FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Effective solver of linear inequalities:
Student:	Tomáš Janecký
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2019/20

Instructions

1) Study Conflict Resolution algorithm (CRA) for solution of system of linear inequalities [1].
2) Discuss parallelization of the algorithm.
3) Implement parallel (multithreaded) linear inequalities solver with CRA using C++ and OpenMP API [2].
4) Measure program's execution times for a set of testing systems from public repositories and compare performance for dense and sparse representation of a systems of inequalities.

References

[1] Korovin, K., Tsiskaridze, N., Voronkov, A.:Conflict Resolution,15th International Conference on Principles and Practice of Constraint Programming,2009, pp. 509-523, Lecture Notes in Computer Science, 5732
[2] B. Chapman, G. Jost, R. Pas: Using OpenMP: Portable Shared Memory Parallel Programming, Amazon, ISBN 978-0262533027

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

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

Prague January 10, 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Efektivní řešič lineárních nerovnic

Tomáš Janecký

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

2. června 2020

Poděkování

Rád bych poděkoval mému vedoucímu práce doc. Ing. IvanuŠimečkovi, Ph.D. za ochotu a pomoc při vypracování mé bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, 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 2. června 2020 . . . . . .

Odkaz na tuto práci

Janecký, Tomáš. *Efektivní řešič lineárních nerovnic*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato práce se zabývá implementací algoritmu pro řešení soustav lineárních nerovnic pomocí řešení konfliktů. Algoritmus je implementován v C++17, jeho paralerizace je docílena pomocí knihovny OpenMP.

Klíčová slova Conflict Resolution Algorithm, soustava nerovnic, C++, řešič soustav, řídké matice, husté matice, paralelizace, OpenMP

Abstract

This thesis deals with implementation of Conflict resolution algorithm, which solves systems of linear inequalities. Algorithm is implemented in C++17 and is parallelized using OpenMP.

Keywords Conflict Resolution Algorithm, system of linear inequalities, C++, solver of inequalities, sparse matrices, dense matricies, parallelization, OpenMP

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza a návrh **5**
2.1 Základní definice 5
2.1.1 Množiny 5
2.1.2 Lineární omezení 5
2.1.3 Uspořádání proměnných 5
2.1.4 Úroveň(Level) nerovnice 5
2.1.5 Přiřazení 6
2.1.6 Normalizace 6
2.1.7 Soustava lineárních nerovnic 6
2.1.8 Řešení soustavy lineárních nerovnic 6
2.1.9 Řídké a husté matice 6
2.1.10 Dolní mez nerovnice 7
2.1.11 Horní mez nerovnice 7
2.1.12 Dolní mez množiny nerovnic 7
2.1.13 Horní mez množiny nerovnic 7
2.1.14 Mezní interval 7
2.1.15 k-konflikt 7
2.1.16 Pravidlo pro změnu přiřazení 7
2.1.17 Pravidlo pro vyřešení konfliktu 8
2.2 Algoritmus 8
2.2.1 Volba algoritmu 8
2.2.2 Idea algoritmu 8
2.2.3 Převod nerovnice na normalizovaný tvar 8
2.2.4 Conflict Resolution Algorithm 9
2.3 Složitost CRA 9

2.4 Příklad . . . 9
2.4.1 Zadání . . . 10
2.4.2 Výpočet . . . 10
2.5 Detailní popis algoritmu . . . 11

3 Realizace **15**
3.1 Uložení dat . . . 15
3.1.1 Reprezentace čísel . . . 15
3.1.1.1 Specializace šablon . . . 16
3.1.2 Reprezentace soustvy lineárních nerovnic . . . 16
3.2 Zdroj matic . . . 17
3.3 Formát vstupu . . . 17
3.3.1 Příklad vstupu . . . 17
3.4 Seřazení proměnných . . . 18
3.5 Strategie pro změnu přiřazení . . . 18
3.6 Paralelizace . . . 18
3.6.1 Rozhraní OpenMP . . . 19
3.6.1.1 Paralerní blok . . . 19
3.6.1.2 Paralerní for cyklus . . . 19
3.6.1.3 Kritická sekce . . . 19
3.6.2 Implementační detaily . . . 20
3.6.2.1 For cykly . . . 20
3.6.2.2 Přerušení paralelního běhu for cyklů . . . 21
3.7 Paralelizace CRA . . . 22
3.7.1 Kontrola přiřazení . . . 22
3.7.2 Hledání konfliktů . . . 23
3.7.3 Strategie pro hledání konfliktů . . . 23
3.8 Normalizace . . . 24
3.9 Optimalizace . . . 24
3.9.1 Přeuspořádní strojového kódu . . . 24
3.9.1.1 Porovnání vyprodukovaného strojového kódu v x86-x64 . . . 25

4 Vyhodnocení **27**
4.1 Velikost matic . . . 27
4.2 Použitý hardware a software . . . 27
4.3 Vstupní data . . . 27
4.4 Použité optimalizace . . . 27
4.5 Sekvenční řešič . . . 28
4.6 Paralelní řešič . . . 28
4.7 Grafy pro matice 25x25 . . . 29
4.7.1 Řídké matice . . . 29
4.7.2 Husté matice . . . 30
4.8 Porovnání s jiným řešičem . . . 31

4.8.1 SciPy 31
4.8.2 Wolfram Mathematica 31

Závěr **33**

Literatura **35**

A Seznam použitých zkratek **37**

B Obsah přiloženého CD **39**

Seznam obrázků

4.1 Naměřené hodnoty pro řídké matice. 29
4.2 Naměřené hodnoty pro řídké matice. 30
4.3 Naměřené hodnoty pro husté matice. 30
4.4 Naměřené hodnoty pro husté matice. 31

Seznam tabulek

4.1 Zaplnění matic. 28
4.2 Porovnání mého řešiče s Mathematicou. 32

Úvod

Nalezení řešení soustavy rovnic/nerovnic je jedním z nejběžnějších matematických problémů, které je třeba řešit. Pokud ale máme dostatečný výpočetní výkon a přesné počítače, tak můžeme vždy nalézt řešení, což se rozhodně nedá říci o všech matematických problémech. Hledání řešení soustav rovnic/nerovnic je hlavním cílem lineárního programování.

Pro řešení soustav lineárních nerovnic, existuje několik známých metod jako například Fourier-Motzkinova metoda, nebo Černikovův algoritmus, které si jsou velmi podobné. Tato práce se ale zabývá implementací Conflict Resolution Algorithm.[1]

Algoritmus byl publikován v roce 2009 Konstantinem Korovinem, Nestanem Tsiskaridzem a Andrejem Voronkovem.[1] Tento algoritmus by měl být v některých případech i o řád efektivnější, než předchozí zmíněné algoritmy.

