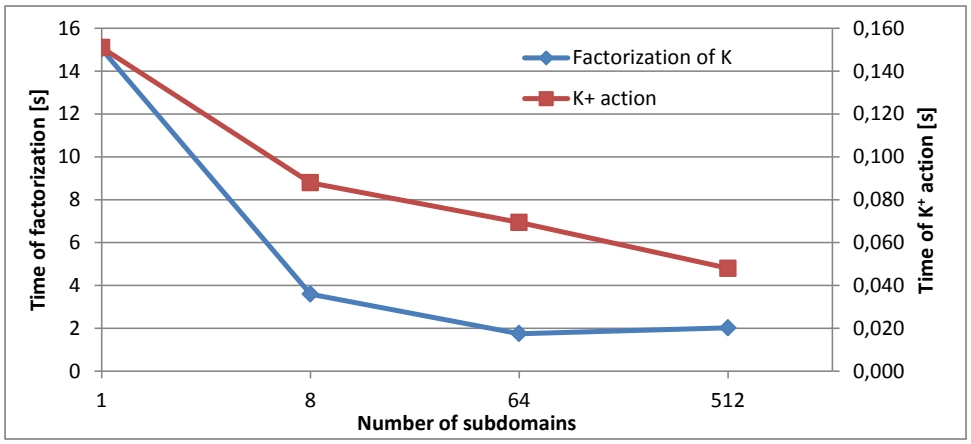

Figure 11: Numerical scalability, factorization of $\mathbf{K}$ and $\mathbf{K}^+$ action on one core.

Another test determines the optimal size of the subdomain with fixed number of computing cores. Table 3 shows the times for 64 cores. The factorization of $\mathbf{K}$ and $\mathbf{K}^+$ action are significantly faster for large number of subdomains. However, time of preprocessing of the coarse problem becomes dominant. We can see decrease of iterations for cases with the equal ratio of N_x, N_y and N_z, e.g. for regular decompositions (4^3, 8^3, 12^3). The best time was observed for 512 subdomains (8 per computational core) with 12 elements on the edge of each subdomain. Note that the tests above were not possible with PERMON until the presented new matrix type `MATBLOCKDIAGSEQ` was implemented.

N. of subdom.	64	128	256	512	768	1024	1728
N_x	4	8	8	8	12	16	12
N_y	4	4	8	8	8	8	12
N_z	4	4	4	8	8	8	12
Primal DOFs	3,000,000	3,120,000	3,244,800	3,374,592	3,504,384	3,634,176	3,779,136
Preproc. time	6.131	3.610	3.895	6.119	7.837	10.945	18.278
Fact. of $\mathbf{K}$	4.887	2.337	1.645	1.263	1.062	0.949	0.870
Preproc. CP	0.598	0.720	1.288	3.211	3.762	6.689	11.741
Solution time	9.562	11.156	7.700	4.303	4.5892	5.642	3.123
K+ action	0.107	0.080	0.064	0.052	0.045	0.041	0.035
Total time	15.693	14.766	11.595	10.422	12.426	16.587	21.401
Iterations	77	112	98	62	74	90	54

Table 3: Numerical scalability with fixed discretization size $h = {}^1/_{96}$ and varying decomposition on 64 cores.

