ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Asynchronní iterativní řešiče
Student:	Martin Quarda
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2020/21

Pokyny pro vypracování

1) Nastudujte formáty řídkých matic např. COO, CSR.
2) Nastudujte iterativní Jacobiho, Gauss-Seidelovu a modifikaci SOR Gauss-Seidelovy metodu hledání řešení soustavy rovnic.
3) Naimplementujte metody z bodu 2) sekvenčně pro husté a řídké matice.
4) Naimplementujte nastudované metody paralelně s využitím OpenMP s různým plánováním cyklů.
5) Změřte a porovnejte rychlost běhu a rychlost konvergence jednotlivých metod pro různé vstupní matice (např. z [3]).
6) Porovnejte rychlost běhu implementovaných metod oproti ostatním open-source implementacím.

Seznam odborné literatury

[1] Fiedler, M.: Speciální matice a jejich použití v numerické matematice. SNTL,Praha, 1981
[2] Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd Edition. Society for Industrial and Applied Mathematics, 2003
[3] The SuiteSparse Matrix Collection, https://sparse.tamu.edu/

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 15. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Asynchronní iterativní řešiče

Martin Quarda

Katedra teoretické informatiky
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

9. ledna 2020

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. ledna 2020 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2020 Martin Quarda. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Quarda, Martin. *Asynchronní iterativní řešiče.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020. Dostupný také z WWW: ⟨`https://beta.quarda.cz/BP.zip`⟩.

Abstrakt

Práce se zabývá třemi algoritmy hledajícími řešení lineárních soustav rovnic. Ty zvládnou vyřešit omezenou skupinu matic. Vybrané algoritmy jsou Jacobiho, Gauss-Seidelova a SOR metoda. Implementoval jsem je sekvenčně a paralelně. Při sekvenční implementaci se blížím rychlostí alternativní implementaci. Paralelní zrychlení se škáluje skoro lineárně s počtem jader až do velikosti matice odpovídající velikosti cache paměti.

Klíčová slova implementace algoritmu, Jacobiho metoda, SOR metoda, Gauss-Seidelova metoda, paralelní programování, Python, Cython, C++, OpenMP

Abstract

The work deals with three algorithms for solving linear system of equations. Matrix indicating the system of equations must meet a few prerequisites before selected algorithms handle the problem. Selected algorithms are Jacobi method, Gauss-Seidel method and SOR method. Algorithms are being implemented sequentially, parallel and then I compare their convergence speed with each other.

Keywords algorithm implementation, Jacobi method, Gauss-Seidel method, SOR method, parallel programming, Python, Cython, C++, OpenMP

Obsah

Úvod **1**
Cíl práce 1
Struktura práce 2

1 Teoretická část **3**
1.1 Formáty řídkých matic 3
1.2 Definice problému 5
1.3 Iterační metody 6
1.4 Jacobiho metoda 7
1.5 Gauss-Seidlova metoda 8
1.6 SOR modifikace metody 8
1.7 Zastavovací kritéria 9
1.8 Paralelní programování 9
1.9 Paralelizace iterativních metod 10
1.10 Možnosti optimilizace 11

2 Použité technologie **13**
2.1 OpenMP 13
2.2 Python 14
2.3 Cython 15
2.4 Představení ostatních implementací 15

3 Praktická část **17**
3.1 Použití matic v jazyce Python 17
3.2 Definice rozhraní knihovny 18
3.3 Vnitřní reprezentace matic 20

3.4 Sekvenční implementace 20
3.5 Paralelní implementace 21
3.6 PETSc4py 22

4 Měření výkonu a testování **23**
4.1 Testovací stroj 23
4.2 Matice pro měření a metodika 24
4.3 Testování optimilizací 24
4.4 Porovnání rychlosti běhu a konvergence metod na jedné matici 25
4.5 Testování na několika maticích 26
4.6 Testování velké matic 28
4.7 Porovnání oproti jiné implementaci 29

Závěr **31**

A Seznam použitých zkratek **33**

Seznam použité literatury **35**

B Obsah přiloženého média **39**

Seznam obrázků

1.1 CSR a CSC formát 5

2.1 Graf vývoje jazyka Python a dalších jazyků na StackOverflow. Porovnání jazyků 14

Seznam tabulek

4.1 Použité matic se základníma vlastnostma 24
4.2 Porovnání zhoršení konvergence v asynchronní verzi 25
4.3 Rychlost konvergence metod na matici obstclae 26
4.4 Porovnání rychlosti konvergence na několika maticích 27
4.5 Zrychlení na velké matici 28
4.6 Porovnání rychlosti oproti PETSc na několika maticích 29

Úvod

Soustavy rovnic mají uplatnění ve řadě odvětví matematiky, například ekonomie, fluidní dynamiky, lineárního programovaní a mnohých dalších. Jejich řešení je důležité při následovaných vyhodnocováních.

Iterativní metody hledající řešení se používají tam, kde klasické metody selhávají nebo jsou příliš pomalé. Selhávají pro vysoce rozměrné matice, kde klasické metody mají příliš velkou náročnost. Bohužel není možné je využít na všechny matice, protože potřebují splnit určité charakteristiky matic.

Iterativní metody jsou efektivní v řídkých maticích, kde pracují pouze s nenulovými hodnotami. Některé metody konstrukce matic pro jev, který popisují, konstruují řídké a rozměrné matice ze své povahy. V takovém případě má smysl uvažovat nad iterativními metodami.

Téma jsem si vybral, neboť mě zajímá programování paralelních aplikací a nenašel jsem existující řešení, které by implementovalo vybrané metody paralelně.

Cíl práce

Cílem teoretické práce je nastudování formátů řídkých matic, iterativní Jacobiho, Gauss-Seidelovy a SOR modifikace Gauss-Seidelovy metody hledání řešení soustavy rovnic. Také zde představuji několik použitých technologií.

V praktické části je implementace nastudovaných metod sekvenčně, paralelně a porovnání implementací mezi sebou a proti konkurenci. Implementace proběhne v programovacím jazyce `C` a `Cython`.

Implementované metody otestuji mezi sebou i oproti jiné implementaci.

Struktura práce

V kapitole 1 zadefinuji problém řešení soustavy lineárních rovnic. Představil jsem podmínky, za kterých mohou algoritmy konvergovat, a představil způsob, jakým se určuje velikost chyby. V sekcích 1.4 - 1.6 jsou postupně popsány jednotlivé metody – Jacobiho, Gauss-Seidlova a SOR.

Následně jsem představil, jak se reprezentují řídké matice v paměti. Na konci jsem shrnul proč se zvyšuje počet jader v procesoru a jaké jsou limity a úskalí paralelních programů. V poslední sekci je vysvětlený čím je limitován paralelní výkon.

V kapitole 2 jsem ukázal použité technologie. První jsem se věnoval `OpenMP`. To je standard pro psaní paralelních programů. Poté jsem představil `Python` a kde se začíná prosazovat. V sekci 2.3 jsem popsal krátce `Cython`. Na konci jsem se věnoval konkurenčním implementacím s kterou se budu porovnávat.

V praktické části (kapitola 3) první navrhuji vhodné rozhraní, které je pro toto použití vhodné pro řídké a husté matice. Tím ušetřím čas při implementaci jednotlivých metod tím, že je stačí implementovat jednou a automaticky fungují pro řídké i husté matice. Poté implementuji metody první sekvenčně, poté paralelně.

V kapitole o testování zkouším ze začátku nějaké optimalizace. Hned poté v sekci 4.4 testuji jednotlivě dvě matice. Důkladné testování začíná až v následující sekci, kde měřím o kolik jsou rychlejší paralelní verze. Na konci porovnávám svojí implementaci s knihovnou `PETSc`.

Kapitola 1

Teoretická část

V této kapitole jsou ukázány formáty pro řídké matice. Dál je zadefinován problém a jsou představeny iterační metody a způsob výpočtu chyby. Představuji jednotlivé metody (Jacobiho, Gauss-Seidlovu a SOR metodu) spolu se zastavovacíma kritériema, které používám. Na konci jsou charakteristiky paralelního programování a ukázány možnosti, jak je možné optimalizovat implementaci.

1.1 Formáty řídkých matic

Pro ukládání řídkých matic se převážně používají 3 formáty. Ty jsou uvedené v obou publikacích [21] a [14]. Nejjednodušší je formát `COO`. Následuje formát `CSC` a formát `CSR`, kde první je vhodnější na sloupcové a druhý na řádkové přístupy.