V první kapitole představím problém a zadefinuji pojmy pro snadnější porozumění algoritmu. V druhé kapitole popíši implementaci algoritmu v C++ a jeho paralelizaci pomocí OpenMP.[6] V závěrečné kapitole se zabývám použitím programu pro řešení soustav linárch nerovnic a nalezením řešení soustav lineárních nerovnic, které jseu reprezentovány řídkými maticemi. Následně prezentuji naměřené výsledky.

Kapitola 1

Cíl práce

Cílem práce je uvést čtenáře do problematiky algortimu[1] pro řešení soustav lineárních nerovnic. Implementovat ho, provést jeho paralelizaci pomocí OpenMP[6] a změřit jeho výkon na řídkých a hustých maticích.

Kapitola 2

Analýza a návrh

V této kapitole zadefinuji pojmy použité v tomto textu za použití informací z [1] . Dále popíši, jak algoritmus funguje.

2.1 Základní definice

2.1.1 Množiny

Množinu racionálních čílsel označíme jako $\mathbb{Q}$ a množinu přirozených čísel jako $\mathbb{N}$. Pro každé n v tomto textu platí $n \in \mathbb{N}$. Dále konečnou množinu proměnných $\{x_1, \ldots, x_n\}$ označíme jako X.

2.1.2 Lineární omezení

Jako linearní omezení nad Xoznačíme

$$a_1x_1 + \ldots + a_nx_n + b \diamond 0$$

kde $a_i \in \mathbb{Q}$ pro $1 \leq i \leq n$, $b \in \mathbb{Q}$, nebo je v jednom z tvarů $\top$,nebo $\bot$. A platí $\diamond \in \{\geq, >, =, \neq\}$. Dále v textu budeme uvažovat pouze případ kdy $\diamond \in \{\geq\}$. Budeme je tedy nazývat lineární nerovnice.

2.1.3 Uspořádání proměnných

Nechť $\succ$ je uspořádání nad X. Bez újmy na obecnosti můžeme předpokládat:

$$x_n \succ x_{n-1} \succ \ldots \succ x_1$$

2.1.4 Úroveň(Level) nerovnice

Nechť A je nerovnice v normalizovaném tvaru nad X s uspořádánim $\succ$, potom jako úroveň(level) označíme maximální pro maximální proměnnou, jejíž koeficient není roven nule.

$$level(a_1x_1 + \ldots + a_nx_n + b \geq 0) = n; a_n \neq 0$$

2.1.5 Přiřazení

Přiřazení σ nad množinou proměnných X definujeme jako zobrazení $X \to \mathbb{Q}$. Při daném přiřazení σ, proměnné $x \in X$ a hodnotě $v \in \mathbb{Q}$, označíme σ_x^v jako přiřazení získané ze σ změnou hodnoty x na v beze změny přiřazení ostatních proměnných.
Například:

$$x_1 \leftarrow 2, x_2 \leftarrow 0, x_3 \leftarrow -1 \ldots$$

$$\sigma_{x_2}^7$$

$$x_1 \leftarrow 2, x_2 \leftarrow 7, x_3 \leftarrow -1 \ldots$$

Získaní výsledku z levé strany nervonice A pomocí přiřazení σ:

$$A(\sigma)$$

2.1.6 Normalizace

Nerovnice je normalizována, pokud je v jednom z náseledujících tvarů:

$$\{\top, \bot, a_1x_1 + \ldots + a_{n-1}x_{n-1} + x_n + b \geq 0, a_1x_1 + \ldots + a_{n-1}x_{n-1} - x_n + b \geq 0\}$$

Například $2x_1 + x_3 + 1 \geq 0$ je normalizována, $2x_1 - x_3 + 6x_4 \geq 0$ není.
Dále budeme uvažovat pouze normalizováné nerovnice.

2.1.7 Soustava lineárních nerovnic

Soustava lineárních nerovnic lze zapsat:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & x_n + & b_1 \geq & 0 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & x_n + & b_2 \geq & 0 \\ \vdots & & \vdots & & \ddots & & \vdots & \vdots & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \ldots & + & x_n + & b_m \geq & 0 \end{array}$$

Množinu všech lineárních nerovnic (všechny úrovně) budeme označovat jako S. Množinu soustav úrovně pouze k budeme označovat S_k.

2.1.8 Řešení soustavy lineárních nerovnic

Přiřazení σ soustavy lineárních nerovnic je jejím řešením, pokud pro každou nerovnic $A \in S$ platí $A(\sigma) > 0$.

2.1.9 Řídké a husté matice

Matice označíme za řídkou, pokud většina jejich prvků je rovna nule. V opačném případě hovoříme o husté matici.

2.1.10 Dolní mez nerovnice

Nechť A je nerovnice a σ nějaké její přiřazení a n je její level. Potom dolní mez je rovna:

$$lower(A, \sigma) = \begin{cases} -A(\sigma) & ; a_n = 1 \\ -\infty & ; a_n = -1 \end{cases}$$

2.1.11 Horní mez nerovnice

Nechť A je nerovnice a σ nějaké její přiřazení a n je její level. Potom horní mez je rovna:

$$upper(A, \sigma) = \begin{cases} A(\sigma) & ; a_n = 1 \\ \infty & ; a_n = -1 \end{cases}$$

2.1.12 Dolní mez množiny nerovnic

Nechť S_k je množina nerovnic levelu k a σ přiřazení potom dolní mez množiny je rovna:

$$U = max\{lower(A, \sigma) | A \in S_k\}$$

2.1.13 Horní mez množiny nerovnic

Nechť S_k je množina nerovnic levelu k a σ přiřazení potom horní mez množiny je rovna:

$$L = min\{upper(A, \sigma) | A \in S_k\}$$

2.1.14 Mezní interval

Nechť S_k je množina nerovnic $L(S_k, \sigma)$ je její dolní mez a $U(S_k, \sigma)$ horní mez potom mezní iterval:

$$I = \langle L(S_k, \sigma), U(S_k, \sigma) \rangle$$

2.1.15 k-konflikt

Nechť $A, B \in S_k$, σ je nějaké přiřazení dále platí $(a_k \in A) = 1$ a $(a_k \in B) = -1$. Potom A a B jsou v k-konfliktu pokud

$$A(\sigma) + B(\sigma) < 0$$

2.1.16 Pravidlo pro změnu přiřazení

$$(S, \sigma) \Rightarrow (S, \sigma^v_{x_k})$$

1. σ vyhovuje všem nerovnicím v S_k pro všechny úrovně $0, \ldots, k-1$
2. σ nevyhovuje alespoň jedné nerovnici v S úrovně k
3. σ vyhovuje všem nerovnicím v S_k

2.1.17 Pravidlo pro vyřešení konfliktu

Nechť $A, B \in S$ a zároveň jsou v k-konfliktu, potom konflikt vyřešíme jejich sečtením A, B a přidáním nově vzniklé nerovnice do S.