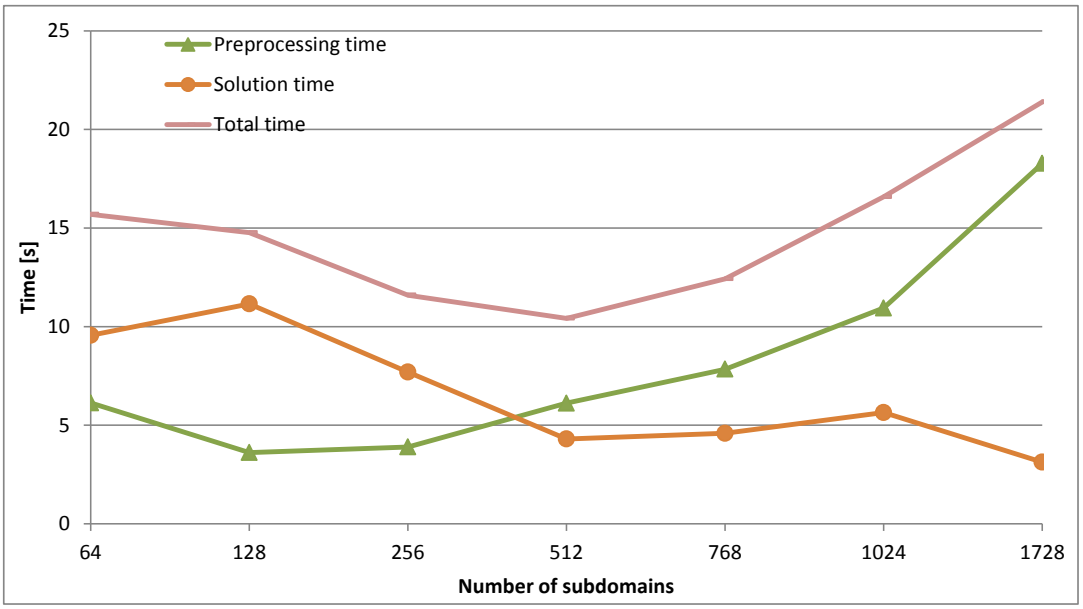

Figure 12: Numerical scalability, preprocessing, solution and total time on 64 cores.

6.2 Parallel solvers for the coarse problem

The coarse problem has relatively small dimension even for cases when original domain is decomposed to large number of subdomains. It allows to store matrix $\mathbf{G}\mathbf{G}^T$ in memory of each node. This test compares pure MPI with shared memory parallelization of the coarse problem solution inside one node. Various direct solvers, which are supported by PETSc, were used. Cube benchmark was decomposed into 8,000 subdomains, so dimension of the coarse problem is 48,000. Numbers of nonzero elements of matrix $\mathbf{G}\mathbf{G}^T$ is approximately 2.1 millions. Tables 4 and 5 show strong scalability of factorization and solve phase of direct solver.

The best threaded solver is Intel MKL Pardiso. It reached the best times for factorization and solution among threaded solvers. Best times among distributed memory direct solvers were reached by MUMPS. Factorization by MKL Pardiso with 16 threads was faster than factorization by MUMPS using 16 MPI processes. However, MUMPS achieved much better time in solve phase. Because the coarse problem action is executed in each TFETI-1 iteration, using MUMPS should be better option.

Number of threads	1	2	4	8	16
Factorization of $\mathbf{G}\mathbf{G}^{\mathrm{T}}$					
MKL Pardiso - LU	9.45E+00	5.42E+00	3.14E+00	2.15E+00	1.68E+00
MUMPS -LU	8.64E+00	5.67E+00	4.46E+00	4.48E+00	4.17E+00
PASTIX - LU	6.48E+00	4.27E+00	3.19E+00	2.69E+00	2.42E+00
MKL Pardiso - Cholesky	9.22E+00	5.23E+00	2.95E+00	2.02E+00	1.57E+00
MUMPS - Cholesky	5.89E+00	4.27E+00	3.96E+00	3.31E+00	3.46E+00
PASTIX - Cholesky	6.37E+00	4.20E+00	3.12E+00	2.62E+00	2.33E+00
$(\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-1}$ action					
MKL Pardiso - LU	6.64E-02	6.10E-02	4.10E-02	3.44E-02	3.44E-02
MUMPS -LU	6.29E-02	5.25E-02	4.24E-02	4.19E-02	5.37E-02
PASTIX - LU	2.04E-01	1.10E-01	6.74E-02	5.58E-02	4.83E-02
MKL Pardiso - Cholesky	6.51E-02	6.08E-02	4.12E-02	3.45E-02	3.40E-02
MUMPS - Cholesky	6.54E-02	5.91E-02	4.62E-02	4.43E-02	6.18E-02
PASTIX - Cholesky	2.00E-01	1.10E-01	6.74E-02	5.65E-02	4.41E-02

