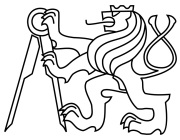
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Effective solver of linear inequalities
Student:	Jan Legner
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 2016/17

Instructions

Design an effective linear inequalities solver using Conflict Resolution algorithm (by Korovin, Tsiskaridze and Voronkov, see [1]). Use C++ for the implementation. Discuss parallelization of the algorithm and modify the program into a parallel one using OpenMP API. Measure program's speedup for a set of testing inputs from public repositories and compare performance for dense and sparse representations of a system of inequalities.

References

[1] http://www.cs.man.ac.uk/ korovink/my_pub/cra09.pdf

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

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague December 13, 2015

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Bachelor's thesis

Effective solver of linear inequalities

Jan Legner

Supervisor: Ing. Ivan Šimeček, Ph.D.

16th May 2017

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis, especially to my dear wife.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 16th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Legner, Jan. *Effective solver of linear inequalities.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Tato práce je zaměřena na algoritmus Conflict resolution, který je určený k řešení systémů lineárních nerovnic. Účelem práce je nalézt efektivní implementaci algoritmu s použitím bežných optimalizačních technik a porovnat výkonnost algoritmu pro řídké a husté systémy nerovnic. Práce obsahuje popis algoritmu, popis procesu optimalizace a výsledky měření výkonnosti.

Klíčová slova řešič, lineární nerovnice, Conflict resolution algoritmus, paralelní, OpenMP, C++, řídké matice

Abstract

This thesis is focused on the Conflict resolution algorithm, which is used to solve systems of linear inequalities. The purpose of this thesis is to find an effective implementation of the algorithm using common optimization techniques and to compare the performance of the algorithm for sparse and dense representations of linear systems. The thesis contains the description of the algorithm, the description of the optimization process and the results of performance measurements.

Keywords solver, linear inequalities, Conflict resolution algorithm, parallel, OpenMP, C++, sparse matrices

Contents

List of Figures

List of Tables

Introduction

To solve a set of linear inequalities is arguably one of the most common tasks in linear optimization. The aim of a linear optimization process is to find a best fitting solution for a mathematical model represented by linear relationships. Solutions for such models are often required in economics, business and many other industries (transportation, manufacturing etc.).

There are a few existing algorithms designed to solve these particular problems. For example: Fourier-Motzkin method (from the year 1827) or Chernikov algorithm (which improves over the Fourier-Motzkin). However, both of these algorithms solve a lesser amount of problems than *CRA*. [1]

K. Korovin, N. Tsiskaridze and A. Voronkov created and published *CRA* in 2009. This algorithm performs better than Fourier-Motzkin method and Chernikov algorithm – in some cases even by order of magnitude. [2] The goal of this thesis is to design an effective version of this algorithm and compare representations of sparse and dense matrices.

In the first chapter I introduce the reader of this thesis to basic definitions, algorithm description and algorithm's real world usages. In the second chapter I present design of programs which are results of this thesis. In the third chapter I evaluate the performance of my implementation and in the final chapter I give the overall conclusion to this thesis.

Chapter 1

Problem description

In this chapter I present basic definitions (as presented in [2]) and explain how *CRA* works. By the end of the chapter I show how the algorithm is applicable on real world problems.

1.1 Basic Definitions

Let $\mathbb{Q}$ denote the set of rationals. Throughout this thesis I denote by n a positive integer and by X a finite set of variables $\{x_1, \ldots, x_n\}$.

1.1.1 Linear constraint

A rational linear constraint over X is either a formula [1]

$$a_n x_n + \ldots + a_1 x_1 + b \geq 0$$

where $b \in \mathbb{Q}$ and $a_i \in \mathbb{Q}$ for $1 \leq i < n$, or one of the formulas $\top$, $\bot$. The formula $\top$ is always true and $\bot$ is always false. Such rational linear constraints over X are called simply linear constraints throughout this thesis.

1.1.2 System of linear inequalities

A system of linear inequalities can be formally expressed as

$$\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \text{ with } \boldsymbol{A} \in \mathbb{R}^{n,m}, \boldsymbol{b} \in \mathbb{R}^n$$

Usually, we need to find a solution $\boldsymbol{x} \in \mathbb{R}^m$. Throughout this thesis a system of linear inequalities is viewed as a set of linear constraints.

[1] Although any of symbols in $\{>, \geq, \neq, =\}$ could stand instead of a symbol $\geq$, I allow only linear constraints written in this specific format as input for the Conflict resolution algorithm as I implemented it. Therefore only linear constraint with symbol $\geq$ are discussed in this thesis.

1.1.3 Sparse and dense matrices

A matrix is considered to be a *sparse* matrix if the majority of elements is equal to 0. If the majority of elements in a matrix is not equal to 0, we speak about a *dense* matrix.

Let M be a matrix with m rows, n columns and z elements equal to 0. The *sparsity* and *density* of M are equal to $\frac{z}{m \cdot n}$ and $1 - \frac{z}{m \cdot n}$ respectively.

1.1.4 Level of a constraint

Let $\succ$ be a total order on X. Let us assume that $x_n \succ x_{n-1} \succ \ldots \succ x_1$. Let c be a linear constraint. The level of a linear constraint $level(c)$ is defined to be 0 if c contains no variables. Otherwise the level of c is k if x_k is the maximal variable in c.

1.1.5 Level of a system of constraints

Let S be a system of constraints. Level of a system of constraints $level(S)$ is defined as follows:

$$level(S) = max\ \{level(c) \mid c \in S\}$$

1.1.6 Normalized constraint

Let c be a linear constraint and let k be a level of c. If c is of the form $\top$, $\bot$ or $a_1x_1 + \cdots + a_{k-1}x_{k-1} \pm 1 \cdot x_k + b \geq 0$ (where $b \in \mathbb{Q}$ and $a_i \in \mathbb{Q}$ for $1 \leq i < k$). It is evident that every linear constraint can be easily normalized.[2]

1.1.7 k-system

Let S be a system of linear constraints and k a level of S. Then I denote such S by $S_{(k)}$.

1.1.8 Assignment

An assignment over X is defined as a mapping from X to $\mathbb{Q}$.

By σ_x^v, given an assignment σ, a variable $x \in X$ and a value $v \in \mathbb{Q}$, I denote the assignment obtained from σ where the former value of x is replaced by v and the rest of values stays unchanged. The denotation σ_x^v is referred to as *an update of* σ *at* x *by* v throughout this thesis.

Let q be a polynomial over X. By $q\sigma$ I denote a value of q where every variable was replaced by a corresponding value from σ.

[2] The normalization can be accomplished by dividing the right side and every coefficient on the left side by $|a_k|$.

Let S be a set of linear constraints and σ an assignment. σ is said to be a solution of (or to satisfy) a constraint $q \geq b$ if $q\sigma \geq b$ is true. If σ is a solution for every constraint in S, σ is said to be a solution of (or to satisfy) S.

If a solution σ exists for S, S is said to be satisfiable. Otherwise it is said to be unsatisfiable.