$$S \leftarrow S \cup (A + B)$$

2.2 Algoritmus

2.2.1 Volba algoritmu

Nalezení řešení soustavy lineárních nerovnic je úlohou lineárního programování. Základní metodou pro vyřešení soustavy nerovnic je Fourier-Motzkinova eliminace .[7] Další metodou je Černikovův algorimus, který je vylepšením Fourier-Motzknovy eliminace.[1] Conflict Resolution Algorithm (CRA), který v rámci této práce implementuji, by měl být v praktických případech řádově efektivnější.[1]

2.2.2 Idea algoritmu

Algoritmus[1] dostane na vstupu soustavu lineárních nerovnic a jednotlivým proměnným $x_1, \ldots, x_n$ přiřadí číslo (přiřazení σ 2.1.5). Na počátečním přiřazení z pohledu algortimu nezáleží. Pokud soustava obsahuje kontradikci (např. $-2 > 0$), tak již lze rohodnou o její nesplnitelnosti v této počáteční fázi. Algoritmus dále přiřazení aktualizuje, dokud nenajde řešení, nebo zjistí, že řešení neexistuje. Aktualizace řešení algoritmus docílí nejdříve nalezením k-konfliktu (pokud existuje), přidáním nové nerovnice na základě pravidla pro vyřešení konfliktu 2.1.17 a následně aktualizuje přiřazení pomocí pravidla pro změnu přiřazení 2.1.16.

Takto algoritmus postupuje po jednotlivých úrovních S_k (s případnými návraty do nižších úrovní při přidání nové nerovnice), dokud nevyřeší všechny konflikty a najde přiřazení, které vyhovuje všem nerovnicím, nebo rozhodne o neřešitelnosti soustavy.

2.2.3 Převod nerovnice na normalizovaný tvar

Nerovnici snadno převedeme na normalizovaný tvar vydělením všech koeficientů $|a_n|$.

$$\begin{aligned} &normalize(a_1x_1 + \ldots + a_{n-1}x_{n-1} + a_nx_n + b \geq 0) = \\ &\frac{a_1}{|a_n|}x_1 + \ldots + \frac{a_{n-1}}{|a_n|}x_{n-1} + \frac{a_n}{|a_n|}x_n + \frac{b}{|a_n|} \geq 0 \end{aligned}$$

[1] Anglický název je Conflict Resolution Algorithm.

2.2.4 Conflict Resolution Algorithm

Algoritmus předpokládá na vstupu normalizováné nerovnice a po sečtení dvou nerovnic je nutné výsledek také normalizovat.

Data: Množina S lineárních nerovnic.
Result: Řešení S, nebo rozhodnutí o neřesitelnosti.
if $\perp \in S$ **then**
  return *"nemá řešení"*
end
$\sigma \leftarrow$ "libovolné přiřazení"
$k \leftarrow 1$
while $k \leq n$ **do**
  if $\neg S_k(\sigma)$ **then**
    `// přiřazení nevyhovuje úrovni`
    while *k-konflikt:*
     $(a_{1i}x_1 + \ldots + x_k + b_i \geq 0, a_{1j}x_1 + \ldots - x_k + b_j \geq 0) \in (S_k, \sigma)$
     do
      $A \leftarrow a_{1i}x_1 + \ldots + x_k + b_i \geq 0$ `// v konfliktu s` B
      $B \leftarrow a_{1j}x_1 + \ldots - x_k + b_j \geq 0$ `// v konfliktu s` A
      $S \leftarrow S \cup (A + B)$ `// aplikace resoluce konfliktu`
      $k \leftarrow level(A + B)$
      if $k = 0$ **then**
        return *"nemá řešení"*
      end
    end
    $\sigma \leftarrow \sigma^v_{x_k}(S_k, \sigma)$ `// nové přiřazení pro` x_k
    `// na základě horní a dolní meze`
  end
  $k \leftarrow k + 1$
end
return σ

Algoritmus 1: Conflict resolution algorithm (CRA).

2.3 Složitost CRA

Složitost CRA je stejná jako Fourier-Motzkinova algoritmu, tedy exponenciální. Podle [1] má v praktických případech lepší složitost, někdy i rozdílem řádu, protože nevytváří zbytečné nové nerovnice.

2.4 Příklad

Následuje příklad použití algoritmu.

2.4.1 Zadání

$$\begin{aligned} -2x_1 + x_2 + x_3 - 2 &\geq 0\ (r_1) \\ x_1 - x_2 - x_3 &\geq 0\ (r_2) \\ x_1 - 2x_2 - 2 &\geq 0\ (r_3) \\ 4 &\geq 0\ (r_4) \end{aligned}$$

2.4.2 Výpočet

Předpokládáme přiřazení $\sigma \leftarrow \{x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 0\}$ a uspořádání $x_3 \succ x_2 \succ x_1$.

1. V S_0 je pouze r_4, což není kontradikce, můžeme pokračovat. Inkrementujeme k.
2. V S_1 je prázdná množina. Tudíž v ní nemůže existovat konflikt a přiřazení musí vyhovovat. Inkrementujeme k.
3. V S_2 se nachází r_3, sama se sebou nemůže být v konfliktu. Přiřazení σ ale nevyhovuje r_3 ($1 \cdot 0 - 2 \cdot 0 + 0 \cdot 0 - 2 \not\geq 0$).
   - Vypočteme mezní interval pro $k = 2$ interval $I = (-\infty, -1\rangle$
   - aktualizujeme přiřazení $x_2 \leftarrow -2$ na základě vypočteného intervalu, neboli $\sigma_{x_2}^{-2}$

   Inkrementujeme k.
4. V S_3 r_2 přiřazení vyhovuje, ale r_1 ne, jsou spolu v konfliktu.
   - sečtením r_1 a r_2, vnikne nová nerovnice $-x_1 - 2 \geq 0$
   - je již v normalizovaném tvaru
   - přidáme ji do S_1 jako r_5

   k nastavíme na $level(r_5)$ tudíž 1
5. Vypočteme mezní interval pro $k = 1$ a podle něj aktualizujeme přiřazení $x_1 \leftarrow -4$. Inkrementujeme k.
6. Nové přiřazení nevyhovuje r_3.
   - Vypočteme mezní interval pro $k = 2$, který je $I = (-\infty, -3\rangle$
   - Aktualizujeme přiřazení $x \leftarrow -6$

   Inkrementujeme k
7. Přiřazení již vyhovuje všem nerovnicím, tudíž konečné řešení je:

$$\{x_1 \leftarrow -4, x_2 \leftarrow -6, x_3 \leftarrow 0\}$$

2.5 Detailní popis algoritmu

Tato podkapitola detailně popisuje jednotlivé kroky algoritmu a jakým způsobem jsou provedeny.

Výraz *nerovnice*[k] představuje množinu nerovnic z S, které mají úroveň k, nebo koeficient dané nerovnice v závislosti na kontextu. $[a, b]$ je interval od a do b.

