VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

DIPLOMOVÁ PRÁCE

2010 Václav Hapla

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

Vytvoření rozhraní knihovny MatSol pro paralelní řešení kontaktních úloh

2010 Václav Hapla

Poděkování

Děkuji panu Doc. Ing. Tomášovi Kozubkovi, Ph.D. za podporu a vstřícnost při vedení mé diplomové práce. Díky patří rovněž Ing. Pavle Kabelíkové za podporu a spolupráci. Ing. Davidu Horákovi, Ph.D., děkuji za poskytnutí starších kódů využívajících PETSc.

Prohlášení o autorství

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 7.5.2010 Podpis:

Abstrakt

Práce se zabývá implementací paralelně a numericky škálovatelných algoritmů pro kontaktní úlohy pružnosti s využitím paralelizované knihovny PETSc a vytvořením rozhraní, které umožňuje komunikovat s knihovnou MatSol napsanou v MATLABU a přesunout část výpočtu z MATLABu na tuto implementaci. Motivací jsou především omezující licenční podmínky MATLABu, limitující počet výpočetních uzlů, a vyšší efektivita C kódu.

V prvé řadě jsou algoritmy představeny v matematické rovině včetně teoretických východisek, avšak s ohledem na praktickou implementaci. Dále je čtenář obeznámen obecně s problematikou paralelních počítačů a jejich programování. Speciálně jsou představeny některé již existující paralelizované numerické knihovny. Zvláštní důraz je pak kladen na diskuzi možností programování rozsáhlých paralelních aplikací s využitím metod objektově orientovaného programování ve světle budoucí implementace v C++ knihovně OOSol, která je vyvíjena na Katedře aplikované matematiky.

Závěrečná kapitola je věnována popisu zmíněné implementace. Konkrétně je do C převedena metoda TFETI z třídy metod dekompozice oblastí včetně vnitřně volaných QP algoritmů SMALBE a MPRGP.

Klíčová slova

kvadratické programování, QP, TFETI, dekompozice oblastí, SMALBE, MPRGP, paralelní výpočty, kontaktní úlohy pružnosti, PETSc, MatSol, MATLAB, objektově orientované programování, OOP, OOSol

Abstract

The thesis deals with implementation of scalable parallel numerical algorithms for contact problems of elasticity. It presents creation of interface which makes it possible to communicate with MatSol library written in MATLAB and to move the computation partly from MATLAB to this implementation. This effort is motivated by restrictive MATLAB licensing, limiting number of computing nodes, and higher efficiency of C code.

Firstly, the algorithms are presented from mathematical point of view including theoretical base, but with regard to practical implementation. Further, reader is made familiar with parallel computers and their programming. Particularly, some existing parallel numerical libraries are introduced. Emphasis is given on the discussion of programming of large parallel applications using methods of object-oriented programming having in mind future implementations within OOSol C++ library.

In the last chapter, attention is paid to description of implementation mentioned. Concretely, TFETI domain decomposition method is ported to C including internally called SMALBE and MPRGP QP algorithms.

Keywords

quadratic programming, QP, TFETI, domain decomposition, SMALBE, MPRGP, parallel computing, contact problems of elasticity, PETSc, MatSol, MATLAB, object-oriented programming, OOP, OOSol

Obsah

Přehled použitých zkratek

API application programming interface; aplikační rozhraní

BLAS Basic Linear Algebra Subprograms/Subroutines; základní podprogramy pro lineární algebru

DDM Domain Decomposition Methods; třída metod pro rozdělení úlohy definované na jisté oblasti na menší úlohy definované na podoblastech této původní oblasti

FETI Finite Element Tearing and Interconnecting; DDM metoda, jež realizuje spojitost řešení mezi podoblastmi tím, že na jejich hranicích zavádí lepicí podmínky, vynucované pomocí Lagrangeových multiplikátorů. Kromě původní stejnojmenné metody je tak nazývána celá skupina metod, které z ní vzešly.

TFETI Total FETI; jedno ze specifických rozšíření původní FETI metody, jež přistupuje k Dirichletovým podmínkám jako ke zvláštnímu případu podmínek lepicích

OOP object oriented programming; objektově orientované programování

QP quadratic programming; kvadratické programování

Úvod

Práce se zabývá implementací paralelně a numericky škálovatelných algoritmů pro kontaktní úlohy pružnosti. V první kapitole jsou popsána teoretická matematická východiska pro efektivní řešení složitých kontaktních úloh, avšak s jistým ohledem na praktickou implementaci. Další kapitola se věnuje paralelním počítačovým výpočtům a jejich vývoji. V kapitole třetí je pozornost zaměřena na paralelní knihovny numerické - věnujeme se zde rozsáhlým možnostem, které na tomto poli nabízí matematický balík MathWorks MATLAB a představíme MatSol, což je knihovna Katedry aplikované matematiky, vyvíjená v tomto prostředí-jazyce. Z C++ knihoven je prezentována PETSc, v níž byla realizována paralelní implementace algoritmů první kapitoly, a osobní zkušenosti s touto knihovnou a nakonec objektově orientovaná C++ knihovna OOSol, jež je rovněž vyvíjena na naší katedře. Zde dochází rovněž k hlubšímu zamyšlení nad problematikou vývoje matematického software mimo jiné z pohledu softwarového procesu.

Kapitola 1

Matematická východiska pro efektivní řešení kontaktních úloh

V této části se budeme zabývat matematickými algoritmy, které umožňují efektivně počítat mimo jiné složité kontaktní úlohy rovnovážného systému mnoha pružných těles, jako je např. analýza namáhání kuličkového ložiska, a využít přitom potenciálu moderních paralelních počítačů.

Toto je samozřejmě velmi nelehký a vyzývavý úkol. Složitost úloh tohoto typu spočívá v tom, že předem nevíme, kde se tělesa vzájemně dotýkají. Dirichletovy a Neumannovy podmínky jsou tedy fakticky známy až poté, co už bylo nalezeno řešení. K dalším obtížím, se kterými se v praxi běžně setkáváme, patří, že „plovoucí“ tělesa systému, jejichž Dirichletovy okrajové podmínky umožňují tuhé pohyby, způsobují singularitu příslušných matic tuhosti.

Celý myšlenkový postup si ukážeme na jednoduchém modelovém semikoercivním problému pružnosti bez tření, který patří do třídy úloh, jež vedou na parciální diferenciální rovnice a lze je formulovat jako tzv. variační nerovnosti.

Dále se budeme zabývat diskretizací a rozdělením diskretizační sítě úlohy na podoblasti (domain decomposition - DD), které umožňuje její efektivní paralelní řešení. Konkrétně projdeme teoretické základy jedné ze slibných metod DD, TFETI (Total Finite Element Tearing and Interconnecting), pomocí níž lze převést problémy tohoto typu na dobře podmíněné problémy kvadratického programování s jednoduchými omezeními na rovnost a nerovnost, navíc s podstatně menší dimenzí, než je velikost původní sítě.

Následně si představíme algoritmus SMALBE, který je v jistém smyslu optimální pro řešení celé třídy QP problémů s omezeními na rovnost a nerovnost vystupujících z TFETI. SMALBE ve vnitřní smyčce potřebuje vyřešit jistý QP problém s omezením pouze na nerovnost. Proto je následně ukázán algoritmus MPRGP, který je k tomu vhodný.

V závěru se krátce zamyslíme nad pojmy paralelní a numerické škálovatelnosti a jejich aplikovatelnosti na zmíněné algoritmy.

1.1 Spojitá formulace modelové kontaktní úlohy

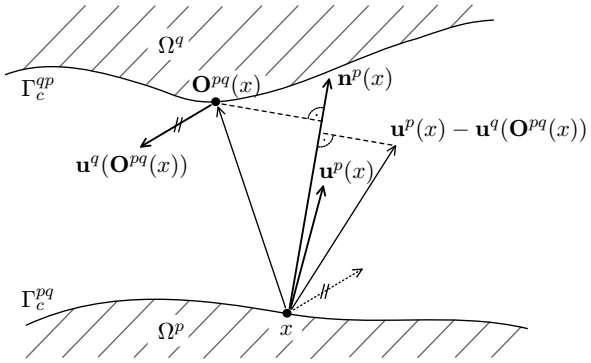
Uvažujme systém $\Omega \subset \mathbb{R}^d, d = 2, 3$, P homogenních izotropních pružných těles, z nichž každé zabírá v referenční konfiguraci právě jednu z oblastí Ω^p, $\overline{\Omega} = \bigcup_{p=1,2,\ldots,P} \overline{\Omega}^p$. Nechť má každá oblast Ω^p dostatečně hladkou hranici Γ^p, a ta ať se skládá z disjunktních částí Γ^p_U, Γ^p_F, Γ^p_C, tedy $\Gamma^p = \overline{\Gamma}^p_U \cup \overline{\Gamma}^p_F \cup \overline{\Gamma}^p_C$. Na všech Γ^p_U máme dána požadovaná posunutí $\mathbf{u}^p_D : \Gamma^p_U \to \mathbb{R}^d$ neboli Dirichletovy podmínky. Na každé Γ^p_F jsou zadány působící síly $\mathbf{F}^p : \Gamma^p_F \to \mathbb{R}^d$.

Γ^p_C je část hranice, která může přijít do kontaktu s nějakou jinou oblastí (tělesem). Dále označme konkrétněji $\Gamma^{pq}_C \subset \Gamma^p_C$ tu část, která může přijít do kontaktu s oblastí Ω^q.

Abychom zabránili nechtěnému pronikání těles na kontaktní části hranice, zavádíme linearizované podmínky nepronikání. Pro každé $p < q$ zavádíme vzájemně jednoznačné spojité zobrazení $\mathbf{O}^{pq} : \Gamma^{pq}_C \to \Gamma^{qp}_C$, které každému $\mathbf{x} \in \Gamma^{pq}_C$ přiřazuje jistý „blízký“ bod $\mathbf{O}^{pq}(\mathbf{x}) \in \Gamma^{qp}_C$. „Blízkost“ může být realizována různými způsoby, které si dovolíme zde nerozebírat. Linearizovanou podmínku nepronikání v každém bodě $\mathbf{x} \in \Gamma^{pq}_C$, $p, q = 1, 2, \ldots, P$, $p < q$ pak definujeme

$$(\mathbf{u}^p(\mathbf{x}) - \mathbf{u}^q(\mathbf{O}^{pq}(\mathbf{x})))\mathbf{n}^p(\mathbf{x}) \leq (\mathbf{O}^{pq}(\mathbf{x}) - \mathbf{x})\mathbf{n}^p(\mathbf{x}), \tag{1.1}$$

kde $\mathbf{n}^p(\mathbf{x})$ je vnější jednotkový normálový vektor oblasti Ω^p v bodě $\mathbf{x}$. Viz Obrázek 1.1.



Obrázek 1.1: Linearizované podmínky nepronikání ve 2D. S laskavým svolením Marie Sadowské [23]

Nechť $c^p_{ijk\ell} : \Omega^p \to \mathbb{R}^d$ označují prvky tenzoru elasticity a $\mathbf{g}^p : \Omega^p \to \mathbb{R}^d$ prvky vektoru objemových sil. Pro každé dostatečně hladké posunutí $\mathbf{u} : \overline{\Omega}^1 \times \ldots \times \overline{\Omega}^s \to \mathbb{R}^d$ je celková potenciální energie definována tzv. energetickým funkcionálem

$$J(\mathbf{u}) = \sum_{p=1}^{s} \left\{ \frac{1}{2} \int_{\Omega^p} a^p(\mathbf{u}^p, \mathbf{u}^p) d\Omega - \int_{\Omega^p} (\mathbf{g}^p)^T \mathbf{u}^p d\Omega - \int_{\Gamma^p_F} (\mathbf{F}^p)^T \mathbf{u}^p d\Gamma \right\}, \tag{1.2}$$

kde

$$a^p(\mathbf{u}^p, \mathbf{v}^p) = \frac{1}{2}\int_{\Omega^p} c^p_{ijk\ell} e^p_{ij}(\mathbf{u}^p) e^p_{k\ell}(\mathbf{v}^p) d\Omega, \tag{1.3}$$

$$e^p_{k\ell}(\mathbf{u}^p) = \frac{1}{2}\left(\frac{\partial u^p_k}{\partial x^p_\ell} + \frac{\partial u^p_\ell}{\partial x^p_k}\right) \tag{1.4}$$

je bilineární forma, resp. lineární funkcionál. Předpokládáme, že tenzor pružnosti vyhovuje přirozeným fyzikálním omezením, tedy

$$a^p(\mathbf{u}^p, \mathbf{v}^p) = a^p(\mathbf{v}^p, \mathbf{u}^p) \quad \text{a} \quad a^p(\mathbf{u}^p, \mathbf{u}^p) \geq 0. \tag{1.5}$$

Zavedeme kartézský součin Sobolevových prostorů

$$\mathcal{V} = H^1(\Omega^1)^d \times \ldots \times H^1(\Omega^s)^d, \tag{1.6}$$

a množinu kinematicky přípustných virtuálních posunutí $\mathcal{K} = \mathcal{K}^E \bigcap \mathcal{K}^I$, kde

$$\mathcal{K}^E = \{\mathbf{v} \in \mathcal{V} : \ \mathbf{v}^p = \mathbf{u}^p_D \ \text{ na } \ \Gamma^p_U, \ \ \mathbf{v}^p = \mathbf{v}_{|\overline{\Omega}^p}\}$$

je množina posunutí vyhovujících Dirichletovým podmínkám a

$$\mathcal{K}^I = \{\mathbf{v} \in \mathcal{V} : \ \forall \mathbf{x} \in \Gamma^{pq}_C, \ \forall p < q : \quad (\mathbf{v}^p(\mathbf{x}) - \mathbf{v}^q(\mathbf{O}^{pq}(\mathbf{x})))^T \mathbf{n}^p \leq (\mathbf{O}^{pq}(\mathbf{x}) - \mathbf{x})^T \mathbf{n}^p\}$$

množina posunutí vyhovujících podmínkám nepronikání. Posunutí $\mathbf{u} \in \mathcal{K}$ systému těles v rovnováze splňuje

$$J(\mathbf{u}) = \min_{\mathbf{v} \in \mathcal{K}} J(\mathbf{v}). \tag{1.7}$$

Podmínky, jež garantují existenci a jednoznačnost řešení $\mathbf{u}$, souvisí s koercivitou energetického funkcionálu J a pojednává o nich např. [12]. Lze uvažovat i obecnější okrajové jako např. předepsaná normálová posunutí.