1.1.9 Lower bound of a constraint

Let c be a normalized linear constraint and let k be a level of c. Then a lower bound $l(c, X)$ of a linear constraint c is defined as follows:

$$l(c, X) = \begin{cases} -(a_1 x_1 + \cdots + a_{k-1} x_{k-1}) + b, & \text{if } a_k > 0 \\ -\infty, & \text{otherwise} \end{cases}$$

1.1.10 Upper bound of a constraint

Let c be a normalized linear constraint and let k be a level of c. Then an upper bound $u(c, X)$ of a linear constraint c is defined as follows:

$$u(c, X) = \begin{cases} (a_1 x_1 + \cdots + a_{k-1} x_{k-1}) - b, & \text{if } a_k < 0 \\ \infty, & \text{otherwise} \end{cases}$$

1.1.11 Lower and upper bound of a set of linear constraints

Let S be a set of n linear constraints, $k \in \mathbb{N}, 1 \leq k \leq n$. Then lower and upper bounds $L(S, k, X)$, $U(S, k, X)$, respectively, are defined as follows:

$$L(S, k, X) = max\ \{l(c, X) \mid c \in S_{(k)}\}$$

$$U(S, k, X) = min\ \{u(c, X) \mid c \in S_{(k)}\}$$

1.1.12 k-conflict

Let S be a set of n linear constraints, $k \in \mathbb{N}, 1 \leq k \leq n$. S is said to contain a *k-conflict* if:

$$L(S, k, X) > U(S, k, X)$$

1.1.13 Boundary interval

Let S be a set of n linear constraints, $k \in \mathbb{N}, 1 \leq k \leq n$. Then a boundary interval $I\langle S, k, X\rangle$ is defined as follows:

$$I(S, k, X) = \langle L(S, k, X), U(S, k, X)\rangle$$

This interval is non-empty if S has no *k-conflict*. [2]

1.2 Conflict Resolution algorithm

The algorithm's approach to finding the solution is fairly simple. The basic idea is that the algorithm comes up with an initial assignment. (That assignment could be very well any random assignment.)

Of course the initial assignment (or rather a random guess of the solution) is mostly completely incorrect. The algorithm therefore gradually adjusts the assignment while adding new constraints during the process based on well defined rules until a single solution is found or the given system of linear constraints is decided to be unsatisfiable. It is quite obvious that the adjustment of an assignment[3] and the process of creating new constraints[4] are crucial for the algorithm.

A detailed description of the algorithm and its rules is found in the following sections.

1.2.1 Input and output of the algorithm

The input of the Conflict Resolution algorithm is a set of linear inequalities and the output is either a single solution of the system or a statement that the system of given inequalities in not solvable.

1.2.2 Algorithm's description

Let S be the input of the algorithm, S is a set of linear inequalities. Let us assume that n is the count of variables occurring in S.

Firstly, the algorithm chooses an initial value for each variable occurring in the system of linear inequalities (an assignment σ). The algorithm then selects a subset $S_{(0)}$ of linear inequalities and checks whether every constraint in the subset is $\top$ or $\bot$. Constraints in $S_{(0)}$ are trivial to be checked, because they contain only inequalities without variables in them. If $\bot$ is found among them, the problem offers no solution.

The algorithm then works with a subset $S_{(k)}$, starting from a subset $S_{(1)}$ and iterating up to $S_{(n)}$. For every subset $S_{(k)}$ it checks whether σ (current assignment) satisfies every constraint in the subset.

If σ satisfies every constraint in a subset, the algorithm then proceeds to the next iteration. Otherwise, if σ does not satisfy $S_{(k)}$, the algorithm may decide that the S is not satisfiable or update the value of σ.

Definition 1.2.2.1 (Assignment refinement) *Let S be a set of linear constraints and let σ be an assignment. The assignment refinement rule at level k is then*

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

[3] Assignment refinement (AR), see 1.2.2.1.

[4] Conflict resolution (CR), see 1.2.2.2.

where

1. *σ satisfies all constraints in $S_{(0)}, S_{(1)}, \dots S_{(k-1)}$,*
2. *σ violates at least one constraint in $S_{(k)}$,*
3. *and $\sigma^{v}_{x_k}$ satisfies $S_{(k)}$.*

Definition 1.2.2.2 (Conflict resolution) *Let S be a set of linear constraints, let c_i and c_j be two constraints in a k-conflict and let σ be an assignment. The conflict resolution rule at level k is then*

$$(S, \sigma) \rightarrow (S \cup Normalized(c_i) + Normalized(c_j), \sigma),$$

where the sum of normalized forms of c_i and c_j does not contain x_k. This implies that one of the normalized constraints contains $-1 \cdot x_k$ while the other contains $+1 \cdot x_k$.[5]

Before the algorithm applies the assignment rule and thus adjusts the value of σ, it must resolve all *k-conflicts*.

Therefore, it tries to find a *k-conflict*. If a *k-conflict* is found, the algorithm applies the conflict resolution rule and changes k (a level of constraints' subset) to a level of the newly added constraint. If the level k is decreased to 0, the algorithm ends and sets the statement that S is not satisfiable as its output. If k is decreased to a positive integer, the algorithm then proceeds to resolve all *k-conflicts* for the assignment σ, one after another, while again possibly decreasing the value of k.

Finally, if no *k-conflict* is found, the boundary interval $I\langle S, k, X\rangle$ must be non-empty. [2] The algorithm then applies the assignment refinement rule. The assignment may be updated by any value in the boundary interval.

After the assignment is updated, σ satisfies all constraints $\{c|level(c) \leq k\}$. The value of k is incremented and the algorithm repeats all the steps above starting by checking if the assignment satisfies the $S_{(k)}$.

The algorithm ends when $k > n$ (or when k drops to 0 during the conflict resolution, as mention above).

1.2.3 Example

The example given in the following paragraphs demonstrates how exactly the conflict resolution algorithm processes its input and how it obtains a solution (or decides that no solution exists).

[5] An occurrence of a *k-conflict* implies that the lower bound of one constraint is greater than the upper bound of the other. Therefore two constraint with the same sign by the x_k variable cannot be in a *k-conflict*. See 1.1.9 and 1.1.10.

Algorithm 1 Conflict resolution algorithm CRA

Input: A set S of linear constraints
Output: A solution of S or "unsatisfiable"

```
1: procedure CRA(S)
2:     if ⊥ ∈ S_(0) then
3:         return "unsatisfiable"
4:     end if
5:     k ← 1
6:     n ← maximum level of constraints in S
7:     σ ← {x_1 ↦ 0, x_2 ↦ 0 ... x_n ↦ 0}                ▷ initial assignment
8:     while k ≤ n do
9:         if not isSatisfied(S, k, σ) then
10:            if containsConflict(S, k, σ) then
11:                (c_i, c_j) ← getConlict(S, k, σ)
12:                c_new ← {normalize(c_i) + normalize(c_j)}
13:                S ← S ∪ c_new                          ▷ conflict resolution rule
14:                k ← getLevel(c_new)
15:                if k = 0 then
16:                    return "unsatisfiable"
17:                end if
18:            end if
19:            I ← getBoundingInterval(S, k, σ)           ▷ See 1.1.13
20:            σ ← σ^v_{x_k}, v ∈ I                       ▷ assignment refinement rule
21:        end if
22:        k ← k + 1
23:    end while
24:    return σ
25: end procedure
```