Function *normalizuj(nerovnice)* **begin**
  foreach *(n ∈ nerovnice)* **do**
    normalize(n) `// //viz. 2. kapitola`
  end
end

Algoritmus 2: Funkce normalizuje všechny nerovnice v množině.

Function *aktualizuj-prirazeni(interval,level,prirazeni)* **begin**
  prirazeni[level] ← nahodne-cislo(interval)
end

Algoritmus 3: Aktualizuje přiřazení na náhodné číslo ze zadaného intervalu pro daný level. (Jen jedna z možných strategií.)

Function *obsahuje-konflikt(prirazeni,level,nerovnice)* **begin**
  d-mez ← $-\infty$
  h-mez ← ∞
  foreach $n \in$ *nerovnice[level]* **do**
    d-mez ← max(d-mez,dolni-mez(prirazeni,n))
    h-mez ← min(h-mez,horni-mez(prirazeni,n)))
  end
  return *d-mez* > *h-mez*
end

Algoritmus 4: Vyhledá konflikt v zadaném levelu(množině) nerovnic.

```
Function vyres-konflikt(prirazeni,level,nerovnice) begin
    foreach (a,b ∈ nerovnice) do
        d-mez ← dolni-mez(prirazeni,a)
        h-mez ← horni-mez(prirazeni,b)
        if d-mez > h-mez then
            c ← a + b
            normalize(c)
            level ← level(c)
            nerovnice ← nerovnice ∪ c
        end
    end
end
```

Algoritmus 5: Vyřeší konflikt mezi dvěma nerovnicemi a a b.

```
Function aktualizuj-interval(nerovnice,prirazeni,level) begin
    return [dolni-mez-urovne(nerovnice,prirazeni,level),
      horni-mez-urovne(nerovnice,prirazeni,level)]
end
```

Algoritmus 6: Na základě mezí pro daný level nerovnic vrátí funkce nový interval.

```
Function obsahuje-kontradikci(nerovnice) begin
    foreach (n ∈ nerovnice[0]) do
        if n[0] < 0 then
            return true
        end
    end
    return false
end
```

Algoritmus 7: Zjistí, jestli množina nerovnic, která obsahuje pouze konstantní členy, obsahuje záporné číslo.

```
Function dolni-mez(prirazeni,nerovnice) begin
    if nerovnice[level(nerovnice)] = −1 then
        return −∞
    end
    hodnota ← 0
    foreach (p,n in prirazeni,nerovnice) do
        hodnota ← hodnota + p · n
    end
end
```

Algoritmus 8: Vypočítá dolní mez jedné nerovnice.

Function *horni-mez(prirazeni,nerovnice)* **begin**
  if $nerovnice[level(nerovnice)] = 1$ **then**
    return ∞
  end
  hodnota $\leftarrow 0$
  foreach *(p,n in prirazeni,nerovnice)* **do**
    hodnota $\leftarrow$ hodnota + p $\cdot$ n
  end
end

Algoritmus 9: Vypočítá horní mez jedné nerovnice.

Function *horni-mez-urovne(nerovnice,prirazeni,level)* **begin**
  hodnota $\leftarrow \infty$
  foreach *(n* $\in$ *nerovnice[level])* **do**
    hodnota $\leftarrow$ min(hodnota,horni-mez(prirazeni,n))
  end
  return *hodnota*
end

Algoritmus 10: Vypočítá horní mez množiny nerovnic.

Function *dolni-mez-urovne(nerovnice,prirazeni,level)* **begin**
  hodnota $\leftarrow -\infty$
  foreach *(n* $\in$ *nerovnice[level])* **do**
    hodnota $\leftarrow$ max(hodnota,horni-mez(prirazeni,n))
  end
  return *hodnota*
end

Algoritmus 11: Vypočítá dolní mez množiny nerovnic.

```
Function main(nerovnice) begin
    normalizuj(nerovnice)
    level ← 0
    max-level ← max-level(nerovnice)
    prirazeni ← (0, 0, 0, ...)
    if obsahuje-kontradikci(nerovnce[0]) then
        return "nema reseni"
    end
    while level < max-level do
        while ¬vyhovuje-urovni(nerovnice,level,prirazeni) do
            if obsahuje-konflikt(prirazeni,level,nerovnice) then
                vyres-konflikt(prirazeni,level,nerovnice)
            end
            if level = 0 then
                return "nema reseni"
            end
            interval ← aktualizuj-interval(level,nerovnice,prirazeni)
            aktualizuj-prirazeni(interval,level,prirazeni)
        end
        level ← level +1
    end
end
```

Algoritmus 12: Conflict Resolution Algorithm detailně.

Kapitola 3

Realizace

V této kapitole popíši implementaci řešiče, způsob uložení čísel a soustavy. Dále popíši paralelizaci algorimu a detaily jeho fungování. Pro implementaci je použito C++17 a pro paralelizaci OpenMP.[6] C++ je vhodný jazyk pro náročné výpočty, protože díky své nízkoúrovňové povaze lze dosáhnout mnoha optimalizací.

3.1 Uložení dat

3.1.1 Reprezentace čísel

Pro uložení čísel jsem použil boost rational [3], třídu z knihovny boost [4], distribuovavou pod licencí boost.[5] Jedná se o šablonu s jedním parametrem, který je použit na reprezentaci čitatele a jmenovatele a jako další jsem použil třídu gmp_rational [18], která je frontend pro knihovnu GNU Multiple Precision Arithmetic Library [17].

Moje implementace používá pro tyto třídy obalovací třídu s rozhraním a reprezentací:

Listing 3.1: Třída obalující číselnou reprezentaci.

```
class FractionWrap {
private:
    boost::rational<int64_t> val_;  //ruzne
public:
    FractionWrap();

    FractionWrap &operator=(FractionWrap other);

    bool operator<(const FractionWrap &other) const;

    bool operator<=(const FractionWrap &other) const;
```

```
    bool operator>=(const FractionWrap &other) const;

    bool operator==(const FractionWrap &other) const;

   //...
};
```

Dále je pomocí specializace šablon docíleno, že stačí na jednom místě ve zdrojovém kódu změnit typ a pro reprezentaci je použit daný typ. Stačí pouze, pokud dodržuje zadané rozhraní. Takto se lze vyhnout dynamické vazbě.

3.1.1.1 Specializace šablon

Tato kapitola popisuje dedukci typu pro uložení čísel na základě specializace šablon.

Listing 3.2: Dedukce použití typu na základě specializace šablon.

```
using Type = FractionWrap; //vyber typu

template<typename T>
struct NumberType;

template<>//specializace sablony
struct NumberType<FractionWrap> {
    using type = FractionWrap;

    constexpr NumberType() = default;
    //...
};

//...

constexpr NumberType<Type> f;

//alias pro cislo
using NUMBER = NumberType<Type>::type;
```

3.1.2 Reprezentace soustvy lineárních nerovnic

Pro reprezentaci jednotlivých nerovnic je použit std::vector z knihovny STL tak, že na nultém indexu je uložen konstantní člen a na vyšších jsou uloženy jednotivé koeficienty lineární nerovnice.

