FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

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 letního semestru 2019/20

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.

15. května 2019

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 15. května 2019 ……

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Quarda, Martin. *Asynchronní iterativní řešiče*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019. Dostupný také z WWW: ⟨`https://beta.quarda.cz/BP.zip`⟩.

Abstrakt

Práce se zabývá třemi algoritmy na řešení lineární soustavy rovnic. Algoritmy řeší pravou stranu matice udavající soustavu rovnic. Matice musí splňovat jeden ze dvou předpokladů, aby to zvládly vyřešit. Vybrané algoritmy jsou Jacobiho metoda, Gauss-Seidelova metoda a SOR metoda. Algoritmy budu implementovány sekvenčně a paralelně. Algoritmy porovnám mezi sebou, rychlosti konvergence i rychlost proti konkurenci.

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

Seznam obrázků

Seznam tabulek

Úvod

Soustavy rovnic mají uplatnění ve spoustě odvětvích matematiky, například ekonomie, fluidní dynamiky, lineárního programovaní a spoustě dalších. Jejich řešní je kritické při jejích interpretacích.

Iterativní metody hledání řešení se používají tam, kde klasické metody selhávají nebo jsou příliš neefektivní. Selhávají pro vysoce rozměrné matice, kde klasické metody mají příliš velkou složitost. 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. Ty se téměr nevyskytují v písemce ve škole, ale reálné matice jsou mnohem rozměrnější a většinu hodnot mají nevyplněné už z povahy, jak jsou sestavené.

Téma jsem si vybral, neboť mě baví programovat paralelní programy a existující řešení neimplementují vybrané metody paralelně.

V teoretické části se analyzuji algoritmy a technologie. V praktické části implementuju algoritmy prvotně sekvenčně a poté je zkouším paralelizovat. Metody budu implementovat tak, aby byly použitelné v prostředí Python.

Kapitola 1

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.

Cílem praktické části je implementace nastudovaných metod sekvenčně, paralelně a porovnání implementací mezi sebou a proti konkurenci. Implementace proběhne v C a Cythonu.

Implementované metody otestuju mezi sebou i oproti ostatním implementacím.

1.1 Struktura práce

V kapitole 2 je zezačátku zadefinován problém, který metody řeší. Představil jsem podmínky, za kterých mohou algoritmy konvergovat, a představil způsob, jakým se určuje velikost chyby. V sekcích 2.2 - 2.4 jsou postupně popsány jednotlivé metody – Jacobiho, Gauss-Seidlova a SOR.

Následně jsem ukázal, jak se reprezentují řídké matice v paměti. Nakonci 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 a proč se vlastně paralelní systémy prosazují.

V kapitole 3 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 3.3 jsem popsal krátce `Cython`. Nakonci jsem se věnoval konkurenčním implementacím s kterou se budu porovnávat.

V praktické části (kapitola 4) první vymýšlím vhodné rozhraní, které sdílí 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. Nakonec jsem měl dostatek času na implementaci adaptivní SOR metody. Ta je popsána v sekci 4.6.

V kapitole o testování zkouším zezačátku nějaké optimalizace. Hned poté v sekci 5.2 testuji jednotlivě dvě matice. Důkladné testování začíná až v následující sekci, kde měřím o kolik jsou rychlejší paralelní verze. Nakonci porovnávám svojí implementaci s knihovnou `PETSc`.

Kapitola 2

Teoretická část

V této kapitole je první zadefinován problém, poté popsány jednotlivé algoritmy a nakonec popsané řídké matice. Popsané algoritmy jsou Jacobiho, Gauss-Seidelova a SOR metoda. Nakonci popisuji řídké matice a paralelní programování.

2.1 Definice problému

Pro danou matici A a vektor b, chceme najít vektor x pro který platí:

$$Ax = b \tag{2.1}$$

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

Matice A je čtvercová matice o velikosti n, vekotory b a x jsou vektory velikosti n. Z obecného zápisu je odvodit návod na implementaci. Proto budu u každé metody uvádět také alternativní zápis přes sumy, kde rovnice platí pro každý řádek (zde označen jako i):

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i \tag{2.2}$$

Vektor x lze najít analyticky, nicméně pro rozsáhlé nebo řídké (případně obojí) je to výpočetně náročné. Uvedené metody umí za speciálních podmínek najít vektor x rychleji. V této práci se zabývám iterativním hledáním

řešení, kde zkusím uhodnout vektor x a pomocí následujících metod se postupně zlepšuje, dokud není dostatečně malá chyba řešení.

2.1.1 Chyba řešení

Chybu nalezeného $\hat{x}$ lze porovnávat pomocí normy. Vektor x zobrazíme pomocí matice A a od toho odečteme b. Tíme nezískáme přímo chybu x, ale získáme tím měřítko, jak přesně je zobrazený na b:

$$||A\hat{x} - b|| = \sqrt{\sum_{i=1}^{n} \Big(\sum_{j=1}^{n} a_{i,j}\hat{x_j} - b_i \Big)^2} \tag{2.3}$$

Neduh této metody je prokletí dimenzionality. Čím rozměrnější matice, tím větší bude chyba, ikdyž v jednotlivých rozměrech bude chyba stále porovnatelná.

2.1.2 Podmínky nalezení řešení

Postačující podmínka, během kterých uvedené metody najdou řešení, je že matice A je striktně diagonálně dominantní. Tato podmínka je velice omezující a matice splňující tuto podmínku jsou spíše vzácné. Definice strikní diagonální dominance je následující, pro každý řádek i platí následující vzoreček:

$$|a_{i,i}| > \sum_{j!=i} a_{i,j}$$

Alternativní podmínka je, že matice A je pozitivně definitní a symetrická. Postačující, že iterační matice má spektrální poloměr menší jak 1. Spektrální poloměr je definován jako největší vlastní číslo matice v absolutní hodnotě. Najít vlastní číslo je srovnatelný problém s vyřešením rovnice a proto je lehčí zkusit použít metodu na matici a z toho usoudit, jestli se zmenšuje chyba řešení a podle toho určit jestli funguje. Pozitivně definitní matice má všecny vlastní čísla kladní.

Následující metody jsou všechny iterativní. Odhadneme nějaké řešení (třeba vektor v bodě 0), které postupně vylepšujeme každou iterací. To jestli metoda zkonverguje může ovlivnit počáteční volba x. Rychlost konvergence záleží na metodě. Jacobiho je typicky nejpomalejší a ostatní se snaží vylepšovat nedostatky předchozí metody.

2.2 Jacobiho metoda

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

Matici A rozdělíme na Diagonálu D a zbytek R, platí $A = D + R$. Poté iterativně řešíme podle rovnice 2.4. Z rovnice 2.5 vyplívá to, jak se metoda implementuje.