Algorithm 2 Helper procedure – checking whether the *k-system* is satisfied

Input: A set S of linear constraints, a level k, an assignment σ
Output: true – σ satisfies $S_{(k)}$; false – otherwise

```
1: procedure isSatisfied(S, k, σ)
2:     for all c ∈ S_(k) do        ▷ c is in a form of a_1x_1 + a_2x_2 ... a_nx_n ≥ b
3:         b ← b value from c
4:         if cσ < b then                                 ▷ See 1.1.8
5:             return false
6:         end if
7:     end for
8:     return true
9: end procedure
```

Algorithm 3 Helper procedure – finds constraints in *k-conflict*

Input: A set S of constraints with a *k-conflict*, a level k, an assignment σ
Output: two constraints in *k-conflict*

```
1: procedure getConlict(S, k, σ)
2:     for all c_i ∈ {c | c ∈ S_(k), a_k > 0} do
3:         for all c_j ∈ {c | c ∈ S_(k), a_k < 0} do
4:             l ← lowerBound(c_i, σ)                    ▷ See 1.1.9
5:             u ← upperBound(c_j, σ)                    ▷ See 1.1.10
6:             if l > u then
7:                 return (c_i, c_j)
8:             end if
9:         end for
10:    end for
11: end procedure
```

Algorithm 4 Helper procedure – checking whether the *k-system* contains conflicts

Input: A set S of linear constraints, a level k, an assignment σ
Output: true – $S_{(k)}$ contains at least one conflict; false – otherwise

```
1: procedure containsConflict(S, k, σ)
2:     L ← −∞
3:     U ← ∞
4:     for all c ∈ S_(k) do
5:         L ← Max(lowerBound(c, σ), L)                  ▷ See 1.1.9
6:         U ← Min(upperBound(c, σ), U)                  ▷ See 1.1.10
7:     end for
8:     if L > U then
9:         return true
10:    end if
11:    return false
12: end procedure
```

Algorithm 5 Helper procedure – normalizing a constraint of a given level

Input: A constraint c in a form $a_1x_1 + a_2x_2 + \cdots a_kx_k \geq b$, a level k of the constraint
Output: An equal constraint in the normalized form.

```
1: procedure normalize(c, k)
2:     c_norm = (a_1x_1)/|a_k| + (a_2x_2)/|a_k| + ··· + (a_kx_k)/|a_k| ≥ b/|a_k|
3:     return c_norm
4: end procedure
```

1.2.3.1 Problem statement

Let the input of the algorithm be the system S of following inequalities:

$$\begin{aligned} 4x_1 + 2x_2 - x_3 &\geq 4 \quad (c_1) \\ 2x_1 - 2x_2 + x_3 &\geq 2 \quad (c_2) \\ x_1 - x_2 &\geq 2 \quad (c_3) \end{aligned}$$

1.2.3.2 Problem processing

1. A total order $x_3 \succ x_2 \succ x_1$ and an initial assignment $\sigma \ : \ x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 0$ are assumed.[6]

2. The variable k is set to 0. Because every constraint contains at least one variable, $S_{(0)}$ is empty. Therefore $S_{(0)}$ is implicitly satisfied. Let us increment k.

3. The variable k is set to 1. Because no constraint contains only x_1, $S_{(1)}$ is empty. Therefore $S_{(1)}$ is implicitly satisfied. Let us increment k.

4. The variable k is set to 2. $S_{(2)}$ contains only a constraint c_3. The constraint c_3 is violated by the assignment $x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 0$. No conflict exists in $S_{(2)}$, because there is only one constraint and it is impossible for one constraint to be in a conflict.[7] In order to update the assignment, the bounding interval must be calculated.

   $$I(S, 2, X) = (-\infty, -2\rangle$$

   The assignment refinement rule is applied and the assignment is updated:

   $$\sigma \leftarrow \sigma_{x_2}^{-4}$$

   $$\sigma \ : \ x_1 \mapsto 0, x_2 \mapsto -4, x_3 \mapsto 0$$

   Let us increment k.

5. The variable k is now set to 3. $S_{(3)}$ contains two constraints: c_1 and c_2. The constraint c_1 is violated by the assignment. Let us check for conflicts. Constraints c_1 and c_2 are in the *k-conflict*:

   $$L(S, 3, X) > U(S, 3, X)$$

   $$-6 > -12$$

[6] This particular initial assignment is also used by the authors of the algorithm in their paper, which investigates various heuristics for optimization of the algorithm.[3]

[7] For any constraint c of the level k, the bounding interval is always in a form of $\langle i, \infty)$ or $(-\infty, i\rangle$, where $i \in \mathbb{Q}$.

Let us apply the conflict resolution rule by adding a new constraint c_4:

$$\begin{aligned} 4x_1 + 2x_2 - x_3 &\geq 4 \quad (c_1) \\ 2x_1 - 2x_2 + x_3 &\geq 2 \quad (c_2) \\ x_1 - x_2 &\geq 2 \quad (c_3) \\ 6x_1 &\geq 6 \quad (c_4) \end{aligned}$$

Let us set k to the level of the newly added constraint c_4, $k \leftarrow 1$. Because $S_{(1)}$ contains no conflicts, the algorithm may calculate the bounding interval and apply the assignment rule.

$$I(S, 1, X) = \langle 1, \infty)$$

$$\sigma \leftarrow \sigma^2_{x_1}$$

$$\sigma \ : \ x_1 \mapsto 2, x_2 \mapsto -4, x_3 \mapsto 0$$

Let us increment k.

6. The variable k is now set to 2. The assignment satisfies every constraint (a constraint c_3) in $S_{(2)}$. Let us increment k.

7. The variable k is now set to 3. $S_{(3)}$ contains constraints c_1 and c_2. The constraint c_1 is violated by the assignment $x_1 \mapsto 2, x_2 \mapsto -4, x_3 \mapsto 0$. No conflict exists in $S_{(3)}$.

$$L(S, 3, X) \leq U(S, 3, X)$$

$$-10 \leq -4$$

$$I(S, 3, X) = \langle -10, -4\rangle$$

The assignment refinement rule is now applied and the assignment is updated:

$$\sigma \leftarrow \sigma^{-7}_{x_3}$$

$$\sigma \ : \ x_1 \mapsto 2, x_2 \mapsto -4, x_3 \mapsto -7$$

Let us increment k.

8. The variable k is now set to 4. The maximum level of constraints in S is only 3, therefore the algorithm finishes its task and returns the final result.

1.2.3.3 Final solution

Although this example was very trivial and only one constraint was added to the set of linear inequalities (the conflict resolution rule was applied only once), it shows well how the algorithm processes the input step by step and how conflicts are resolved.