Písmenem n značím řád matice (rozměr). Všechny matice budou čtvercové. Písmenem k značím počet nenulových hodnot.

1.1.1 COO

Formát `COO` (compressed coordinate list, přeloženo jako kompresovaný seznam souřadnic) je tvořen 3 poli o velikosti k(počet nenulových hodnot). Pole pro matici A označené A_{data} udává hodnoty a pole A_{rows} a A_{cols} udávají index pozice v poli. Pořadí prvků není definicí určeno, protože kdyby bylo důležité pořadí, tak je lepší použít další formáty. Vzhledem k tomu, že nemá pořadí, tak je náročné hledat různé hodnoty na stejné pozici, a proto je povoleno mít duplicitní záznamy, které se sečtou při převodu na jiný formát.

Formát není příliš vhodný na aritmetické operace, protože po nich může vzniknout množství duplicitních hodnot na totožných pozicích.

Pro matici A (prázdná místa představují hodnotu 0):

$$A = \begin{bmatrix} 10 & & & 8 & \\ & & 5 & & \\ & & & & 3 \\ 2 & 1 & & & \\ & & 7 & & \end{bmatrix}$$

Vypadají jednotlivá pole takto:

$$A_{data} = \begin{bmatrix} 10 & 5 & 8 & 3 & 2 & 1 & 7 \end{bmatrix}$$
$$A_{cols} = \begin{bmatrix} 1 & 3 & 4 & 5 & 1 & 2 & 3 \end{bmatrix}$$
$$A_{rows} = \begin{bmatrix} 1 & 2 & 1 & 3 & 4 & 4 & 5 \end{bmatrix}$$

Všechny tři formáty mají paměťovou složitost $O(k)$ při předpokladu, že mají více nenulových hodnot, než je řád matice $n > k$. Formát COO je nejnáročnější na místo z uvedených formátů pro matice.

Formát zabírá přinejmenším $n \cdot d + 2 \cdot k \cdot i$ `B` místa, kde d značí velikost nenulových hodnot (typicky 8 pro datový typ `double` nebo 4 pro datový typ `float`) a i značí velikost indexu (typicky 4 nebo 2).

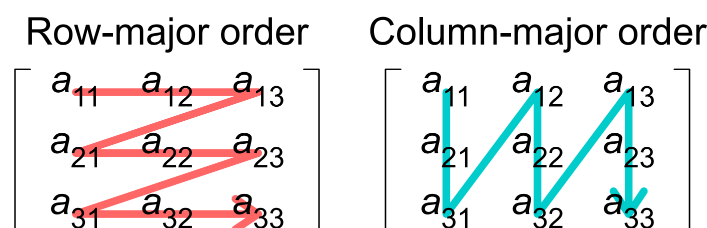
1.1.2 CSR

`CSR` (compressed sparse rows, přeloženo jako kompresované řídké řádky) je formát. Je také reprezentován 3 poli. Pole označená A_{data} a A_{cols} jsou shodné s formátem `COO` až na jeden rozdíl. Pole A_{cols} je seřazené po celých řádcích od shora dolů. Díky tomu si nemusíme pro každou hodnotu ukládat index řádků, ale stačí si pamatovat indexy, kde řádky začínají. To je v poli $A_{rowindex}$.

Pro matici A uvedenou v podkapitole 1.1.1 vypadají jednotlivá pole takto:

$$A_{data} = \begin{bmatrix} 10 & 8 & 5 & 3 & 2 & 1 & 7 \end{bmatrix}$$
$$A_{cols} = \begin{bmatrix} 1 & 4 & 3 & 5 & 1 & 2 & 3 \end{bmatrix}$$
$$A_{rowindex} = \begin{bmatrix} 1 & 3 & 4 & 5 & 8 \end{bmatrix}$$

Formát `CSR` zabírá)$n \cdot d + (n + k) \cdot i$. Je úspornější o $(k - n) \cdot i$ oproti formátu `COO`.



Obrázek 1.1: Ukázka porovnání CSR a CSC formátu. Napravo je naznačeno řazení CSR formátu a nalevo CSC. [4]

1.1.3 CSC

`CSC` formát je stejný jak `CSR` z podkapitoly 1.1.2, pouze transponovaný. Tj. uchovává indexy řádků a sloupce jsou po sobě.

1.2 Definice problému

Problém, který metody řeší, je nalezení řešení soustavy lineárních rovnic. To znamená, že pro zadanou matici A a vektor b, chceme najít vektor x pro který platí:

$$Ax = b \tag{1.1}$$

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Matice A je čtvercová matice o řádu n, vektory b a x jsou vektory řádu n. Pro všechny metody budu uvádět dva zápisy. První obecný přes matice, z kterého se vyvodí iterační matice. Druhý zápis, který je přes sumy, je bližší implementaci. V rovnici 1.2 je uveden zápis problému přes sumu.

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i \tag{1.2}$$

1.2.1 Chyba řešení

Správná chyba by měla tvar $||x - \hat{x}||$, kde $\hat{x}$ značí aproximované řešení a x značí skutečné řešení. Takovou ale neumíme využít, protože známe pouze

vektor $\hat{x}$, kdybychom znali vektor x, tak nemá smysl ho hledat. Proto pro určování chybovosti budeme používat rezidualní chybu. Reziduální chyba je vzdálenost zobrazeného x a pravé strany rovnice(b).

$$||b - A\hat{x}|| = \sqrt{\sum_{i=1}^{n} \left(b_i - \sum_{j=1}^{n} a_{i,j}\hat{x_j} \right)^2} \tag{1.3}$$

1.3 Iterační metody

Pro řešení soustav rovnic se používají přímé a aproximáční metody. Přímé metody najdou řešení během předem známého počtu kroků a paměti. Pro řídké matice mají většinou během nějaké fáze výpočtu v paměti matici, která není řídká, a proto pro ně mají velkou paměťovou i výpočetní náročnost. Klasickým představitelem je Gaussova eliminační metoda.

Druhou skupinou jsou iterativní metody, ty postupným opakováním kroků zkoušejí vylepšovat řešení do té doby, než je chyba přijatelná.

Já se v této práci zabývám statickými iteračními metodami. Tyto iterační metody mají tvar iterace podle rovnice 1.4. V této iteraci se matice G označuje iterační maticí a z její vlastnosti se odvozuje to, jestli nalezne řešení (konverguje).

$$x_{(k+1)} = Gx_{(k)} + f \tag{1.4}$$

1.3.1 Podmínky k nalezení řešení

Tvar rovnice metod je zajimavý pro zkoumání toho, jestli daná metoda konverguje k řešení. Pokud metoda najde řešení, tak x_* z rovnice 1.5 je hodnota, ke které x_k konverguje s přibývajícími iteracemi.

$$x_* = Gx_* + f \tag{1.5}$$

Odečtením ronice 1.5 od 1.4 dostaneme rovnici 1.6. Z té jde vidět, že pokud iterační matice se mocněním přibližuje k 0, tak hodnota $x_{(k)}$ se blíží k x_*.

$$x_{(k+1)} - x_* = G(x_{(k)} - x_*) = \cdots = G^{k+1}(x_{(0)} - x_*) \tag{1.6}$$

Iterační matice se mocněním blíží k 0, právě když platí $|\rho(G)| < 1$, kde ρ značí spektrální poloměr matice, což je největší vlastní číslo matice. [21].

Nalezení vlastního čísla matice je srovnatelně náročné s řešením přes iterativní metody, proto se tato podmínka nevyužívá. V publikaci [21] je

zmíněna lepší metoda na ověření, zda-li může konvergovat. Platí vzoreček $\rho(G) \leq ||G||$, kde $||G||$ značí libovolnou normu matice G, a proto stačí ověřit některé normy. Jednoduše vypočítatelná norma je Frobienova norma s následujícím vzorcem:

$$||G||_F = \sqrt{\sum_i \sum_j |g_{ij}|^2}$$

Do konvergujících matic patří všechny striktně diagonálně dominantní. Toto pravidlo je velmi striktní a pro naprostou většinu matic nedostačující. Striktně diagonální matice jsou dané následující podmínkou:

$$|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$$

V publikaci [21] je zmíněna věta, že pro symetrickou matici A konverguje SOR s parametrem ω mezi hodnotami 0 a 2 pro libovolnou počáteční volbu x_0 pouze tehdy, když A je pozitivně definitní. Tato věta platí i pro Gauss-Seidelovu.