1.2 TFETI: Dekompozice oblastí a přechod ke QP s jednoduchými omezeními

1.2.1 Stručný úvod do DDM

Podstatou matematických metod rozložení oblastí (Domain Decomposition Methods – DDM) v úlohách mechaniky je dekompozice rovnovážného systému pružných těles na jednotlivá tělesa (domény) a jejich další dělení na podoblasti (subdomény) tak, aby bylo možno hledat části řešení na každé podoblasti odděleně, nezávislé na ostatních, což poté umožňuje velmi snadnou paralelizaci. Tento postup je člověku vlastní a dobře pochopitelný, neboť při řešení jakéhokoliv

problému v oblasti lidské činnosti vždy usilujeme o jeho zjednodušení, dekompozici na dílčí problémy.

Matematický rámec a souvislosti napříč různými DDM jsou dobře podchyceny v [11]. Jednou z úspěšných DDM je FETI (Finite Element Tearing and Interconnecting). Byla poprvé představena Farhatem a Rouxem v roce 1992 [10]. Spočívá v „roztržení“ jednotlivých oblastí (pružných těles) Ω^p našeho systému Ω na vzájemně disjunktní podoblasti. Pro každou podoblast se pak definuje samostatný eliptický problém s Neumannovými podmínkami. Podoblasti jsou diskretizovány metodou konečných prvků.

Vzájemné nezávislosti podoblastí při zachování spojitosti řešení se dosahuje zavedením nových okrajových podmínek na hranicích mezi podoblastmi. Nazýváme je „lepicími“ podmínkami (gluing conditions), jež jsou vynucovány Lagrangeovými multiplikátory.

Tyto základní myšlenky byly dále rozvíjeny. Klíčovou se ukázala především aplikace teorie duality konvexního programování a KKT systémů. Výsledný QP problém má pak Hessián mnohem menší dimenze a s výrazně lepším číslem podmíněnosti, než je tomu u primárního QP problému. V neposlední řadě se také zjednodušuje tvar omezení na nerovnost tak, že dostáváme jednoduché omezení na proměnnou [7, 5]. Dále byly zavedeny ortogonální projektory a předpodmiňovače, zaručující numerickou škálovatelnost.

Postupně vznikla celá skupina metod, nazývaná jednoduše FETI metody. V tomto kontextu je někdy původní metoda nazývána FETI-1. Dalšího zlepšení se dospělo vynucením spojitosti v bodech křížení (tj. sdílených dvěma a více oblastmi). Toho se dosáhlo projekcí Lagrangeových multiplikátorů na jistý pomocný prostor (FETI-2) nebo neúplnou dekompozicí, kdy podoblasti nejsou odděleny zcela, ale sdílejí tyto body (FETI-DP) [8]. Podrobnější přehled těchto metod je např. v [13].

Metodu TFETI, předloženou Dostálem, Horákem a Kučerou [6], kterou se budeme dále zabývat, charakterizuje to, že vynucuje Lagrangeovými multiplikátory nejen splnění lepicích podmínek, ale také splnění Dirichletových okrajových podmínek, a tyto neovlivňují matici tuhosti. Dá se říci, že se na Dirichletově hranici jednoduše chová, jako by tam bylo další těleso, což odpovídá i fyzikální představě překážky, s níž je náš systém v kontaktu. Ukazuje se, že to znamená jednodušší implementaci, protože matice tuhosti všech podoblastí pak mají předem známá jádra a můžeme s nimi zacházet jednotně, což zjednodušuje implementaci. Numerické experimenty navíc ukazují, že metoda je také efektivnější a numericky stabilnější, tedy vhodnější pro řešení složitých koercivních a semikoercivních problémů [7]. Praktické dopady jsou podrobněji rozepsány též např. v [6, 5, 13].

Nakonec poznamenejme, že velikost a tvar podoblastí nejsou FETI metodami předepsány a jsou to více méně empirické parametry, které však ovlivňují efektivitu výpočtu. Navíc podotkněme, že není žádný zvláštní důvod, aby tyto parametry byly uniformní. V praxi se pro

samotné geometrické rozdělení používá specializovaný software jako je např. METIS/ParMETIS [15], který pro optimalizaci parametrů rozdělení využívá grafových algoritmů. Cílem je samozřejmě co nejrovnoměrnější rozložení výpočetní zátěže.

Jednou z usilovně zkoumaných alternativ je diskretizace metodou hraničních prvků a třída DD metod, která ji využívá, se nazývá BETI (Boundary Elements Tearing and Interconnecting). Obdobou TFETI pro hraniční prvky je TBETI, popsaná např. v [23], kde se lze dočíst, že výsledný QP problém je v podstatě stejný jako u TFETI a lze použít stejných efektivních řešičů.

1.2.2 TFETI – dekompozice, diskretizace a primální QP problém

Přistupme tedy ke konkrétnější formulaci postupu při dekompozici a diskretizaci původního spojitého problému metodou TFETI. V celém zbytku kapitoly o této metodě se odkazujeme zejména na [7]. Budeme se však snažit, aby text byl o něco bližší implementaci. V tomto světle některé spíše teoretické teze vynecháváme a některá místa zase rozvíjíme podrobněji.

Každé těleso Ω^p systému Ω, $\overline{\Omega} = \bigcup_{p=1,2,\ldots,P} \overline{\Omega}^p$, tedy pomyslně oddělíme od ostatních a rozdělíme na P_p podoblastí, $\overline{\Omega^p} = \bigcup_{r=1,2,\ldots,P_p} \overline{\Omega}^{pr}$. Všechny podoblasti $\overline{\Omega}^{pr}$, $p = 1, 2, \ldots, P$, $r = 1, 2, \ldots, P_p$ očíslujeme globálně jednoznačným číslováním, číslování původních oblastí můžeme zapomenout a vrátit se k původnímu značení.

Dále umělou hranici mezi podoblastmi Γ_p a Γ_q, která je součástí Γ_p a je „lepena“ k sousední hranici Γ_q, označme Γ_{pq}.

V dalším pak zavedeme linearizované podmínky nepronikání a jednotné lepicí podmínky pro hranice mezi podoblastmi, kde vynucují spojitost posunutí, i pro hranice s Dirichletovými podmínkami, kde vynucují předepsaná posunutí. Tato vynucení budeme realizovat pomocí Lagrangeových multiplikátorů.

Symbolem d rozumíme počet stupňů volnosti, v našem případě je $d = 3$, ale ponechme tento obecný symbol.

Konečně-prvkovou diskretizací oblasti $\overline{\Omega} = \bigcup_{p=1,2,\ldots,s} \overline{\Omega}_p$ při vhodném očíslování uzlů přechází spojitý problém na QP problém

$$\min. \quad \frac{1}{2}\mathbf{u}^\top \mathbf{K}\mathbf{u} - \mathbf{f}^\top \mathbf{u} \quad \text{za podm.} \quad \mathbf{B}_\mathcal{I}\mathbf{u} \le \mathbf{c}_\mathcal{I} \quad \text{a} \quad \mathbf{B}_\mathcal{E}\mathbf{u} = \mathbf{c}_\mathcal{E}, \tag{1.8}$$

kde $\mathbf{K} = \mathrm{diag}(\mathbf{K}_1, \ldots, \mathbf{K}_s)$ je symetrická pozitivně semidefinitní blokově diagonální matice řádu $n \times n$ a $\mathbf{f}$ je sloupcový vektor velikosti n, kde s je celkový počet podoblastí vzniklých dekompozicí, $n = N_{\mathrm{nodes}} \cdot d$, N_{nodes} je celkový počet uzlů. Číslo n je *primární dimenzí* naší úlohy. Zřejmě hledané řešení $\mathbf{u}$ má tutéž dimenzi.

Symboly $\mathcal{I}$, resp. $\mathcal{E}$, nám označují objekty figurující v nerovnostních, resp. rovnostních vazbách. Nerovnostní omezení $\mathbf{B}_\mathcal{I}\mathbf{u} \le \mathbf{c}_\mathcal{I}$ je maticový zápis linearizovaných podmínek nepro-

nikání, kde $\mathbf{B}_\mathcal{I}$ je matice rozměrů $m_\mathcal{I} \times n$ a plné hodnosti a $\mathbf{c}_\mathcal{I}$ je sloupcový vektor velikosti $m_\mathcal{I}$, kde $m_\mathcal{I}$ označuje globální počet zadaných podmínek nepronikání.

Rovnostní omezení $\mathbf{B}_\mathcal{E}\mathbf{u} = \mathbf{c}_\mathcal{E}$ je maticový zápis lepicích a Dirichletových podmínek. Zde $\mathbf{B}_\mathcal{E}$ je matice rozměrů $m_\mathcal{E} \times n$, plné hodnosti, a $\mathbf{c}_\mathcal{E}$ je sloupcový vektor velikosti $m_\mathcal{E}$, kde $m_\mathcal{E} = M_\mathcal{E} \cdot d$ s $M_\mathcal{E}$ značícím celkový počet zadaných lepicích a Dirichletových podmínek. Celkový počet podmínek na rovnost a nerovnost, $m = m_\mathcal{I} + m_\mathcal{E}$, je nazýván *duální dimenzí* úlohy. Typicky je $m \ll n$, což má svůj význam, jak se ukáže později.

Nyní rozebereme podrobněji původ těchto objektů. Budeme se zabývat pouze konformním dělením. Alternativním přístupem je využití nekonformních sítí, více v [13], str. 25.

Vektor řešení $\mathbf{u}$, hledaného posunutí, sestává z N_{nodes} d-tic pod sebou, i-tému uzlu odpovídá i-tá d-tice, jež odpovídá geom. souřadnicím posunutí i-tého uzlu.

Vektor zatížení $\mathbf{f}$ sestává z celkových sil působících na každý uzel, které jsou výslednicemi objemových sil a/nebo předepsaného napětí v případě uzlů na Γ_F. Každému i-tému uzlu v něm odpovídá i-tá d-tice hodnot pod sebou, odpovídající geom. souřadnicím působící síly.

Diagonální bloky $\mathbf{K}_p$ jsou matice tuhosti odpovídající jednotlivým podoblastem $\overline{\Omega}_p$. Jsou to symetrické pozitivně semidefinitní řídké matice se známými jádry díky tomu, že metoda TFETI nevpravuje zadané Dirichletovy podmínky do matice $\mathbf{K}$, ale vynucuje je externími rovnostními podmínkami, jak bude uvedeno dále. Bloky $\mathbf{K}_p$, tedy i celá $\mathbf{K}$, mohou být efektivně rozloženy pomocí singulární Choleského faktorizace, lépe však pomocí Choleského-SVD faktorizace [4], která se lépe vypořádá s maticemi, jež jsou špatně podmíněné a pouze pozitivně semidefinitní, nikoliv pozitivně definitní, což je často případ $\mathbf{K}_p$.

V následujícím využijeme také značení zavedené v sekci 1.1 a navíc označíme $\mathbf{x}^{(i)}$ i-tý uzel v globálním indexování.

Matice $\mathbf{B}_\mathcal{E}$ a vektor $\mathbf{c}_\mathcal{E}$ vynucují předepsaná posunutí na části hranice s Dirichletovými podmínkami a spojitost posunutí na umělých hranicích mezi podoblastmi podmínkou $\mathbf{B}_\mathcal{E}\mathbf{u} = \mathbf{c}_\mathcal{E}$. Abychom popsali přesněji jejich konstrukci, rozdělme $\mathbf{c}_\mathcal{E}$ vertikálně na bloky $\mathbf{r}_i(\mathbf{c}_\mathcal{E})$ výšky d, $\mathbf{B}_\mathcal{E}$ na řádkové bloky $\mathbf{R}_i(\mathbf{B}_\mathcal{E})$ výšky d a ty dále na čtvercové $d \times d$ bloky $\mathbf{S}_{ij}(\mathbf{B}_\mathcal{E})$, $i, j = 1, 2, \ldots, M_\mathcal{E}$.

Nejprve se věnujme realizaci Dirichletových podmínek. Je-li i-tá Dirichletova podmínka zadána v nějakém bodě $\mathbf{x}^{(j)} \in \Gamma_U^p$, bude $\mathbf{R}_i(\mathbf{B}_\mathcal{E})$ nulový až na $\mathbf{S}_{ij}(\mathbf{B}_\mathcal{E}) = \mathbf{I}$. A $\mathbf{r}_i(\mathbf{c}_\mathcal{E})$ bude odpovídat vektoru předepsaného posunutí, $\mathbf{r}_i(\mathbf{c}_\mathcal{E}) = \mathbf{u}_D^p(\mathbf{x}^{(j)})$. Tímto skutečně máme Dirichletovy podmínky vynuceny pro každé posunutí $\mathbf{u}$:

$$\begin{array}{c} \mathbf{B}_\mathcal{E}\mathbf{u} = \mathbf{c}_\mathcal{E} \\ \Updownarrow \\ \forall p \in \{1, 2, \ldots, P\}, \forall \mathbf{x}^{(j)} \in \Gamma_U^p : \quad \mathbf{u}^p(\mathbf{x}^{(j)}) = \mathbf{R}_i(\mathbf{B}_\mathcal{E})\mathbf{u} = \mathbf{r}_i(\mathbf{B}_\mathcal{E}\mathbf{u}) = \mathbf{r}_i(\mathbf{c}_\mathcal{E}) = \mathbf{u}_D^p(\mathbf{x}^{(j)}). \end{array}$$

Dále se věnujme realizaci lepicích podmínek při analogickém značení. Lepíme-li i-tou lepicí podmínkou dva body se stejnými souřadnicemi $\mathbf{x}^{(j)} \in \Gamma_G^p$ a $\mathbf{x}^{(k)} \in \Gamma_G^q$, bude $\mathbf{R}_i(\mathbf{B}_\mathcal{E})$ nulový až na $\mathbf{S}_{ij}(\mathbf{B}_\mathcal{E}) = \mathbf{I}$ a $\mathbf{S}_{ik}(\mathbf{B}_\mathcal{E}) = -\mathbf{I}$. Vektor $\mathbf{r}_i(\mathbf{c}_\mathcal{E})$ bude nulový. V případě bodů křížení, které jsou sdíleny více podoblastmi, se postupuje analogicky, ale je potřeba dát pozor, aby řádky $\mathbf{B}_\mathcal{E}$ zůstaly nezávislé.