Table 4: Shared memory parallelism, strong scalability of factorization and solve phase of direct solvers for coarse problem with 8,000 subdomains.

number of process	1	2	4	8	16
Factorization of $\mathbf{G}\mathbf{G}^{\mathrm{T}}$					
MUMPS -LU	8.64E+00	6.43E+00	4.28E+00	3.02E+00	2.43E+00
PASTIX - LU	6.48E+00	4.27E+00	3.40E+00	2.92E+00	2.85E+00
SuperLU_DIST - LU	2.17E+01	1.05E+01	6.07E+00	3.99E+00	2.97E+00
MUMPS - Cholesky	5.89E+00	3.75E+00	2.72E+00	2.07E+00	1.87E+00
PASTIX - Cholesky	6.37E+00	4.27E+00	3.39E+00	2.96E+00	2.82E+00
$(\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-1}$ action					
MUMPS -LU	6.29E-02	4.77E-02	2.59E-02	2.29E-02	2.28E-02
PASTIX - LU	2.04E-01	1.10E-01	6.86E-02	6.03E-02	3.32E-02
SuperLU_DIST - LU	2.27E-01	8.90E-02	5.74E-02	3.62E-02	3.08E-02
MUMPS - Cholesky	6.54E-02	4.13E-02	3.17E-02	1.98E-02	1.70E-02
PASTIX - Cholesky	2.00E-01	1.08E-01	6.54E-02	5.63E-02	3.32E-02

Table 5: MPI parallelism, strong scalability of factorization and solve phase of direct solvers for coarse problem with 8,000 subdomains.

7 Conclusion

In the thesis, I dealt with the hybrid parallelization of TFETI-1 method implemented in PermonFLLOP library. I presented various parallel programming models and I focused on the hybrid MPI + OpenMP model in my thesis. The thesis also summarizes several libraries in terms of hybrid programming support. This is library PETSc on which is based PermonFLLOP and direct solvers which are supported by PETSc.

The main objective of the thesis was real implementation of TFETI-1 in the PermonFLLOP library. It included also the modification of interface for loading data from binary files and for connection with benchmark tool PermonCube. The greatest benefit of new implementation is the ability to perform computation for more subdomains on one core. It enables to divide original domain into larger number of smaller subdomains than was previously possible and to exploit good properties of numerical scalability of FETI methods. This benefit has been demonstrated on numerical experiments. This extension of implementation of PermonFLLOP can be also a basis for the H-TFETI-1 method [28] implementation. Then this modification allowed hybrid parallelization. Other code optimization of the hybrid parallelization will be necessary to prove its better scalability.

Further work can focus on parallelization of other techniques of solution of the coarse problem, for example on QR factorization in shared memory. Another current issue is the optimization of the code for Intel MIC Accelerators.

Radim Sojka

8 References

[1] Farhat, C., Roux, FX., *An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems.* SIAM Journal on Scientific Computing 13, 1992; 379-396.

[2] Dostál, Z., Horák, D. and Kučera, R., *Total FETI — an easier implementable variant of the FETI method for numerical solution of elliptic PDE*. Commun. Numer. Meth. Engng., 22: 1155–1162. 2006, doi: 10.1002/cnm.881

[3] Brzobohatý, T., Dostál, Z., Kozubek, T., Kovář, P. and Markopoulos, A., *Cholesky decomposition with fixing nodes to stable computation of a generalized inverse of the stiffness matrix of a floating structure*. Int. J. Numer. Meth. Engng., 88: 493–509. 2011, doi: 10.1002/nme.3187