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \tag{2.4}$$

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

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(k)}\right) \tag{2.5}$$

2.3 Gauss-Seidlova metoda

Gauss-Seidlova metoda je trošku složitější. Pro výpočet $x^{(k+1)}$ už nepoužíváme jen $x^{(k)}$, ale začnou se tam používat už vypočítané hodnoty $x^{(k+1)}$. [1]

Matici rozdělíme na dolní trojúhelník včetně diagonály L_* a horní trojúhelník bez diagonály U. Poté iterativně řešíme podle rovinice 2.6. Zápis stejné rovnice přes sumy je v rovnici 2.7. Algoritmus se snaží co nejdříve používat aktuálnější $\hat{x}$.

$$x^{(k+1)} = L_*^{-1}(b - Ux^{(k)}) \tag{2.6}$$

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

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(k)}\right) \tag{2.7}$$

2.4 SOR modifikace metody

Successive over-relaxation (rovnice 2.8) metoda zavádí koeficient ω. Tento koeficient slouží k zvětšení ($\omega > 1$) nebo zmenšení ($\omega < 1$) skoku oproti Gauss-Seidlově metodě a závisí na konkrétní matici. Optimální hodnota parametru záleží na matici. Pohybuje se v rozsahu $0 < \omega < 2$. V rovnici 2.9 je představen intuitivnější zápis pomocí sum. [2]

$$x^{(k+1)} = (D + \omega L)^{-1}(\omega b - [\omega U + (\omega - 1)D]x^{(k)}) \tag{2.8}$$

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

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(k)}\right) \tag{2.9}$$

Speciální případy nastávají při $\omega = 1$ (metoda degraduje na Gauss-Seidlovu) a $\omega = 0$ (přestane se zlepšovat řešení, protože bude platit: $x_j^{(k+1)} = x_j^{(k)}$).

2.5 Paralelizace iterativních metod

Paralelizace Jacobiho metody je jednoduchá. Z rovnice 2.5 jde vidět, že jediný co potřebuju na každý řádek je matice A a $x^{(k)}$. Ty už znám z předchozí iterace a tedy každý řádek můžu paralelizovat a nemusím se obávat, že by to ovlivnilo výpočet.

Paralelizaci Gauss-Seidelovi metody budu provádět modifikovaným způsobem. Striktní GS metoda by brala pouze $x_i^{(k+1)}$ takové, že i je menší než aktuálně počítaný index. Já budu využívat co nejvíce aktuálních hodnot $x_i^{(k+1)}$. Ty které budou k dispozici se použijí, jinak se využije hodnota starší iterace $x_i^{(k)}$.

Touto modifikací dosáhnu intuitivní implementací paralelní metody. Měl bych tím zaplatit lehce pomalejší konvergencí, která doufám nebude mít až tak velký vliv. Modifikace SOR metody proběhne obdobně modifikovaně.

2.6 Formáty řídkých matic

Pro ukládání řídkých matic se převážně používají 3 formáty. Nejjednoduší je `COO`, ale není příliš vhodný pro aritmetické operace. Další jsou `CSR` a `CSC`, kde jeden je vhodnější na sloupcové přístupy a další na řádkové přístupy. [2]

2.6.1 COO

`COO` (compressed coordinate list, přeloženo jako kompresovaný seznam souřadnic) je ten nejjednodušší. Umožnuje duplikativní záznamy, které se typicky vyřeší, při převodu do jiného formátu, sečtením. Je reprezentován 3 stejně dlouhýma polema. První pole jsou samotné hodnoty, další 2 řádky představují index sloupce a řádku. `COO` formát neurčuje nějaké pořadí elementů, takže můžou být seřazené libovolně. Občas může být vhodné ukládat velikost, ale typicky se zvládne vyčíst z prvků a 2 čísla nejsou takový rozdíl.

Pro matici A:

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

Vypadají jednotlivá pole takto:

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

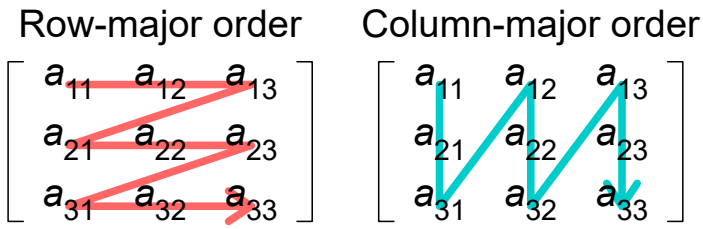
$$A_{cols} = \begin{bmatrix} 1 & 3 & 4 & 5 & 1 & 2 & 3 \end{bmatrix}$$

$$A_{rows} = \begin{bmatrix} 1 & 2 & 1 & 3 & 4 & 4 & 5 \end{bmatrix}$$

Formát je dost žravý na místo a vezme přinejmenším $3k$ místa, kde k znamená množství vyplněných čísel. V horším případě při mnoha duplicitách může vzít i mnohem víc místa.

2.6.2 CSR

`CSR` (compressed sparse rows, přeloženo jako kompresované řídké řádky) je formát, který je nejvhodnější na mojí aplikaci. Opět je reprezentován 3 polama, ale poslední pole je kratší a má délku n. První 2 pole mají stejný význám jak v `COO`, první reprezentuje samotný data a další index sloupců. Poslední pole je pole indexů na začátky řádků. Nakonec je připojen končící index ukazující hned za poslední řádek. Navíc jsou ale data seřazený



Obrázek 2.1: Ukázka porovnání CSR a CSC formátu. Napravo je naznačeno řazení CSR formátu a nalevo CSC. [3]

sestupně podle řádků. V jednotlivých řádcích můžou být uspořádany libovolně, ale dává se přednost tomu, že jsou take setřízeny.

Pro matici A uvedenou výše vypadají jednotlivá pole takto:

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

$$A_{cols} = \begin{bmatrix} 1 & 4 & 3 & 5 & 1 & 2 & 3 \end{bmatrix}$$

$$A_{rowindex} = \begin{bmatrix} 1 & 3 & 4 & 5 & 8 \end{bmatrix}$$

Místa zabírá méně jak `COO`, konkrétně $2k + n$ a má příjemnější vlasnosti pro aplikaci matice na vektor x (Ax). Pro aplikaci matice je celý řádek po kupě. Pro efektivní ukládání řídké matice je potřeba, aby měla hustotu $< 50\%$ a u `COO` dokonce $< 33\%$. U matic malých rozměrů nemusí být příliš výhodné, nicméně pro řídké matice velkých rozměrů už může být úspora ohromná.

`CSC` formát je stejný jak `CSR`, akorát transponovaný. Tj. uchovává indexy řádků a sloupce jsou po kupě.

2.7 Paralelní programování

Paralelní programování je aktuální programovací disciplína. Předpověď Gordona Moora (jeden ze zakladatelů Intelu) v roce 1975, že počet tranzistorů integrovaných v jednom čipu se zdvojnásobí každý rok (od roku 1965 tvrdil každý rok, který později změnil) platila ještě donedávna. Díky limitům velikoti atomů, a toho že od nějaké velikosti hradel platí spíš kvantová mechanika (V kvantové mechanice fungují jevy jako Kvantové tunelování, které jsou neslučitelné s aktuálním fungováním hradel.) než klasická mechanika na které stojí mikročipy, se už lehce spomaluje. Čipy by museli být neúměrně velké pro svoje použití. Ty je potřeba lépe chladit a jsou dražší.

Aktuálně jsou hradla v procesoru úzká několik atomů a předpokládá se, že zákon přestane platit do roku 2025. [4]

Dřív procesory zrychlovali tím, že se zvyšovala frekvence jednoho jádra, bohužel u Pentium 4 zjistiji, že přílišná frekvence a rozdělení instrukcí na miniinstrukce není výhodné. [5] 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. [6]

2.7.1 Amdahlův zákon

Amdahlův zákon údává 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×. [7]

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

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

2.7.2 Úskalí paralelního programování

Paralelní programování má pár specifických úskalí, se kterýma se v klasickém sekvenčním programování nedá setkat.

Deadlock je čistě chyba programu. Je to když dvě vlákna pasivně čekají na uvolnění nějakého prostředku co má to druhé vlákno a ani jedno ho neumí uvolnit tomu druhému. Opačná situace je livelock, to je aktivní čekání způsobené stejnou chybou, akorát obírá o čas procesoru ostatní procesy.

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

Falešné sdílení je stav, kdy procesory sdílí stejný pamětový prostor do kterého zapisují a přehazují si jeho část v rámci. Z MESI koherenčního modelu může mít pouze jedno vlákno blok v modifikovaném stavu a ostatním vláknům ho tímto zápisem zneplatní. Pokud ho ale zároveň využívají ostatní vlákna, tak je tam nadměrná komunikace, která zajištuje znovusdílení bloku.

Kapitola 3

Použité technologie

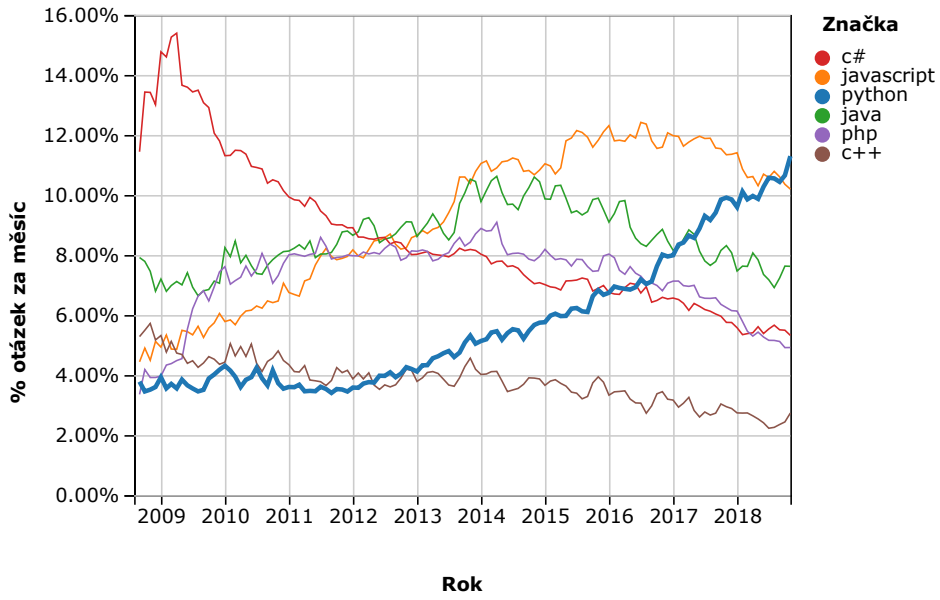
V téte kapitole popíšu jednotlivé použité technologie. Jako základní jazyk byl zvolen `C/C++`, který je defacto standard pro nízkoúrovnové a efektivní aplikace. Použití knihovny bude prostřednictvím `Python`. To je interpretovaný jazyk, který je pomalejší jak `C/C++`. Iterování a výpočet metod bude probíhat v `C/C++`. `Python` bude zajišťovat nahrávání matic a nastavování parametrů metod. Nakonci se věnuji existujícím konkurenčním řešením.

3.1 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 simulovat bariéry. Podobně jednoduše umožnuje použití všech paralelních konstruktů. [8]

`OpenMP` obsahuje bohaté nastavení paralelizace smyček. Při nastavení `dynamic` plánovač volí za běhu na kterém poběží jednotlivé iterace. Nevýhodou je, že se vlákna můžou při příliš rychlém iterování potkat při dotazu na plánovač. Poté vlákno, které se dotázalo později skončí dočasně ve spánkovém módu. Tomu se předchází nastavením většího `chunk-size`, který určuje po kolika iteracích přiděluje plánovač.

Plánování `static` předem určí rozsahy pro jednotlivá vlákna. Tím zmizí riziko spánku vlákna při souběžném dotazu na plánovač. Nastavením parametru `chunk-size` se dá změnit velikost rozdělovaných bloků. Při nastavení 1 se střídají vlákna po každé iteraci, komu bude přidělena. Výchozí nastavení je $\frac{\text{délka cyklu}}{\text{počet jader}}$, které rozdělí do souvislých bloků a každý blok má za úkol



Obrázek 3.1: Porovnání jednotlivých jazyků na StackOverflow. [11]

vlastní vlákno. Nastavení `static` je nevýhodné, když jednotlivé iterace trvají rozdílnou dobu. [9]

Při nastavení `guided` plánovač funguje na pomezí `static` a `dynamic`. Zezačátku rozděluje větší bloky, aby omezil množství dotazů na plánovač. Nakonci rozděluje menší bloky pro vlákna, které stihli větší bloky dokončit rychle. Nastavení `chunk-size` omezuje velikost nejmenšího bloku.

3.2 Python

`Python` je interpretovaný jazyk, který v poslední době získává popularitu i ve vědecké společnosti. V rámci Jupyter notebooku je možné lehce kombinovat Python kód a text práce. Při prohlížení notebooku si můžete okamžitě sám experimentovat a zkoušet spouštět kód.

Není proto divu, že byl na stránce Kaggle nejpopulárnější, která si klade za cíl spojovat datové vědce po světě, tím že jim dává platformu, kde jsou k dispozici zajimavá data a vyhlašuje nejlepší řešení. Python je taky nejdoporučovanější jazyk pro začínající datové vědce. [10]

Na Stackoverflow si všimli jazyka `Python` jako nejrychleji rostoucího hlavního programovacího jazyka ve státech vysokého přijmu (podle definice Světové banky. Náleží mezi ně i Česká Republika od roku 2006). [12]

Samotný Python by byl příliš pomalý a má pár omezení, nicméně ve výchozí implementaci (CPython) má velice dobré API a je jednoduché na něj navázat efektivnější program v libovolném jazyce, který podporuje volací konvenci C. Toho aktivně využívá velké množství kvalitních knihoven. Nejznáměnší knihovny pro datové vědce jsou Numpy, SciPy, Pandas a TensorFlow.