Matice $\mathbf{B}_\mathcal{I}$ a vektor $\mathbf{c}_\mathcal{I}$ popisují linearizované podmínky nepronikání. Mějme libovolné $i \in \{1, 2, \ldots, m_\mathcal{I}\}$ a ať i-tá podmínka nepronikání je zadána pro nějaký uzel $\mathbf{x}^{(j)} \in \Gamma_C^p$ a obecný bod $\mathbf{O}^{pq}(\mathbf{x}^{(j)}) \in \Gamma_C^q$. Označme i-tý řádek matice $\mathbf{B}_\mathcal{I}$ symbolem $\mathbf{r}_i(\mathbf{B}_\mathcal{I})$. Pak $\mathbf{B}_\mathcal{I}$ a $\mathbf{c}_\mathcal{I}$ se sestaví tak, aby platilo

$$\begin{gathered}
\mathbf{r}_i(\mathbf{B}_\mathcal{I})\mathbf{u}^p(\mathbf{x}^{(j)}) \le (\mathbf{c}_\mathcal{I})_i \\
\Updownarrow \\
(\mathbf{u}^p(\mathbf{x}^{(j)})) - \mathbf{u}^p(\mathbf{O}^{pq}(\mathbf{x}^{(j)}))\mathbf{n}^p(\mathbf{x}^{(j)}) \le (\mathbf{u}^p(\mathbf{x}^{(j)}) - \mathbf{u}^p(\mathbf{O}^{pq}(\mathbf{x}^{(j)})))\mathbf{n}^p(\mathbf{x}^{(j)})
\end{gathered}$$

pro všechna možná posunutí $\mathbf{u}^p$. Konkrétních provedení se používá několik.

Všimněme si, že matice $\mathbf{B}_\mathcal{E}$ i $\mathbf{B}_\mathcal{I}$ jsou velmi řídké.

1.2.3 TFETI – přechod k duálnímu problému

Problém (1.8) je sice standardním problémem kvadratického programování, avšak ukazuje se, že není vhodný pro numerické řešení, protože matice $\mathbf{K}$ je typicky velmi špatně podmíněná a přípustná množina je definována dostatečně složitě na to, aby to znemožňovalo efektivně počítat projekce na ni, což by mařilo efektivitu výpočtu.

Tyto potíže lze významně redukovat použitím teorie duality konvexního programování [5]. Lagrangián problému (1.8) je

$$L(\mathbf{u}, \boldsymbol{\lambda}_\mathcal{I}, \boldsymbol{\lambda}_\mathcal{E}) = \frac{1}{2}\mathbf{u}^\top \mathbf{K}\mathbf{u} - \mathbf{f}^\top \mathbf{u} + \boldsymbol{\lambda}_\mathcal{I}^\top(\mathbf{B}_\mathcal{I}\mathbf{u} - \mathbf{c}_\mathcal{I}) + \boldsymbol{\lambda}_\mathcal{E}^\top(\mathbf{B}_\mathcal{E}\mathbf{u} - \mathbf{c}_\mathcal{E}), \tag{1.9}$$

kde $\boldsymbol{\lambda}_\mathcal{I}$, resp. $\boldsymbol{\lambda}_\mathcal{E}$ je Lagrangeův multiplikátor pro nerovnostní, resp. rovnostní vazby. Zápis můžeme zjednodušit zavedením

$$\boldsymbol{\lambda} = \begin{bmatrix} \boldsymbol{\lambda}_\mathcal{I} \\ \boldsymbol{\lambda}_\mathcal{E} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_\mathcal{I} \\ \mathbf{B}_\mathcal{E} \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} \mathbf{c}_\mathcal{I} \\ \mathbf{c}_\mathcal{E} \end{bmatrix},$$

takže Lagrangián pak můžeme psát ve tvaru

$$L(\mathbf{u}, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{u}^\top \mathbf{K}\mathbf{u} - \mathbf{f}^\top \mathbf{u} + \boldsymbol{\lambda}^\top(\mathbf{B}\mathbf{u} - \mathbf{c}). \tag{1.10}$$

Je známo, že primární QP problém (1.8) je ekvivalentní sedlobodovému problému

$$\text{najdi } (\widehat{\mathbf{u}}, \widehat{\boldsymbol{\lambda}}) \text{ tak, že} \quad L(\widehat{\mathbf{u}}, \widehat{\boldsymbol{\lambda}}) = \sup_{\boldsymbol{\lambda}_\mathcal{I} \ge \mathbf{o}} \inf_{\mathbf{u}} L(\mathbf{u}, \boldsymbol{\lambda}). \tag{1.11}$$

Zaveďme matici $\mathbf{R}$ plné hodnosti, jejíž obor hodnot je roven jádru $\operatorname{Ker}\mathbf{K}$. Jinak řečeno, sloupce $\mathbf{R}$ tvoří bázi $\operatorname{Ker}\mathbf{K}$.

Na základě tvrzení 2.22 v [5] dostáváme, že vektor Lagrangeových multiplikátorů $\widehat{\boldsymbol{\lambda}}$, je řešením duálního QP problému s omezeními na rovnost a nerovnost

$$\text{min.} \quad \Theta(\boldsymbol{\lambda}) \quad \text{za podm.} \quad \boldsymbol{\lambda}_{\mathcal{I}} \geq \mathbf{o} \quad \text{a} \quad \mathbf{R}^\top(\mathbf{f} - \mathbf{B}^\top\boldsymbol{\lambda}) = \mathbf{o}, \tag{1.12}$$

kde

$$\Theta(\boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{\lambda}^\top\mathbf{B}\mathbf{K}^\dagger\mathbf{B}^\top\boldsymbol{\lambda} - (\mathbf{B}\mathbf{K}^\dagger\mathbf{f} - \mathbf{c})^\top\boldsymbol{\lambda}, \tag{1.13}$$

$\mathbf{K}^\dagger$ je libovolná levá zobecněná inverze matice $\mathbf{K}$, tedy matice splňující

$$\mathbf{K}\mathbf{K}^\dagger\mathbf{K} = \mathbf{K}.$$

Zdůrazněme, že Hessián $\mathbf{B}\mathbf{K}^\dagger\mathbf{B}^\top$ a hledané řešení $\boldsymbol{\lambda}$ jsou dimenze m (duální dimenze), jež je rovna počtu podmínek na rovnost a nerovnost, která bývá mnohem menší než primární dimenze n. Navíc již tento Hessián je zpravidla lépe podmíněný než Hessián primárního problému (1.8).

Pokud je řešení $\widehat{\boldsymbol{\lambda}}$ problému (1.12) známo, získáme i řešení primární úlohy pomocí vztahů

$$\mathbf{u} = \mathbf{K}^\dagger(\mathbf{f} - \mathbf{B}^\top\boldsymbol{\lambda}) + \mathbf{R}\boldsymbol{\alpha}, \tag{1.14}$$

$$\boldsymbol{\alpha} = (\mathbf{R}^\top\widetilde{\mathbf{B}}^\top\widetilde{\mathbf{B}}\mathbf{R})^{-1}\mathbf{R}^\top\widetilde{\mathbf{B}}^\top(\widetilde{\mathbf{c}} - \widetilde{\mathbf{B}}^\top\mathbf{K}^\dagger(\mathbf{f} - \mathbf{B}^\top\widetilde{\boldsymbol{\lambda}})), \tag{1.15}$$

$$\widetilde{\mathbf{B}} = \begin{bmatrix}\widetilde{\mathbf{B}}_{\mathcal{I}} \\ \mathbf{B}_{\mathcal{E}}\end{bmatrix}, \quad \widetilde{\mathbf{c}} = \begin{bmatrix}\widetilde{\mathbf{c}}_{\mathcal{I}} \\ \mathbf{c}_{\mathcal{E}}\end{bmatrix}, \tag{1.16}$$

kde matice $\widetilde{\mathbf{B}}_{\mathcal{I}}$ je tvořena řádky $\mathbf{B}_{\mathcal{I}}$, které odpovídají kladným prvkům $\widehat{\boldsymbol{\lambda}}_{\mathcal{I}}$, a podobně $\widetilde{\mathbf{c}}_{\mathcal{I}}$.

Vraťme se ještě ke $\mathbf{K}^\dagger$. O aplikovatelnosti různých druhů pseudoinverzí pojednává [4], kde je prokázána vhodnost SVD-Choleského rozkladu. Pro ilustraci však nyní zvolme relativně jednoduchou pseudoinverzi pomocí singulárního Choleského rozkladu $\mathbf{K} = \mathbf{L}\mathbf{L}^\top$. Ten existuje, protože $\mathbf{K}$ je symetrická a pozitivně semidefinitní. Důležitým pozorováním je, že zde není potřeba provádět explicitní vyčíslení $\mathbf{K}^\dagger$ ani její explicitní násobení s jinou maticí, které by způsobilo zaplnění. Skutečně, v našem případě vždy stačí provést akci násobení vektorem zprava, a tu lze provést tímto způsobem:

$$\mathbf{x} = \mathbf{K}^\dagger\mathbf{b} = (\mathbf{L}\mathbf{L}^\top)^\dagger\mathbf{b} = (\mathbf{L}^\top)^\dagger\mathbf{L}^\dagger\mathbf{b} = \mathbf{L}^\top\backslash(\mathbf{L}\backslash\mathbf{b}).$$

1.2.4 TFETI – předpodmínění

I když je problém (1.12) mnohem vhodnější k praktickému numerickému výpočtu, než je problém (1.8), dalšího zlepšení dosáhli Farhat, Mandel a Roux [9] jistým předpodmíněním, které si dále představíme.

Označme

$$\mathbf{F} = \mathbf{B}\mathbf{K}^{\dagger}\mathbf{B}^{\top}, \tag{1.17}$$

$$\widetilde{\mathbf{d}} = \mathbf{B}\mathbf{K}^{\dagger}\mathbf{f} - \mathbf{c}, \tag{1.18}$$

$$\widetilde{\mathbf{G}} = \mathbf{R}^{\top}\mathbf{B}^{\top}, \tag{1.19}$$

$$\widetilde{\mathbf{e}} = \mathbf{R}^{\top}\mathbf{f}. \tag{1.20}$$

Dále mějme regulární matici $\mathbf{T}$, takovou, že matice

$$\mathbf{G} = \mathbf{T}\widetilde{\mathbf{G}} \tag{1.21}$$

má ortogonální řádky. Ještě označme

$$\mathbf{e} = \mathbf{T}\widetilde{\mathbf{e}}. \tag{1.22}$$

Otázkou nyní je, jak získat $\mathbf{T}$. Přirozeným nápadem je aplikace Gramm-Schmidtova ortogonalizačního procesu přímo na řádky $\widetilde{\mathbf{G}}$. Ovšem vhodnější volbou se ukazuje použití Choleského rozkladu matice $\widetilde{\mathbf{G}}\widetilde{\mathbf{G}}^{\top} = \widetilde{\mathbf{L}}\widetilde{\mathbf{L}}^{\top}$. Matice $\mathbf{T} = \widetilde{\mathbf{L}}^{-1}$ ortogonalizuje $\widetilde{\mathbf{G}}$:

$$\widetilde{\mathbf{G}}\widetilde{\mathbf{G}}^{\top} = \widetilde{\mathbf{L}}\widetilde{\mathbf{L}}^{\top} \quad \Leftrightarrow \quad (\widetilde{\mathbf{L}}^{-1}\widetilde{\mathbf{G}})(\widetilde{\mathbf{G}}^{\top}\widetilde{\mathbf{L}}^{-\top}) = (\widetilde{\mathbf{L}}^{-1}\widetilde{\mathbf{G}})(\widetilde{\mathbf{L}}^{-1}\widetilde{\mathbf{G}})^{\top} = \mathbf{G}\mathbf{G}^{\top} = \mathbf{I}.$$

Akci násobení matice $\mathbf{G}$ vektorem $\mathbf{x}$ pak ovšem realizujeme takto:

$$\mathbf{G}\mathbf{x} = \widetilde{\mathbf{L}}\backslash(\widetilde{\mathbf{G}}\mathbf{x}). \tag{1.23}$$

Při uvedeném značení problém (1.12) pak přechází v

$$\text{min.} \quad \frac{1}{2}\boldsymbol{\lambda}^{\top}\mathbf{F}\boldsymbol{\lambda} - \boldsymbol{\lambda}^{\top}\widetilde{\mathbf{d}} \quad \text{za podm.} \quad \boldsymbol{\lambda}_{\mathcal{I}} \geq \mathbf{o} \quad \text{a} \quad \mathbf{G}\boldsymbol{\lambda} = \mathbf{e}. \tag{1.24}$$

Dále lze, jak je psáno v [5], redukovat rovnostní podmínku

$$\mathbf{G}\boldsymbol{\lambda} = \mathbf{e}$$

s obecnou pravou stranou na rovnostní podmínku s nulovou pravou stranou, jak dále ukážeme. Množina vektorů, které ji splňují,

$$\Omega_E = \{\boldsymbol{\lambda} \in \mathbb{R}^m : \mathbf{G}\boldsymbol{\lambda} = \mathbf{e}\},$$

je afinním podprostorem ve tvaru

$$\Omega_E = \widetilde{\boldsymbol{\lambda}} + \operatorname{Ker}\mathbf{G},$$

kde $\widetilde{\boldsymbol{\lambda}}$ je libovolný vektor splňující

$$\mathbf{G}\widetilde{\boldsymbol{\lambda}} = \mathbf{e},$$

např. jej můžeme volit jednoduše $\widetilde{\boldsymbol{\lambda}} = \mathbf{G}^\top \mathbf{e}$ z ortogonality $\mathbf{G}$. Zavedeme-li substituci

$$\boldsymbol{\lambda} = \boldsymbol{\omega} + \widetilde{\boldsymbol{\lambda}}, \tag{1.25}$$

kde $\boldsymbol{\omega} \in \operatorname{Ker} \mathbf{G}$, můžeme, po odstranění konstantního členu, redukovat (1.24) na minimalizaci na $\operatorname{Ker} \mathbf{G}$

$$\text{min.} \quad \frac{1}{2}\boldsymbol{\omega}^\top \mathbf{F}\boldsymbol{\omega} - \boldsymbol{\omega}^\top \mathbf{d} \quad \text{za podm.} \quad \boldsymbol{\omega}_\mathcal{I} \geq \ell_\mathcal{I} \quad \text{a} \quad \mathbf{G}\boldsymbol{\omega} = \mathbf{o}. \tag{1.26}$$

Zde jsme označili

$$\mathbf{d} = \widetilde{\mathbf{d}} - \mathbf{F}\widetilde{\boldsymbol{\lambda}}, \quad \ell_\mathcal{I} = -\widetilde{\boldsymbol{\lambda}}_\mathcal{I}.$$

Finálním zlepšením je zavedení ortogonálních projektorů

$$\mathbf{Q} = \mathbf{G}^\top \mathbf{G} \quad \text{a} \quad \mathbf{P} = \mathbf{I} - \mathbf{Q}$$

na $\operatorname{Im} \mathbf{G}^\top$, resp. na $\operatorname{Ker} \mathbf{G}$ pro vylepšení numerické stability. Jejich aplikací na problém (1.26) získáváme ekvivalentní podobu duálního problému (1.26)

$$\text{min.} \quad \frac{1}{2}\boldsymbol{\omega}^\top (\mathbf{PFP} + \rho\mathbf{Q})\boldsymbol{\omega} - \boldsymbol{\omega}^\top \mathbf{Pd} \quad \text{za podm.} \quad \boldsymbol{\omega}_\mathcal{I} \geq \ell_\mathcal{I} \quad \text{a} \quad \mathbf{G}\boldsymbol{\omega} = \mathbf{o}. \tag{1.27}$$

Z řešení $\boldsymbol{\omega}$ tohoto problému získáme snadno duální řešení $\boldsymbol{\lambda}$ problému (1.24) dosazením do (1.25) a primární řešení problému (1.8) dosazením $\boldsymbol{\lambda}$ do (1.14)-(1.16). Regularizační člen $\rho\mathbf{Q}$, $\rho > 0$, umožňuje použití QP řešičů, které vyžadují regularitu Hessiánu. Dá se ukázat, že Hessián $\mathbf{PFP}$ má shora i zdola omezené spektrum [9]. Optimálním řešičem pro tuto třídu problémů je QP algoritmus SMALBE, který bude představen v následující sekci.