The algorithm returned the correct solution of the given system

$$\sigma \ : \ x_1 \mapsto 2, x_2 \mapsto -4, x_3 \mapsto -7.$$

The solution can be easily verified:

$$\begin{array}{rl} 4 \cdot 2 + 2 \cdot (-4) - (-7) \geq 4 & (c_1) \\ 2 \cdot 2 - 2 \cdot (-4) + (-7) \geq 2 & (c_2) \\ 2 - (-4) \geq 2 & (c_3) \\ \hline 7 \geq 4 & (c_1) \\ 5 \geq 2 & (c_2) \\ 6 \geq 2 & (c_3) \end{array}$$

1.3 Real world applications

What are linear inequalities actually good for and why is it important to solve them?

With systems of linear inequalities one can design a mathematical model of their business' expenses, of a town's public transportation means, of a manufacturing process. Biologists and chemists may simulate how fast different chemical reactions are occurring in cells and there are plenty of other examples of applications of theses systems, which can be found.

Linear systems are generally used to find the best possible solution in allocating limited resources to achieve maximum profit (or minimum cost).

Although in simple examples (often used in economics textbooks) computing a solution to a system of linear inequalities may be an easy task to accomplish, the complexity of the real world is not limited to thousands or even tens of thousands of variables and constraints. [4]

The process of finding optimal solutions to such problems is often called *Linear optimization* or *Linear programming*. The linear programming problems are often solved for example by Fourier-Motzkin elimination. However, Fourier-Motzkin elimination does not solve systems of linear interval very effectively. The authors of Conflict resolution algorithm compared both their algorithm and Fourier-Motzkin elimination and discovered that the former outperforms the latter in some cases by orders of magnitude. [2]

The computers today are faster than in the past due to a vast technology development. A programmer may take advantage of multi-core processors and parallel computation to speedup the execution times of many algorithms.

The objective of this thesis is to implement an effective parallel version of Conflict resolution algorithm and compare it to a sequential version of the implementation.

The second objective is to compare the performance of the algorithm when dealing with sparse and dense matrices.

The results of these objectives may help to determine whether this algorithm can be used by companies and businesses on daily basis when dealing with their problems.

Chapter 2

Implementation

In this section I provide general details of the implementation. Next, I describe specific optimization techniques used for the sequential and the parallel solver. The chapter is finished by the description of sparse representation of matrices.

2.1 Language choice

Both, the sequential and the parallel, versions of the algorithm are written in C++, because the C++ language is very suitable for numerical computations. Also the language is quite understandable and offers a reasonable control of how the code is executed on the *low level*.

2.2 Basic system representation

I decided to use a vector (from the STL library) of arrays for the implementation of the systems of linear inequalities for dense matrices. The vector provides basic operations such as adding new constraints during conflict resolution phase[8] of the algorithm, while the array in every item of the vector ensures sufficient speed while accessing coefficients of a constraint.

I use a vector of linked lists to represent sparse matrices.

The data type of the constraints' coefficients is a number in the single-precision floating point format (*float*). I decided not to use any library for computing in an arbitrary precision, because the objective of this thesis is to parallelize this algorithm, not to investigate its numerical properties. [9]

The special case of the representation of sparse matrices is presented by the end of this chapter.

[8] See the conflict resolution rule 1.2.2.2.

[9] However, this approach may in some cases create an issue with the finiteness of the algorithm.2.4

2.3 Problem of a choice

The authors of this algorithm noticed that the algorithm's execution time and efficiency is significantly affected not only by the input data but also by the choice of some parameters of the algorithm.

2.3.1 Assignment update

If the assignment σ does not satisfy the subset $S_{(k)}$, a value is chosen from the boundary interval $I\langle S, k, X\rangle$. (1.2.2.1) The authors of the algorithm have analyzed several possibilities such as choosing the middle point of the interval, a random point, a maximum/minimum point and a few others. [3] Their final decision was influenced by working with a library with arbitrary precision (GMP), because it was profitable to minimize denominator in rationals used in the assignment. In this thesis however, my primal objective is to design a fast implementation. (1.3) Since numbers are stored as *floats* in my implementation, I decided to use the following function to determine which value to use:

$$v = \begin{cases} \frac{(i+j)}{2}, & \text{if the bounding interval is } \langle i, j\rangle \\ i + |i|, & \text{if the bounding interval is } \langle i, \infty) \\ j - |j|, & \text{if the bounding interval is } (-\infty, j\rangle \\ 0, & \text{if the bounding interval is } (-\infty, \infty), \end{cases}$$

where $i, j \in \mathbb{Q}$.

This assures that even if the interval is open on either side, the selected value remains within a reasonable range.

2.3.2 Conflicting constraints

If a *k-conflict* exists in $S_{(k)}$, two constraints are chosen to be resolved. The authors of the algorithm explored a few possibilities for this problem too. Firstly, I decided to implement a simple solution using the first conflict which could be found. However, later I decided to change the implementation and I used the method choosing the conflict with constraints c_i and c_j such that:

$$L(S, k, X) = l(c_i, X) \text{ and } U(S, k, X) = u(c_j, X)$$

I chose this method, because it has performed very well in tests made in [3] and my test runs also showed a significant performance gain.

Among other options considered by the authors were for example a random conflict or the first conflict. [3]

2.3.3 Order on variables

The third parameter of the algorithm is the choice of the total order on X. The authors paid attention to a random order and to a length-based order. [3] I decided to use the order in which the variables are loaded from the input file.

2.4 Floating point arithmetic

The algorithm's finiteness is not guaranteed when using the floating point arithmetic because of the limited precision provided. [5]

When the *k-conflict* is detected between two constraints, it must be resolved by the conflict resolution rule. The algorithm specifies that new constraints are added and the assignment is updated in a way that the two previously conflicting constraints cannot be in a conflict. But with the usage of the floating point arithmetic sometimes the assignment cannot be updated in such way. This results in finding of the same conflict (a conflict among the two same constraints) over and over again.

My implementation contains a simple detection of this cyclic behavior. Every time a conflict is resolved, indexes of both constraints are added to a set. If they were stored in the set previously, the algorithm resolves the conflict by adjusting the right sides of both conflicting constraints. They are adjusted by number d from the interval:

$$I_d = \langle \tfrac{l-u}{2}, \infty),$$

where $l > u$ and l, u are lower and upper bounds of conflicting constraints. The constraints are adjusted as follows:

$$\begin{array}{rl} a_{1,i}x_1 + a_{2,i}x_2 \dots a_{n,i}x_n \geq b_i & (c_i) \\ a_{1,j}x_1 + a_{2,j}x_2 \dots a_{n,j}x_n \geq b_j & (c_j) \\ \hline c_i \leftarrow a_{1,i}x_1 + a_{2,i}x_2 \dots a_{n,i}x_n \geq b_i - d & \\ c_j \leftarrow a_{1,j}x_1 + a_{2,j}x_2 \dots a_{n,j}x_n \geq b_j - d & \end{array}$$

This adjustment assures that the conflict is resolved. The constraints are changed, though, and the system's set of possible solutions changed as well. Nevertheless, this change is tolerated and the algorithm's finiteness is now assured.

This specific solution to a problem caused by the floating point arithmetic was proposed in [5].

Algorithm 6 Conflict resolution algorithm CRA with a few upgrades.