1.4 Jacobiho metoda

Jacobiho metoda je nejjednodušší, pro výpočet další iterace $x^{(k+1)}$ stačí znát hodnoty předchozí iterace $x^{(k)}$. [14]

Matici A rozdělíme na diagonálu D a zbytek R, platí $A = D + R$. Poté iterativně řešíme podle rovnice 1.7.

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \tag{1.7}$$

$$D = \begin{bmatrix} a_{1,1} & 0 & 0 & \dots & 0 \\ 0 & a_{2,2} & 0 & \dots & 0 \\ 0 & 0 & a_{3,3} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{n,n} \end{bmatrix}, R = \begin{bmatrix} 0 & a_{1,2} & a_{1,3} & \dots & a_{1,n} \\ a_{2,1} & 0 & a_{2,3} & \dots & a_{2,n} \\ a_{3,1} & a_{3,2} & 0 & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & 0 \end{bmatrix}$$

Pro Jacobiho metodu je iterační matice rovna $-D^{-1}R$. Díky tomu, že D je jen diagonála, tak invertovat jí je jednoduché.

1.5 Gauss-Seidlova metoda

Gauss-Seidlova metoda je modifikace Jacobiho metody. Pro výpočet $x^{(k+1)}$ už nepoužíváme jen $x^{(k)}$, ale používá se k části výpočtu používat už aktulizované hodnoty $x^{(k+1)}$ z předchozích řádků. [14]

Matici rozdělíme na dolní trojúhelník včetně diagonály L_* a horní trojúhelníkovou matici bez diagonály U. Poté iterativně řešíme podle rovnice 1.8.

$$x^{(k+1)} = L_*^{-1}(b - Ux^{(k)}) \tag{1.8}$$

$$L_* = \begin{bmatrix} a_{1,1} & 0 & 0 & \dots & 0 \\ a_{2,1} & a_{2,2} & 0 & \dots & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & a_{n,n} \end{bmatrix}, U = \begin{bmatrix} 0 & a_{1,2} & a_{1,3} & \dots & a_{1,n} \\ 0 & 0 & a_{2,3} & \dots & a_{2,n} \\ 0 & 0 & 0 & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Pro Gauss-Seidlovu metodu je iterační matice $-L_*^{-1}U$.

1.6 SOR modifikace metody

Successive over-relaxation (rovnice 1.9) metoda zavádí koeficient ω. Tento koeficient slouží k zvětšení ($\omega > 1$) nebo zmenšení ($\omega < 1$) skoku oproti Gauss-Seidlově metodě a volí se v závislosti na konkrétní matici. Více o volbě ω je v podkapitole 1.3.1.

$$x^{(k+1)} = (D + \omega L)^{-1}(\omega b - [\omega U + (\omega - 1)D]x^{(k)}) \tag{1.9}$$

$$D = \begin{bmatrix} a_{1,1} & 0 & \dots & 0 \\ 0 & a_{2,2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{n,n} \end{bmatrix}, L = \begin{bmatrix} 0 & 0 & \dots & 0 \\ a_{2,1} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & 0 \end{bmatrix}, U = \begin{bmatrix} 0 & a_{1,2} & \dots & a_{1,n} \\ 0 & 0 & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix}$$

Iterační matice je $\Big(D+\omega L\Big)^{-1}\Big(\omega U+(\omega-1)D\Big)$. Při volbě $\omega = 1$ degraduje na Gauss-Seidelovu.

1.7 Zastavovací kritéria

Počítat celou chybu po každé iteraci by bylo zbytečně náročné, a proto je potřeba vymyslet lepší způsob. Jeden ze způsobů je vzít vzdálenost kroku, který udělala metoda během poslední iterace, a jakmile se přestane zlepšovat dostatečně rychle, tak výpočet zastavím. Vzdálenost kroku můžeme lehce vypočítat pomocí normy rozdílu vektorů daný vzorcem: $||x^{(k)} - x^{(k+1)}||$. Ten poslouží jako zastavovací kritério pro GS a SOR.

Pro Jacobiho metodu můžu využít samotný algoritmus metody a s malou změnou vypočítám chybu předcházející iterace. Z rovnice 1.7 vezmu výraz $b - Rx^{(k)}$, který znormuji a tím získám chybu.

1.8 Paralelní programování

Paralelní programování je stále aktuálnější programovací disciplína. Předpověď Gordona Moora (jeden ze zakladatelů Intelu) v roce 1975, že počet tranzistorů integrovaných v jednom čipu se zdvojnásobí každý rok, ale už přestává platit. Od dostatečně malé velikosti hradel začíná převažovat kvantová mechanika oproti klasické, a proto se začíná zastavovat jejich zmenšování. V kvantové mechanice probíhá kvantové tunelování, které je neslučitelné s klasickým fungováním hradla. Pro platnost zákona by se musely zvětšovat čipy, které by spotřebovaly více elektřiny a navíc jsou drahé. [13]

Dříve procesory zrychlovaly tím, že se zvyšovala frekvence jednoho jádra, bohužel u Pentium 4 zjistili, že přílišná frekvence a rozdělení instrukcí na miniinstrukce není výhodné. [18] Proto aktuální procesory směřují k většímu počtu výpočetních jader. Grafické karty jsou extrémním příkladem paralelního procesoru, kde každá skupinka pixelů může být počítána na vlastním jádru. Nejnovější GPU mají až 4600 specializovaných výpočetních jader. [8]

1.8.1 Amdahlův zákon

Amdahlův zákon udává maximální zrychlení při rozdělení výpočtu na několik jader. Paralelizovatelná část programu je p, počet jader je k. Zrychlení programu oproti sekvenční verzi je funkce S s parametrem k, který udává počet jader. Zajimavá vlastnost je její limita, stačí mít 5% neparalelizovatelného kódu a maximální zrychlení je 20×. [1]

$$S(k) = \frac{1}{1 - p + \frac{p}{k}}$$

$$\lim_{k->\infty} \frac{1}{1-p}$$

1.8.2 Nástroje paralelního programování

Vzájemné vyloučení (mutex) je synchronizační prostředek k zabránění vykonávání operací více vlákny nad jedním prostředkem (proměnná, vypisování do souboru nebo konzole apod.) najednou. Současným vykonáváním operací nad jedním prostředkem může lehce dojít k nekonzistenci dat.

Kritická sekce je podobná mutexu, pouze se zamyká určitá část kódu místo prostředku. Pro svoje fungování potřebuje nějaké synchronizační primitivum, které může zastávat právě mutex.

Bariéra je synchronizační prostředek, přes které může pokračovat pouze celá skupina vláken, která pracuje na jedné části algoritmu. První vlákno (a všechny další), které dorazí k bariéře, musí počkat na poslední vlákno, než dodělá předcházející práci, a tím překonají bariéru zároveň.

1.8.3 Úskalí paralelního programování

Paralelní programování má pár specifických úskalí, se kterými se v klasickém sekvenčním programování není možné potkat.

Prvním takovým úskalím je uváznutí (Deadlock). Deadlock je čiště chyba programu. Je to stav, když dvě vlákna pasivně čekají na uvolnění stejného prostředku, co má druhé vlákno, a ani jedno ho neumí uvolnit tomu druhému. Stejná situace je livelock, to je aktivní čekání způsobené stejnou chybou, pouze místo nic nedělání zbytečně zatěžuje procesor.

Souběh (Race condition) je souběžný zápis a čtení paměti, kde se jedna operace nedokončí a paměť uvázne v nesprávném stavu. Proto je nutné proměnné zamykat a nebo zapisovat přes atomické operace.

Falešné sdílení (False sharing) je stav, kdy různá jádra procesoru sdílí stejný paměťový prostor, do kterého zapisují, a navzájem si ho zneplatňují, a proto nastávají nadbytečné přesuny mezipaměťových řádků (cache line) mezi jádry.

1.9 Paralelizace iterativních metod

1.9.1 Jacobiho metoda

Paralelizace Jacobiho metody je jednoduchá. Na začátku iterace potřebuje algoritmus znát celý vektor $x^{(k-1)}$ a s tím si vystačí každé vlákno po celou dobu iterace. Jediné, co je potřeba synchronizovat, jsou přechody mezi

jednotlivými iteracemi. Tj. aby začátek nové iterace na libovolném vlákně proběhl až po konci iterace na všech ostatních vláknech, to je synchronizace pomocí bariéry.