1.3 SMALBE: Optimální QP řešič pro TFETI

Nyní si představíme optimální algoritmus pro řešení problému s rovnostním a nerovnostním omezením (1.27). Jeho vnější smyčka generuje aproximace Lagrangeových multiplikátorů pro rovnostní omezení, zatímco vnitřní smyčka řeší pomocný problém s nerovnostním omezením. Přeznačíme-li $\boldsymbol{\omega}$ na $\boldsymbol{\lambda}$ a zavedeme-li nový vektor Lagrangeových multiplikátorů $\boldsymbol{\mu}$ pro rovnostní vazby, můžeme psát Lagrangián pro problém (1.27)

$$L(\boldsymbol{\lambda}, \boldsymbol{\mu}, \rho) = \frac{1}{2}\boldsymbol{\lambda}^\top (\mathbf{PFP} + \rho\mathbf{Q})\boldsymbol{\lambda} - \boldsymbol{\lambda}^\top \mathbf{Pd} + \boldsymbol{\mu}^\top \mathbf{G}\boldsymbol{\lambda} \tag{1.28}$$

a jeho gradient

$$\mathbf{g}(\boldsymbol{\lambda}, \boldsymbol{\mu}, \rho) = \nabla L_{\boldsymbol{\lambda}} = (\mathbf{PFP} + \rho\mathbf{Q})\boldsymbol{\lambda} - \mathbf{Pd} + \mathbf{G}^\top \boldsymbol{\mu}. \tag{1.29}$$

Projektovaný gradient $\mathbf{g}^P = \mathbf{g}^P(\boldsymbol{\lambda}, \boldsymbol{\mu}, \rho)$ Lagrangiánu L v $\boldsymbol{\lambda}$ je definován po prvcích

$$g_i^P = \begin{cases} g_i & \text{pro} \quad \lambda_i > \ell_i \quad \text{nebo} \quad i \in \mathcal{E}, \\ g_i^- & \text{pro} \quad \lambda_i = \ell_i \quad \text{a} \qquad i \in \mathcal{I}. \end{cases} \tag{1.30}$$

Zde g_i jsou prvky $\mathbf{g}$ a $g_i^- = \min\{g_i, 0\}$.

Přistupme k samotnému algoritmu SMALBE. Prezentovaná varianta bývá označována jako SMALBE-M, neboť se od původního SMALBE liší v tom detailu, že zachovává po celou dobu výpočtu penaltu ρ konstantní a místo ní je aktualizován parametr M, který řídí přesnost ve vnitřní smyčce. Tato varianta je i autorem [5] více doporučována a poslední články (např. [7], [23]) se zabývají právě jí. U parametrů uvádíme jejich rozumné počáteční hodnoty, které by měly zaručit dobrou konvergenci a mohou být implementovány jako výchozí. Dovolíme si převzít názvy proměnných z TFETI, i když SMALBE je univerzální řešič pro danou třídu QP problémů.

Algoritmus 1 (SMALBE-M)**.**
Require: $\varepsilon_{\text{out}} > 0$ $[\varepsilon_{\text{out}} = 10^{-6}]$ {*numerická přesnost pro vnější smyčku*}
$\eta > 0$ $[\eta = 0,1||\mathbf{Pd}||]$,
$\beta > 1$ $[\beta = 10]$,
$M_0 > 0$ $[M_0 = 1]$,
$\rho > 0$ $[\rho \approx ||\mathbf{PFP}||]$,
$\boldsymbol{\lambda}^0$ $[\lambda_i^0 = \max\{\ell_i, 0\}$ pro $i \in \mathcal{I}$,jinak $\lambda_i^0 = 0]$,
$\boldsymbol{\mu}^0$ $[\boldsymbol{\mu}^0 = 0]$,
$k = 0$.
Ensure: $\boldsymbol{\lambda} = \arg\min \frac{1}{2}\boldsymbol{\lambda}^\top(\mathbf{PFP} + \rho\mathbf{Q})\boldsymbol{\lambda} - \boldsymbol{\lambda}^\top\mathbf{Pd}$ s.t. $\boldsymbol{\lambda}_{\mathcal{I}} \geq \ell_{\mathcal{I}}$ and $\mathbf{G}\boldsymbol{\lambda} = \mathbf{o}$

1: **for** $k := 0, 1, \ldots$ **do**
2: {*vnitřní iterace s adaptivní kontrolou přesnosti:*}
3: $\boldsymbol{\lambda}^k := \arg\min\ L(\boldsymbol{\lambda}, \boldsymbol{\mu}, \rho)$ s.t. $\boldsymbol{\lambda}_{\mathcal{I}} \geq \ell_{\mathcal{I}}$ s parametry $\varepsilon_{\text{in}} := \min\{M_k||\mathbf{G}\boldsymbol{\lambda}^k||,\ \eta\}$ a $\boldsymbol{\lambda}_{\text{in}}^0 := \boldsymbol{\lambda}^k$
4: **if** $||\mathbf{g}^P(\boldsymbol{\lambda}^k, \boldsymbol{\mu}^k, \rho)|| < \varepsilon_{\text{out}}$ **and** $||\mathbf{G}\boldsymbol{\lambda}^k|| < \varepsilon_{\text{out}}$ **then**
5: $\boldsymbol{\lambda} := \boldsymbol{\lambda}^k$
6: **return** $\boldsymbol{\lambda}$ {*řešení nalezeno, konec*}
7: **end if**
8: $\boldsymbol{\mu}^{k+1} := \boldsymbol{\mu}^k + \rho\mathbf{G}\boldsymbol{\lambda}^k$ {*aktualizace Lagrangeových multiplikátorů*}
9: **if** $k > 0$ **and** $L(\boldsymbol{\lambda}^k, \boldsymbol{\mu}^k, \rho) < L(\boldsymbol{\lambda}^{k-1}, \boldsymbol{\mu}^{k-1}, \rho) + \rho||\mathbf{G}\boldsymbol{\lambda}^k||^2/2$ **then**
10: $M_{k+1} := M_k/\beta$ {*aktualizace vyvažovacího parametru M*}
11: **else**
12: $M_{k+1} := M_k$
13: **end if**
14: **end for**

Krok 3 může být implementován libovolným algoritmem, který je schopen řešit problém

$$\text{min.} \quad L(\boldsymbol{\lambda}) = L(\boldsymbol{\lambda}, \boldsymbol{\mu}, \rho) \quad \text{za podm.} \quad \boldsymbol{\lambda}_{\mathcal{I}} \geq \ell_{\mathcal{I}}, \tag{1.31}$$

s numerickou přesností ε_{in} a počáteční aproximací $\boldsymbol{\lambda}_{\text{in}}^0$ a zaručuje konvergenci projektovaného gradientu $\mathbf{g}^P$ k nule. Unikátní vlastností SMALBE je, že počet iterací nutných k nalezení přibližného řešení problému (1.27) je v mezích daných spektrem Hessiánu

$$\mathbf{A} = \mathbf{PFP} + \rho\mathbf{Q}.$$

Podrobně o SMALBE a jeho optimálních vlastnostech pojednává [5].

Abychom dostali omezení celkového počtu násobení Hessiánu vektorem, je potřeba realizovat *krok 3* vhodným algoritmem, který řeší problém (1.31) rovněž s počtem kroků omezeným ve smyslu extrémů spektra Hessiánu $\mathbf{A}$. V [5] je ukázáno, že jedním takovým algoritmem je MPRGP a že SMALBE s *krokem 3* implementovaným algoritmem MPRGP k nalezení přibližného řešení potřebuje pouze $O(1)$ násobení Hessiánu vektorem. Algoritmus MPRGP si představíme v následující sekci.

1.4 MPRGP: Efektivní vnitřní iterace SMALBE

Zde bude prezentován algoritmus MPRGP pro QP problémy s jednoduchým omezením na nerovnost,

$$\text{min.} \quad \mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{x}^\top \mathbf{b} \quad \text{za podm.} \quad \mathbf{x} \in \Omega_B = \{\mathbf{x} \in \mathbb{R}^m : x_i \geq \ell_i \text{ pro } i \in \mathcal{I}\}, \tag{1.32}$$

jako je vnitřní smyčka algoritmu SMALBE (1.31). MPRGP rozšiřuje tradiční algoritmus sdružených gradientů (conjugated gradients, CG) tím, že kromě kroku sdružených gradientů (CG step) používá navíc tzv. expanzního kroku (expansion step) a úměrnostního kroku (proportioning step). Dovolíme si dále používat anglické názvy těchto kroků, protože uvedené české překlady se v praxi nepoužívají.

Gradient problému (1.32) je

$$\mathbf{g}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}.$$

Označme $\mathcal{A}(\mathbf{x})$ *aktivní množinu* indexů $\mathbf{x}$,

$$\mathcal{A}(\mathbf{x}) = \{i \in \mathcal{I} : x_i = \ell_i\}.$$

Projektovaný gradient $\mathbf{g}^P(\mathbf{x})$ je pak definován po prvcích

$$g_i^P(\mathbf{x}) = \begin{cases} g_i(\mathbf{x}) & \text{pro} \quad i \notin \mathcal{A}(\mathbf{x}), \\ g_i^-(\mathbf{x}) & \text{pro} \quad i \in \mathcal{A}(\mathbf{x}). \end{cases}$$

Projektovaný gradient můžeme rozložit na *volný (free) gradient* $\boldsymbol{\varphi}(\mathbf{x})$ a *useknutý (chopped) gradient* $\boldsymbol{\beta}(\mathbf{x})$, $g_i^P(\mathbf{x}) = \boldsymbol{\varphi}(\mathbf{x}) + \boldsymbol{\beta}(\mathbf{x})$,

$$\varphi_i(\mathbf{x}) = \begin{cases} g_i(\mathbf{x}) & \text{pro} \quad i \notin \mathcal{A}(\mathbf{x}), \\ 0 & \text{pro} \quad i \in \mathcal{A}(\mathbf{x}), \end{cases} \quad \text{a} \quad \beta_i(\mathbf{x}) = \begin{cases} 0 & \text{pro} \quad i \notin \mathcal{A}(\mathbf{x}), \\ g_i^-(\mathbf{x}) & \text{pro} \quad i \in \mathcal{A}(\mathbf{x}). \end{cases}$$

A konečně zavádíme *redukovaný volný (reduced free) gradient* $\widetilde{\boldsymbol{\varphi}}(\mathbf{x})$,

$$\widetilde{\varphi}_i(\mathbf{x}) = \begin{cases} \min\left\{\frac{x_i - \ell_i}{\overline{\alpha}}, \varphi_i(\mathbf{x})\right\} & \text{pro} \qquad i \in \mathcal{I}, \\ \varphi_i(\mathbf{x}) & \text{jinak.} \end{cases}$$

Samotný algoritmus MPRGP je Algoritmus 2. V hranatých závorkách u vstupních proměnných je uvedeno mapování na proměnné SMALBE, liší-li se označení. Abychom mohli vybrat $\overline{\alpha}$, potřebujeme co nejlevněji určit $||\mathbf{A}||$. K tomu lze použít Algoritmus 3.

Algoritmus 2 (MPRGP - Modified proportioning with reduced gradient projections)**.**

Require: $\mathbf{A} \in \mathbb{R}^{m\times m}$ {*symetrická pozitivně definitní*}
$\mathbf{b} \in \mathbb{R}^m \quad [\mathbf{b} := \mathbf{Pd} - \mathbf{G}^\top \boldsymbol{\mu}]$
$\boldsymbol{\ell} \in \mathbb{R}^m$
$\mathcal{I}$ {*množina indexů omezených prvků*}
$\mathbf{x}^0 \in \Omega_B = \{\mathbf{x} \in \mathbb{R}^m : x_i \geq \ell_i \text{ pro } i \in \mathcal{I}\} \quad [\mathbf{x}^0 := \boldsymbol{\lambda}_\mathbf{k}]$
$\overline{\alpha} \ \in (0, 2/||\mathbf{A}||\rangle \quad [\overline{\alpha} \approx 2/||\mathbf{A}||]$
$\Gamma > 0$
$\varepsilon \quad [\varepsilon := \min\{M_k||\mathbf{G}\boldsymbol{\lambda}^k||,\ \eta\}]$ {*numerická přesnost*}
Ensure: $\mathbf{x} = \arg\min\ \mathbf{x}^\top\mathbf{Ax} - \mathbf{x}^\top\mathbf{b} \quad \text{s.t. } \mathbf{x} \in \Omega_B$

$\mathbf{x} := \mathbf{x}^0$
$\mathbf{g} := \mathbf{Ax} - \mathbf{b}$
$\mathbf{p} := \boldsymbol{\varphi}(\mathbf{x})$
while $||\boldsymbol{\varphi}(\mathbf{x}) + \boldsymbol{\beta}(\mathbf{x})|| > \varepsilon$ **do**
  if $||\boldsymbol{\beta}(\mathbf{x})||^2 \ \leq \ \Gamma^2\widetilde{\boldsymbol{\varphi}}(\mathbf{x})^\top\boldsymbol{\varphi}(\mathbf{x})$ **then**
    {$\mathbf{x}$ *is strictly proportional*}
    $\alpha_{cg} := \mathbf{g}^\top\mathbf{p}/\mathbf{p}^\top\mathbf{Ap}$
    $\alpha_f := \min\{(x_i - \ell_i)/p_i : p_i > 0\}$ {$\alpha_f := \max\{\alpha : \mathbf{x} - \alpha\mathbf{p} \in \Omega_B\}$}
    if $\alpha_{cg} \leq \alpha_f$ **then**
      $\mathbf{x} := \mathbf{x} - \alpha_{cg}\mathbf{p}$ {*conjugate gradient step*}
      $\mathbf{g} := \mathbf{g} - \alpha_{cg}\mathbf{Ap}$
      $\gamma := \boldsymbol{\varphi}(\mathbf{x})^\top\mathbf{Ap}/(\mathbf{p}\top\mathbf{Ap})$
      $\mathbf{p} := \boldsymbol{\varphi}(\mathbf{x}) - \gamma\mathbf{p}$
    else
      $\mathbf{x} := \mathbf{x} - \alpha_f\mathbf{p}$ {*feasible half-step*}
      $\mathbf{g} := \mathbf{g} - \alpha_f\mathbf{Ap}$
      $\mathbf{x} := \mathbf{x} - \overline{\alpha}\widetilde{\boldsymbol{\varphi}}(\mathbf{x})$ {*expansion step*}
      $\mathbf{g} := \mathbf{Ax} - \mathbf{b}$
      $\mathbf{p} := \boldsymbol{\varphi}(\mathbf{x})$
    end if
  else
    {$\mathbf{x}$ *is not strictly proportional*}
    $\mathbf{d} := \boldsymbol{\beta}(\mathbf{x})$
    $\alpha_p := \mathbf{g}^\top\mathbf{d}/(\mathbf{d}^\top\mathbf{Ad})$
    $\mathbf{x} := \mathbf{x} - \alpha_p\mathbf{d}$ {*proportioning step*}
    $\mathbf{g} := \mathbf{g} - \alpha_p\mathbf{Ad}$
    $\mathbf{p} := \boldsymbol{\varphi}(\mathbf{x})$
  end if
end while
return $\mathbf{x}$