Jednotlivé nerovnice stejné úrovně jsou spolu uloženy také ve vektoru a tyto vektory jsou podle úrovně množiny uloženy také ve vektoru. Kde na nultém indexu je množina nerovnic, které mají pouze konstantní člen atd. Výsledkem tedy je trojrozměrné pole.

Listing 3.3: Trojrozměrné pole čísel.

```
std::vector<
        std::vector<
                std::vector<NUMBER>>> constraints;
```

Stejným způsobem je uložené přiřazení.

Listing 3.4: Přiřazení.

```
std::vector<NUMBER> assignment;
```

3.2 Zdroj matic

Jako zdroj zadání jsem použil [11]. Jedná se o repozitář obsahující řídké matice z různých odvětví a dále poskytuje nástroj pro generování nových matic.

Alternarivně, kvůli technickým problémům [11], jsem generoval matice pro benchmark pomocí SciPy[15].

3.3 Formát vstupu

Použité matice jsou Matrix Market formátu. Jedná se o textový formát výhodný pro uložení řídkých matic, protože ukládá pouze nenulové pozice, ostatní jsou implicitně nuly. Komentáře v souborech jsou uvozeny %, mohou být použity pouze jako hlavička. Pozice elementů v matici zičínají od 1.

3.3.1 Příklad vstupu

Pro matici:

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 4 & 0 \\ 7 & 0 & 2 \end{pmatrix}$$

je její odpovídající uložení:

Listing 3.5: Uložení matice ve formátu Matrix Market.

```
%Toto je komentar
%pocet radku, sloupcu a nenulovych elementu
 3 3 5
    1   1  1.000e+00
    1   3  3.000e+00
    2   2  4.000e+00
    3   1  7.000e+00
    3   3  2.000e+00
```

3.4 Seřazení proměnných

Implementace algoritmu ponecháva pořadí proměnných stejné, jako byly uloženy ve vstupní matici. Tedy zleva doprava pro 3.3.1 pro první řádek.

$$1 \cdot x_2 + 0 \cdot x_1 + 3 \geq 0$$

$$x_2 \succ x_1$$

$$\dots$$

3.5 Strategie pro změnu přiřazení

Pro změnu přiřazení na základě nově vypočteného mezního intervalu, také existuje několik strategií. Různé strategie diskutované v [2] jsou:

- minimum/maximum intervalu
- náhodná hodnota z intervalu
- střed intervalu
- nejblíže k binárnímu středu intervalu

Pro svoji implementaci jsem si zvolil výběr středu intervalu, s výjimkou, kdy právě jedna mez je nekonečno, potom je nové přiřazení o absolutní hodnotu druhé meze v intervalu pro snadnější implementaci a debugování.

3.6 Paralelizace

Jak již zmíněno, paralelizace je dosaženo pomocí knihovny OpenMP.[6]

3.6.1 Rozhraní OpenMP

Interakce s knihovnou je založena na direktivách preprocesoru *#pragma*, pomocí kterých lze jednoduše dosáhnou paralelizace kódu. Jedná se o multiplatformní knihovnu, takže výsledný kód je přenositelný. Pokud kompilátor překládá zdrojový kód s *#pragma* direktivami, které nepodporuje, tak je ignoruje. Tedy v případě OpenMP by výsledný program běžel pouze v jednom vlákně, ale výstup by byl stejný.

3.6.1.1 Paralerní blok

Blok paralelizovaného kódu musí být uvozen makrem *#pragma omp parallel*.

Listing 3.6: OpenMp paralelní blok.

```
 #pragma omp parallel
{
    //...
}
```

Alternativně lze přidávat parametr *parallel* ke všem direktivám OpenMP a tento blok vynechat.

3.6.1.2 Paralerní for cyklus

Běh for cyklu lze paralelizovat pomocí makra *#pragma omp for*.

Listing 3.7: Příklad paralerizovaného for cyklu.

```
 #pragma omp parallel
{
     #pragma omp for
    for(size_t i = 0 ; i < 100000 ; i++){
        array[i] = 0
    }
 }
```

Při běhu programu knihovna rozdělí for cyklus mezi jednotlivá vlákna

3.6.1.3 Kritická sekce

Makro *#critical* uvozuje blok kódu uvnitř paralelního bloku, který může být vykonáván pouze jedním vláknem, ne více vlákny najednou. Jedná se tedy o podobný způsob synchronizace jako jsou mutexy.

Listing 3.8: Kritická sekce v OpenMP.

```
#pragma omp parallel
{
//...

    #pragma omp critical
    {
        flag = false;
    }
//...
}
```

3.6.2 Implementační detaily

3.6.2.1 For cykly

Pro paralelizaci for cyklů pomocí OpenMP je nutné, aby byly v kanonickém tvaru. Pro cykly ve tvaru:

Listing 3.9: Range based for smyčka.

```
for(auto& it : container){
    //...
}
```

Musí platit, že je lze rozvinout do kanonického tvaru a kontejner musí poskytovat iterátor s náhodným přistupem.[13]

Listing 3.10: Příklady použití for cyklů s OpenMP.

```
std::vector<int> v;
std::list<int> l;

//...

#pragma omp parallel{

    #pragma omp for
    for(auto& it v){
        //OK
    }

    #pragma omp for
    for(size_t i = 0 ; i < v.size() ; i++){
        //OK
    }
```

```
    #pragma omp for
    for(auto& it : l){
        //chyba, list neposkytuje
        //iterator s nahodnym pristupem
    }
}
```

3.6.2.2 Přerušení paralelního běhu for cyklů

Pro paralelní hledání jehly v kupce sena může být příhodné, pokud jedno vlákno nalezne jehlu, přerušit běh ostatních vláken a pokračovat dále ve výpočtu. Na druhou stranu každé vlákno musí kontrolovat signál o svém ukončení, což přidává režii, tudíž může v konečném důsledku výpočet trvat déle.

Přerušení for cyklu v OpenMP je dosaženo pomocí maker *#pragma omp cancel* pro signalizaci ukončení a *#pragma omp cancellation point for* pro kontrolu, jestli dané vlákno má ukončit.

Příklad:

Listing 3.11: Příklad přerušení paralerního cyklu.

```
int matrix[20][20];
int  i_f = -1, j_f = -1;
//...
#pragma omp parallel
   {
        //paralerizovan je pouze prvni for cyklus
        #pragma omp for
        for (int i = 0 ; i < 20 ; ++i) {
            int sum = 0;
            for (int j = 0; j < 20; ++j) {

            }
            if (sum > 0) {
                #pragma omp critical
                {
                    i_f = i;
                    j_f = j;
                }
                //nema cenu jiz dale pokracovat
                //ukonci prvni for cyklus
                #pragma omp cancel for
            }
        //kontrola, jestli ma jiz vlakno skoncit
        #pragma omp cancellation point for
```

```
        }
    }
```

Pro správné fungování přerušení je nutné nastavit proměnnou prostředí.