Input: A set S of linear constraints
Output: A solution of S or an error message

1: **procedure** CRA(S)
2: Resolved = {}
3: **if** $\perp \in S_{(0)}$ **then**
4: **return** "unsatisfiable"
5: **end if**
6: $S \leftarrow$ NormalizeEveryConstraint(S)
7: $k \leftarrow 1$
8: $n \leftarrow$ maximum level of constraints in S
9: $\sigma \leftarrow \{x_1 \mapsto 0, x_2 \mapsto 0 \dots x_n \mapsto 0\}$ ▷ initial assignment
10: **while** $k \leq n$ **do**
11: **if not** isSatisfied(S, k, σ) **then**
12: **if** containsConflict(S, k, σ) **then**
13: $(c_i, c_j) \leftarrow$ getConlict(S, k, σ)
14: **if** $(i, j) \in$ Resolved **then**
15: $(c_i, c_j) \leftarrow$ correct(c_i,c_j)
16: **continue**
17: **end if**
18: Resolved $\leftarrow$ Resolved $\cup\, (i, j)$
19: $c_{new} \leftarrow \{$normalize(c_i) + normalize(c_j)$\}$
20: $c_{new} \leftarrow$ normalize(c_{new})
21: **if** size(S) = MAX_CONSTRAINTS **then**
22: **return** "limit reached"
23: **end if**
24: $S \leftarrow S \cup c_{new}$ ▷ conflict resolution rule
25: $k \leftarrow$ getLevel(c_{new})
26: **if** $k = 0$ **then**
27: **return** "unsatisfiable"
28: **end if**
29: **end if**
30: $I \leftarrow$ getBoundingInterval(S, k, σ) ▷ See 1.1.13
31: $\sigma \leftarrow \sigma^v_{x_k}, v \in I$ ▷ assignment refinement rule
32: **end if**
33: $k \leftarrow k + 1$
34: **end while**
35: **return** σ
36: **end procedure**

2.5 Input data

I chose two possibilities how to provide input data for my programs. The data is read from the standard input device in both cases.

1. The input contains a set of systems of linear constraints. Each system consists of two numbers (n, m) and of m rows and $n+1$ columns, where n is a maximum level of constraints in the system. The fields in rows are separated by spaces. On the i-th row, in the j-th column (where $j \leq n$) is a coefficient a_j in the i-th constraint in the system. The last column is dedicated for the b value in the i-th constraint. All the constraints have their 0 coefficients present in the input file.

2. The input consists of appended files. (The format's description is available at [6].) Only *coordinate format* with real numbers as values is accepted. This format is particularly useful when dealing with sparse matrices.

I have tested the programs with random generated data and with real world test data available at the *Matrix Market*[10].

2.5.1 Normalized constraints

I realized that most constraint in the system are accesses at least once while being required to be in a normalized form.

If the assignment does not satisfy constraints at level k and no conflicts are found, the assignment must be updated to a value from the boundary interval (see the 20^{th} line in Algorithm 1). Finding the boundary interval at the level k requires all of the constraints from the level k to be normalized. (1.1.13)

If a *k-conflict* is found, the conflict is resolved with the normalized forms of conflicting constraints. (1.2.2.2)

Therefore I decided to normalize every constraint in the system at the beginning of the algorithm. (See the 6^{th} line in Algorithm 6.) Also every time a new constraint is added, it is added in a normalized form. (See the 20^{th} line in Algorithm 6.)

2.6 Sequential solver

After a *naive* implementation of Algorithm 6, which is referred to as *naive CRA* throughout the rest of this thesis, I decided to improve the implementation in the following steps:

1. **Level of constraints** A subset of constraints of the level k is required in the algorithm very often, it is suitable to split the system of constraints

[10] See http://math.nist.gov/MatrixMarket/

into multiple sets of constraints: one set for every level k. The speedup caused by this enhancement was immense – dense matrices of size 15x15 took about 6 times less to solve and sparse matrices of size 4241x4241 took about 250 times less time to solve.[11]

2. **Conflict detection** The *naive CRA* looks for conflicts and returns the first one to appear. I changed the behavior and used a different method: finding maximally overlapping constraints. (2.3.2)

   This change sped up the program by another third. (The speedup varied a lot depending on a particular set of constraints.)

3. **Compiler options** To gain even better performance, I added the following parameters for *g++*

   - *march=native* – this parameter selects the CPU to generate code for at compilation time by determining the processor type of the compiling machine. The code generated with this parameter may use every feature of the CPU it was compiled on. [7]
   - *Ofast* – this parameter enables all *O3* optimizations as well as others, for example *ffast-math*. This ensures that floating point operations are simplified and that the loops are vectorized if possible. [8]
   - *funroll-loops* – this parameter enables loop unrolling of the loops whose number of iterations can be determined at compile time or upon entry to the loop. Loop unrolling helps predict branch-jumping in programs and reduces the count of instructions needed during the execution of a program.

   These parameters improved the performance by a factor of two for some system, for others the difference was much smaller, sometimes a factor of eight could be noticed.

4. **Division** – I changed the code of the function normalizing constraints in a way which favors multiplication over division. The function first computes *divisor*, then its inversed value $multiplier = \frac{1}{divisor}$. Every coefficient in a constraint is then multiplied instead of being divided.

   This improvement reduced the execution time by roughly 15 %. Most significant performance gain can be seen when dealing with systems containing a lot of *k-conflicts*.

5. **Manual loop unroll, loop tiling** – I tried to manually apply both of these techniques on the source code but but I found no way how to produce a code with a speedup improvement using these techniques.

[11] Although I mention comparison with sparse matrices, the representation of the system was a general one – a vector of simple arrays

Since the most of the execution time is spent in parts, which try to find conflicts, calculate boundary interval and check satisfaction of the system, and I did not succeed to improve these parts by these techniques, I did not try to apply them on different parts of the code as the speedup would not be notable even if other loops were apt to to be unrolled or tiled.

6. **Restrict, const** – I declared every read-only variable to be *const* in order to provide hints for the compiler about how to work with them. I used *__restrict__* type qualifier whenever dealing with pointers to arrays which are not aliased[12] to enable better optimization by the compiler. This enhancement improved the performance of the solver by another 5 %.

7. **Boundary interval** – Every time a *k-system* is not satisfied, the algorithm tries to find a conflict. Because I use a method finding a maximal overlap, the conflict finding function also finds lower and upper bound of the system. After the conflicts are resolved, the assignment is updated by a value from the boundary interval.

   This is the simplified version of CRA:

```
1  while (k <= n) {
2    if (not isSatisfied(KSystem[k], n, k,
        assignment)) {
3      float * constraint1, * constraint2;
4      int i, j;
5      float lower, upper;
6      while (findConflict(KSystem[k], n, k,
          assignment, constraint1, constraint2, i,
          j, lower, upper)) {
7        // resolve conflict
8      }
9      float L, U;
10     getBoundaryInterval(KSystem[k], n, k,
          assignment, L, U);
11     assignment[k - 1] = getUpdateValue(L, U);
12   }
13   k++;
14 }
```

   I noticed that in fact, lines number 9 and 10 can be removed completely if *getUpdateValue* is called with *lower* and *upper* arguments (declared on the 5$^{\text{th}}$ line).