Algoritmus 3 (Odhad $||\mathbf{A}||$)**.**
Require: $\mathbf{A} \in \mathbb{R}^{m\times m}$ {*symetrická pozitivně definitní*}
$\mathbf{x} \in \mathbb{R}^m \setminus \{\mathbf{o}\}$
$n_{\text{it}} > 0$ {*předem daný počet iterací*}
Ensure: $a \approx ||\mathbf{A}||$

```
1: for i := 0 to n_it do
2:     y := Ax
3:     x := ||y||^{-1} y
4: end for
5: a := ||Ax||
6: return a
```

1.5 Numerická a paralelní škálovatelnost

Numerická škálovatelnost znamená, že doba běhu algoritmu je přibližně přímo úměrná počtu neznámých. Paralelní škálovatelnost je pak taková vlastnost, že doba běhu je nepřímo úměrná počtu procesorů, které úlohu řeší paralelně. Algoritmům, které mají obě tyto vlastnosti, se říká prostě škálovatelné. Takové vlastnosti jsou velmi významné, protože výpočetní náročnost rozsáhlých úloh bez využití efektivních algoritmů může růst rychleji nežli samotný výkon počítače.

Otázkou je, zda i algoritmy prezentované dříve v této kapitole mají tyto vlastnosti. A právě pro SMALBE s vnitřní smyčkou implementovanou algoritmem MPRGP je teoreticky dokázáno Větou 8.3 v [5] a také diskuzí na konci kapitoly 8.2 a 8.4, že tuto vlastnost mají. Následné numerické experimenty to potvrzují.

Kapitola 2

Paralelní počítačové výpočty

V této kapitole se budeme zabývat problematikou paralelních počítačových výpočtů a jejich programování. *Paralelní počítačové výpočty* (lépe snad angl. *parallel computing*) obecně spočívají tom, že původní problém, jenž je řešen sekvenčním programovým kódem, se rozloží na menší podproblémy, do jisté míry nezávislé, a odpovídající části kódu lze poté řešit souběžně, čímž se dosahuje výrazného snížení času výpočtu.

Nejprve se si paralelizaci rozdělíme na čtyři základní úrovně. Dále přejdeme k představení základních paradigmat paralelního programování, a to přímo na konkrétních příkladech API a knihoven, které tato paradigmata reprezentují. Větší část sekce bude věnována MPI, což je API reprezentující model *distribuovaná paměť/předávání zpráv*. Zmíníme však i některá další masově používaná paradigmata a jejich realizace.

V další sekci se budeme věnovat specifikům *vývoje paralelních aplikací* z hlediska odlišností jejich softwarového procesu oproti sekvenčním aplikacím a poukážeme na nové problémy, které při jejich vývoji vyvstávají.

Následně se ve stručném přehledu seznámíme s některými paralelními *numerickými* knihovnami. Nejvíce se budeme zabývat tím, co na tomto poli nabízí MATLAB, potažmo knihovna MatSol, jež je v tomto jazyce/prostředí vyvíjena. Pak přejdeme k C knihovně PETSc a C++ knihovně OOSol, zejména se zaměříme na jejich srovnání, výhody a nevýhody. V případě knihovny OOSol, která je vyvíjena na Katedře aplikované matematiky a autor se na ní podílí, jsou navíc diskutovány některé současné problémy a předloženy návrhy na jejich možná řešení. V závěru si velmi stručně představíme některé alternativní paralelní numerické knihovny.

2.1 Úrovně paralelizace

Výpočty lze paralelizovat na několika úrovních:

1. na úrovni bitového kódu (*bit-level*)

2. na úrovni instrukcí (*instruction-level*)
3. na úrovni dat (*data-level*)
4. na úrovni úloh (*task-level*)

Úroveň 1 – paralelizace bitového kódu – spočívá ve zvyšování délky bitového slova, se kterou pracuje procesor. Jejím zvyšováním se snižuje počet instrukcí, jež musí procesor vykonat, aby provedl operaci na proměnných, jejichž bitová délka je větší než délka slova. U osobních počítačů se dnes setkáváme s 32-bitovou délkou, která byla nejčastější celá dvě desetiletí, a 64-bitovou délkou, která ji postupně nahrazuje. Velké počítače (serverů, superpočítačů) mají již prakticky výhradně 64-bitovou architekturu. Oproti tomu více jak polovina produkovaných procesorů je pouze 8-bitových, jedná se např. o prvky pro spotřební elektroniku. Tato nejnižší úroveň paralelizace se zjevně odehrává na úrovni hardware, operačního systému a překladače. Programátor koncových aplikací je nemůže a vlastně ani netouží nijak ovlivnit. A stěží bychom dnes nazvali nějakou aplikaci paralelní jen proto, že je optimalizována pro delší bitové slovo. Touto úrovní se tedy dále nebudeme zabývat.

Úroveň 2 – paralelizace instrukcí – spočívá v odhalení toho, které z primitivních numerických operací, jako je sčítání, násobení, modulo, negace apod., daného programového kódu, lze provádět nezávisle na sobě a tedy i souběžně. Máme-li např. posloupnost operací

1. `e = a + b`
2. `f = c + d`
3. `g = e * f`,

lze zřejmě operace 1 a 2 vykonat souběžně, zatímco operaci 3 lze korektně vypočítat až poté, co operace 1 a 2 skončily a daly mezivýsledky. Úkolem návrhářů procesorů a překladačů je potom co nejvíce tento potenciál využít. Příkladem může být *pipelining*. Všimněme si, že tato úroveň se týká opět hardwaru až překladače, zatímco programátora koncových aplikací také nemusí příliš zajímat, protože jeho zdrojový kód zůstává nedotčen. Ovšem vzhledem k současným reáliím, kdy jedna paměťová operace má časové nároky odpovídající třeba několika stům procesorových cyklů, je tato úroveň paralelizace neadekvátní pro paralelizaci dat mimo čip a je potřeba se zamýšlet nad sofistikovanějšími způsoby, které dokáží efektivně využít architektur s více jádry, procesory nebo uzly. Ty už se programátora koncových aplikací většinou týkají, a to dosti markantním způsobem.

Úroveň 3 – paralelizace dat, někdy také nazývána *paralelizace smyček (loop-level parallelism)*, se realizuje tak, že každá výpočetní jednotka (jádro, procesor nebo i celý počítač – výpočetní uzel) provádí stejnou úlohu se stejným kódem nad různými částmi dat. Často však má jedna jednotka řídící roli a koordinuje celý průběh výpočtu, nazýváme ji *master*. Ostatní jednotky se mu podřizují, říkáme jim *slave* nebo *worker*. *Master* tedy fakticky vykonává jiné

části kódu. Toho se ale zpravidla dosahuje tím, že kód je jednotný a jsou v něm logické podmínky typu *jestliže jsem master, tak ..., jinak ...*, což zabraňuje zbytečným potížím s udržováním dvou zdrojových kódů a spouštěním dvou binárních kódů. Zpravidla výpočet probíhá tak, že je úloha rozdělena *masterem* na podúlohy a ty jsou přiřazeny jednotlivým *slavům*, ty nezávisle na sobě (tj. bez komunikace) počítají „svou“ podúlohu a nakonec svůj výsledek pošlou *masterovi*, který sestaví z dílčích výsledků kompletní výsledek původní úlohy. Tento způsob paralelizace vyžaduje takovou povahu vstupních dat, aby je bylo možné rozdělit zhruba na stejné části, které mohou být zpracovány nezávisle. Abychom uvedli jednoduchý konkrétní příklad, uvažujme dvě matice $\mathbf{A}$ a $\mathbf{B}$ stejné velikosti, nad nimiž provádíme operaci sčítání

$$\mathbf{C} = \mathbf{A} + \mathbf{B},$$

přičemž máme k dispozici n výpočetních jednotek. Povaha matic vyhovuje předpokladům datové paralelizace. Postupovat můžeme třeba tak, že $\mathbf{A}$ a $\mathbf{B}$ rozdělíme na n horizontálních pásů o zhruba stejném počtu řádků tak, aby k-tý pás $\mathbf{R}_k(\mathbf{A})$ matice $\mathbf{A}$ měl stejný počet řádků jako k-tý pás $\mathbf{R}_k(\mathbf{B})$ matice $\mathbf{B}$. Pak stačí, aby k-tá výpočetní jednotka počítala

$$\mathbf{R}_k(\mathbf{C}) = \mathbf{R}_k(\mathbf{A}) + \mathbf{R}_k(\mathbf{B}),$$

a výslednou matici sestaví *master* jednoduše:

$$\mathbf{C} = \begin{bmatrix} \mathbf{R}_1(\mathbf{C}) \\ \mathbf{R}_2(\mathbf{C}) \\ \vdots \\ \mathbf{R}_n(\mathbf{C}) \end{bmatrix}.$$

V ideálním případě, kdy můžeme zanedbat časové nároky komunikace mezi výpočetními jednotkami, je výpočetní čas přímo úměrný velikosti dat a nepřímo úměrný počtu výpočetních jednotek. Je zřejmé, že takový způsob paralelizace již má jisté nároky na programátora, zdrojový kód takto paralelizované aplikace se liší od její čistě sekvenční podoby v tom, že je potřeba odlišit úlohy *mastera* a *slavů* a zajistit dekompozici dat.

Úroveň 4 – paralelizace úloh, lze se setkat také s pojmenováním *funkční paralelizace (function parallelism)*, přidává oproti předchozí úrovni distribuci funkcionality. Programová vlákna pracují nad stejnými nebo různými daty a vykonávají obecně různé úlohy s různým kódem. Nejsou tedy jen role *mastera* a *slavů*, ale rolí může být více, až tolik, kolik je vláken. Z toho také vyplývá, že čas výpočtu nemusí být přímo úměrný velikosti problému a nepřímo úměrný počtu výpočetních jednotek.

2.2 MPI

MPI [17] je standardizovaná API specifikace pro komunikaci mezi procesy, primárně zaměřená na architektury s distribuovanou pamětí. To jsou architektury, kde každý procesor může přistupovat jen do paměti, kterou vlastní, ale procesory jsou propojeny rychlou sítí a mohou sdílet data předáváním zpráv. Některé implementace MPI však obsahují též rozšíření architektury se sdílenou pamětí, o nichž bude řeč v další kapitole. Je produktem MPI fóra [16], které zastřešuje řadu výrobců, vládních i nevládních výzkumných organizací a vysokých škol. Cílem MPI je vysoký výkon, škálovatelnost a přenositelnost. Je v dnešní době dominantním modelem v oblasti paralelních výpočtů. Je jazykově nezávislá a existuje řada jejích volně dostupných i komerčních implementací v C/C++, Javě, Pythonu a dalších jazycích. Za referenční implementaci lze považovat MPICH [3]. Poslední verze MPI specifikace je 2.2 a její kompletní dokumentace je volně ke stažení [18].

MPI je představitelem modelu *sdílená paměť/předávání zpráv*. MPI je de fakto standardem pro tento model paralelizace a komunikace mezi procesy. Dle Flynnovy taxonomie reprezentuje paradigma *SPMD (Single Program Multiple Data)*. Tedy jeden zdrojový kód je přeložen do jednoho binárního kódu a ten je spuštěn z jednoho místa. Z hlediska hierarchie prezentované v kapitole 2.1 pracuje MPI s 3. a 4. úrovní.

Program na bázi MPI je nutno spouštět pomocí speciální utility `mpiexec` (dříve `mpirun`) s parametrem `-np n`, která zajistí souběžné spuštění `n` procesů, každý s jednou instancí programu, které jsou rovnoměrně rozmístěny na dostupné procesory. Tomu se říká statické vytváření procesů. Zpravidla se pro maximální výkon volí `n` rovno počtu procesorů. V MPI-2 však již existuje i možnost dynamického vytváření procesů za běhu programu. Procesy lze shlukovat do tzv. komunikátorů.

Základem komunikace mezi procesy je, jak již bylo řečeno, předávání zpráv. Zprávy vždy obsahují identifikátor vysílajícího procesu, cílového procesu a komunikátoru, v němž se nachází cílový proces. Dále obsahuje identifikátor zprávy. Zpráva také může obsahovat data, a to primitivní datové typy nebo jejich pole. V základním MPI API není přímo podporováno posílání C++ objektů. C struktury podporovány jsou, i když poněkud nepohodlně. Komunikace (výměna zpráv) může být synchronní (blokující) nebo asynchronní. Na straně odesílatele lze specifikovat, zda bude zpráva bufferována, což umožňuje pokračovat v běhu programu, i když druhá strana nezačala přijímat. Kromě přímé komunikace MPI podporuje také kolektivní komunikaci. Operací *broadcast* může jeden proces poslat svá data na všechny procesy zadaného komunikátoru, operacemi *scatter* a *gather* lze elegantně realizovat distribuci dat na procesy daného komunikátoru a následné sesbírání jejich částečných výsledků. Operace *barrier* realizuje synchronizaci mezi procesy.

Kompletní výčet všech funkcí lze nalézt v [18]. Přívětivý tutoriál pro základy MPI lze nalézt např. na [2].