1.9.2 Gauss-Seidelova a SOR metoda

Paralelizace GS je hodně náročná. Pro plnou paralelizaci je potřeba nějak ověřit, že jiné vlákno už připravilo novou hodnotu $x_j^{(k)}$ pro aktuální iteraci u $j < i$, kde i značí index aktuálně počítaného čísla. Alternativně je možné iteraci rozložit na dvě poditerace a první vypočítat část $b - Ux^{(k)}$ a až jako druhý krok provést $D^{-1}(b - Ux^{(k)})$. Nicméně to by zvýšilo cenu synchronizace, protože by byly potřeba 2 bariéry. Případně se používají techniky, kde se počítá víc iterací najednou. Já jsem zvolil jiný způsob. Výpočet budu provádět asynchronně, tj. nebudu čekat na výpočet celého předchozího $x^{(k)}$, jako to vyžaduje definice, ale použiji pouze co největší část $x^{(k)}$, co je aktuálně k dispozici (best effort taktika).

1.10 Možnosti optimilizace

1.10.1 Falešné sdílení

Falešné sdílení nastává, když do jedné cache line zapisuje víc vláken najednou. Aktuální procesory mají cache line o velikosti 64 bytů, tedy 8 čísel typu `double` nebo 16 čísel typu `int`. V algoritmech se zapisuje akorát do vektoru x a do pomocných proměnných. Stačí, aby algoritmus nebyl prokládaný, tedy aby se vlákna nestřídala a v počítání vektoru x po každém řádku a problém falešného sdílení by neměl nastat. Zároveň jednotlivá vlákna budou sdílet co nejméně proměnných, do kterých se zapisuje.

Pořád bude nastávat skutečné sdílení, když mezi tím, než znovu stihne zapsat vlákno do jedné cache line, tak jiné vlákno ji bude chtít přečíst. Tomu je náročné až nemožné předejít.

1.10.2 Nerovnoměrná zátěž

Nerovnoměrná zátěž by nastala, kdyby jedno vlákno mělo na výpočet mnohem delší řádky než ostatní vlákna. U plně vyplněných matic tento problém nemůže vznikat, ale počítám i s řídkými maticemi. Předejdu problému tak, že budu řídké matice rozdělovat po souvislých řádcích pro jednotlivá vlákna tak, aby v každé skupině řádků bylo podobné množství nenulových hodnot.

1.10.3 Pomalé dělení

V dokumentu [7] popisující instrukce procesorů Intel se uvádí propustnost a zpoždění jednotlivých instrukcí. Konkrétně budu porovnávat instrukce `mulsd` a `divsd`, které slouží k násobení repektivně dělení čísla v přesnosti `double`. Budu brát v úvahu architekturu `Skylake`, které mají uvedené nejlepší hodnoty ve pospěch dělení. V dokumentu se uvádí, že zpoždění `mulsd` jsou 3 cykly procesoru oproti 14 cyklům u `divsd`. Rozdíl v propustnosti je ještě větší ve prospěch `mulsd`, a to dokonce 8× ve prospěch násobení.

Jediné dělení, které se v metodách provádí, je dělení čísly na diagonále. Pokud budu ukládat inverzi diagonály, tak můžu provádět násobení místo dělení, a tím zrychlím algoritmus převážně na velmi řídkých maticích.

1.10.4 Asynchronnost

Náročná operace pro procesor je synchronizace bariérou. Mohlo by se vyplatit méně synchronizovat vlákna mezi sebou tak, aby neprobíhala každou iteraci, ale dal se tento parametr volit. Prerekvizita je rozdělit množství práce na jednotlivá vlákna rovnoměrně, což je obsahem podkapitoli 1.10.2.

Kapitola 2

Použité technologie

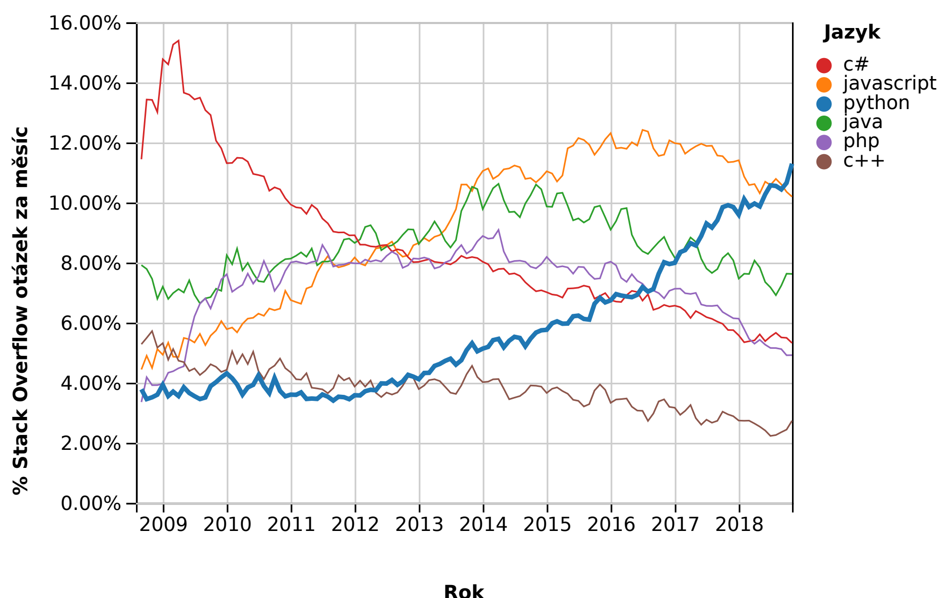
V této kapitole popíšu jednotlivé použité technologie. Jako základní jazyk byl zvolen `C/C++`, který je de facto standard pro nízkoúrovňový a efektivní aplikace. Použití knihovny bude prostřednictvím jazyka `Python`. To je interpretovaný jazyk, který je pomalejší než `C/C++`. Výpočet metod bude probíhat v `C/C++`. Prostředí jazyka `Python` bude zajišťovat nahrávání matic a nastavování parametrů metod.

2.1 OpenMP

`OpenMP` je knihovna se soustavou direktiv pro překladač, která umožňuje jednoduché paralelní programování v `C/C++`. Prostředky paralelního programování jsou dostupné přes jednořádkový příkaz *#pragma omp*. Například použití bariéry je v `OpenMP` jednořádkové s volitelným pojmenováním, zatímco v některých programovacích jazycích je nutné je simulovat. Podobně jednoduše umožňuje použití všech paralelních konstruktů. [15]

`OpenMP` obsahuje bohaté nastavení paralelizace smyček. Při nastavení `dynamic` plánovač volí za běhu, na kterém vláknu poběží jednotlivé iterace. Nevýhoda je, že se můžou při příliš rychlém iterování blokovat při dotazu na plánovač a pozdější vlákno skončí dočasně ve spánkovém módu. Tomu se předchází nastavením většího `chunk-size`, které určuje počet iterací, než se znova zeptá plánovače.

Plánování `static` předem určí rozsahy pro jednotlivá vlákna. Tím se vyloučí blokování plánovače, protože ho vlákna nepotřebují vůbec. Nastavením parametru `chunk-size` se dá změnit velikost rozdělovaných bloků. Výchozí nastavení je $\frac{\text{délka cyklu}}{\text{počet vláken}}$, které rozdělí do souvislých bloků, a každý



Obrázek 2.1: Porovnání jednotlivých jazyků na StackOverflow. [19]

blok má za úkol vlastní vlákno. Nastavení `static` je nevýhodné, když jednotlivé iterace trvají rozdílnou dobu. [20]

Nastavení `guided` se snaží omezit dotazy na plánovač, ale zároveň předchází vyhladovění práce. Ze začátku rozděluje větší bloky, aby omezil množství dotazů na plánovač. Na konci rozděluje menší bloky pro vlákna, která stihla větší bloky dokončit rychle. Nastavení `chunk-size` omezuje velikost nejmenšího bloku.

2.2 Python

`Python` je interpretovaný, vyšší programovací jazyk z roku 1991. Jeho síla je vyšší abstrakce a rychlé proto-typování, ale není příliš rychlý tam, kde je kritická výkonnost. Více o rychlosti `Python` je v článku [16], kde vychází kód v čistém `Python` (tím myslím bez externích knihoven) oproti implementaci `C` více jak 1000 pomalejší během maticové operace. Ve své výchozí implementaci `CPython` má dobré `API` a vzniká pro něj spousta efektivních nativních knihoven. V těchto knihovnách slouží `Python` pouze pro vysokoúrovňové řízení programů a výpočet probíhá ve výkonném kódu `C` a případně dalších programovacích jazyků. Dneska se používá primárně tam, kde buď rychlost není až tak důležitá a nebo se používají nativní knihovny.