[12] Two pointers are not aliased when they are the only thing used to access the underlying object in memory.

This upgrade improved the performance by another third.

2.7 Parallel solver

I chose *OpenMP* library[13] for the parallelization of the algorithm because it uses a simple and flexible interface, the changes made to the parallelized code are minimal and the library supports multiple platforms. Therefore the resulting code is portable, which is a great benefit.

The parallelization of the code is achieved by using a *#pragma omp* directive. If the compiler does not support OpenMP's directive, *#pragma omp* should be skipped and the code should still be working.[14]

A number of threads created by OpenMP can be easily set by *omp_set_num_threads* – this functions allows the programmer to keep control of the programs' execution speed and the usage of resources.

To parallelize a simple for-loop, one can write the following:

```
1 #pragma omp parallel
2 {
3   #pragma omp for
4   for (i = 0; i < 100; ++i)
5     a[i] = a[i] + b;
6 }
```

#pragma omp parallel starts a block of commands which should be executed by multiple threads. *#pragma omp for* specifies that the following loop is the one to be run in parallel.

1. **Conflict detection** The most of the execution time was spent in the function for finding the conflicts. This function contains a single for loop: for every constraint in $S_{(k)}$ it finds it lower and upper bounds. The function finds a constraint with the highest lower bound and a constraint with the lowest upper bound. I present the code of this function in a simplified form which finds only maximum lower bound.

```
1 int indexI = -1;
2 float maxLBound = -FLT_MAX;
3 int iter_i, iter_ii = KSystem.size();
4 #pragma omp parallel if(k * iter_ii > 50000)
      firstprivate(n, k)
5 {
6   int private_indexI = -1;
7   float private_maxLBound = -FLT_MAX;
```

[13] https://gcc.gnu.org/projects/gomp/ and https://computing.llnl.gov/tutorials/openMP

[14] This is not true for every piece of code but many usages of the OpenMP library allow the *#pragma omp* directive to be removed without changing the output of the program.

```
 8   #pragma omp for nowait schedule(static, 256)
 9   for (iter_i = 0; iter_i < iter_ii; iter_i++) {
10     const float * constraint = KSystem[iter_i];
11     float lbound = lowerBound(c, n);
12     if (private_maxLBound < lbound) {
13       private_maxLBound = lbound;
14       private_indexI = iter_i;
15     }
16   }
17   #pragma omp critical
18   {
19     if (maxLBound < private_maxLBound) {
20       maxLBound = private_maxLBound;
21       indexI = private_indexI;
22     }
23   }
24 }
```

The 4$^{\text{th}}$ line starts a parallel block. The *if* condition assures that the code is parallelized only if a certain amount of iterations is expected. (Note that the function call on the 11$^{\text{th}}$ line executes a loop with k iterations.) The value presented in a for loop was found experimentally to behave best on the test machine.

Also note that variables on the 6$^{\text{th}}$ and 7$^{\text{th}}$ lines are local for each thread.

The *nowait* keyword on the line 8 removes implicit barrier at the end of the for loop, so the running threads do not wait for each other to finish the for loop. They join at the end of the *parallel* block instead.

The *schedule(static, 2048)* parameter on the same line states that each tread will do 2048 consequential iterations before executing another part of the for loop.[15] I found this setting to be vital for the speedup of this loop – the processor can work with its cache memory more efficiently.

The 17$^{\text{th}}$ line starts a critical section of the code. The critical section can be accessed by at most one thread at the time. If two or more threads accessed the critical section the function might not return expected and correct values. [9]

Unfortunately, I needed not only to find minimum and maximum bounds but also to find indexes of these extreme constraints. *OpenMP* has a *reduction* keyword, which simplifies dealing with critical sections:

[15] The last executed part does not have to consist of 2048 consequential iterations.

```
1 int max_val = -1;
2 #pragma omp parallel for reduction(max:max_val)
3 for (i = 0; i < n; i++)
4   if (a[i] > max_val)
5     max_val = a[i];
```

Reduction cannot be used in this case, therefore a slightly more complex code must be written.

2. **System satisfaction** The function checking whether an assignment satisfies a set of constraints was the second function I parallelized. The parallelization was accomplished in a similar manner as in the case above.

   The code must have been altered a little: The sequential implementation returned from the execution of the function as soon as a single unsatisfied constraint was found, skipping the rest of constraints.

   However, this approach is not applicable with OpenMP, because the parallel block must not contain a *return* statement. Therefore all constraints are checked in parallel and then the final result is found.

   Although this approach may realize unnecessary checks, the overall speedup was notable.

2.8 Sparse solver

I implemented the sparse matrix solver's matrix representation as a linked list. Each item in the list contains a non-zero value of coefficient and an index of its column in matrix.

This list is wrapped in a structure containing also the right side of a constraint and the level of a constraint.

This representation is very efficient when dealing with sparse matrices but fails miserably when dealing with general matrices. Also, this representation cannot benefit from loop vectorization and other similar optimization techniques.

Nevertheless, this representation of matrices is very effective if there are only few non-zero elements present.

Chapter 3

Evaluation

I present measured values for different sets of constraints and different levels of optimization. I run the solvers on Intel Core i7 machine with 2.8 GHz and with 8 GB of RAM.

3.1 Metrics

I used three types of metrics for the purpose of comparison of different implementations described in the previous chapter.

1. **Memory** – I measured the maximum memory allocated when solving systems of inequalities with different matrix representations. The memory usage was tracked with *GNU time 1.7*.

2. **MFLOPS** – This is an acronym for *millions of floating-points operations executed per second*. The *MFLOPS* performance metric tries to correct the primary shortcoming of the *MIPS* metric (millions of instructions per second) by more precisely defining the unit of 'distance' traveled by a computer system when executing a program. [10] I used *perf 4.4.59* to measure this metric. However, I noticed this tool does not work very well with vectorized operations and I had to recalculate counts of floating point operations by myself.

3. **Speedup** – When comparing parallel versions I calculated the ratio of *MFLOPS* for different counts of running threads relatively to the *MFLOPS* of the best sequential version.

3.2 Results

In this section, I will present measured results for different versions of solvers (see Table 3.1) on different data sets (see Table 3.2).

Short name	Details
optimized	optimized code (*const, restrict*, maximum overlapping conflicts, ...), compiled with *-Ofast*
march	same as optimized but compiled with *-march=native*
unroll	same as optimized but compiled with *-march=native -funroll-loops*
sparse	sequential version with sparse representation of matrices
1 thread	parallel version of *unroll* with a single thread
2 threads	parallel version of *unroll* with 2 threads
4 threads	parallel version of *unroll* with 4 threads

Table 3.1: Short names of measured solvers

Short name	Specification
wide	matrices with 25 rows, 2000 columns $density = 0.95$
long	matrices with 5000 rows, 17 columns $density = 0.95$
sparse	matrices with different sizes (from 236 x 236 to 17281 x 17281) $sparsity$ from 0.4 to 0.002
square	matrices with 22 rows, 22 columns $density = 0.95$

Table 3.2: Short names of data sets and their specifications

3.2.1 Sequential solver

Although I was not able to optimize the solver by manually unrolling or tiling loops, the speedup gained by *march=native* and *funroll-loops* was notable. (See Figure 3.1.) The largest speedup caused by these parameters can be seen when processing the *wide* data set. The probable cause is that the CPU does not need to branch-jump so often when going through a long array.