2.2.1 Objektově orientovaná rozšíření MPI

MPI má také své slabé stránky. Programátorovi, který je zvyklý na tzv. vyšší programovací jazyky, jejichž éra začala s představením C++, bude základní MPI API připadat nezvykle staromilské a nepřívětivé. Primárními jazyky jsou totiž Fortran a ANSI C, které patří k nejstarším používaným jazykům. Jedním z důsledků je z dnešního pohledu nepříliš elegantní ošetření chyb způsobem, že veškeré rutiny vracejí hodnotu typu `int`, která je 0, pokud rutina proběhla v pořádku, nebo nenulové číslo značící chybový stav. Dále to znamená již zmíněnou neúplnou podporu uživatelsky definovaných typů, což může narušovat celkovou architekturu objektově orientovaného systému. Dokonce i předávání řetězce z jednoho procesu do druhého je dosti nepohodlné – znamená sérii několika zpráv (nejprve je potřeba zjistit délku řetězce!) a explicitní bufferování.

Naštěstí existují rozšíření MPI, která přidávají plnou podporu objektově orientovaných postupů. K nejuznávanějším patří knihovna *Boost.MPI*. Odborná veřejnost se shoduje na tom, že knihovny *Boost* [25] patří k tomu nejlepšímu, co svět C++ nabízí. Zárukou kvality je již to, že všechny procházejí procesem *peer review*, tedy hodnocením obecně uznávanými experty v dané oblasti. Dokonce velká část těchto knihoven předznamenává právě dokončovaný mezinárodní standard *C++0x* [1], který by měl nahradit stávající standard *ISO/IEC 14882*, publikovaný roku 1998 a aktualizovaný v roce 2003. Více o tomto standardu se lze dočíst např. na Wikipedii.

Boost.MPI není celkově nová knihovna, ale spíše C++ rozhraní ke standardnímu MPI. Je proto potřeba připojit také některou impelementaci MPI. *Boost.MPI* je psána plně v moderním objektově orientovaném stylu. Ošetření chybových stavů je řešeno systémem výjimek. Podpora uživatelských typů je zajištěna využitím serializace, což sice může obnášet určitou režii, na druhou stranu se tím ušetří komunikace a knihovna má i jisté možnosti další optimalizace. Knihovna obsahuje také vylepšenou správu komunikátorů a další vylepšení.

2.3 GPGPU/CUDA – od herního průmyslu k lineární algebře

Zajímavým trendem poslední doby je využívání procesorů grafických karet (GPU – Graphic Processing Unit), které byly původně určeny a používány výhradně ke zpracování 2D a 3D grafiky a jejímu vykreslování, k jiným než grafickým výpočtům. Poté, co se grafická jádra začala vyrábět jako plně programovatelná, se totiž ukázalo, že jejich výkon a vysoce paralelní struktura vykazuje mnohem vyšší efektivitu než univerzální CPU u celé řady algoritmů.

Dochází tak k zajímavé situaci, kdy výsledky výzkumu a vývoje v oblasti dobře dotovaného herního průmyslu náhle mají význam i pro „seriózní“ odvětví využívající masivně paralelních počítačů. Pro využívání GPU k obecným účelům se vžila zkratka *GPGPU (General-purpose computing on graphics processing units)*.

GPU již tradičně staví na paradigmatu *stream processingu* (proudové zpracování), které pracuje s představou *streamu* (proudu), což je množina datových prvků, jež vyžadují podobné zpracování. *Kernel* (jádro) je funkce, která má být vykonána na každém prvku *streamu*. *Stream processing* potom probíhá tak, že každý prvek se načte ze vstupního *streamu*, aplikuje se na něj posloupnost *kernelů* a zapíše se do výstupního *streamu*. GPU obsahuje velké množství (až stovky) výpočetních jednotek a každá z nich je schopna provádět tyto akce nezávisle na ostatních. Proti tomu CPU dnes mají obvykle jednu až čtyři výpočetní jednotky, které jsou ale mnohem rychlejší. Dále rozhraní mezi pamětí a výpočetními jednotkami grafické karty je mnohem rychlejší, propustnější a paralelizovanější, než je tomu u rozhraní CPU k RAM. L2-cache procesoru je sice velmi rychlá, ale má jen velmi omezenou kapacitu několika MB. Z uvedeného vyplývá, že výpočet na GPU může být třeba o dva řády rychlejší než na CPU téhož počítače, avšak pouze pro masivně paralelizovatelné úlohy, které mohou být spuštěny ve stovkách vláken s co nejmenší komunikací mezi vlákny. Ovšem třeba BLAS rutiny jsou právě takovými algoritmy. Nabízí se tedy využití GPU pro některé k tomu vhodné části numerických výpočtů.

Aktuálně asi nejvýznamnějším představitelem GPGPU je *CUDA* společnosti NVIDIA. Jedná se o API a jeho implementaci v jazyce C, umožňující programovat masivně paralelní aplikace spouštěné na grafických kartách této společnosti. Programují se zcela standardně jako běžné C aplikace, pouze na vhodných místech se použijí CUDA rutiny pro kopírování dat do GPU, spuštění výpočtu a kopírování výsledků zpět do hlavní paměti. CUDA je zdarma ke stažení ve formě SDK včetně dokumentace [21]. V rámci CUDA SDK je dodávána knihovna CUBLAS, což je implementace BLAS level 1,2,3 pro husté matice [22], ke stažení je nově i implementace BLAS pro řídké matice [26]. Možností, jak využít knihovnu CUDA ke zrychlení výpočtů existuje mnoho. Nic nebrání tomu použít ji k urychlení některých výpočtů v MATLABu přes MEX rozhraní [24, 19, 20] o jeden až dva řády na témže počítači. Další zajímavou možností je kombinace s MPI, kdy uzly jsou spojeny do clusteru a komunikují pomocí MPI a na každém uzlu jsou nainstalovány GPU podporující CUDA [14]. Zdá se, že o tomto novém trendu ještě mnoho uslyšíme.

Kapitola 3

Paralelní numerické knihovny

3.1 Paralelní výpočty v MATLABu

Matematický balík MATLAB společnosti MathWorks nabízí zajímavé a docela intutivní možnosti paralelních výpočtů. Klíčové jsou dva toolboxy:

- Parallel Computing Toolbox
- Distributed Computing Server

Parallel Computing Toolbox umožňuje rozložit výpočet z uživatelského sezení MATLABu, kterému se říká *client* na jiná sezení, zvaná *workers* (dělníci). Sám o sobě dovoluje spuštění nejvýše čtyř procesů současně (tedy jednoho klienta a tři dělníky), a to na tom samém počítači. Taková konfigurace se tedy hodí pouze pro urychlení výpočtu na osobním počítači se dvěma nebo čtyřmi jádry. Pro delegování výpočtu na výpočetní cluster je potřeba mít na tomto clusteru nainstalován *Distributed Computing Server*, na němž je pak počet dělníků omezen zakoupenými licencemi. Na serverové straně je ještě jakýsi prostředník mezi klientem a dělníky, který koordinuje přidělování práce dělníkům. Je jím buď standardní plánovač *scheduler* třetí strany (LSF, PBS Pro, TORQUE nebo CCS) nebo jeho specializovanější obdoba od MathWorks, tzv. *job manager*.

3.1.1 Paralelizace smyček

Parallel Computing Toolbox podporuje 3. i 4. úroveň paralelizace, jak jsme si je zavedli v kapitole 2.1. Pro 3. úroveň poskytujes jako základní nástroj paralelizovanou smyčku `parfor`, jejíž syntaxe i sémantika je obdobná jako u běžné smyčky `for`:

Pro alokaci požadovaného počtu dělníků a dalších prostředků pro paralelní běh je ještě před použitím `parfor` nutné spustit příkaz

```
for i = 1:n
   ...
end
```

```
parfor i = 1:n
   ...
end
```

`matlabpool`

Bez něj by `parfor` fungoval jako běžný `for`.

MATLAB vykonává sérii příkazů (tělo cyklu) na nějakém rozsahu hodnot indexové proměnné. MATLAB se však automaticky postará o rozložení iterací na dostupné procesory – část iterací počítá klient a část dělníci. Data, která tělo cyklu používá, jsou rozeslána z klienta na dělníky, kteří odvedou většinu výpočetní práce a pošlou své výsledky zpět klientovi, který je sestaví do celkového výsledku. Klient má tedy úlohu *mastera*.

Dělníci vyhodnocují jednotlivé iterace nezávisle na sobě, v nedefinovaném pořadí a asynchronně. Z toho plyne, že konstrukci `parfor` nelze použít tehdy, pokud na sobě iterace závisejí. Chování `parfor` je tedy přece jen poněkud odlišné od sekvenčního `for`. Navíc se chování `parfor` odlišuje v tom, jak se chová k dočasným proměnným. K těm patří iterační proměnná `i` stejně jako každá jiná, jež je cílem přiřazení uvnitř cyklu. Tyto mají po skončení cyklu nastavenu původní hodnotu před cyklem. Viz příklad:

```
d = 77; i = 0;
parfor i = 1:4
   d = i*2;
   A(i) = d;
end
A %změněno
d % = 77
i % = 0
```

Pokud požadujeme, aby proměnná byla tzv. redukční, tedy aby byla akumulována přes všechny iterace, smíme do ní v těle cyklu pouze přiřazovat a smí záviset pouze na své předchozí generaci. Vlevo je korektní kód, vpravo chybný kód, který nelze ani spustit.

```
x = 0;
parfor i = 1:10
   x = x + i;
end
x % = 55
```

```
x = 0; y = 0;
parfor i = 1:10
   x = x + i;
   y = x + 1;
end
x
y
```

3.1.2 Další možnosti paralelizace

Zatím jsme se seznámili s paralelizací na 3. úrovni, Parallel Computing Toolbox má však i rozsáhlé možnosti paralelizace na 4. úrovni, tedy na úrovni funkční dekompozice. S tím je spojeno určité pojmosloví: Zavádí se pojem *job*, což je nějaká velká operace, kterou chceme paralelizovat. Předpokládá se, že `job` lze dekomponovat na dílčí podúlohy, *tasky*. Ty obecně nemusejí být identické. *Joby* jsou dvojího druhu:

1. distributed job
2. parallel job

Distributed job je takový, že jeho *tasky* mezi sebou přímo nekomunikují. Nepotřebují běžet současně ani býti nějak synchronizovány, dokonce mohou být vykonávány za sebou.

Oproti tomu v *parallel job* je definován jen jeden *task*, jehož přesné kopie zpracovává souběžně větší počet dělníků, každý nad jiným segmentem dat, a mohou spolu komunikovat i během výpočtu. V tomto případě se dělníkům říká *laby*.

Obě paradigmata se při použití Parallel Computing Toolbox programují velmi podobným způsobem:

1. vytvořit místní objektovou reprezentaci plánovače nebo najít *job manager*
2. vytvořit *job*
3. vytvořit *tasky*
4. předat *job* plánovači nebo *job manageru*
5. obdržet výsledky *jobu*

Tím, co bylo dosud uvedeno, možnosti Parallel Computing Toolboxu zdaleka nekončí, avšak jako příklad jeho možností to postačí. Viděli jsme, že paralelizace je implementována na vysoké úrovni abstrakce, s dobře zavedenými pojmy, které jsou v dokumentaci konzistentně používány. Parallel Computing Toolbox je nadprůměrně rychle pochopitelný a odstiňuje uživatele od mnoha implementačních detailů, jako je použitý protokol předávání zpráv, takže jeho použití je přímočaré. Ve spojení s Distributed Computing Serverem a dalšími funkcemi MATLABU, jako je výborný debugger, jednoduchá konstrukce matic, jednoduché vstupně-výstupní operace a řada dalších se zřejmě jedná o velmi silný nástroj umožňující rychlý vývoj aplikací.

3.2 MatSol

Matsol je knihovna vyvíjena na Katedře aplikované matematiky v prostředí a jazyce MathWorks MATLAB, s cílem umožnit nahrazení standardních řešičů dodávaných v komerčních i nekomerčních balících pro FEM moderními paralelně a numericky škálovatelnými algoritmy

založenými na DDM, konkrétně FETI a BETI. MatSol může být na jednu stranu zajímavou alternativou pro uživatele těchto balíků, na stranu druhou dává příležitost vývojářům algoritmů otestovat nové metody na realistických úlohách. MatSol aktuálně obsahuje řešiče určené zejména pro koercivní i semikoercivní kontaktní úlohy, úlohy tvarové optimalizace a další mechanické úlohy. Paralelizace je realizována prostřednictvím technologií MATLABParallel Computing Toolbox / Distributed Computing Server.

3.3 PETSc

PETSc [petsí] je zkratkou pro *Portable, Extensible Toolkit for Scientific Compitation*. Je to paralelní numerická knihovna psaná ve variantě pro ANSI C a FORTRAN. Poskytuje rozsáhlou sadu datových struktur a rutin pro implementaci numerické řešení zejména parciálních diferenciálních rovnic a souvisejících problémů na paralelních počítačích. Pro meziprocesovou komunikaci využívá MPI standard (kapitola 2.2 na straně 22). Je dílem lidí z University of Chicago a Argonne National Laboratory.

Knihovna je organizována hierarchicky, do nabalujících se modulů, čímž je umožněno použít úroveň abstrakce, jež je adekvátní řešenému problému. PETSc obsahuje paralelních lineárních i nelineárních řešičů a k tomu potřebných rutin nižší úrovně. K nim patří rutiny pro sestavování paralelních hustých a řídkých matic v několika formátech uložení a vektorů, s nimi kompatibilní paralelizované BLAS úrovně 1, 2 a 3, rutiny pro souborový a obrazovkový vstup a výstup, funkce pro měření času, logování, ladění a profilování aj. V neposlední řadě obsahuje rozhraní k mnoha jiným knihovnám, jako je rozdělovací grafový nástroj ParMETIS, nástroj pro automatické derivování ADIC a řada externích řešičů, a k velkým matematickým balíkům, k nimž mezi jinými patří MATLAB, či Mathematica.

Základní moduly PETSc lze rozdělit podle objektů, s nimiž pracují:

- indexové množiny (`IS`)
- vektory (`Vec`) – paralelní i sekvenční
- matice (`Mat`) – řídké i husté, paralelní i sekvenční
- distribuovaná pole (`DA`) – (pro paralelizaci gridových metod, např. konečných diferencí)
- metody Krylovových podprostorů (`KSP`) – např. CG
- předpodmiňovače (`PC`) – např. ILU
- nelineární řešiče (`SNES`) – např. Trust Region

- řešiče časově závislých parciálních diferenciálních rovnic, tzv. *timesteppery* (TS)

Rutiny odpovídající jednotlivým modulům jsou odlišeny prefixem uvedeným v závorkách, obecné a podpůrné funkce mají prefix `Petsc`.

PETSc umožňuje do značné míry přizpůsobení vlastním potřebám, např. integrovat vlastní solvery.