Na Stackoverflow si všimli jazyka Python jako nejrychleji rostoucího hlavního programovacího jazyka ve státech vysokého příjmu (podle definice Světové banky. Náleží mezi ně i Česká republika od roku 2006). [5]

Mezi omezení Python patří nemožnost plné paralelizace. Python prostředí má jeden velký zámek, který může mít maximálně jedno vlákno, ve kterém běží kód, co manipuluje s Python objekty najednou. Pro moje použití není vhodné tento zámek uvolňovat, protože manipuluji s polem, které spravuje Python.

2.3 Cython

Cython je nadstavba Python, která může zkompilovat většinu kódu jazyka Python a k tomu přidává vlastní konstrukce, které jsou rychlejší [3]. Funguje tak, že vezme svůj kód, ten zkompiluje do C kódu využívající CPython API, které svým voláním imituje výsledek chování původního Python kód. Více o Cython ve zdroji [3].

Kouzlo je v tom, že některé svoje konstrukce zkompiluje do běžného C kódu a ušetří se volání CPython API. Když se tímto způsobem optimalizuje smyčka, která zabere nejvíce času, tak se dosáhne na smyčce efektivita kompilovaného kódu. Cython navíc umí přímo využívat zároveň prostředky jazyka C/C++.

2.4 Představení ostatních implementací

Konkurencí pro mě jsou numerické systémy jako MATLAB a NumPy. Nicméně naivní implementace v těchto systémech by mně neměla konkurovat v rychlosti, a proto jsem pro vhodné porovnání spíš hledal knihovnu implementovanou v nízkoúrovňovém jazyce. Našel jsem dva vhodné představitele.

První je DLAP. Knihovna je implementovaná v jazyce Fortran. Fortran je jazyk z poloviny 20. století, který se dnes už používá jen na specifické úlohy. Mezi tyto úlohy patří termodynamické simulace a je to jeden z používaných jazyků pro superpočítače. Já Fortran neovládám a DLAP má implementovanou pouze sekvenční verzi algoritmů. [17]

Další je PETSc. Knihovna je implementovaná v jazyce C. PETSc má podle webu implementovanou Jacobiho a SOR metodu. Gauss-Seidelova metoda je dosažitelná přes SOR metodu s $\omega = 1$. Jacobiho metodu je možné použít paralelně přes OpenMPI. SOR metoda je bohužel pouze sekvenční. [2]

Přímo na stránkách knihovny PETSc zmiňují Python navázání pojmenované PETSc4py. To jsem zvolil používat pro moje testování. PETSc4py je

jednoduché navázání napojené přímo na C rozhraní a při nesprávném použití jsem dosáhl `segmantation fault`, při kterých se toho moc nedozvím. Tohle není typické chování knihoven pro Python. [9]

Jako konkurenční implementaci budu zkoušet pouze sekvenční Gauss-Seidel a SOR z knihovny `PETSc` použité z `Python` přes `PETSc4py`.

Kapitola 3

Praktická část

Implementace algoritmů bude probíhat v `C/C++` a implementace programového rozhraní v jazyce `Cython`. V první kapitole ukážu, jak se dají použít libovolné matice v jazyce `Python`. V druhé představím rozhraní a co vše se tam dá nastavovat. V dalších kapitolách popíšu, jak moje implementace fungují uvnitř a jak jsem postupoval.

3.1 Použití matic v jazyce Python

Pro husté matice je potřeba mít balíček `NumPy`. Pro práci s řídkými maticemi se v `Python` používá balíček `SciPy`. `Python` umožňuje matice zadávat přímo, případně načítat ze souboru v Matrix Market formátu (`.mtx`). Následující matice bude použita pro ukázky.

$$A = \begin{bmatrix} 13 & & 3 & 6 & \\ & 11 & -5 & & 4 \\ & -1 & 7 & & \\ & -5 & & 8 & 2 \\ 3 & 3 & & & 9 \end{bmatrix}$$

Pro zadání matice a její interpretaci jako hustou matici je možné použít konstruktor. Případně má `NumPy` svoje metody pro načítání hustých matic ze souborů. Je možné zadat řídkou matici s argumentem husté matice, případně je možné použít přímo pole z definice (indexovanými od 0):

```
matrix = scipy.sparse.csr_matrix((
    [13,3,6,11,-5,4,-1,7,-5,8,2,3,3,9],
    [0,2,3,1,2,4,1,2,1,3,4,0,1,4],
    [0,3,6,8,11,14]))
```

V modulu `scipy.io` jsou funkce pro načítání matic. Na SuiteSparse Matrix Collection [11] jsou matice dostupné ve 3 formátech a všechny zvládne SciPy načíst.Funkce `scipy.io.mmread` slouží k načtení Matrix Market formátu. [6]

Pro načtení stažené matice stačí napsat následující kód:

```
mymatrix = scipy.io.mmread('jmenomatice.mtx')
```

3.2 Definice rozhraní knihovny

Cílem mého rozhraní je jednoduché použití. Proto celé řídí jediná třída, která se jmenuje `Solver`. Ta se inicializuje podle parametrů konstruktoru a po ní už má pouze omezené možnosti, jak měnit svoje vlastnosti. Následuje výčet parametrů konstruktoru. Vzhledem k jednoduchosti jsou všechny nepovinné až na matici.

matrix *povinný parametr* Udává matici, nad kterou je prováděn algoritmus. Může být 2 rozměrné pole nebo řídká reprezentace skrz nějakou třídu ze `scipy.sparse`.

method *volitelný parametr* Název metody, která má být použita na řešení. Metody jsou vysvětlené v kapitole 1. Při nezadání se zvolí automaticky Gauss-Seidel když $\omega = 1$, jinak SOR. Možné hodnoty jsou: `'jacobi'`, `'gaussseidel'` a `'sor'`.

threads *volitelný parametr* Počet vláken použitých na výpočet. Při nezadání se použije počet vláken systému.

threshold *volitelný parametr* U Jacobiho metody značí hranici chyby, po které se zastaví výpočet. U ostatních metod značí hranici rozdílu jednotlivých iterací. Když nastane $||x^{(k+1)} - x^{(k)}|| < \texttt{threshold}$, tak skončí výpočet a aktuální x se vrátí jako řešení. Při nezadání se zvolí 0 (skončí se při dosáhnutí maximálního počtu iterací).

iterations *volitelný parametr* Zadaný počet iterací algoritmu. Jakmile by se překročil počet, tak se vrátí aktuální výsledek bez ohledu na správnost. Při nezadání se použije neomezený počet iterací.

omega *volitelný parametr* Hodnota ω pro algoritmus SOR. Na jiném algoritmu nemá žádný vliv. Při nezadání se použije neutrální hodnota 1.

asyncskips *volitelný parametr* Parametr pro asynchronní Gauss-Seidelovu a SOR metodu, který určuje počet iterací bez synchronizace a bez ověřování podmínek konvergence.

Metody třídy `Solver` jsou následující:

solve Metoda udělá výpočet podle parametrů zadaných v konstruktoru. Parametry jsou vektor b (povinný) a případně vektor x (nepovinný, při nezadání se použije automaticky vytvořený nulový vektor). Vektor x se při zadání modifikuje na místě (zadaný vektor x se změní).

setIterations Metoda změní `iterations` podle parametru.

setThreshold Metoda nastaví `threshold`.

setOmega Metoda nastaví parametr omega.

Třída `Solver` má jedinou podstatnou metodu. Tou je `solve`. Metoda bere 1 až 2 argumenty. První argument je povinný a je to jednorozměrné pole představující pravou stranu rovnice. Druhý argument je nepovinný a je to počáteční x z rovnice. Při nezadání se použije nulový vektor. Vrací nalezený vektor x a chybu, jak je zadefinovaná v teoretické části ve dvojcig. Následuje příklad použití.

Jednoduchost rozhraní je možné vidět v příkladu použití. Stačí pouze 5 řádků a provede se výpočet. Stejným způsobem se používají všechny metody, takže pro jakoukoliv metodu stačí 5 řádků. Uživatel se tak může víc soustředit na jeho použití než na to, jak se používá knihovna.

```
from iterativesolvers import Solver
matrix = scipy.io.mmread('matrix/494_bus.mtx')
b = np.ones(mat.shape[0])  # vytvoří pole jedniček
solver = Solver(mat, method='jacobi', threads=1, iterations=99)
x, err = solver.solve(b)
```

Vnitřně funguje tak, že přijme parametry, na základě toho zvolí vhodnou třídu, která dědí z `AbstractSolver` a tu si pamatuje. Při volání `solve` zavolá metodu `solve`, která podle parametrů najde řešení a vrátí výsledek. `AbstractSolver` je jednoduchá třída, která neumí nic kromě zapamatování parametrů algoritmu, nicméně má neimplementovanou virtuální třídu `solve` a tu implementují její potomci představující jednotlivé algoritmy.

3.3 Vnitřní reprezentace matic

Je několik způsobů, jak několik tříd sjednotit se stejným chováním. Nejvíce přímočará je přes mateřskou abstraktní třídu a virtuální metody. To používám na `AbstractSolver`. Každá podtřída má pak vlastní tabulku virtuálních metod a instance třídy má ve svojí struktuře adresu na tuto tabulku. Implementovat funkci používající abstraktní třídu jako parametr je přímočaré, ale abstraktní třída matic by zabránila kompilátoru optimalizovat krátké funkce a bylo by potřeba vytvořit abstraktní třídy i pro iterátory.

Další možností jsou šablony (template). To naprogramuji 2 třídy zvlášť, kde jedna bude reprezentovat husté matice a další řídké matice. Když budou mít stejné rozhraní, tak je možné naprogramovat funkci, která jako argument šablony bude brát jednu z třídy matice a vygenerují se tím implementace pro hustou nebo řídkou matici podle potřeby automaticky (nebo se použije už vygenerovaná).

Moje požadavky na matice byly malé, tak umí pouze iterovat po řádcích a v řádcích po jednotlivých hodnotách a umí zjistit, kde se nachází. Paměť nad poli v mém případě spravuje `Python` a `NumPy`, proto moje třídy matic slouží jen jako obalové třídy pro lehčí procházení jednotlivých hodnot.

Implementace matic umožnuje využívat jiný datový typ, než `double`, ale třída `AbstractSolver` změnu neumožňuje, takže je možné využívat pouze `double`.

3.4 Sekvenční implementace

V této podkapitole postupně popisuji implementaci sekvenční verze jednotlivých metod popsaných v kapitole 1.

3.4.1 Jacobiho metoda

Rovnice 3.1 je Jacobiho metoda, která vznikne vyjádřením řádku z rovnice 1.7 s definicí maticového násobení. Implementovat metodu pomocí tohoto vzorce je přímočaré.

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(k)}\right) \tag{3.1}$$

Jedna z optimalizací, kterou jsem během této metody uplatnil, je prohazování rolí vektoru $x^{(k)}$ a $x^{(k+1)}$, díky které se vyhnu jednomu kopírování vektoru x během každé iterace, místo toho stačí prohodit pouze 2 ukazatele v paměti.

Při sekvenční implementaci není potřeba optimalizovat nerovnoměrnou zátěž. Dělení se dá vyhnout, protože jediný člen, kterým se dělí je diagonála a tak je možné uložit inverzi diagonály a tu použít místo dělení.

3.4.2 Gauss-Seidelova metoda

Sekvenční implementace GS probíhá podle vzorce 3.2.

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}\Big(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(k)}\Big) \tag{3.2}$$

Optimalizace při sekvenční verzi jsou shodné s Jacobiho metodou, pouze si vystačím pouze s jedním vektorem x.

3.4.3 SOR metoda

Poslední metoda SOR je příbuzná Gauss-Seidelově. V podstatě sdílí celý kód s Gauss-Seidelem, pouze dosazení do x je modifikováno koeficientem ω. Hodnota $||x^{(k)} - x^{(k+1)}||$ je ovlivněná omegou, protože tahle metoda mění velikost skoku oproti Gauss-Seidelovi.

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{i,i}}\Big(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(k)}\Big) \tag{3.3}$$

3.5 Paralelní implementace

3.5.1 Jacobiho metoda

Jacobiho metodu jsem implementoval první. Je tam provedena optimalizace na spravedlivé rozdělení, ale není asynchronní. Problém, kvůli kterému není asynchronní je, že má 2 vektory x a je potřeba prohazovat jejich role. Předpokladem pro většinu optimilizací je sloučení těchto 2 vektorů x, ale tím ztratíme možnost počítat přesnou chybu za běhu a navíc by se metoda začala chovat jak asynchronní Gauss-Seidel, proto si další optimalizace nechám až na příští metody.

3.5.2 Asynchronní Gauss-Seidelova a SOR metoda

Dále jsem implementoval Gauss-Seidelovu metodu. Vzorec Gauss-Seidlovy metody je zadefinovaný sekvenčně. Pro paralelní prostředí je nevýhodné,

takže budu programovat metodu, který *vlastně není Gauss-Seidel*, ale pouze se jí blíží svým chováním. Ze vzorce z kapitoly 1.5 je vidět, jak funguje Gauss-Seidelova metoda. Metoda vezme co nejvíce hodnot z aktuální iterace a tam, kde zatím nemá hodnoty aktuální iterace použije hodnotu z minulé iterace.

Asynchronnost je ta vlastnost, že nečeká na vypočtení části $x^{(k+1)}$, na které by měla podle definice, ale snaží se co nejvíce pokročit ve výsledku i za cenu horší konvergence.

Paralelní verze použije všechny hodnoty z aktuální iterace, co má k dispozici (nejen předchozí, ale pokud nějaké vlákno vypočítalo nějakou hodnotu, co je víc vepředu, tak ji také využije) a ostatní doplní hodnotami z minulé iterace. Vzhledem k tomu, že výpočet probíhá paralelně, tak využije méně hodnot z aktuální iterace než regulérní Gauss-Seidlova metoda. Proto konverguje pomaleji, nicméně výpočet probíhá paralelně a tedy rychleji.

Asynchronní SOR přebírá všechny vlastnosti asynchronního Gauss-Seidelovu. Pouze při dosazení závěrečné hodnoty proběhne modifikace x po dle parametru ω.

3.6 PETSc4py

Knihovna se soustředí na pokročilé metody (především na KSP, Krylov subspace methods) a mé vybrané metody se tam obvykle používají jen na před-počítávání (preconditioner) optimálního začátku pro metody, na které se soustředí. SOR metodu se mě povedlo použít, bohužel je implementovaná jen sekvenčně. Jacobiho metodou se chlubí na stránkách, ale po prozkoumání zdrojových kódů zjišťuji, že tam je pouze zjednodušená Jacobiho metoda. Implementovaná tam je jen první iterace s nulovým počátečním vektorem x, alespoň pro řídké matice. Je irelevantní porovnávat rychlost, protože tam je implementovaná tak, že se před-počítá diagonála a první iterace se poté provede jedním vynásobením vektorů navzájem.

KAPITOLA 4

Měření výkonu a testování

Prvotně měřím metody na jednotlivých maticích. Poté měřím několik matic najednou a zjišťuji, jak hodně paralelizace zvyšuje výkon. Na konci je porovnání s implementací v knihovně `PETSc`.

4.1 Testovací stroj

Měření paralelních testů bude probíhat na školním serveru:

GCC gcc version 8.3.0

PYTHON Python 2.7.5

CPU 2× E5-2620 v2 22`nm` 2,2 - 2,6 `GHz`

CORES 2 × 6 jader, 2 × 6 vláken, HyperThreading je deaktivován

CACHE 15`MB` `L3`, 2 × 6× 256`kB` `L2`, 2 × 6× 64`kB` `L1`

Ostatní parametry by měly být irelevantní. Teoretické zrychlení by se mohlo blížit u paralelní verze 12×, když zanedbám komunikaci mezi jádry a procesory.

Název	řád	nenulové hodnoty	velikost v paměti
494_bus	494	1 666	22 kB
bcsstk21	3 600	26 600	333 kB
nasa1824	1 824	39 208	477 kB
bodyy4	17 546	121 938	1,5 MB
bodyy5	18 589	129 281	1,6 MB
boddy6	19 366	134 748	1,7 MB
obstclae	40 000	197 608	2,5 MB
pdb1HYS	36 417	4 344 765	52 MB

Tabulka 4.1: Použité matic se základníma vlastnostma.

4.2 Matice pro měření a metodika

Matice jsou uvedené v tabulce 4.1. Všechny matice jsem stahoval z SuiteSparse Matrix Collection (zdroj: [10]).Všechny časy měření jsou 10× změřené a je uvedený průměr všech měření.

4.3 Testování optimilizací

V této podkapitole proběhne vyhodnocení možností k optimalizaci přednesených v podkapitole 1.10.

4.3.1 Optimalizace nerovnoměrné zátěže

Otestoval jsem matici `494_bus`, na které jsem provedl 100 iterací a tento pokus jsem provedl 1000× a s aktivovaným lepším rozdělením Jacobiho metoda počítala o 17% rychleji.

4.3.2 Optimalizace pomalého dělení

Na matici `494_bus` udělala optimalizace pomalého dělení zrychlení o 4,5%. Je to částečně tím, že je matice malá a na větších maticích to udělá pravděpodobně menší rozdíl, protože na každý řádek se provádí pouze jedno dělení v neoptimalizovaný verzi.

4.3.3 Optimalizace asynchronnosti

Asynchronnost ovlivňuje parametr `asyncskips`, ten zkusím různě nastavovat a budu porovnávat změny v čase a chybě. V tomto testu zafixuji počet

Název matice	**algoritmus**	`asyncskips`	**čas[s]**	**chyba**
`nasa1824`	Gauss-Seidel	sekvenční verze		8,163
		1	0,405	8,701
		2	0,383	9,009
		5	0,363	9,067
		40	0,345	9,307
`bcsstk21`	SOR $\omega = 1{,}9$	sekvenční verze		$2{,}03 \cdot 10^{-9}$
		1	0,172	$3{,}178 \cdot 10^{-9}$
		2	0,158	$1015 \cdot 10^{-9}$
		5	0,166	$129222 \cdot 10^{-9}$
`494_bus`	Gauss-Seidel	sekvenční verze		78,80
		1	0,145	80,11
		2	0,124	80,51
		5	0,112	79,99
		10	0,108	80,24
`bodyy6`	SOR $\omega = 1{,}9$	sekvenční verze		$2{,}97 \cdot 10^{-12}$
		1	0,500	$3{,}23 \cdot 10^{-12}$
		2	0,489	$14{,}56 \cdot 10^{-6}$
		5	0,476	$0{,}14 \cdot 10^{-6}$
		40	0,471	$4{,}44 \cdot 10^{-12}$

Tabulka 4.2: Porovnání zhoršení konvergence v asynchronní verzi.

iterací na 1 000. Výsledek je v tabulce 4.2. Pro malou matici `494_bus` parametr urychlil metodu s tím, že chyba se skoro nezměnila. Pro větší matici `bcsstk21` se už při nepatrném zvýšení parametru zvětšila chyba o tolik, že nedává smysl ho nastavovat na víc, než 1.

4.4 Porovnání rychlosti běhu a konvergence metod na jedné matici

Pro spoustu velkých matic Jacobiho metoda divergovala, ale Gauss-Seidlova už konvergovala. Pro porovnání zrychlení jednotlivých matic jsem zvolil matici `obstclae`, u které konvergovala i Jacobiho metoda. Můžu tím porovnat zrychlení u Jacobiho oproti Gauss-Seidelovi.

Následuje tabulka počtu iterací a časů, které byly potřeba ke konvergenci, zde zvolenou jako chybě menší než 10^{-12}. Tato hodnota byla zvolena, protože pro menší hodnoty se nějaké metody zasekly a vlivem chyby výpočtu se přestalo řešení zlepšovat. Matice je netypická tím, že na ní Jacobiho

Algoritmus	sekvenční verze iterací	sekvenční verze čas [ms]	paralelní verze iterací	paralelní verze čas [ms]	**parelelní zrychlení**
Jacobi	676	241,9	676	34,6	6,97
Gauss-Seidel	345	140,5	345	23,4	6,02
SOR $\omega = 1{,}5$	116	64,4	116	10,2	6,33

Tabulka 4.3: Rychlost konvergence na matici obstclae.

metoda konverguje skutečně rychle.

4.5 Testování na několika maticích

Následně jsem naprogramoval test, který hledá optimální počet iterací a následně měří čas, který trvá dosažení toho počtu iterací. Překvapilo mě, že matice o kterých jsem si myslel, že divergují ve výsledku po dostatku iteracích nakonec přesto konvergovali. Parametr ω u SOR metody jsem volil z rozmezí 1,1 až 1,9 odkrokovaně po 0,1. Zvolil jsem takovou hodnotu, která trvala nejméně iterací. Hledám takové řešení, které má menší chybu než 10^{-9}. Testoval jsem sekvenční a paralelní verze Jacobiho, asynchronní Gauss-Seidelovu a SOR metodu.

Cílem tohoto porovnání je nejen porovnat algoritmy mezi sebou, ale hlavně porovnat zrychlení paralelních verzí. Použil jsem matice bcsstk21, `bodyy4`, `bodyy5`, `bodyy6`, `nasa1824` a `obstclae` ze stránky SuiteSparse Matrix Collection a zprůměroval jejich výsledek. Několik matic je vypsaných speciálně, ale všechny matice by zabraly příliš místa.

Z tabulky 4.4 je vidět, že zrychlení paralelní verze Jacobiho je o dost vyšší. Vlákna si nezneplatnují vektor x, v cache paměti a tak tam není tak vysoká nadbytečná synchronizace. Asynchronní Gauss-Seidelova metoda musela průměrně vykonat o 1,03% víc iterací, asynchronní SOR o 1,26%. To mně přijde v pořádku, když vezmeme v úvahu, že se celý výpočet zrychlí.

Název matice	**sekvenční verze**		**asynchronní verze**		**paralelní**
Algoritmus	**iterací**	**čas [s]**	**iterací**	**čas [s]**	**zrychlení**
`bcsstk21`					
Jacobi	367 200	15,05	367 200	6,23	2,41
Gauss-Seidel	183 080	7,82	183 650	3,05	2,56
SOR $\omega = 1{,}9$	10 290	0,49	10 480	0,18	2,74
`bodyy4`					
Jacobi	4 970	1,08	4 970	0,18	6,05
Gauss-Seidel	2 080	0,48	2 090	0,09	5,13
SOR $\omega = 1{,}8$	190	0,055	200	0,010	5,28
`bodyy5`					
Jacobi	32 450	7,27	32 450	1,12	6,48
Gauss-Seidel	13 530	3,21	13 620	0,59	5,48
SOR $\omega = 1{,}9$	670	0,194	690	0,036	5,43
`bodyy6`					
Jacobi	221 080	50,33	221 080	8,35	6,03
Gauss-Seidel	91 430	21,98	91 700	4,49	5,49
SOR $\omega = 1{,}9$	5 160	1,54	5 270	0,26	5,78
`nasa1824`					
Jacobi	metoda diverguje				–
Gauss-Seidel	321 580	16,54	328 860	6,76	2,45
SOR $\omega = 1{,}8$	35 780	1,95	37 520	0,79	2,46
`obstclae`					
Jacobi	532	0,190	532	0,28	6,75
Gauss-Seidel	272	0,110	272	0,019	5,98
SOR $\omega = 1{,}6$	88	0,0491	88	0,0086	5,72
Průměrný výsledek ze 7 matic					
Jacobi	-	14,78	-	3,23	5,54
Gauss-Seidel	-	8,36	-	2,92	4,52
SOR	-	0,71	-	0,21	4,56

Tabulka 4.4: Porovnání rychlosti konvergence na několika maticích.

Název matice **Algoritmus**	**sekvenční verze** **čas na iteraci**	**paralelní verze** **čas na iteraci**	**parelelní** **zrychlení**
pdb1HYS			
Jacobi	6,748 ms/it	1,171 ms/it	5,764
Gauss-Seidel	5,911 ms/it	1,631 ms/it	3,623

Tabulka 4.5: Zrychlení na velké matici.

4.6 Testování velké matic

Pro dostatečně rozlehlé matice, které se nevejdou do `L3` cache by mohla být nejpomalejší část výpočtu samotný přenos dat z RAM paměti. Pro testování použiji matici pdb1HYS z MatrixMarket, která v paměti zabere 52`MB`, teda nemůže se vejít do cache paměti. Metody pro uvedenou matici nekonvergují, tak budu porovnávat rychlost samotné iterace. Zrychlení u Jacobiho metody je 5,8 a u Gauss-Seidelovu 3,6. Z toho nejde poznat, jestli je zpomalení způsobený tím, že se nevejde do cache nebo něčím jiným. Podrobnější výsledky jsou v tabulce 4.5.

Název matice	**počet**	**čas[s]**	
Algoritmus	**iterací**	**moje**	PETSC
`494_bus`			
Gauss-Seidel	501 233	1,19	1,41
SOR $\omega = 1{,}9$	29 863	0,086	0,83
`bcsstk21`			
Gauss-Seidel	183 066	5,58	6,14
SOR $\omega = 1{,}9$	10 290	0,35	0,35
`bodyy4`			
Gauss-Seidel	2071	0,406	0,430
SOR $\omega = 1{,}8$	185	0,047	0,041
`bodyy5`			
Gauss-Seidel	13 523	2,769	3,01
SOR $\omega = 1{,}8$	664	0,181	0,161
`bodyy6`			
Gauss-Seidel	91 429	19,52	20,90
SOR $\omega = 1{,}9$	5 157	1,43	1,18
`nasa1824`			
Gauss-Seidel	321 512	15,75	12,15
`obstclae`			
Gauss-Seidel	272	0,096	0,112
SOR $\omega = 1{,}6$	87	0,044	0,037
průměrný výsledek ze 7 matic			
Gauss-Seidel	113 089	4,22	4,57
SOR	6 611	0,31	0,27

Tabulka 4.6: Porovnání rychlosti oproti PETSc na několika maticích.

4.7 Porovnání oproti jiné implementaci

Jak je zmíněno v kapitole 2.4, budu testovat z knihovny `PETSc` metodu SOR. Matice jsem zvolil stejné jako v předchozí sekci. Navíc je matice 494_bus, která je pro paralelní testování příliš malá, a nepoužil jsem matici nasa1824. Gauss-Seidelovu metodu jsem zvládl počítat o 8,6% rychleji, SOR metodu o 11,7% pomaleji. Podrobnější výsledky jsou v tabulce 4.6.

Pro porovnávání mezi konkurencí nepoužívám školní server, protože jsem nenašel porovnávám pouze sekvenční metody a u těch je přínos dalšího procesoru a jader minimální.

Závěr

Tato práce se zabývá iterativními algoritmy pro řešení soustavy rovnic. Metody fungují pouze na omezeném spektru matic. Postupně byly představeny tři metody, kde každá vylepšuje v nějakém smyslu tu předchozí.

Cílem této práce bylo implementovat nastudované algoritmy sekvenčně a poté paralelně. U paralelní implementace bylo za úkol zjištění vlivu jednotlivých taktik plánování cyklů.

V poslední kapitole jsem podrobně měřil rychlost a zrychlení paralelní verze jednotlivých algoritmů. Výsledky jsem ověřoval na několika maticích. Paralelního zrychlení jsem dosáhnul až $7\times$ s průměrným zrychlením 5,54 pro Jacobiho metodu, 4,52$\times$ pro Gauss-Seidlovu metodu a $4,56\times$ pro SOR. Asynchronní SOR a asynchronní Gauss-Seidlova metody potřebovali průměrně o 1,2% víc iterací, než sekvenční verze.

Příloha A

Seznam použitých zkratek

API Application Programming Interface / programové aplikační rozhraní

COO Coordinate list / Souřadnicový formát, vysvětleno v 1.1

CPU Central processing unit / Centrální procesorová jednotka

CSC Compressed sparse column / komprimované řídké sloupce, vysvětleno v 1.1

CSR Compressed sparse rows / Komprimované řídké řádky, vysvětleno v 1.1

GPU Graphics processing unit / Grafická výpočetní jednotka

SOR Successive over-relaxation / Opakovaná nadměrná relaxace, více v 1.6

Seznam použité literatury

[1] Gene M. Amdahl. „Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities“. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, s. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[2] Satish Balay et al. *PETSc Web page*. 17. dub. 2019. URL: http://www.mcs.anl.gov/petsc.

[3] S. Behnel et al. „Cython: The Best of Both Worlds“. In: *Computing in Science Engineering* 13.2 (2011), s. 31–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118.

[4] Cmglee. *Row and column major order*. Ukázka rozdílu mezi CSR a CSC formátem. URL: https://commons.wikimedia.org/wiki/File:Row_and_column_major_order.svg (cit. 01. 04. 2019).

[5] The SciPy community. *Input and output (scipy.io) — SciPy v1.2.1 Reference Guide*. 6. zář. 2017. URL: https://stackoverflow.blog/2017/09/06/incredible-growth-python/ (cit. 17. 04. 2019).

[6] The SciPy community. *Input and output (scipy.io) — SciPy v1.2.1 Reference Guide*. URL: https://docs.scipy.org/doc/scipy/reference/io.html (cit. 01. 04. 2019).

[7] Intel Corporation. *Intel Intrinsics Guide*. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[8] NVIDIA Corporation. *NVIDIA TITAN RTX*. 17. dub. 2019. URL: https://www.nvidia.com/cs-cz/titan/titan-rtx/ (cit. 17. 04. 2019).

[9] Lisandro D. Dalcin et al. „Parallel distributed computing using Python". In: *Advances in Water Resources* 34.9 (2011). New Computational Methods and Software Tools, s. 1124–1139. ISSN: 0309-1708. DOI: https://doi.org/10.1016/j.advwatres.2011.04.013. URL: http://www.sciencedirect.com/science/article/pii/S0309170811000777.

[10] Tim Davis, Texas A&M University Scott Kolodziej a Yahoo! Labs Yifan Hu. *SuiteSparse Matrix Collection.* URL: https://sparse.tamu.edu/ (cit. 01. 04. 2019).

[11] Timothy A. Davis a Yifan Hu. „The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (pros. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: http://doi.acm.org/10.1145/2049662.2049663.

[12] Bob Hayes. *Programming Languages Most Used and Recommended by Data Scientists.* 13. led. 2019. URL: http://businessoverbroadway.com/2019/01/13/programming-languages-most-used-and-recommended-by-data-scientists/ (cit. 01. 04. 2019).

[13] Arik Hesseldahl. *Moore's Law Hits 50, but It May Not See 60.* 15. dub. 2015. URL: https://www.recode.net/2015/4/15/11561480/moores-law-hits-50-but-it-may-not-see-60 (cit. 17. 04. 2019).

[14] Fiedler M. *Speciální matice a jejich použití v numerické matematice.* SNTL,Praha, 1981.

[15] Tim Mattson. *Hands-on Introduction to OpenMP.* URL: https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf (cit. 17. 04. 2019).

[16] Jean Francois Puget. *A Speed Comparison Of C, Julia, Python, Numba, and Cython on LU Factorization.* 15. led. 2016. URL: https://www.ibm.com/developerworks/community/blogs/jfp/entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization?lang=en (cit. 11. 11. 2019).

[17] Anne Greenbaum; Mark Seager. *DLAP Sparse Linear Algebra Package.* 17. dub. 2019. URL: https://people.sc.fsu.edu/~jburkardt/f_src/dlap/dlap.html (cit. 01. 01. 2010).

[18] Patrick Schmid. *Dothan Over Netburst: Is The Pentium 4 A Dead End?* 25. květ. 2005. URL: https://www.tomshardware.com/reviews/dothan-netburst,1041.html (cit. 17. 04. 2019).

[19] Stackoverflow. *Stack Overflow Trends*. Graf vybraných jazyků na StackOverflow. URL: https://insights.stackoverflow.com/trends?tags=python%2Cjavascript%2Cjava%2Cc%23%2Cphp%2Cc%2B%2B&utm_source=so-owned&utm_medium=blog&utm_campaign=gen-blog&utm_content=blog-link&utm_term=incredible-growth-python (cit. 17. 04. 2019).

[20] Jaka Špeh. *OpenMP: For & Scheduling*. 1. červ. 2006. URL: http://jakascorner.com/blog/2016/06/omp-for-scheduling.htm (cit. 17. 04. 2019).

[21] Saad Y., Society for Industrial a Applied Mathematics. *Iterative Methods for Sparse Linear Systems. Second edition*. Society for Industrial a Applied Mathematics, 2003. ISBN: 9780898715347. URL: https://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.

Příloha B

Obsah přiloženého média

- `readme.txt` ...... stručný popis obsahu CD a intalace
- `thesis.pdf` ...... text práce ve formátu PDF
- `src` ...... zdrojové kódy implementace
  - `solver.pyx` ...... rozhraní Python
  - `matrix.h` ...... definice rozhraní matic
  - `seqsolver.h` ...... implementace sekvenčních řešičů
  - `parallelsolver.h` ...... implementace paralelních řešičů
  - `setup.py` ...... instalační skript knihovny
  - `bench.py` ...... skript obsahujicí funkce měřící výkon
- `thesis` ...... zdrojová forma práce ve formátu LaTeX
- `matrix` ...... ukazkové matice ze SparseMatrix Collection
- `graphs` ... grafy, které jsem generoval, když jsem hledal vhodné matice, většina grafů je generovaný jen paralelníma metodama