[4] Farhat, C. and Roux, F.-X., *A method of finite element tearing and interconnecting and its parallel solution algorithm.* Int. J. Numer. Meth. Engng., 32: 1205–1227. 1991, doi: 10.1002/nme.1620320604

[5] Farhat, C., Mandel, J., Roux, F-X. *Optimal convergence properties of the FETI domain decomposition method*. Computer Methods in Applied Mechanics and Engineering. 1994;115:365–85. doi:10.1016/0045-7825(94)90068-X

[6] Mandel, J., Tezaur, R., *Convergence of a Substructuring Method with Lagrange Multipliers*, Numerische Mathematik, 1995, 73: 473-487.

[7] Kozubek, T., Vondrák, V., Menšík, M., Horák, D., Dostál, Z., Hapla, V., Kabelíková, P., Čermák, M., *Total FETI domain decomposition method and its massively parallel implementation.* Advances in Engineering Software, 2013, DOI: 10.1016/j.advengsoft.2013.04.001.

[8] OpenMP ARB: OpenMP Home Page. `http://www.openmp.org`

[9] OpenMP ARB: OpenMP 4.0 API.
`http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`

[10] Message Passing Interface Forum: Message Passing Interface Forum Home Page. `http://www.mpi-forum.org/`

[11] Message Passing Interface Forum: MPI 3.0 documentation. `http://www.mpi-forum.org/docs/mpi-3.0`

[12] Hoefler, T., Dinan, J., Buntinas, D., Balaji, P., Barrett, B., Brightwell, R., Gropp, W., Kale, V., Thakur, R., *MPI + MPI: a new hybrid approach to parallel programming with*

MPI plus shared memory. Computing, Volume 95, Issue 12 , pp 1121-1136. 2013, doi: 10.1007/s00607-013-0324-2

[13] Rabenseifner, R., Hager, G., Jost, G. *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes.* Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on: 427 - 436, 2009, doi: 10.1109/PDP.2009.43

[14] Documentation of supercomputer Anselm, `https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview`

[15] Stringfellow, N., *MPI/OpenMP Hybrid Parallelism for Multi-core Processors.* Slides available online at `http://www.speedup.ch/workshops/w39_2010/slides/SpeedupTutorial2010Stringfellow.pdf`

[16] Gropp, W., Thakur, R., *Thread-safety in an MPI implementation: Requirements and analysis* Parallel Computing Volume 33, Issue 9, p. 595–604, 2007, doi:10.1016/j.parco.2007.07.002

[17] PERMON web page, `http://industry.it4i.cz/produkty/permon/`

[18] PermonFLLOP web page, `http://industry.it4i.cz/produkty/permon/fllop/`

[19] PermonCube web page, `http://industry.it4i.cz/produkty/permon/cube/`

[20] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., Zhang, H., PETSc Web page. http://www.mcs.anl.gov/petsc

[21] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., Zhang, H., *PETSc Users Manual*, Argonne National Laboratory

[22] MUMPS web page, `http://mumps-solver.org/`

[23] SuperLU web page, `http://crd-legacy.lbl.gov/ xiaoye/SuperLU/`

[24] PasTiX web page, `http://pastix.gforge.inria.fr/`

[25] Intel MKL Pardiso web page, `https://software.intel.com/en-us/node/470282`

[26] CHOLMOD web page, `https://developer.nvidia.com/cholmod`

[27] Kozubek, T., Horák, D., Hapla, V., *FETI Coarse Problem Parallelization Strategies and Their Comparison.* Available: http://www.praceproject.eu/IMG/pdf/feticoarseproblemparallelization.pdf, 2012.

[28] Kozubek, T., Jarošová, M., Menšík, M., Markopoulos, A., *Hybrid Total FETI Method.* Available: http://www.prace-ri.eu/IMG/pdf/hybridtotalfetimethod.pdf, 2012.