3.3.1 PETSc – Ukázkový příklad

```
#include "petscksp.h"

int main(int argc,char **args)
{
    Vec            x, b, u;      /* approx solution, RHS, exact solution */
    Mat            A;            /* linear system matrix */
    KSP            ksp;         /* linear solver context */
    PC             pc;           /* preconditioner context */
    PetscReal      norm;         /* norm of solution error */
    PetscErrorCode ierr;
    PetscInt       i,n = 10,col[3],its;
    PetscMPIInt    size;
    PetscScalar    neg_one = -1.0,one = 1.0,value[3];

    PetscInitialize(&argc,&args,(char *)0,help);
    ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);

    /* Get value of CLI argument -n */
    ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);

    /* Create vectors x, b, u. */
    ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
    ierr = PetscObjectSetName((PetscObject) x, "Solution");CHKERRQ(ierr);
    ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
    ierr = VecSetFromOptions(x);CHKERRQ(ierr);
    ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
    ierr = VecDuplicate(x,&u);CHKERRQ(ierr);

    /* Create matrix A. */
    ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
    ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n);CHKERRQ(ierr);
    ierr = MatSetFromOptions(A);CHKERRQ(ierr);

    /* Assemble matrix A. */
    value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
    for (i=1; i<n-1; i++) {
        col[0] = i-1; col[1] = i; col[2] = i+1;
        ierr = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
    }
    i = n - 1; col[0] = n - 2; col[1] = n - 1;
```

```
41      ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
42      i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
43      ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
44      ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
45          /* here can be some code to overlapping communication */
46      ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
47
48      /* Set exact solution u = ones(n); then compute right-hand-side vector b = Au.*/
49      ierr = VecSet(u,one);CHKERRQ(ierr);
50      ierr = MatMult(A,u,b);CHKERRQ(ierr);
51
52      /* Create linear solver context. */
53      ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);
54
55      /* Set operators. Matrix of linear system serves also as preconditioning matrix. */
56      ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);
57
58      /* Set linear solver defaults for this problem. */
59      ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
60      ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
61      ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT);CHKERRQ(ierr);
62
63      /* Solve linear system. */
64      ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
65
66      /* View solver info. */
67      ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
68
69      /* Check solution. */
70      ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
71      ierr  = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
72      ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
73      ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %A, Iterations %D\n",
74          norm,its);CHKERRQ(ierr);
75
76      /* Clean-up. */
77      ierr = VecDestroy(x);CHKERRQ(ierr); ierr = VecDestroy(u);CHKERRQ(ierr);
78      ierr = VecDestroy(b);CHKERRQ(ierr); ierr = MatDestroy(A);CHKERRQ(ierr);
79      ierr = KSPDestroy(ksp);CHKERRQ(ierr);
80      ierr = PetscFinalize();CHKERRQ(ierr);
81      return 0;
82  }
```

3.3.2 PETSc – Výhody, nevýhody, doporučení

V této kapitole se autor rozhodl prezentovat své názory na práci s knihovnou PETSc, jak je vidí po několika měsících užívání.

Výhody

Rozhraní – PETSc má implementovánu celou řadu rozhraní k externím softwarům, z oblasti našeho zájmu např. k ParMETISu, MATLABu.

Polymorfismus – uniformní přístup k sekvenčním i paralelním maticím a různým formátům uložení. Např. použití téhož řešiče.

Rozfázování náročných operací – PETSc poskytuje `*Begin/*End` bloky, jako na řádcích 44–46 příkladu na str. 29, které umožňují překrývání komunikace a výpočtu např. při sestavování velké globální matice.

Kontexty zefektivňující kód. Lze těžit z toho, že po sobě jdoucí matice, nad nimiž probíhá výpočet, mají stejnou strukturu nenulových čísel apod., viz příklad na str. 29, řádek 52 a dál.

Odstínění od MPI – programátor jen výjimečně potřebuje explicitně volat MPI rutiny.

Zabudovaná podpora CLI přepínačů a argumentů.

Možnost korektně pracovat přímo s vnitřním polem matice nebo vektoru pomocí párových rutin `VecGetArray` a `VecRestoreArray` (resp. s prefixem `Mat` u matic). Druhá rutina se postará a o dealokaci dočasného pole a případné zkopírování změněných hodnot zpět do objektu.

Systém *viewerů*, které umožňují jednotným způsobem např. tisknout hodnoty vektoru na obrazovku nebo jej uložit v MATLAB ASCII, či binárním formátu.

Množství formátů matic nad rámec specifikace BLAS včetně blokových.

Množství příkladů – ke knihovně se dodává přes sto příkladů ke všem modulům.

Podpora logování, ladění a profilování kódu.

Množství již hotových řešičů a předpodmiňovačů.

Konzistentní pojmenování a kódové konvence.

Odstínění od pointerů pomocí `typedefů`, které odstraňují nutnost explicitního používání operátorů `*, &, ->`

```
1                func(Vec v) {...}
2                ...
3                Vec v;
4                VecCreate(...,&v);
5                func(v);
```

Nevýhody

Lpění na ANSI C, které často vede k méně přehlednému, těžkopádnému a jen částečně OO kódu. Jsem přesvědčen, že v dnešní době, kdy všechny dostupné překladače umí C++ a existuje řada knihoven, které jej posouvají dál, už to nemá smysl. To bychom mohli zůstat u FORTRANu.

Ošetření chybových stavů pouze pomocí návratové hodnoty, která je pak odchytávána makrem `CHKERRQ`, což znepřehledňuje kód - viz příklad na str. 29.

Místy nedostatečná dokumentace – některé možná zajímavé funkce jsou devalvovány tím, že o nich není zmínka v manuálu, nejsou k nim příklady a API dokumentace je celkově přespříliš stručná. Nebo je jejich popis zjevně zastaralý. Špatně jsou dokumentovány třeba i různé formáty uložení matic. Některé důležité skutečnosti se člověk dozví až při brouzdání zdrojovým kódem nebo metodou pokus-omyl. Nicméně základní funkčnost je dokumentována slušně v manuálu.

Svázanost s Linuxem – ač by PETSc měla být dle informací na webu plně přenositelná na platformu Windows, není zřejmě pro tuto platformu příliš testována. Instalace pod Windows byla velmi, velmi zdlouhavá a netriviální, do značné míry cestou pokusů a omylů a zahrnovala mimo jiné značné úpravy instalačních skriptů. Pro jejich spuštění je potřeba mít nainstalováno alespoň pseudolinuxové prostředí Cygwin. Navíc občas některá funkce zjevně nefunguje, jak by měla (např. přímé ukládání matic do binárního formátu MATLABu způsobuje neznámou běhovou chybu).

Svázanost s konkrétními implementacemi MPI PETSc je testována primárně pro MPICH a dále už jen pro OpenMPI. Ukazuje se, že na tom záleží, i když všechny implementace MPI mají být sjednoceny MPI standardem. Testy s jinými implementacemi skončily nezdarem.

Zmatky s nedodělanou funkčností – třeba některé funkce, které „se tváří“, že fungují pro všechny typy matic, a dokumentace tomu odpovídá, ve skutečnosti fungují jen pro pár základních typů.

Časté změny API, vynucující změny v kódu. U vyšších jazyků s podporou přetížených funkcí se to řeší označením staré verze funkce jako `deprecated` a případným smazáním až po delší přechodné době, kdy obě verze koexistují. ANSI C však neumožňuje dvě funkce se stejným názvem, byť s různou signaturou.

Špatná správa verzí – v popisech funkcí se nelze dočíst, kdy byly zavedeny, ani kdy byly naposledy aktualizovány. Někdy se tak míchají již překonané funkce s jejich novějšími

a lepšími alternativami, které byly dobře otestovány, ale i s funkcemi, jež jsou zřejmě vývojové a špatně funkční. A uživatel knihovny je zmaten.

Doporučení

PETSc poměrně dost vnucuje svůj styl kódu. Člověk by si řekl, že když už je celá knihovna napsaná v ANSI C, tak to ve svém programu nebude narušovat. Zkušenosti však ukázaly, že to nemusí být nejlepší cesta. Komplexnější kód, který jsem tvořil, by se prokazatelně dal napsat mnohem elegantněji, stručněji a méně náchylný na chyby, kdybych k němu nepřistupoval jako k *nadstavbě* PETSc, ale jako samostanému programu, který pouze *využívá* PETSc ve formě wrapperů, ale má svůj vlastní, objektově orientovaný styl využívající možností C++.

PETSc se nám může hodit jako referenční knihovna, s níž budeme poměřovat náš OOSol, a možná k překonání doby, než bude OOSol dospělejší. Měli bychom se pokusit vyvarovat nedostatků, jež zde byly nastíněny. Je ale potřeba si dát pozor, aby nám PETSc nebrala příliš mnoho času, aby se vývoj neštěpil do třetí větve mimo MatSol a OOSol.

3.3.3 Implementace paralelních řešičů kontaktních úloh v PETSc

V rámci praktické části diplomové práce jsem se zabýval implementací algoritmů uvedených v kapitole 1 a rozhraní pro knihovnu MatSol psanou v MATLABu s využitím paralelní knihovny PETSc představené na začátku této kapitoly.

Vstupní data programu odpovídají vstupům problému (1.8), jsou to tedy matice $\mathbf{K}$ a $\mathbf{B}$, vektory $\mathbf{f}$ a $\mathbf{c}$ a indexová množina $\mathcal{I}$. Tato data je program schopen načíst z binárního souboru uloženého MatSolem.

Dále byl naimplementován TFETI preprocesor, jenž realizuje všechny kroky popsané v sekcích 1.2.2–1.2.4, str. 9–12, a tím převádí primární úlohu (1.8) na předpodmíněnou duální úlohu (1.27). Výstupem je tedy vektor $\mathbf{b} = \mathbf{Pd}$ a matice $\mathbf{A} = \mathbf{PFP} + \rho\mathbf{Q}$, avšak ve faktorizované podobě, tedy jako struktura A obsahující skalár ρ a matice $\widetilde{\mathbf{L}}, \widetilde{\mathbf{G}}$ (viz (1.23)), $\mathbf{B}, \mathbf{K}^{\dagger}$, přesněji jejich lokální části příslušné danému procesu.

Dále bylo potřeba realizovat algoritmus SMALBE-M, jehož pseudokód je Algoritmus 1 na str. 15. Ten má na vstupu zmíněné objekty A a $\mathbf{b}$ a vrací řešení $\boldsymbol{\lambda}$ problému (1.27). Toto řešení je uloženo do binárního souboru a vráceno MatSolu, který jej načte a pokračuje ve výpočtu.

Posledním implementovaným algoritmem je MPRGP (Algoritmus 2 na str. 17), který realizuje vnitřní smyčku algoritmu SMALBE a řeší problém (1.32).

Kód jsem vyvíjel v integrovaném vývojovém prostředí Visual Studio 2008. Vycházel jsem z přesvědčení, že kód není možné kvalitně vyvíjet přímo na cílovém počítači, jímž byl cluster Comsio, ale že je nutné mít nějaký adekvátní vývojový nástroj, možnosti ladění, bezprostřední

kontakt s kódem. Troufám si říci, že se mi takovýto postup během mé programátorské praxe vždy vyplatil. Koneckonců, též vývojáři knihovny PETSc doporučují vývoj na lokálním počítači. Samozřejmě je nutné program testovat i na cílové platformě, avšak lépe je tak činit pouze na konci každé vývojové iterace.

Počátečním bodem byla implementace, kterou jsem dostal k dispozici, která se v první chvíli zdála prakticky kompletní. Postupně při studiu kódu jsem však objevil celou řadu vážných nedostatků. Ty jsem postupně odstraňoval a doimplementoval zmíněné rozhraní.

V první řadě program nebylo možné zkompilovat ve Visual Studiu, protože byl špatně rozdělený. Neobsahoval hlavičkové soubory a kód byl rozdělen na úrovni C souborů. Ty nebyly samy o sobě zkompilovatelné, protože spoléhaly na to, že budou vnořeny do hlavního souboru. To je samozřejmě úplně špatně a některé překladače odmítají takové C soubory přeložit. Vytvořil jsem tedy zcela standardní strukturu s hlavičkovými soubory.

Kód byl dále „zanesený“ stovkami zakomentovaných řádků, které jsem musel odstranit, protože znemožňovaly orientaci v kódu. Přejmenovával jsem proměnné, aby odpovídaly textům, a stejně jsem upravoval i komentáře. Odstraňoval jsem redundantní argumenty funkcí, aby se ukázalo, jaké vlastně mají vstupy a výstupy. Měnil jsem pořadí příkazů tak, aby výpočet zůstal ekvivalentní, ale aby byla zřejmá cesta mezivýsledků. A čistě implementační mezivýsledky jsem pojmenoval poněkud odlišně od proměnných odpovídajících textům. Snažil jsem se jasně vymezit, jaké vstupy a výstupy dané algoritmy mají, aby byla každá funkčnost zasazena do správné funkce.

Teprve po tomto „velkém úklidu“ vykrystalizovaly nesrovnalosti na vyšší úrovni, zejména redundantní poměrně náročné operace – např. mnoho kopírování z paralelních vektorů do sekvenčních, které se ukázalo jako zbytečné. Naopak některé operace zcela chyběly, jmenovitě třeba řádek 1.22.

Pro vylepšení výkonu při vysokém počtu iterací jsem vytvořil globální struktury, v nichž jsou alokovány vektory použité pro mezivýsledky, s životností po celou dobu výpočtu. Tím jsem zabránil tomu, aby byly pravidelně v každém cyklu znovu alokovány a pak dealokovány.

Implementaci jsem dosud nestihl otestovat na clusteru, nicméně předběžné testy na lokálním počítači dávají slušné výsledky, srovnatelné s čistým MatSolem, a přitom cítím ještě hodně příležitostí ke zlepšení a existují i možnosti, jak odbourat předávání dat prostřednictvím souboru a místo toho je předávat in-memory. Pro ilustraci přikládám aktuální podobu

algoritmu SMALBE:

```
#include "qp_smalbe.h"

/* code for inititalization and finalization, handling global vars */

PetscErrorCode QPSMALBE(MatFact A, MatFact G, Vec b, PetscScalar epsilon, Vec lmbd)
{
    PetscScalar epsilon_in;

    while (Counter.out < P.max_out) {
        TRY( AugLag_b(Gt, d, mi, b) );              /* b = P*d - G'*mi */

        TRY( MatMultTranspose(Gt, lmbd, Glmbd) );   /* Glmbd = G*lmbd   */
        TRY( VecNorm(Glmbd, NORM_2, norm_Glmbd) );  /* norm_Glmbd = norm(G*lmbd)*/
        epsilon_in = PetscMin(eta, M*norm_Glmbd);   /* epsilon_in = min{eta, M*||G*lmbd||}*/

        /* Inner iteration with adaptive precision control.*/
        TRY( QPMPRGP(A, b, l, epsilon_in, lmbd) );

        /* Stopping criterion. */
        if (norm_gp < epsilon && norm_Glmbd < epsilon)
            break;

        /* Update of the Lagrange multipliers mi. */
        TRY( MatMultTranspose(Gt, lmbd, Glmbd) );   /* Glmbd = G*lmbd = Gt'*lmbd */
        TRY( VecAXPY(mi, rho, Glmbd) );             /* mi = mi + rho*Glmbd    */

        /* Compute augmented Lagrangian al = 0.5*lmbd'*A*lmbd - lmbd'*P*d + lmbd'*G'mi. */
        al_old = al;
        TRY( AugLag(A, b, lmbd, &al) );

        /* Update rho provided the increase of augmented Lagrangian is not sufficient. */
        if (Counter.out > 0
                && al < al_old + 0.5*rho* norm_Glmbd * norm_Glmbd)
            M = M / P.beta;

        Counter.out++;
    }

    return 0;
}
```

3.4 OOSol

OOSol je zkratkou Object Oriented Solvers (objektově orientované řešiče). Je to projekt, jehož hlavním cílem je vývoj moderní objektově orientované knihovny pro lineární algebru v jazyce C++. Objektově orientovaný přístup se týká jak datových struktur, tak numerických

algoritmů. Měl by umožnit jednoduché, přirozené a rychlé využití těchto algoritmů zejména pro řešení praktických počítačově řešených optimalizačních problémů v oblasti konstrukčních návrhů. Vyvíjen je na Katedře aplikované matematiky. OOSol obsahuje jak BLAS rutiny v sekvenčních a paralelních verzích a řešiče zejména pro QP.