Listing 3.12: Nastavení proměnné prostředí OpenMP.

```
$ export OMP_CANCELLATION=true
```

3.7 Paralelizace CRA

V programu jsem paralelizoval dvě jeho hlávní části je to.

3.7.1 Kontrola přiřazení

Funkce funguje velmi jednoduše. Na začatku nastaví flag indikující, že přiřazení vyhovuje levelu, následuje pomocí OpenMP paralerizovaný for cyklus, kdy jednotlivá vlakna určují hodnotu nerovnic s daným přiřazením. První vlákno, které objeví nerovnici, jež přiřazení nevyhovuje nastaví flag a ukončí ostatní vlákna[2]

Listing 3.13: Paralelizovaná funkce pro kontrolu přiřazení.

```
bool assignment_satisfies_level_pl(
    const std::vector<NUMBER> &assignments,
    size_t curr_level,
    const ConstraintSet &constraints) {

    bool satisfies = true;

#pragma omp parallel
    {
#pragma omp for
        for (auto &it_level:constraints[curr_level]) {
            NUMBER result = 0;
            auto it_assign = assignments.begin();
            for (auto &it_constraint : it_level) {
                result += it_constraint * *it_assign;
                ++it_assign;
            }
            if (result < 0) {
            #pragma omp critical
                {
```

[2]Ve skutečnosti OpenMP nezaručuje, že se jedná o první vlákno, které dosáhlo kritické sekce.[13]

```
                    satisfies = false;
                }
                #pragma omp cancel for
            }
#pragma omp cancellation point for
        }
    }
    return satisfies;
}
```

Sekvenční funkce funguje na stejné principu, s tím rozdílem, že při nalezení nevyhovující nerovnice se okamžitě vrátí.

3.7.2 Hledání konfliktů

3.7.3 Strategie pro hledání konfliktů

Oba případy jsem paralerizoval způsobem hledání jehly v kupce sena. Tedy nalezení prvního konfliktu, nebo nalezení první nerovnice, které přiřazení nevyhovuje. Zatímco hledání prvního nevyhovujícího přiřazení jiným způsobem řešit nelze, hledání konfliktů je možné řešit více způsoby. Autoři CRA[1] v [2] diskutují další možnosti vybrání konfliktu k vyřešení. Jsou to:

- první nalezený konflikt (strategie, kterou jsem zvolil)
- náhodný výběr konfliktu (přístup založený na heuristice)
- maximální konflikt
- relaxační metoda

Hledání konfliktů v mé implementaci funguje velmi podobně, jako kontrola přiřazení. Program prochází všechny dvojice nerovnic v levelu a kontroluje, jestli mezi nimi neexistuje konflikt. Podobně jako u kontroly přiřazení první vlákno, které nalezne konflikt ho vyřeší a ukončí ostatní vlákna.

Listing 3.14: Paralerizovaná funkce pro hledání konfliktů.

```
//...
    bool constraint_added = false;
#pragma omp parallel
{
    #pragma omp for
    for (size_t i = 0; i < curr_level_cnt; ++i) {
        for (size_t j = 0; j < curr_level_cnt; ++j) {
            NUMBER lower_bound = get_lower_bound(
                    assignments,
                    curr_level_constrains[i],
```

```
                    curr_level);
            NUMBER upper_bound = get_upper_bound(
                    assignments,
                    curr_level_constrains[j],
                    curr_level);
            if (lower_bound > upper_bound) {
                #pragma omp critical
                {
                    if (!constraint_added) {
                        resolve(constraints,
                        curr_level);
                        constraint_added = true;
                    }
                }
                #pragma omp cancel for
                }
            }
        #pragma omp cancellation point for
    }
}
//...
```

V tomto kódu je paralelizován pouze první for cyklus. OpenMP nabízí i paralelizaci vnořených cyklů pomocí *#pragma omp parallel for collapse*(n)[13], kde n je počet cyklů, ale použití tohoto parametru je složitější.

3.8 Normalizace

Normalizace všech vstupních nerovnic je provedena na začátku algoritmu a nová nerovnice, vzniklá součtem dvou již normalizovaných nerovnic, je také normalizována.

3.9 Optimalizace

3.9.1 Přeuspořádní strojového kódu

Jedna z optimalizací, kterou nabízí gcc je přeuspořádání strojového kódu pomocí *__builtin_expect*.[14] Jedná se o vestavěnou funkci gcc, která pomáhá kompilátoru s optmilaizací kódu pro branch predictor a pipelining. Pomocí ní může kompilátor přeskládat výsledný strojový kód podmínky, u které očekáváme, že nastane velmi zřídka, nebo naopak velmi často.

Další podobnou funkcí je *__builtin_expect_with_probability*[14], která má stejnou funkci jako*__builtin_expect* s tím rozdílem, že má navíc parametr

propability typu double, který může být v rozsahu $\langle 0.0, 1.0 \rangle$, který určuje pravděpodobnost podmínky.

Implicitní pravděpodobnost podmínky je 90%.[14] Tyto optimalize lze použít u podmínek, u kterých čekáme, že budou nastávat často, nebo naopak zřídka.

3.9.1.1 Porovnání vyprodukovaného strojového kódu v x86-x64

Normální podmínka:

Listing 3.15: Strojový kód bez __builtin_expect.

```
;if(i > 20){
85:   mov    eax,DWORD PTR [rbp-0xc]
88:   cmp    eax,0x14
8b:   jle    99 <main+0x3e>
;...
;else{
;...
```

Podmínka s optimalizací[3]:

Listing 3.16: Strojový kód s __builtin_expect.

```
;if( __builtin_expect(i > 20,0)){
85:   mov    eax,DWORD PTR [rbp-0xc]
88:   cmp    eax,0x14
8b:   setg   al
8e:   movzx  eax,al
91:   test   rax,rax
94:   je     a2 <main+0x47>
;...
;else{
;...
```

[3]Oba programy byly kompilovány s přepínačem $-O0$.

Kapitola 4

Vyhodnocení

V této kapitole přestavím naměřené hodnoty s různými optimalizacemi.

4.1 Velikost matic

Při generování velkých matic a následném měření jsem narazil na problém, kdy při řešení soustav, které mají více než 8 až 10 proměnných v závislosti na hustotě matic dochází k přetečení, nebo podtečení $int64\_t$, který je použit jako čitatel a jmenovatel pro zlomky s konečnou přesností.

Měření jsem tedy prováděl pouze se s zlomky s potenciálně nekonečnou přesností.

4.2 Použitý hardware a software

Výsledky byly naměřeny na již starším počítači s procesorem Xeon E5-2640 2.50GH a 6GB RAM.