Mezi omezení patří nemožnost plné paralelizace. Python prostředí má jeden velký zámek, který může mít maximálně jedno vláko 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. Můj kód, co bude počítat běží v C++ a proto mě neomezuje zámek Python.

3.3 Cython

Cython je nadstavba Python. Zvládne zkompilovat většinu kódu v Python a přidává svoje konstrukce, které jsou rychlejší. Funguje tak, že vezme svůj kód, ten zkompiluje do C kódu využivající CPython API, které svým voláním dělá stejný výsledek jak původní Python kód.

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 ušetří spousta času a navíc sám nechá uvolněný zámek Python.

Další výhoda je, že z Cythonu je možné přímo volat efektivně C a C++ funkce. [13]

3.4 Představení ostatních implementací

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

První je DLAP. Knihovna je implementovaná v jazyce Fotran. Fotran je jazyk z poloviny 20. století, který se dneska už používá jen na specifické úlohy. Mezi tyto úlohy patří termodynamické simulace a je to jeden z používaných jazyků pro superpočítače. Já Fotran neovládám a DLAP má implementovanou pouze sekvenční verze algorimů. [14]

Další je PETSc. Knihovna je implementovaná v jazyce C. PETSc má implementované Jacobiho a SOR metodu. Gauss-Seidelova metoda je doažitelná přes SOR metodu s $\omega = 1$. Jacobiho metodu je možné použít i para-

lelně přes `OpenMPI`. Vzhledem k problémům, které se mě vyskytli u sekvenční verze SOR jsem paralelního Jacobiho netestoval. `OpenMPI` je knihovna na paralelizaci programů, která funguje na principu posílání zpráv. Je důležité neplést si jí s `OpenMP`. [15]

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 dosáhnul segmantation fault, při kterých se ani nedozvím na kterém řádku `Python` to zkolabovalo. Vypíše akorát `C` stack volání funkcí. [16]

Knihovna se soustředí na pokročilé metody (především na KSP, Krylov subspace methods) a mé vybrané metody se tam obvykle používají jen na předpočítání (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 zjištuji, že tam je pouze zjednodušená Jacobiho metoda. Implementovaná tam je jen první iterace s nulovým počátečním vektorem x a zjednodušené blokové varianty. Je irelevantní porovnávat rychlost, protože tam je implementovaná tak, že se předpočítá diagonála a první iterace se poté provede jedním vynásobením vektorů navzájem. Blokové varianty poté mají implementovaný už běžnou jacobiho metodu, ale pouze na omezeném bloku matice do maximálně 7×7.

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

Kapitola 4

Praktická část

Implementace algoritmů bude probíhat v `C/C++` a implementace rozhraní v Cythonu. V první kapitole ukážu jak se dají použít libovolné matice v `Python`. V druhé představím rozhraní a co vše se tam dá nastavovat. V dalších kapitolách popíšu jak to funguje uvnitř a jak jsem postupoval. V poslední kapitole je představená metoda Adaptivní SOR metoda.

4.1 Použití matic v Python

Pro husté matice stačí balíček `NumPy`. Pro práci s řídkýma maticema se v `Python` používá balíček `SciPy`. `Python` umožnuje matice zadávat přímo, případně načítat ze souboru, v libovolném formátu. 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. Přípaně 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 [17] jsou matice dostupné ve 3 formátech a všechny zvládne SciPy načíst.

Pro načtení stažené matice stačí napsat nasledující kód:

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

Funkce `scipy.io.mmread` sloučí k načtení Matrix Market formátu. [18]

4.2 Definice rozhraní knihovnu

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

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

method *volitelný* Název Metody která má být použita na řešení. Metody jsou vysvětlené v 2. Při nezadání se zvolí automaticky Gauss-Seidel při $\omega = 1$, jinak SOR. Možné hodnoty jsou: `'jacobi'`, `'gaussseidel'`, `'sor'` a `'adaptivesor'`. Adaptivní SOR má několik speciálních parametrů vysvětlených v kapitole 4.6.1.

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

threshold *volitelný* U jacobiho metody značí hranici chyby, po které se zastaví výpočet (chybu minulé iterace je možné efektivně počítat během iterování). U ostatních metod značí hranici rozdílu jednotlivých iterací. Když nastane $||x^{(k+1)} - x^{(k)}|| <$ `threshold`, tak skončí výpočet a aktuální x se vrátí jako řešení. Při nezadání se zvolí 0 (neskončí se nikdy a nebo se dosáhne maximálního počtu iterací).

iterations *volitelný* Maximální počet jednotlivých iterací algoritmu. Jakmile by se překročil počet, tak se vrátí aktuální výsledek bez ohledu na správnost. Při nezadání se použije neomezený počet iterací, což nebude vhodný nápad pro rozlehlé matice.

omega *volitelný* Hodnota ω z algoritmu SOR, při jiném než algoritmu SOR nedělá nic. Při nezadání se použije neutrální hodnota 1.

Metody třídy `Solver` jsou následující:

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

setIterations Metoda nastaví maximálního počtu iterací. Po dosáhnutí tohoto počtu pokaždé skončí.

setThreshold Metoda nastaví threshold.

setOmega Metoda nastaví parametr omega při SOR, při jiné metodě vyhodí vyjímku.

Jednoduchost rozhraní jde 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 tímpádem 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])
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 pár parametrů algoritmu, nicméně má neimplementovanou virtuální třídu `solve` a tu implementují její potomci představující jednotlivé algoritmy.

4.3 Vnitří 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 tuhle tabulku.

Implementovat funkci používající abstraktní třídu jako parametr je přímočarné, ale abstraktní třída matic by zabránila kompilátorovy optimalizovat krátké funkce a bylo by potřeba vytvořit abstraktní třídy i pro iterátory.

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

Moje požadavky na matice byli malé, tak umí akorát iterovat po řadcích a v řádcích po jednotlivých hodnotách a umí zjistit, kde se nachází. Paměť nad polema v mém případě spravuje `Python` a `NumPy`, proto moje třídy matic slouží jen jako obalové třídy pro snažší iteraci.

Navíc je možné, aby matice využívala i jiný datový typ, než `double`. Nicméně `AbstractSolver` má ve své definici `double` tak není možné využívat jiné datové typy.

Použití rozhranní matice je vidět na výpočtu chyby:

```
template <typename MatrixType,
  typename T = typename MatrixType::T>
double matrixError(const MatrixType& matrix,
  const T* b, const T* x){
    T err_sqr = 0;
    for(auto row : matrix){
        auto line = - b[row.rowIndex()];
        for(auto col = row.begin(); col != row.end(); ++col){
            line += *col * x[col.columnIndex()];
        }
        err_sqr += line * line;
    }
    return sqrt(err_sqr);
}
```

Poté při použití s `DenseMatrix<Double>` se vygeneruje verze pro hustou matici a při `SparseMatrix<Double>` vygeneruje verze pro řídkou matici.

4.4 Sekvenční implementace

První jsem implementoval Jacobiho metodu. Implementace samotného algoritmu je jednoduchá. Akorát jsem si musel vhodně formulovat to, kdy se algoritmus může zastavit. U Jacobiho metody by šlo počítat chybu minulé

iterace přímo a téměř s nulovým vlivem na výpočet, ale u ostatních metod by to výpočet spomalilo. Proto jsem zvolil jako měřítko $||x^{(k)} - x^{(k+1)}||$, které se lehko implementuje do všech třech algoritmů.

Další vylepšení je prohazování rolí $x^{(k)}$ a $x^{(k+1)}$. Kdyby se neprohazovali role, tak je nutné během každé iterace skopírovat nové hodnoty x do původního pole. K prohození rolí stačí vyměnit pouze 2 ukazatele.

Gauss-Seidelova metoda byla na řadě druhá. Ta je hrozně podobna Jacobiho metodě, akorát si nepotřebuje pamatovat původní hodnoty x. Na sekvěnční naprogramování je jednodušší jak Jacobiho metoda. Je u ní zbytečný trik s měněním významu polí, protože má pouze jedno pole pro hodnoty x.

Poslední metoda SOR je příbuzná Gauss-seidelově. V podstatě sdílí celý kód s Gauss-Seidelem, akorát závěrečný výpočet x je lehce rozdílný. Hodnota threshold je saturovaná omegou, protože tahle metoda mění velikost skoku oproti Gauss-Seidelovi.

4.5 Paralelní implementace

4.5.1 Jacobiho metoda

Jacobiho metoda metodu jsem implementoval jako první. Naprogramoval jsem jí striktně Jacobiho, což se ukázalo těžší, než jsem si myslel. V prvotních implementacích stačilo vynechat jednu synchronizační bariéru a zvýšila se rychlost, ale začlo to fungovat částečně jako Gauss-Seidelova metoda.

4.5.2 modifikovaná Gauss-Seidelova a SOR metoda

Další jsem implementoval Gauss-Seidelovu metodu. Vzorec Gauss-Seidlovi metody je zadefiovaný sekvenčně. Pro paralelní prostředí nevýhodné, takže budu programovat metodu, co *vlastně není Gauss-Seidel*, ale pouze se jí blíží svým chováním. Ze vzorce z kapitoly 2.3 je vidět, jak funguje Gauss-Seidelova metoda. Metoda Vezme co nejvíce hodnot z aktuální iterace a tam kde zatím nemá hodnoty aktuální iterace, tak použije hodnotu z minulé iterace.

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 jí taky využije) a ostatní doplní hodnotama z minulé iterace. Vzhledem k tomu, že výpočet probíhá paralelně, tak využije méně hodnot z aktuální iterace, než regulerní Gauss-Seidlova metoda.

Proto konverguje pomaleji, nicméně výpočet probíhá paralelně a teda rychleji.

Paralelní SOR přebírá všechny vlastnosti paralelního Gauss-Seidelovi metody. Pouze při dosazení závěrečné hodnoty proběhne modifikace x po dle parametru ω.

4.6 Implementace Adaptivní paralelní SOR metody

Na základě předchozích metod jsem zkusil vymyslet vlastní adaptivní metodu. Ta se snaží optimalizovat parametr omega na základě předpokladu vývoje chyby s daným parametrem. Většina iterací probíhá běžně jako modifikovaná paralelní SOR metoda, ale jednou za určený počet iterací se provede optimalizační iterace.

Během optimalizační iterace každé vlákno zkusí svojí hodnotu ω na sekvenční verzi SOR. Sekvenční verzi používá z důvodu stabilnějšího průběhu oproti paralelní verzi. Vypočítají se 2 iterace a chyby po každé iteraci se dosadí do vzorečku 4.1. Chyba po první iteraci je označená ve vzorečku e_1, obdobně chyba po druhé iteraci je e_2. Počítají se dvě iterace, protože některé hodnoty ω dočasně zvýší chybu, ale po několika iteracích svým rychlejším konvergováním přeskočí původně lepší, ale pomaleji konvergujicí ω.

$$e_f = e_1 \cdot \left(\frac{e_2}{e_1}\right)^{\texttt{needy}} \tag{4.1}$$

Tento vzoreček se snaží předpokládat hodnotu chyby za `needy` iterací. Parametr ω zvolí tak, aby minimalizoval předpokládanou chybu e_f. Takový vzoreček jsem zvolil potom, co se ukázalo jako nedostatečně vzít v úvahu pouze chybu nebo vektor kterým se zlepšuje. Pro některé matice je třeba přizpůsobovat parametr `needy`, jinak se adaptivní metoda chová nestabilně.

4.6.1 Parametry metody

Adaptivní metoda umí přizpůsobovat parametr ω sama během běhu. Oproti ostatním metodám má speciální parametry. Metoda je implementovaná pouze paralelně. Pouze paralelně jde počítat spoustu sekvenčních SOR iterací a neztratit mnoho času. Následuje výpis paramterů metody. Ostatní parametry přebírá z ostatních metod, ty jsou popsány v kapitole 4.2.

adaptive_iteration *volitelný* Určuje kolik iterací je pauza mezi přizpůsobením parametru ω. Výchozí hodnota je 100.

needy *volitelný* Určuje kolik iterací dopředu bude předpokládat chybu. Výchozí hodnota je 50.

min *volitelný* Minimální ω, který může algoritmus zvolit. Vychozí hodnota je 0,5.

max *volitelný* Maximální ω, který může algoritmus zvolit. Výchozí hodnota je 1,5.

level *volitelný* Počet úrovní přizpůsobování parametru ω. Větší hodnoty znamenají přesnější optimální určení parametru ω. Výchozí hodnota je 2.

Kapitola 5

Měření výkonu a testování

V první části kapitoli zkouším speciální optimalizace. První zkouším kompenzovat falešné sdílení cache pamětí, poté zlepšuji rozdělování úseků u řídkých matic. Od sekce 5.2 už následuje měření. Prvotně měřím metody na jednotlivých maticích. Poté měřím několik matic najednou a zjištuji jak hodně paralelizace zvyšuje výkon. Nakonci je porovnání s implementací v knihovně `PETSc`.

Měření bude probíhat na počítači s následujícími parametry:

GCC gcc version 8.3.0

PYTHON Python 3.7.3

CYTHON Cython version 0.29.7

CPU i7-7700HQ 14nm 3.4 GHz

CORES 4 jádra, 8 vláken

CACHE 6MB L3, 4× 256 KB L2, 4× 64 KB L1

RAM 8GB DDR4-2400

Ostatní parametry by měli být irelevantní. Teoretické zrychlení by se mělo blížit u paralelní verze 4×. Všechny matice (kromě první) jsem stahoval z SuiteSparse Matrix Collection. Testoval jsem spoustu matic, některé grafy jsou přiložené ve složce `graphs` s bakalářskou prací. [19] Funkce použité pro měření jsou ve skriptu `bench.py`.

5.1 Testování optimalizací během implementace

Během implementace jsem zkoušel několik různých optimalizací. Vliv dvou konkrétních popisuji v další kapitole. Mimo popsané dále jsem také optimalizoval použití bariér, každá bariéra významně spomaluje každou iteraci.

5.1.1 Kompenzace falešného sdílení

Řekl jsem si, že zkusím ještě vlastní způsob kdy se pokusím omezít vliv falešného sdílení, ten je vysvětlen v 2.7.2. To znamená je třeba správně kompenzovat šířku cache line. Ta je typicky `64B` u intelových procesorů. Takže je třeba všechny pole přizpůsobit tomu, aby posun byl takový, aby adresa všech polích měla adresu dělitelnou 64 (nebo-li $0x40$). Tato metoda má potenciál zrychlit husté matice a kdyby se to ukázalo jako úspěšná. S potenciálem rozšiřit to i na řídké matice.

`Numpy` nemá možnost přímo manipulovat s adresou pole. Proto se to musí trošku obcházet přes `slice`, další možnost by byla vnutit přímo adresu, ale to by se ztratila krásná vlastnost. `Numpy` by už automaticky neuvolnoval paměť.

Zkoušel jsem testovat, jestli má zarovnání paměti vektoru x vliv rychlost výpočtu na hustou matici u Jacobiho metody. Vzal jsem matici `494_bus` ze SuiteSparse Matrix Collection, doplnil ji nulama na hustou matici a porovnával koeficient zrychlení sekvenční verze proti paralelní verzi. Tato matice nekonverguje, ale diverguje velice pomalu. Vzhledem k tomu, že pouze porovnávám rychlosti jednotlivých iterací a neporovnávám rychlosti konvergence, tak to nevadí.

Provedl jsem 10000 měření na 100 iteracích Jacobiho metodě. Poté jsem porovnal zrychlení mezi sekvenční a paralelní verzi. Pro zarovnanou verzi vyšel koeficient zrychlení 3,92. Pro nezarovanou verzi vyšel koeficient zrychlení 4,34. Mezi sekvenčníma verzema nazájem byl zanedbatelný rozdíl o koeficientu 1,002 ve prospěch nezarovnané verze. Z koeficientů jde vidět, že zarovnávání je nadbytečné pro husté matice, proto ho nebudu provádět.

5.1.2 Lepší plánování pro řídké matice

Zlepšení pro řídké matice je rozdělit rozmezí cyklů líp jak `OpenMP` static. `OpenMP` nemá jak zjistit kolik čísel je v jednotlivých řádcích řídké matice. Pro husté matice je static plánování to nejvhodnější, protože matice je plná čísel a je třeba vykonat přesně stejně násobení pro každý řádek. Nicméně

u řídkých maticích to tak být nemusí, proto by mohlo být lepší je rozdělit na reálné zaplněnosti jednotlivých řádků.

Proto jsem naprogramoval metodu na nalezení ideálního dělení řádků. Ta pracuje v čase $O(\ln(n))$ pomocí binárního vyhledávání. Hlavní vlákno udělá pole rozdělení matice a poté se tím paralelní část řídí.

Stejně jak v předchozí kapitole jsem provedl měření, tentokrát na řídké verzi stejné matice. Počet iterací jsem zvýšil na 1000. Vzhledem k velikosti matice vychází paralelní verze oproti sekvenční pomalejší. Původní paralelní verze se static plánováním měla koeficient zrychlení 0,42. Vylepšená verze měla koeficient zrychlení 0,49.

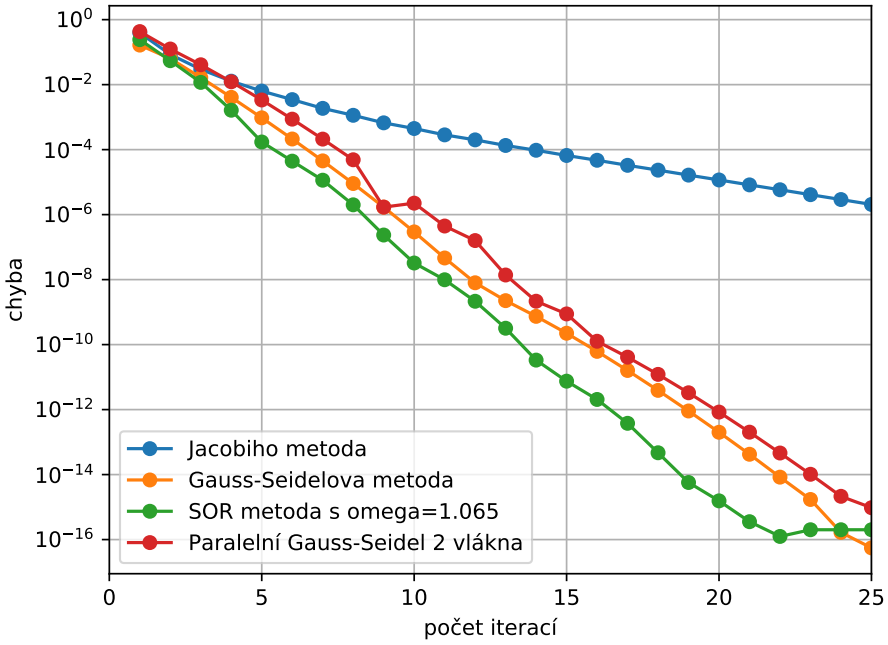
5.2 Porovnání rychlosti běhu a konvergence metod na jedné matici

Jako první jsem zvolil úplně jednoduchou matici. Vzhledem k velikosti matice nebudu uvádět rychlosti běhu, ale paralelní metody byli pomalejší než sekvenční. Z grafu jde vidět jak se postupně metody zlepšují. Paralelní Gauss-Seidelova metoda s 2 vláknama je jen těsně nad obyčejným Gauss-Seidelem. Jacobiho metodu nerozděluji na paralelní a sekvnenční na tomhle grafu, vzhledem k tomu, že rychlost konvergence zůstává stejná u paralelního Jacobiho.

$$\begin{bmatrix} 5 & -1 & 2 & 0 \\ -1 & 6 & -1 & 3 \\ 2 & -1 & 5 & -1 \\ 0 & 3 & -1 & 7 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

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-Seidela.

Matice `obstclae` je velká 40000×40000 a má 197608 nenulových hodnot. Vektor pravé strany byl zvolen jako jednotkový vektor, počáteční x je nulový vektor. Následuje tabulka počtu iterací a časů, které byli potřeba ke konvergenci, zde zvolenou jako chybě menší jak 10^{-12}. Tato hodnota byla zvolena, protože pro menší hodnoty se nějaké metody zasekli a vlivem chyby výpočtu se přestalo řešení zlepšovat. Matice je netypická tím, že na ní Jacobiho metoda konverguje skutečně rychle a dokonce svou rychlostí někdy překonává Gauss-Seidela.



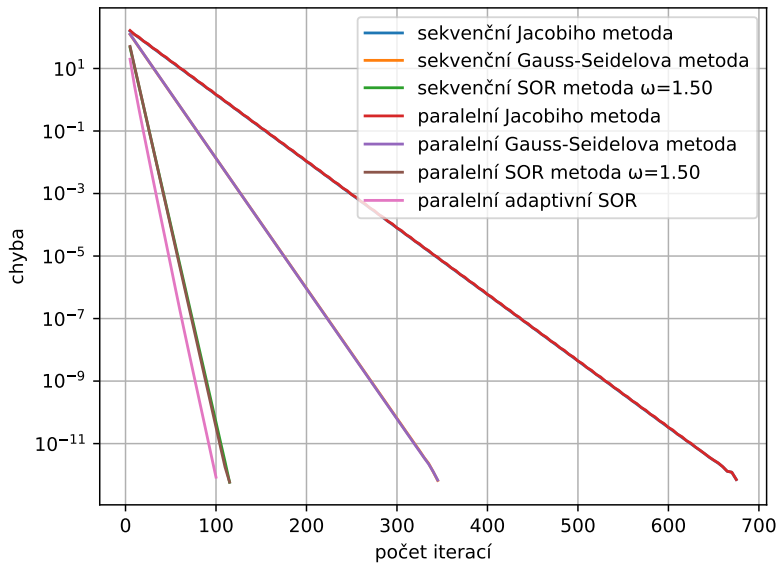
Obrázek 5.1: Rychlost konvergence metod na jednoduché matici.

	sekvenční verze		paralelní verze		**parelelní**
Algoritmus	**iterací**	**čas [ms]**	**iterací**	**čas [ms]**	**zrychlení**
Jacobi	674	152,5	674	48,2	3,16
Gauss-Seidel	344	145,6	344	57,5	2,53
SOR $\omega = 1{,}5$	113	58,1	113	23,3	2,49
adaptivní SOR	–	–	100	39,5	–

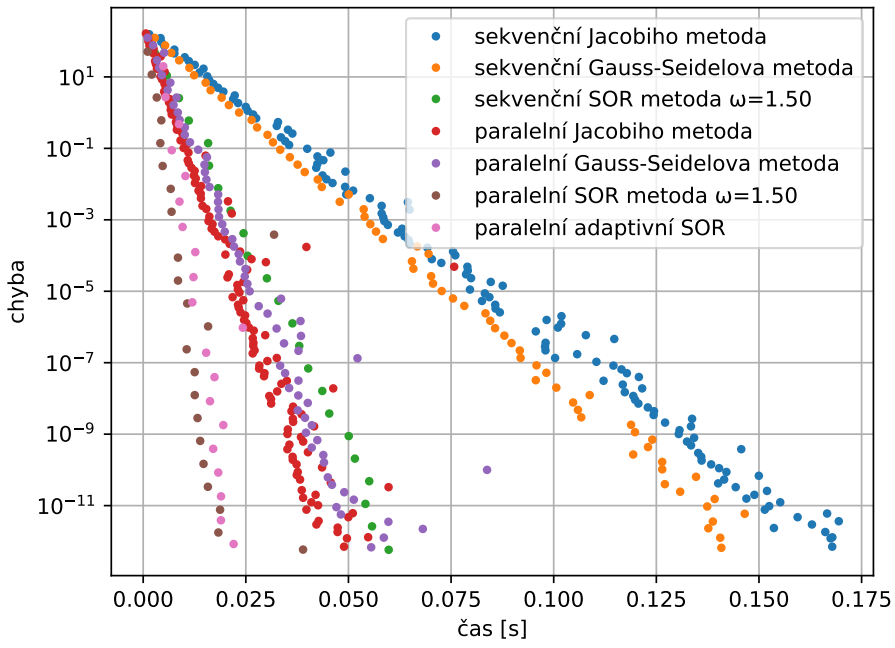
Tabulka 5.1: Rychlost konvergence na matici obstclae.

5.3 Porovnání na několika maticích najednou

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. Konvergencí jsem pro tento test zvolil 1. hodnotu, která má chybu menší jak 10^{-9}. Testoval jsem sekvenční a paralelní verze



Obrázek 5.2: Rychlost konvergence na matici obstclae podle počtu iterací.



Obrázek 5.3: Rychlost konvergence na matici obstclae podle času.

Jacobiho, modifikované Gauss-Seidelovu a SOR metodu. Adaptivní SOR jsem neměřil, protože je výpočetně příliš náročně určit přesnou iteraci konvergence.

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

Z tabulky 5.2 je vidět, že účinost paralelní verze Jacobiho je o dost vyšší. Vlákna si nezneplatnují vektor x v cache paměti a tak tam není tak vysoká nadbytečná synchronizace. Paralelní modifikovaná Gauss-Seidelova metoda musela průměrně vykonat o 1,03% víc iterací, modifikovaná SOR o 1,26%. To mě přijde v pořádku, když vemem v úvahu, že se celý výpočet zrychlí.

Pro zhodnocení zrychlení zkusím dosadit hodnoty do Amdahlova zákona a jako neznámou zvolím poměr, udavající paralelizovatelnou část. Budeme počítat hodnotu p ze vzorce, k je čtyři, protože měření jsem prováděl na čtyřjádrovém počítači, $S(k)$ je paralelní zrychlení z tabulky.

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

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

Pro Jacobiho metodu vychází $p = 90{,}5\%$, pro Gauss-Seidlovu vychází $p = 67{,}4\%$ a pro SOR metodu je $p = 76{,}1\%$. Zrychlení výpočtu je znatelné hlavně u Jacobiho metody. U ostatních metod zažíná být znát vliv větší synchronizace cache pamětí. Ty potřebují opakovaně synchronizovat vektor x i uprostřed iterace.

Název matice **Algoritmus**	**sekvenční verze** **iterací**	**čas [s]**	**paralelní verze** **iterací**	**čas [s]**	**parelelní** **zrychlení**
bcsstk21	3600 × 3600				
Jacobi	365 871	10,14	365 871	4,02	2,52
Gauss-Seidel	183 152	4,89	183 559	3,71	1,32
SOR $\omega = 1{,}9$	10 288	0,31	10 446	0,21	1,46
bodyy5	18589 × 18589				
Jacobi	32 448	5,12	32 448	1,51	3,40
Gauss-Seidel	13 523	3,00	13 567	1,30	2,31
SOR $\omega = 1{,}9$	664	0,18	683	0.07	2.53
nasa1824	1824 × 1824				
Jacobi	metoda diverguje				–
Gauss-Seidel	321 565	13,29	320 575	7,18	1,85
SOR $\omega = 1{,}9$	35 753	1,55	35 967	0,83	1,88
⋮					
Průměrný výsledek ze 7 matic					
Jacobi	124 956	10,35	124 956	3,35	3,11
Gauss-Seidel	87 434	6,11	87 399	3,13	2,02
SOR	7 452	0,51	7 524	0,25	2,33

Tabulka 5.2: Porovnání rychlosti konvergence na několika maticích.

5.4 Porovnání oproti jiné implementací

Jak je zmíněno v kapitole 3.4, tak 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, protože `PETSc` vyhazovalo z mě neznámého důvodu chybu. Gauss-Seidelovu metodu jsem zvládl počítat o 8,6% rychleji, SOR metodu o 11,7% pomaleji. Podrobnější výsledky jsou v tabulce 5.3.

Název matice / **Algoritmus**	**počet iterací**	**čas[s] moje**	**čas[s] PETSC**	**zrychlení**
494_bus	494 × 494			
Gauss-Seidel	501 233	1,19	1,41	17,9%
SOR $\omega = 1{,}9$	29 863	0,086	0,83	−2,6%
bcsstk21	3600 × 3600			
Gauss-Seidel	183 066	5,58	6,14	10,0%
SOR $\omega = 1{,}9$	10 290	0,35	0,35	−1,0%
bodyy6	19366 × 19366			
Gauss-Seidel	91 429	19,52	20,90	7,1%
SOR $\omega = 1{,}9$	5 157	1,43	1,18	−20,8%
⋮				
průměrný výsledek ze 7 matic				
Gauss-Seidel	113 089	4,22	4,57	8,6%
SOR	6 611	0,31	0,27	−11,7%

Tabulka 5.3: Porovnání rychlosti oproti PETSc na několika maticích.

Závěr

Tato práce se zabývá iterativníma algoritmama pro řešení soustavy rovnic. Metody fungují pouze na omezeném spektru matic. Postupně byli představeny tři metody, kde každá vylepšuje v nějakém smyslu tu předchozí. Navíc jsem implementoval svojí adaptivní metodu.

Cílem této práce bylo implementovat nastudované algoritmy sekvenčně a poté paralelně. U paralelní implementace bylo za úkol zjištění vlivu jednotlivých taktik plánování cyklů. To bylo splněno, řídké matice chytře rozděluji, aby každé vlákno mělo podobně velkou práci.

V poslední kapitole jsem podrobně měřil rychlost a zrychlení paralelní verze jednotlivých algoritmů. Výsledky jsem ověřoval na několika maticích. Jacobiho metodu se mě povedlo paralelizovat z 90%, Gauss-Sedelovu z 67% a SOR z 76%.

Příloha A

Seznam použitých zkratek

API Application Programming Interface / programové aplikační rozhraní

COO Coordinate list / Souřadnicový formát, vysvětleno v 2.6

CPU Central processing unit / Centrální procesorová jednotka

CSC Compressed sparse column / komprimované řídké sloupce, vysvětleno v 2.6

CSR Compressed sparse rows / Komprimované řídké řádky, vysvětleno v 2.6

GPU Graphics processing unit / Grafická výpočetní jednotka

SOR Successive over-relaxation / Opakovaná nadměrná relaxace, více v 2.4

Seznam použité literatury

2. Y., Saad; INDUSTRIAL, Society for; MATHEMATICS, Applied. *Iterative Methods for Sparse Linear Systems. Second edition*. Society for Industrial a Applied Mathematics, 2003. ISBN 9780898715347. Dostupné také z: https://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.

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

5. SCHMID, Patrick. *Dothan Over Netburst: Is The Pentium 4 A Dead End?* [online]. 2005 [cit. 2019-04-17]. Dostupné z: https://www.tomshardware.com/reviews/dothan-netburst,1041.html.

6. CORPORATION, NVIDIA. *NVIDIA TITAN RTX* [online]. 2019 [cit. 2019-04-17]. Dostupné z: https://www.nvidia.com/cs-cz/titan/titan-rtx/.

7. AMDAHL, Gene M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. Atlantic City, New Jersey: ACM, 1967, s. 483–485. AFIPS '67 (Spring). Dostupné z DOI: 10.1145/1465482.1465560.

8. MATTSON, Tim. *Hands-on Introduction to OpenMP* [online] [cit. 2019-04-17]. Dostupné z: https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf.

9. ŠPEH, Jaka. *OpenMP: For & Scheduling* [online]. 2006 [cit. 2019-04-17]. Dostupné z: http://jakascorner.com/blog/2016/06/omp-for-scheduling.htm.

10. HAYES, Bob. *Programming Languages Most Used and Recommended by Data Scientists* [online]. 2019 [cit. 2019-04-01]. Dostupné z: http://businessoverbroadway.com/2019/01/13/programming-languages-most-used-and-recommended-by-data-scientists/.

12. COMMUNITY, The SciPy. *Input and output (scipy.io) — SciPy v1.2.1 Reference Guide* [online]. 2017 [cit. 2019-04-17]. Dostupné z: https://stackoverflow.blog/2017/09/06/incredible-growth-python/.

14. SEAGER, Anne Greenbaum; Mark. *DLAP Sparse Linear Algebra Package* [online]. 2019 [cit. 2010-01-01]. Dostupné z: https://people.sc.fsu.edu/~jburkardt/f_src/dlap/dlap.html.

15. BALAY, Satish et al. *PETSc Web page*. 2019. Dostupné také z: http://www.mcs.anl.gov/petsc.

16. DALCIN, Lisandro D.; PAZ, Rodrigo R.; KLER, Pablo A.; COSIMO, Alejandro. Parallel distributed computing using Python. *Advances in Water Resources*. 2011, roč. 34, č. 9, s. 1124–1139. ISSN 0309-1708. Dostupné z DOI: https://doi.org/10.1016/j.advwatres.2011.04.013. New Computational Methods and Software Tools.

17. DAVIS, Timothy A.; HU, Yifan. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 2011, roč. 38, č. 1, s. 1:1–1:25. ISSN 0098-3500. Dostupné z DOI: 10.1145/2049662.2049663.

18. COMMUNITY, The SciPy. *Input and output (scipy.io) — SciPy v1.2.1 Reference Guide* [online] [cit. 2019-04-01]. Dostupné z: https://docs.scipy.org/doc/scipy/reference/io.html.

19. DAVIS, Tim; SCOTT KOLODZIEJ, Texas A&M University; YIFAN HU, Yahoo! Labs. *SuiteSparse Matrix Collection* [online]. [Cit. 2019-04-01]. Dostupné z: https://sparse.tamu.edu/.

Seznam použitých obrázků

3. CMGLEE. *Row and column major order* [online obrázek] [cit. 2019-04-01]. Dostupné z: https://commons.wikimedia.org/wiki/File:Row_and_column_major_order.svg Ukázka rozdílu mezi CSR a CSC formátem.

11. STACKOVERFLOW. *Row and column major order* [online obrázek] [cit. 2019-04-17]. Dostupné z: 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 Graf vybraných jazyků na StackOverflow.

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 obsahující 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