Protože se mě tato problematika jako jednoho z vývojářů OOSolu bezprostředně týká, dovolil jsem si celý zbytek kapitoly napsat v 1. osobě j.č.

V rámci této diplomové práce jsem měl implementovat rozhraní PETSc-MatSol a OOSol-MatSol. Avšak jak v případě PETSc, tak v případě OOSolu se ukázalo, že se nemůžu zabývat jen tímto rozhraním, protože kódy, na něž měla má rozhraní bezprostředně navazovat, nebyly ještě hotové. Proto jsem místo dvou rozhraní, po dohodě s vedoucím dipl. práce, vytvořil jen jedno rozhraní a zkompletoval přilehlé algoritmy, jak bylo řečeno v kapitole 3.3.3. Pro PETSc jsem se rozhodl proto, že jsem se domníval, že hotové kódy TFETI, SMALBE a MPRGP, napsané s využitím této knihovny, k tomu budou zralejší. To byl ovšem omyl. Nicméně bych rád na tomto místě prezentoval své názory na to, co se podle mě musí stát, aby se OOSol po letech razantně hnul dobrým směrem. Tyto postřehy jsem postupně získal, když jsem se začal hlouběji zajímat i o jiný než vlastní kód a o celkové dění.

OOSol sice znamená Object Oriented Solvers, ale dle mého mínění zůstává tomuto označení v lecčem dlužen. Tedy ne, že by OOSol nebyl objektově orientovaný, ale mám pochybnosti, že je korektně objektově orientovaný. Na jednu stranu se zcela ztotožňuji s cíly tohoto projektu a celkovou filozofií. Na druhou stranu se jim podle mě často příliš ustupuje aktuálním, často jen zdánlivým, potřebám a nedodržuje se jednotná architektonická linie. Myslím, že uzrál čas, aby byla provedena celková revize knihovny od BLAS rutin nahoru. Toto je celkem složitý týmový úkol, nicméně předběžně jsem si již ověřil, že by tyto snahy mohly být podpořeny. Ale uveďme si několik příkladů.

Jednou z věcí, kterou podle mě bude nutno provést, je rozdělení OOSolu na několik hierarchicky uspořádaných modulů, v čemž si můžeme vzít příklad z PETSc. Třeba takto:

- sekvenční implementace matic a vektorů a BLAS
- paralelní implementace matic a vektorů a BLAS
- sekvenční lineární řešiče
- paralelní lineární řešiče
- sekvenční nelineární řešiče
- paralelní nelineární řešiče

Tím se mimo jiné jasně vymezí působnosti, např. že implementace sekvenční matice nesmí nic vědět o paralelních maticích nebo o řešičích.

S tím souvisí otázka, zda je opravdu nutné, abychom udržovali sekvenční a paralelní verze napříč celou hierarchií. Třeba PETSc pouze rozlišuje sekvenční a paralelní matice a vektory, zatímco řešiče jsou univerzální. Je opravdu důležité se vždy zamyslet, zda nám údržba dalekosáhlých kódů navíc stojíc za to. Všimněte si, že nepíši naprogramování, ale údržba. Dle mých zkušeností je právě udržování kódu u OOSolu podceňováno, ale žádný životaschopný software na světě nebyl hotový první verzí a je také potřeba počítat s tím, že může vzniknout potřeba přizpůsobení kódu nějakým novým, dříve neznámým podmínkám. Zkrátka vývoj je proces a nikoliv konečná činnost.

Působnosti je však potřeba jasně vymezovat i na daleko jemnější úrovni. Můžu uvést konkrétní příklad. Setkal jsem se s tím, že v OOSolu si každá implementace matice nese řetězec se svým názvem. To mi bylo podezřelé, protože instance různých třídy by měly být odlišeny implicitně právě svou třídní identitou. A navíc podmínkování na třídní příslušnost často indikuje nějakou chybnou úvahu v návrhu, protože by měl stačit polymorfismus metod. Tak jsem se zajímal, jak to vzniklo a vyšlo najevo, že to je kvůli tomu, aby blokové matice věděly, jakého typu jsou jejich diagonální bloky, aby je bloková matice mohla poslat MPI zprávou. Když se nad tím zamyslíme, tak mnohem lepším řešením by bylo, kdyby ty bloky uměly posílat samy sebe a poskytovaly pro to jednotnou metodu, vynucenou rozhraním. Pak by blokovou matici nemuselo zajímat, jakého konkrétního typu jsou její bloky, ale jen to, zda implementují daný interface. Takže by měla jednodušší kód bez zbytečných chyb způsobených tím, že se zapomnělo na některý maticový formát. A navíc získáváme jako bonus možnost smíšených blokových matic, kde každý blok může být obecně jiného typu.

V kódu se poměrně často objevují implementace metod, které pouze vracejí `null`, protože nebyl čas je implementovat. To, že je dosud nikdo neimplementoval, je v pořádku. Mně však vadí, že tyto metody nejsou nijak označeny a přitom vracejí *nějakou hodnotu*. Toto lze mnohem lépe ošetřit způsobem, který znám ze světa Javy - metoda radši vyhodí výjimku `throw new RuntimeException("Not implemented yet.")`, než aby někdo pokládal její návratovou `null` za skutečný výsledek. A pro jistotu by každé takové nedodělané místo mělo křičet: `//TODO doimplementovat`, příp. `//FIXME doimplementovat`.

Závěr

V práci byly prezentovány matematické metody, jež umožňují efektivní paralelní výpočet mimo jiné složitých kontaktních úloh pružnosti. Byl ukázán přechod od spojitého matematického modelu pružnosti k mnohem jednoduššímu problému kvadratického programování, který je vhodný k paralelnímu počítačovému řešení. Dále byly ukázány algoritmy kvadratického programování, které jsou vhodné k řešení takového problému v explicitní pseudokódové podobě.

Problematika diplomové práce úzce souvisí s aktuálními výzkumnými aktivitami Katedry aplikované matematiky a také s připravovaným projektem IT4Innovations. V tomto světle si uvědomuji důležitost toho, aby byly algoritmy vyvíjené na naší katedře dotaženy i z pohledu implementace a dále „profesionalizovány“. Věřím, že to je nutná podmínka „komerčního úspěchu“ naší katedry, a proto se budu zasazovat o odstranění propasti v kvalitě algoritmů na úrovni matematické abstrakce a na úrovni počítačové realizace.

To však nebude možné bez spolehlivé platformy, na které bude možno bez obav dále stavět. Má-li jí být OOSol, je potřeba na něm odvést notný kus práce. Sám se chci v tomto směru angažovat a věnovat tomu podstatnou část svého času během Ph.D. studia. Jednu z velkých výzev pro celý vývojový tým vidím v etablování funkčního softwarového procesu této knihovny. Jsem přesvědčen, že lze tvořit software vysoce výkonný, paralelizovaný a přitom v souladu s principy objektově orientovaného návrhu a s dobře definovaným softwarovým procesem. A nejen, že to lze, ale že to je vlastně nutnost, protože zmíněné principy znamenají také třeba lepší zapojitelnost nových členů týmu a bez nových lidí se nemůžeme obejít.

Samozřejmě tyto snahy na poli spíše počítačové vědy musím a chci snoubit s hlubším poznáním matematickým. Rád bych dále rozvíjel své znalosti v oblastech, jimiž jsem se začal zabývat v souvislosti s tímto textem, tedy algoritmům doménové dekompozice a souvisejícím řešičům. Rád bych se však seznámil i s matematickými modely, jimiž se naše katedra zatím tolik nezabývá (aspoň mi to tak připadá), jako je modelování hydrosféry a klimatu, což by v IT4I odpovídalo tématu SC4Natural&BioSciences. Využívají některých výsledků, s nimiž jsem teď přicházel do kontaktu, třeba konečně prvkové diskretizace, ale také hlubokých poznatků např. funkcionální analýzy a statistiky.

Tím se zase vracím ke svému implementačnímu zaměření, protože uvést tyto matematické modely v „softwarový život“, který budou žít na špičkových superpočítačích, je neskutečná výzva. Máme-li se zabývat takovými problémy, je nutné si neprve zajistit dobrou půdu pod nohama.

Literatura

[1] ISO/IEC N3092 Draft International Standard, Final Committee Draft. 2010.
URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf

[2] anonymous: The Supercomputing Blog.
URL http://supercomputingblog.com/cuda-tutorials/

[3] Argonne National Laboratory: MPICH2 : High-performance and Widely Portable MPI.
URL http://www.mcs.anl.gov/research/projects/mpich2/

[4] Brzobohatý, T.; Dostál, Z.; Kozubek, T.; Kovář, P.; Markopoulos, A.: Cholesky-SVD decomposition with fixing nodes to stable computation of a generalized inverse of the stiffness matrix of a floating structure. *International Journal for Numerical Methods in Engineering*, ročník –, předloženo, zatím nevytištěno, 2010.

[5] Dostál, Z.: *Optimal Quadratic Programming Algorithms: With Applications to Variational Inequalities*, ročník Volume 23. New York: Springer US, první vydání, 2009, ISBN 978-0-387-84805-1 (Print), 978-0-387-84806-8 (Online), doi:10.1007/b138610.

[6] Dostál, Z.; Horák, D.; Kučera, R.: Total FETI - an easier implementable variant of the FETI method for numerical solution of elliptic PDE. *Communications in Numerical Methods in Engineering*, ročník 22, č. 12, 2006: s. 1155–1162, doi:10.1002/cnm.881.

[7] Dostál, Z.; Kozubek, T.; Vondrák, V.; Brzobohatý, T.; Markopoulos, A.: Scalable TFETI algorithm for the solution of multibody contact problems of elasticity. *International Journal for Numerical Methods in Engineering*, ročník –, přijato, zatím nevytištěno, 2009, doi:10.1002/nme.2807.

[8] Farhat, C.; Lesoinne, M.; LeTallec, P.; Pierson, K.; Rixen, D.: FETI-DP: a dual-primal unified FETI method - part I: A faster alternative to the two-level FETI method. *International Journal for Numerical Methods in Engineering*, ročník 50, č. 7, 2001: s. 1523–1544, doi:10.1002/nme.76.

[9] Farhat, C.; Mandel, J.; Roux, F. X.: Optimal convergence properties of the FETI domain decomposition method. *Computer Methods in Applied Mechanics and Engineering*, ročník 115, č. 3-4, 1994: s. 365 – 385, ISSN 0045-7825, doi:10.1016/0045-7825(94)90068-X.

[10] Farhat, C.; Roux, F.-X.: An Unconventional Domain Decomposition Method for an Efficient Parallel Solution of Large-Scale Finite Element Systems. *SIAM Journal on Scientific and Statistical Computing*, ročník 13, č. 1, 1992: s. 379–396, doi:10.1137/0913020.

[11] Fragakis, Y.; Papadrakakis, M.: A unified framework for formulating Domain Decomposition Methods in Structural Mechanics. Technická zpráva, Institute of Structural Analysis & Seismic Research, National Technical University Athens, 2002.

[12] Hlaváček, I.; Haslinger, J.; Nečas, J.; Lovíšek, J.: *Solution of Variational Inequalities in Mechanics*. Springer Verlag, Berlin, 1988.

[13] Horák, D.: *FETI Based Domain Decomposition Methods for Variational Inequalities*. Dizertační práce, VŠB - Technical University of Ostrava, 2007.
URL `http://www.am.vsb.cz/theses/horak_phd.pdf`

[14] Jacobsen, D. A.; Thibault, J. C.; Senocak, I.: An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In *Computations on Multi-GPU Clusters" 48th AIAA Aerospace Sciences Meeting and Exhibit. Orlando, FL. Jan. 2010.*, 2010.
URL `http://works.bepress.com/inanc_senocak/3`

[15] Karypis, G.; Schloegel, K.; Kumar, V.: *PARMETIS - Parallel Graph Partitioning and Sparse Matrix Ordering Library*. University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, 2003.
URL `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`

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

[17] Message Passing Interface Forum: The Message Passing Interface (MPI) standard.
URL `http://www.mcs.anl.gov/research/projects/mpi/`

[18] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard, Version 2.2*. Message Passing Interface Forum.

[19] NVIDIA Corporation: *Accelerating MATLAB with CUDA Using MEX Files*.
URL `http://developer.download.nvidia.com/compute/cuda/1_0/Accelerating%20Matlab%20with%20CUDA.pdf`

[20] NVIDIA Corporation: MATLAB Acceleration.
URL http://www.nvidia.com/object/matlab_acceleration.html

[21] NVIDIA Corporation: NVIDIA GPU Computing Developer Home Page.
URL http://developer.nvidia.com/object/gpucomputing.html

[22] NVIDIA Corporation: CUDA CUBLAS Library. 2010.
URL http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/CUBLAS_Library_3.0.pdf

[23] Sadowská, M.; Dostál, Z.; Kozubek, T.; Markopoulos, A.; Bouchala, J.: Scalable Total BETI based solver for 3D multibody frictionless contact problems in mechanical engineering. *Engineering Analysis with Boundary Elements*, ročník –, předloženo, zatím nevytištěno, 2010.

[24] The MathWorks, Inc.: *Accelerating MATLAB Code Using GPUs*. 2010.
URL http://www.mathworks.com/products/techkitpdfs/91804.pdf

[25] various: *Boost C++ Libraries*. 2010.
URL http://www.boost.org/

[26] various: cusp-library: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. 2010.
URL http://code.google.com/p/cusp-library/