Program byl kompilován na gcc 9.3.0 a operační systém byl Linux 5.4. Procesorový čas byl měřen pomocí nástroje *time*.

4.3 Vstupní data

Jako vstupní data jsem použil matice velikosti 25x25 generované pomocí SciPy[15], hodnoty matic náležely intervalu $(-10, 10)$. Takto jsem vygeneroval dvě sady hustých a řídkých matic, na kterých jsem poté následně prováděl měření.

4.4 Použité optimalizace

Seznam použitých optimalizací:

Procento hodnot, které nejsou rovné 0.	
řídké matice	30%
husté matice	45%

Tabulka 4.1: Zaplnění matic.

- **-Ofast** Jedná se o přepínač kompilátoru, který zapne všechny optimalizace -03.[14] Navíc zapne optimalizace, které nejsou validní pro programy, jež nenásledují standard C++[16]. Pro C++ jsou to *-ffast-math* a *-fallow-store-data-races*.
- **-march=native** Tento přepínač na základě typu CPU, které kompiluje daný program, instruuje kompilátor, aby pro kompilovaný program využil všechny instrukce kompilujícího CPU.[14] Nevýhodou tohoto přepínače je, že výsledný program nemusí fungovat na každém CPU stejné architerktury.
- **__buitin_expect** Vestavěná fukce gcc , detailně popsáno v 3.9.1.

4.5 Sekvenční řešič

Řešič vždy rychleji spočítal řídké soustavy, než husté a to s velmi výrazným rozdílem. Vzhledem k tomu, že se jednalo o náhodně generované soustavy, tak se dalo očekávat rovnoměrné rozložení konfliktů. CRA je navíc algoritmus, který je idealní pro řídké matice. Z pokusů při měření vyplynulo, že okolo 40% zaplnění matic se obtížnost problému rapidně zvýší. V případě matic 25x25 z řádu jednotek sekund na desítky minut.

U sekvenčního řešiče měl nejvyšší vliv na rychlost přepínač $-Ofast$, jak je vidět z grafů v kapitole 4.7. Přidání *__builtin_expect* do funkce hledající konflikty naopak vedlo ke zpomalení. Stejný, i když daleko meší vliv měl přepínač $-march{=}native$, což ale může být způsobeno stářím procesoru, i tak ale pro mě bylo překvapení, že tento přepínač způsobil zpomalení i když velmi malé.

Lze tedy říci, že kompilátor je sám o sobě velmi dobrý v optimalizacích a lepší je se spolehnou na profilování pomoci $-fprofile\text{-}arcs$.[14]

4.6 Paralelní řešič

Nakonec jsem se rozhodl ze smyčky, která hledala konflikty, odstranit direktivy pro zrušení for cyklu, aby nepřidávaly režii.

Paralelizace algoritmu se projevila velmi výrazně, jak je vidět na grafech v kapitole 4.7, zřejmě z důvodu náročnosti výpočtů s čísly s potenciálně nekonečnou přesností. Přepínače kompilátoru měly u paralerní verze daleko menší vliv na výsledný čas.

Pro řídké matice bez použití optimalizací kompilátoru ale byla sekvenční verze rychlejší, což pravděpodobně bylo způsobeno velkou režií vláken.

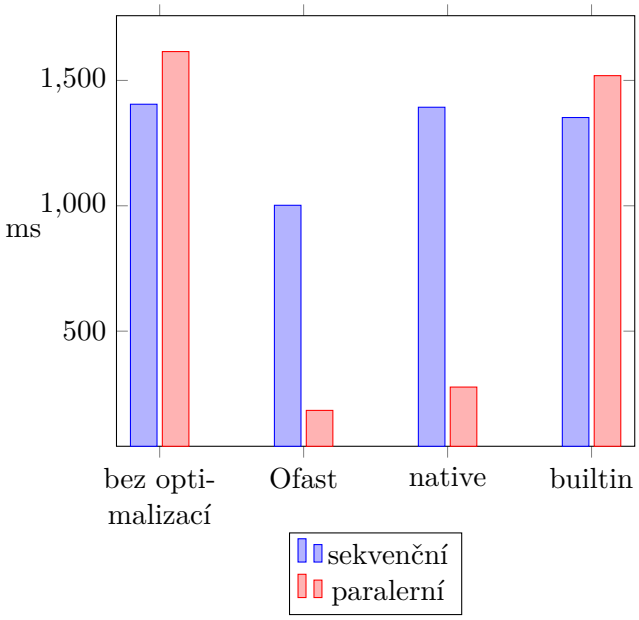
Paralelní verze algoritmu si pravděpodnobně vedla tak dobře na testovacích datech, protože jejich náhodné generování zřejmě zajistilo rovnoměrnější rozložení konfliktů než na reálných soustavách.

4.7 Grafy pro matice 25x25

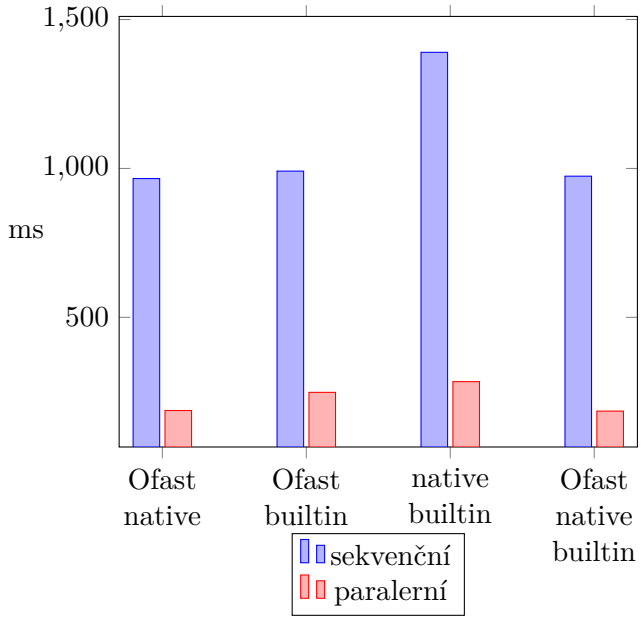
Tato kapitola obshuje obsahuje grafy s naměřenými hodnotami.

4.7.1 Řídké matice

Měřeno v milisekundách.