The *square* data set contains small matrices and the CPU spends more time on jumps. The same applies to the *sparse* set: Only a small number of constraints is present in *k-systems*, which causes more frequent jumps. And processing of *long* data set does not benefit much from the *funroll-loops*, because the inner loops are too short.

The speedup gained by tiny changes in the algorithm (see 2.6) was the most significant of all, however. The *naive CRA* reached only about 250 *MFLOPS* and the execution time took about 800 times longer.

3.2.2 Parallelization

The parallelization process was not as significant as I expected it to be. It is probably caused by a large overhead of threads – the parallelized loops in the sequential version were executed quite often but the loops themselves took very little time to execute. The parallelization by OpenMP also disables some of the compiler's optimizations – for example, the vectorization may be limited.

The presence of the overhead is evident from the results presented in Figure 3.2. The speedup of the version with 1 thread is less than 1 – which means that the parallel version with 1 thread is actually slower than the sequential version it is compared to.

The maximum speedup for all data sets was accomplished by running 2 threads. More threads bring more overhead and slow down the solvers.

3.2.3 Sparse matrices

The solver with the sparse representation of matrices was only suitable when dealing with sparse matrices. The solver was able to solve them really quickly. However, the dense representation dealt relatively well with both types (with the exception of the largest sarse matrices, which did not fit into the program's memory – resulting in the program not being able to solve them).

Although the dense representation's *MFLOPS* metric was greater for every input data (see Figure 3.3), the overall execution time of the sparse solver was better then the execution time of the dense solver when dealing with sparse matrices. (See Table 3.3.) For every other data set, however, the sparse solver was significantly slower compared to the best version of sequential solver with dense representation of matrices.

The reason that the *MFLOPS* are so low for the sparse solver is that the following of the linked lists representing sparse matrices is costly for the CPU as it cannot load the whole constraint to its cache.

	wide	**long**	**sparse**	**square**
unroll	9.79	33.92	5.60	37.46
sparse	47.93	91.57	2.16	120.77

Table 3.3: Execution times of solvers with different matrix representations (in seconds)

The differences in the memory usage are presented in Figure 3.4. (Note that the y-axis uses logarithmic scale.) The dense representation uses more memory than the sparse representation, because it stores every zero element in the memory. On the other hand, the sparse representation uses more memory

for every other case, because every item of the linked list is a structure containing two numbers (a value of a coefficient and an identifier of variable it belongs to) as well as a pointer to the next item.

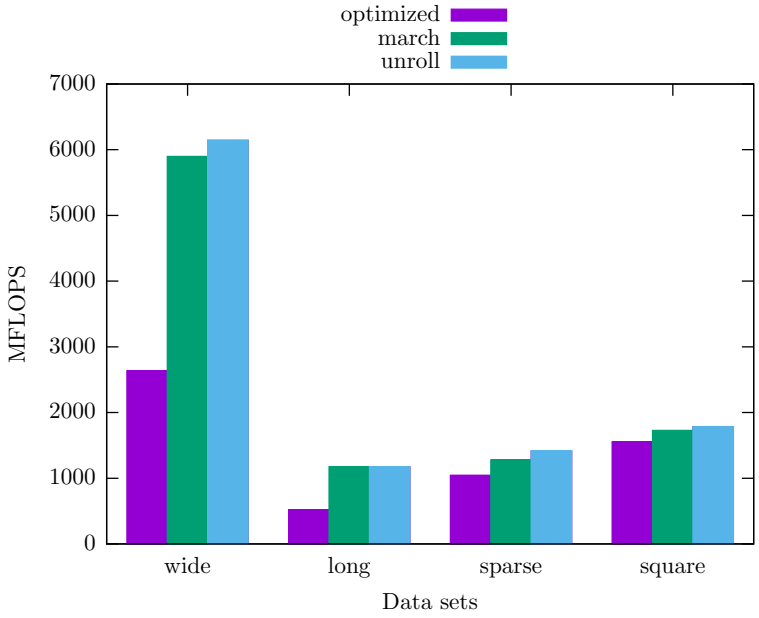


Figure 3.1: MFLOPS for different levels of optimization of the sequential version

3.3 Comparison to other solver

I decided to compare my implementation of the solver of inequalities with another solver. I chose a python library *mystic*[16] which is built around *numpy*[17] and *sympy*[18] libraries. The main reason why I chose this particular solver is that it can be set up to produce a single solution to a set of linear inequalities and my solvers behave in the same way.

Unfortunately, I was only able to compare my solver and the *mystic* solver on small matrices (10 rows, 10 columns, density 0.95), because the *mystic*'s implementation run into the memory problems with larger matrices.

The Table 3.4 provides comparison of how many matrices each solver solved in 20 seconds. The results are one-sided as the *mystic* solver appears to be very inefficient.

[16] https://pypi.python.org/pypi/mystic
[17] http://www.numpy.org
[18] http://www.sympy.org

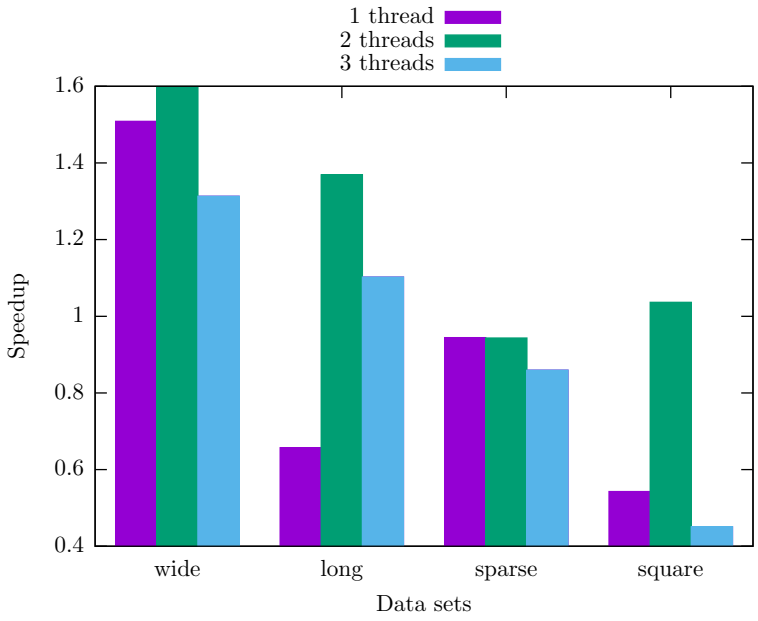


Figure 3.2: Speedup relative to version *unroll* for 1, 2 and 4 threads

However, it is hard to tell what exactly the *mystic* solver calculates in background and whether the single solution returned is the only product created by it.

solver	**count of matrices**
mystic	11
CRA	120105

Table 3.4: Number of matrices solved in 20 seconds by the *mystic* solver and the *2 threads* solver

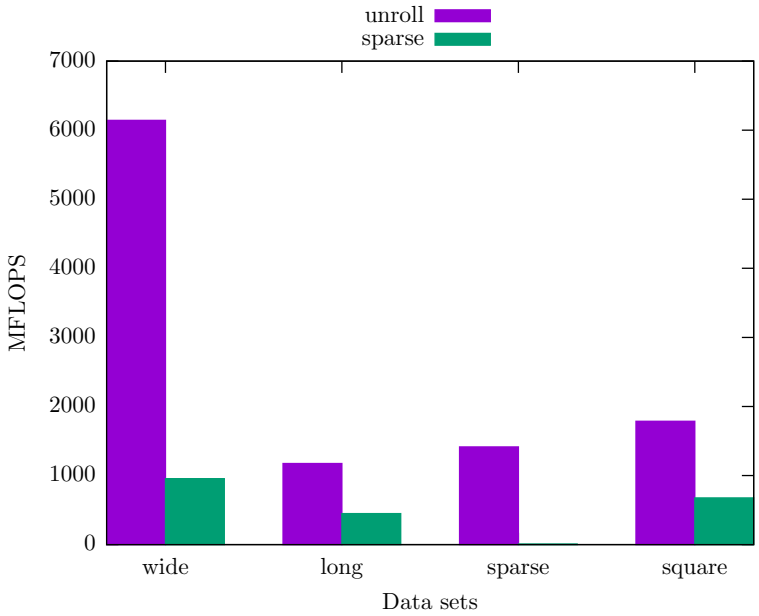


Figure 3.3: MFLOPS for different representations of matrices

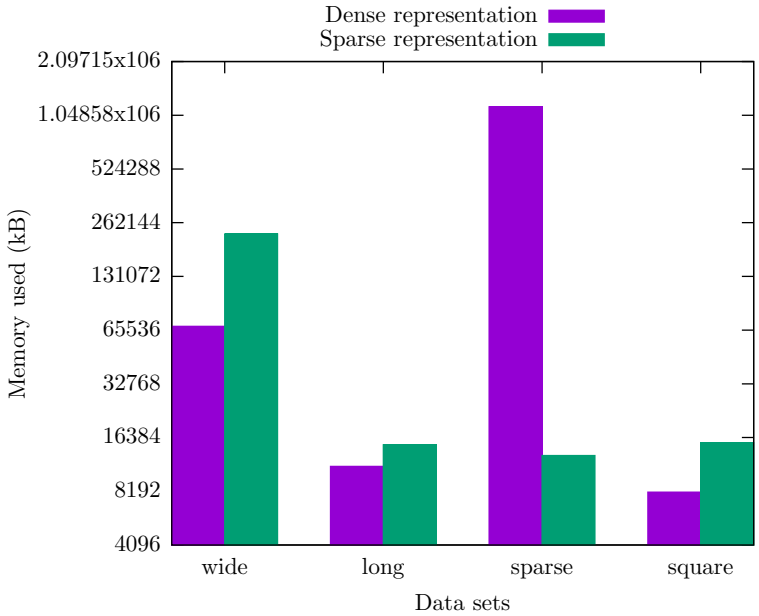


Figure 3.4: Memory allocated by different representations of matrices

Conclusion

I described the conflict resolution algorithm in the first part of this thesis, then, in the next part, I presented the optimization process of the solver. The process consisted of the code enhancements, of setting up the compiler's parameters and of the parallelization. I also designed and implemented a version of the solver using the sparse representation of matrices.

Although the results of the parallelization are not very convincing, the preceding steps showed that it is possible to improve the overall performance of this algorithm by changing implementation details.

Furthermore, the algorithm is able to solve both dense and sparse systems efficiently, as I showed in the evaluation part of the thesis, and may therefore be eligible for solving real world problems.

Bibliography

[1] Gent, I. P.: *Principles and Practice of Constraint Programming - CP 2009*. Springer Science & Business Media, September 2009, ISBN 3-642-04243-0, 521 p.

[2] Korovin, K.; Tsiskaridze, N.; Voronkov, A.: Conflict resolution algorithm [online]. 2009, [Cited: 2017-02-01]. Available at: `http://www.cs.man.ac.uk/~korovink/my_pub/cra09.pdf`

[3] Korovin, K.; Tsiskaridze, N.; Voronkov, A.: Implementing conflict resolution [online]. 2011, [Cited: 2017-02-01]. Available at: `http://www.cs.man.ac.uk/~korovink/my_pub/implementing_cra_psi_2011.pdf`

[4] Durlauf, S. N.; Blum, L. E.: *The New Palgrave of Economics*. Palgrave Macmillan, 2008, ISBN 978-1-349-58804-6, 281 p.

[5] Makara, T.: Vplyv druhov aritmetiky na Korovin/Tsiskaridz/Voronkovov algoritmus pre rieenie linernych nerovnc. 2014.

[6] Matrix Market Exchange Formats [online]. [Cited 2017-05-02]. Available at: `http://math.nist.gov/MatrixMarket/formats.html`

[7] Using the GNU Compiler Collection *GCC* x86 Options [online]. [Cited 2017-05-09]. Available at: `https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html`

[8] Using the GNU Compiler Collection *GCC* Optimize Options [online]. [Cited 2017-05-09]. Available at: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

[9] Gill, P. S.: *Operating System Concepts*. Firewall Media, 2006, ISBN 81-7008-913-1, 25 p.

[10] Lilja, D. J.: *Measuring computer performance: A practitioner's guide*. Cambrindge Unversity Press, 2005, ISBN 0-521-64670-7, 14 p.

Appendix A

User Guide

A.0.1 Makefile

The CD contains *Makefile* to make it easier to compile this project. It would not help to include the binaries, because they are compiled with *march* flag and are compatible only with the machine they were compiled on. The following commands are supported:

- **clean** – Deletes all object (*.o) files and binaries.
- **build_normal** – Build *naive* version (but with *-OFast* flag).
- **build_optimized** – Build the best sequential version with a dense representation of matrices.
- **build_parallel** – Build the parallel version.
- **build_sparse** – Build the version with a sparse representation of matrices.
- **build/all** – Build all.
- **doc** – Create a documentation for source files.

A.0.2 Running solvers

When the build is finished, the four binaries should be created:

- cra
- cra_optimized
- cra_prallel
- cra_sparse

All of these solvers read input data from the standard input device. If the *mtx* argument is passed to the solver, it reads the input in MTX format. Otherwise it expects two integers (columns and rows) and then $rows \cdot columns$ values.

Multiple files can be passed via the standard input device one after another, the solvers will solve them all. After the last input, the execution time is printed.

If the solver fails to load an input matrix due to a memory limit, the matrix is skipped and the following matrix is loaded.

Appendix **B**

Contents of enclosed CD

- `readme` ..... the file with CD contents description
- `sources` ..... the directory of source codes
  - `cra` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `systems` ..... the thesis text directory
  - `test` ..... basic test data
  - `dense` ..... dense matrices
  - `sparse` ..... sparse matrices
- `text` ..... the thesis text directory
  - `doc` ..... Doxygen documentation of the source files
  - `Thesis_CRA_2017_Jan_Legner.pdf` ..... the thesis text in PDF format