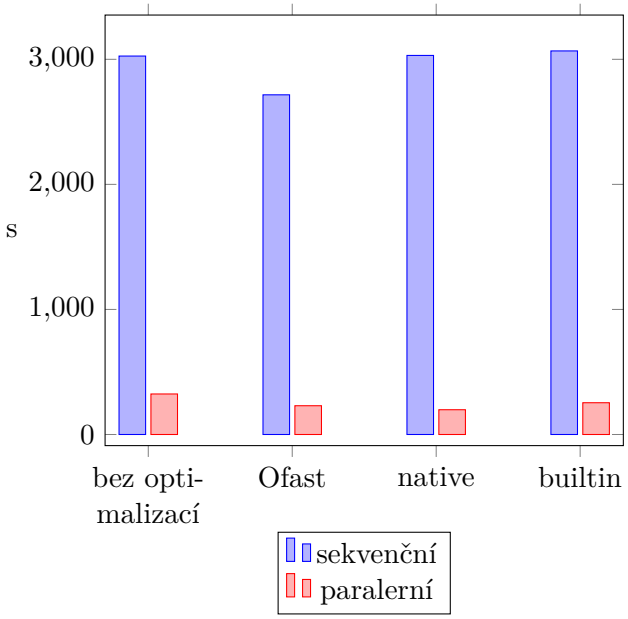
Obrázek 4.1: Naměřené hodnoty pro řídké matice.



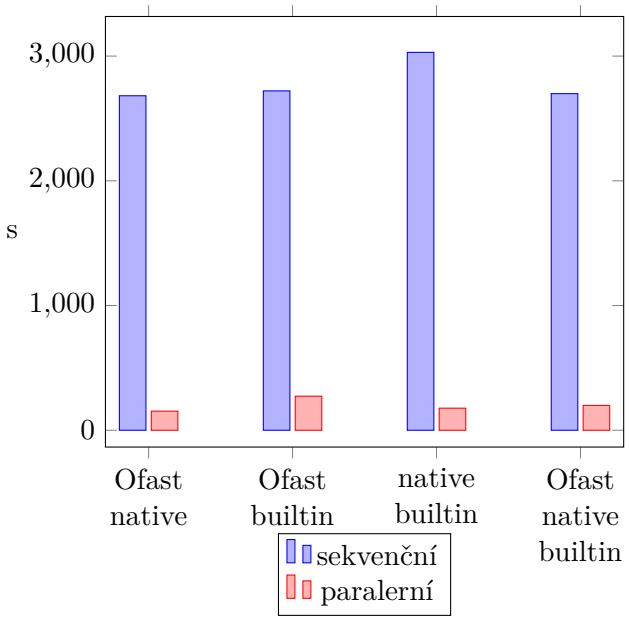
Obrázek 4.2: Naměřené hodnoty pro řídké matice.

4.7.2 Husté matice

Měřeno v sekundách.



Obrázek 4.3: Naměřené hodnoty pro husté matice.



Obrázek 4.4: Naměřené hodnoty pro husté matice.

4.8 Porovnání s jiným řešičem

4.8.1 SciPy

Pro srovnání jsem použil funkci *linprog* z knihovny SciPy[15], která reprezentuje čísla v plovoucí desetinné čárce. Řešení hledá pomocí lineárního programování, minimalizací funkce. Knihovna pro své výpočty používá paralelizaci.

Naneštestí pro větší soustavy funkce není schopna najít řešení, tudíž srovnání je velmi obtížné.

4.8.2 Wolfram Mathematica

Při výpočtu jsem použil funkci FindInstance, která najde jedno řešní pro množinu rovnic a nerovnic. Mathematica[19] stejně jako SciPy používá pro výpočet čísla v plovoucí desetinné čárce. K měření času jsem použil funkci Timing, která je také součástí Mathematicy a měří stějně jako *time* čas využití procesoru.

Mathematica byla schopna nalézt řešení instance, které můj program řešil asi 3 minuty během desítek milisekund (4.2). Ale za cenu toho, že výsledek nebyl přesný a vyčíslení některých nerovnic dalo číslo menší než nula, i když velmi blízko nule.

Čas pro vyřešení hustých soustav.	
Mathematica	82ms
Můj řešič	153s

Tabulka 4.2: Porovnání mého řešiče s Mathematicou.

Závěr

V první části práce jsem zadefinoval potřebnou teorii a detailně i s příklady popsal samotný algortimus CRA. V další části jsem popsal realizaci algoritmu pomocí C++ a OpenMP a některé optimalizace. Poslední část zahrnovala ukázku naměřených výsledků pro řídké a husté matice a porovnání sekvenčního a paralerního algoritmu.

Implementace se zlomky reprezentovanými pomocí $int64\_t$ se ukázala jako nevhodná pro větší problémy. Na druhou stranu implementace používající GMP je na výpočty velmi náročná a je otázkou, jestli je něco takového nutné pro reálné problémy.

Literatura

[1] Korovin, K.; Tsiskaridze, N.; Voronkov, A.: Conflict resolution algorithm [online]. 2009, Dostupné z: http://www.cs.man.ac.uk/~korovink/my_pub/cra09.pdf.

[2] Korovin, K.; Tsiskaridze, N.; Voronkov: Implementing Conflict Resolution [online]. 2011, Dostupné z: http://www.cs.man.ac.uk/~korovink/my_pub/implementing_cra_psi_2011.pdf.

[3] Boost rational [online]. Dostupné z: https://www.boost.org/doc/libs/1_71_0/libs/rational/rational.html.

[4] Knihovna boost [online]. Dostupné z: https://www.boost.org/.

[5] Licence boost [online]. Dostupné z: https://www.boost.org/users/license.html.

[6] Knihovna OpenMP [online]. Dostupné z:https://www.openmp.org/.

[7] Sierksma, G.; Zwols, Y.: Linear and Integer Optimization: Theory and Practice. 3rd Edition. Chapman and Hall/CRC, 2015. ISBN 978-1498710169.

[8] Gent, I. P. : Principles and Practice of Constraint Programming - CP 2009. Springer, 2009. ISBN 978-3-642-04243-0.

[9] Chapman, B. ; Jost, G.; Ruud van der Pas: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).The MIT Press, 2007. ISBN 978-0262533027.

[10] Guide into OpenMP: Easy multithreading programming for C++ [online]. Dostupné z: https://bisqwit.iki.fi/story/howto/openmp/.

[11] Matrix Market [online]. Dostupné z: https://math.nist.gov/MatrixMarket/index.html.

[12] Matrix Market Exchange Formats [online]. Dostupné z: https://math.nist.gov/MatrixMarket/formats.html#MMformat.

[13] Specifikace OpenMP [online]. Dostupné z: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[14] Dokumentace kompilátoru GCC [online]. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc/.

[15] Knihovna SciPy [online]. Dostupné z: https://www.scipy.org/.

[16] Standard C++ [online]. Dostupné z: https://isocpp.org/std/the-standard.

[17] Knihovna GMP [online]. Dostupné z: https://gmplib.org/.

[18] Boost gmp_rational [online]. Dostupné z:https://www.boost.org/doc/libs/1_68_0/libs/multiprecision/doc/html/boost_multiprecision/tut/rational/gmp_rational.html.

[19] Wolfram Mathematica software [online]. Dostupné z: https://www.wolfram.com/mathematica/.

Příloha **A**

Seznam použitých zkratek

RAM random-access memory

CPU central processing unit

GB gigabyte

Příloha **B**

Obsah přiloženého CD

- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX