Computing and Informatics, Vol. 35, 2016, 1307–1337

PARALLEL SOLVER OF LARGE SYSTEMS OF LINEAR INEQUALITIES USING FOURIER-MOTZKIN ELIMINATION

Ivan Šimeček, Richard Fritsch, Daniel Langr, Róbert Lórencz

Department of Computer Systems
Faculty of Information Technologies
Czech Technical University in Prague
160 00 Prague, Czech Republic
e-mail: {xsimecek, fritsric, daniel.langr, robert.lorencz}@fit.cvut.cz

Abstract. Fourier-Motzkin elimination is a computationally expensive but powerful method to solve a system of linear inequalities. These systems arise e.g. in execution order analysis for loop nests or in integer linear programming. This paper focuses on the analysis, design and implementation of a parallel solver for distributed memory for large systems of linear inequalities using the Fourier-Motzkin elimination algorithm. We also measure the speedup of parallel solver and prove that this implementation results in good scalability.

Keywords: Solver, linear inequalities, Fourier-Motzkin elimination, distributed algorithms, MPI, C++

Mathematics Subject Classification 2010: 15A39, 65Y05, 68W15

1 INTRODUCTION

The goal of this paper is to analyze, design and implement a parallel solver for large systems of linear inequalities. The resulting solver should utilise Fourier-Motzkin elimination and be highly optimized for both speed and memory efficiency. Its implementation should be based on the MPI library. The results of the experiments should be compared with theoretical expectations. The performance and correctness will be tested on publicly available and randomly generated data sets from the data-dependency analysis.

1.1 Problem Description

Solving a system of n linear inequalities in m variables can be expressed formally as finding a solution of the system

$$\mathbf{Ax} \leq \mathbf{b} \text{ with } \mathbf{A} \in \mathbb{R}^{n,m},\ \mathbf{b} \in \mathbb{R}^{n}.$$

We are usually interested in finding a real solution $\mathbf{x} \in \mathbb{R}^n$ or an integer solution $\mathbf{x} \in \mathbb{Z}^m$ of $\mathbf{Ax} \leq \mathbf{b}$. Explicit representation of the set of solutions is desirable.

1.1.1 Basic Definitions

A set $\eth \subset \mathbb{R}^n$ is *convex* if

$$\forall \mathbf{x}, \mathbf{y} \in \mathbb{A}, \lambda \in (0,1) : \lambda \mathbf{x} - (1-\lambda)\mathbf{y} \in \mathbb{A}.$$

There are some special cases of convex subsets:

- *hyperplane*, if it can be expressed as $\mathbf{x} \in \mathbb{R}^n | \mathbf{a}^T\mathbf{x} = c$,
- *half-space*, if it can be expressed as $\mathbf{x} \in \mathbb{R}^n | \mathbf{a}^T\mathbf{x} \geq c$,
- *convex polyhedron*, if it is an intersection of finitely many half-spaces.

1.1.2 The Fundamental Theorem of Linear Inequalities

Existence and form of a solution for a system of linear inequalities results from the fundamental theorem of linear inequalities (generalization of the Farka's lemma [1]):

Let $\mathbf{a_1}, \ldots, \mathbf{a_m}, \mathbf{b}$ be n-dimensional vectors, then

- either $\mathbf{b}$ is a non-negative linear combination of linearly independent vectors $\mathbf{a_1}, \ldots, \mathbf{a_m}$
- or there exists a hyperplane $\{\mathbf{x}|\mathbf{cx} = 0\}$, containing $t-1$ linearly independent vectors from $\mathbf{a_1}, \ldots, \mathbf{a_m}$, such that $\mathbf{cb} < 0$ and $\mathbf{ca_1}, \ldots, \mathbf{ca_m} \geq 0$, where $t = \text{rank}\{\mathbf{a_1}, \ldots, \mathbf{a_m}\}$.

It follows from this theorem that a solution to a system of linear inequalities is either a convex polyhedron or an empty set.

1.2 Fourier-Motzkin Elimination

1.2.1 The Algorithm

The Fourier-Motzkin elimination is a mathematical algorithm for eliminating variables from a system of linear inequalities. FME is a generalization of Gaussian elimination and it is capable of finding both real and integer solutions. Its computational complexity is double-exponential. The main disadvantages are restriction to linear inequalities only and rapid growth in the number of them. It was first described by the well-known French mathematician J. B. J. Fourier in 1826 and later rediscovered by American mathematician T. S. Motzkin.

A concise description as found in [2], [3] and [4] follows:

1. A matrix $\mathbf{a} = (a_{ij})_{i,j} \in \mathbb{R}^{n,m}$ and a vector $\mathbf{b} \in \mathbb{R}^n$ represent the system

$$\begin{array}{rl} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m}x_m & \leq b_1, \\ \vdots \qquad\quad \vdots \qquad\quad \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm}x_m & \leq b_n. \end{array}$$

A "working system" is initialized with a matrix $\mathbf{T} \in \mathbb{R}^{n,m}$ and a vector $\mathbf{q} \in \mathbb{R}^n$:

$$t_{ij} = a_{ij} \text{ and } q_i = b_i \text{ where } 1 \leq i \leq n, 1 \leq j \leq m$$

and set variables $r = m$ and $s = n$.

2. The s inequalities are sorted and indices $n_1, n_2 \in \mathbb{N}, 1 \leq n_1 \leq n_2 \leq s$ are determined such that the following holds after index renaming:

$$\begin{array}{ll} t_{ir} > 0 & \text{for } 1 \leq i \leq n_1, \\ t_{i'r} < 0 & \text{for } n_1 + 1 \leq i' \leq n_2, \\ t_{i''r} = 0 & \text{for } n_2 + 1 \leq i'' \leq s. \end{array}$$

3. The first n_2 inequalities are normalized as follows:

$$\begin{array}{ll} t_{ij} = t_{ij}/t_{ir} & \text{for } 1 \leq i \leq n_2, 1 \leq j \leq r-1, \\ q_i = q_i/t_{ir} & \text{for } 1 \leq i \leq n_2. \end{array}$$

The system now looks as follows:

$$t_{i1}x_1 + t_{i2}x_2 + \ldots + t_{i,r-1}x_{r-1} + x_r \leq q_i, \quad 1 \leq i \leq n_1, \tag{1}$$
$$t_{i'1}x_1 + t_{i'2}x_2 + \ldots + t_{i',r-1}x_{r-1} + x_r \leq q_{i'}, \quad n_1 + 1 \leq i' \leq n_2, \tag{2}$$
$$t_{i''1}x_1 + t_{i''2}x_2 + \ldots + t_{i'',r-1}x_{r-1} \leq q_{i''}, \quad n_2 + 1 \leq i'' \leq s. \tag{3}$$

4. From subsystem (1) following inequalities are obtained:

$$x_r \leq q_i - \sum_{j=1}^{r-1} t_{ij}x_j \quad \text{for all } i, 1 \leq i \leq n_1.$$

We define the upper bound B_r^U for x_r as

$$B_r^U(x_1, \dots, x_{r-1}) = \min_{1 \le i \le n_1} \left(q_i - \sum_{j=1}^{r-1} t_{ij} x_j \right),$$

if $n_1 = 0$, then $B_r^U(x_1, \dots, x_{r-1}) = +\infty$.

Similarly, from subsystem (2) following inequalities are obtained:

$$x_r \ge q_{i'} - \sum_{j=1}^{r-1} t_{i'j} x_j \quad \text{for all } i', n_1 + 1 \le i' \le n_2.$$

Thus we define the lower bound B_r^L for x_r as

$$B_r^L(x_1, \dots, x_{r-1}) = \max_{n_1+1 \le i' \le n_2} \left(q_{i'} - \sum_{j=1}^{r-1} t_{i'j} x_j \right),$$

if $n_2 = n_1$, then $B_r^L(x_1, \dots, x_{r-1}) = -\infty$.

The range of feasible values for variable x_r

$$B_r^L(x_1, \dots, x_{r-1}) \le x_r \le B_r^U(x_1, \dots, x_{r-1})$$

is expressed using only variables $x_1, \dots, x_{r-1}$, these bounds are recorded for later use.

Further in the text we will refer to inequality categories (1), (2) and (3) as *positive*, *negative* and *zero* categories, respectively.

5. If $r = 1$, the algorithm has finished, because the bounds B_r^L and B_r^U are constants. The system has a *real* solution $x \in \mathbb{R}^m$ if and only if $B_r^L \le B_r^U$ and $q_{i''} \ge 0$ for all $i'', n_2 + 1 \le i'' \le s$. Otherwise (if $r > 1$) the algorithm continues.
6. The variable x_r is eliminated by adding all inequalities from the negative category to all inequalities from the positive category. This produces $n_1(n_2 - n_1)$ new inequalities in $r - 1$ variables:

$$q_{i'} - \sum_{j=1}^{r-1} t_{i'j} x_j \le x_r \le q_i - \sum_{j=1}^{r-1} t_{ij} x_j$$
$$\text{for all } i,\ i',\ \text{with } 1 \le i \le n_1,\ n_1 + 1 \le i' \le n_2.$$

By putting these inequalities together with $s - n_2$ inequalities from the zero category, a new system is obtained with $s' = s - n_2 + n_1(n_2 - n_1)$ inequalities in $r-1$ variables. If $s' = 0$ the algorithm has finished and the unknowns $x_1, \dots, x_{r-1}$ can be chosen arbitrarily (the system has infinitely many solutions). Otherwise the algorithm continues.

The elimination step can be geometrically interpreted as projection of the input polyhedron onto successive smaller dimensional space [2].

7. In the new system

$$\begin{cases} \sum_{j=1}^{r-1}(t_{ij} - t_{i'j})x_j \le q_i - q_{i'} & \text{for all } i,\ i' \text{ with } 1 \le i \le n_1,\ n_1 + 1 \le i' \le n_2, \\ \sum_{j=1}^{r-1} t_{i''j}x_j \le q_{i''} & \text{for all } i,\ i' \text{ with } n_2 + 1 \le i'' \le s. \end{cases}$$

The coefficients are renumbered as $t_{i,j}$ and q_i, where $1 \le i \le s'$ and $1 \le j \le r-1$. The variables are set $s = s', r = r - 1$ and the algorithm continues with step 2 (sorting step).

The algorithm ends when all unknowns are eliminated or when a column where all r^{th} values have the same sign is reached. In the first case, having all inequalities in the form $x_1 < C$ (where C is an unspecified constant), the unknown is easily computed. In the latter case, unknowns are chosen arbitrarily.

To express the solution the bounds that are saved throughout the computation are used – for re-substitution the subsequently obtained values.

A simplified version of the FME is outlined in Algorithm 1.

```
Input: system of linear inequalities $R$
for $i \leftarrow r$ to $1$ do
    $i$ is the variable to eliminate
    $W_i^0 \leftarrow W_i^+ \leftarrow W_i^- \leftarrow \emptyset$
    for each inequality $e \in R$ according to value of $x$ in column $i$ do
        if $= 0$ then
            insert $e$ into $W_i^0$
        else if $> 0$ then
            insert $e$ into $W_i^+$
        else if $< 0$ then
            insert $e$ into $W_i^-$
        end
    end
    normalize all inequalities in $W_i^+$ and $W_i^-$
    record $W_i^+$ and $W_i^-$ as lower and upper bounds respectively
    $R \leftarrow W_i^0$
    foreach pair $a \in W_i^+$ and $b \in W_i^-$ do
        create a new inequality $a - b$ and insert it into $R$
    end
    $R$ now contains inequalities without variable $i$
end
use the recorded bounds in back substitution
```

Algorithm 1: A simplified FME algorithm

1.2.2 Properties of the Algorithm

As described in [3], if we are interested in a real solution to $\mathbf{Ax} \leq \mathbf{b}$ with $\mathbf{a} \in \mathbb{R}^{n,m}$, $\mathbf{b} \in \mathbb{R}^n$, the FME algorithm tells us whether it exists and provides an explicit representation. It follows from the course of the algorithm that real solutions satisfy the recorded bounds:

$$\begin{array}{rcl} B^L_m(x_1,\ldots,x_{m-1}) \leq & x_m & \leq B^U_m(x_1,\ldots,x_{m-1}), \\ B^L_{m-1}(x_1,\ldots,x_{m-2}) \leq & x_{m-1} & \leq B^U_{m-1}(x_1,\ldots,x_{m-2}), \\ \vdots & \vdots & \vdots \\ B^L_1 \leq & x_1 & \leq B^U_1. \end{array}$$

To obtain one real solution vector the back substitution algorithm (Algorithm 2) should be applied.

```
Input: upper and lower bounds $B^L_1 \ldots B^L_{m-1}$ and $B^U_1 \ldots B^U_{m-1}$
Output: vector $x$ satisfying the recorded bounds
for $i \leftarrow 1$ to $m-1$ do
    $b_L \leftarrow B^L_{i+1}(x_1,\ldots,x_i)$
    $b_U \leftarrow B^U_{i+1}(x_1,\ldots,x_i)$
    $x_i \leftarrow$ a value chosen from the interval $[b_L, b_U]$
end
return $x$
```

Algorithm 2: Back substitution algorithm

As suggested in [5, 6], the value can be chosen from an interval according to the following rule:

$$x = \begin{cases} 0 & \text{for interval } (-\infty,\infty), \\ L + |L| & \text{for interval } [L,\infty), \\ U - |U| & \text{for interval } (-\infty,U], \\ \frac{L+U}{2} & \text{for interval } [L,U]. \end{cases} \tag{4}$$

On the other hand, if our aim is an integer solution (e.g., in data dependency analysis), the fact that the FME algorithm finished successfully does not guarantee the existence of one (for details see [7]). An explicit test of the following system is required:

$$\begin{array}{rcl} \lceil B^L_m(x_1,\ldots,x_{m-1})\rceil \leq & x_m & \leq \lfloor B^U_m(x_1,\ldots,x_{m-1})\rfloor, \\ \lceil B^L_{m-1}(x_1,\ldots,x_{m-2})\rceil \leq & x_{m-1} & \leq \lfloor B^U_{m-1}(x_1,\ldots,x_{m-2})\rfloor, \\ \vdots & \vdots & \vdots \\ \lceil B^L_1\rceil \leq & x_1 & \leq \lfloor B^U_1\rfloor. \end{array} \tag{5}$$

This can be approached in two ways:

- If no upper bounds $B_r^U = +\infty$ and no lower bounds $B_r^L = -\infty$ for some r, $1 \le r \le m$, then the loop nest of Algorithm 3 enumerates the complete solution set. This approach is obviously feasible only when the solution set is of reasonable dimensions. It is therefore desirable to somehow infer the maximum size of the solution set. If one is only interested in the existence of an integer solution set, the loop nest can be terminated after the first suitable x has been found.
- If a symbolic representation is desired, for example in the case when restructuring loop nests during compilation, then bounds for x are directly provided by (5).

```
Input: upper and lower bounds $B_1^L \dots B_{m-1}^L$ and $B_1^U \dots B_{m-1}^U$
Output: the complete solution set
for $x_1 \leftarrow \lceil B_1^L \rceil$ to $\lfloor B_1^U \rfloor$ do
    for $x_2 \leftarrow \lceil B_2^L(x_1) \rceil$ to $\lfloor B_2^U(x_1) \rfloor$ do
        $\ddots$
            for $x_m \leftarrow \lceil B_{m-1}^L(x_1, \dots, x_{m-1}) \rceil$ to $\lfloor B_{m-1}^U(x_1, \dots, x_{m-1}) \rfloor$ do
                print $x$
            end
    end
end
```
Algorithm 3: Complete integer solution set enumeration algorithm

1.2.3 Complexity of the Algorithm

The article [8] states that a weak lower bound on run time of FME was established in a previous print of [2] in 1986. The same paper then provides the first asymptotic complexity analysis of FME. Such analysis is not the objective of this paper, thus a similar analysis given in [3], which is easier to understand, is provided here. The run time of FME can be catastrophic in the worst case:

$T(n, m)$ stands for the run time of FME for a system of n inequalities in m variables. From the normalisation and elimination steps (3 and 6) of the algorithm one can obtain the run time for the general case:

$$T(s, r) = O(sr) + \max_{1 \le n_1 \le n_2 \le s} T(n_1(n_2 - n_1) + s - n_2, r - 1).$$

The worst case scenario, which occurs when $n_1 = n_2 - n_1 = n/2$, produces:

$$\begin{aligned} T(n, m) &\le O(nm) + T(n^2/4, m - 1) \\ &= O\left(\sum_{r=0}^{m-1} (m - r) \frac{n^{2^r}}{4^{2^r - 1}} \right) \end{aligned}$$

The double-exponential complexity is especially detrimental to the performance when m is large. However, the average run time should be significantly better because of two factors:

1. Generally low probability that the first argument of T reaches its maximum in every single step of the algorithm.
2. Because $n_2 \ll s$ if the matrix $\mathbf{A}$ includes a great deal of zeros, run time is significantly affected by the sparsity of $\mathbf{A}$. In the worst-case scenario the proportion of non-zero coefficients with inequalities of positive and negative categories doubles every iteration. The number of zero coefficients does not change for inequalities in the zero category.

1.3 Possible Technologies for Parallelization

1.3.1 OpenMP

OpenMP [9] is a cross-platform standard for parallel processing. The OpenMP API specification is defined as a collection of compiler directives, library routines and environment variables extending the C, C++, and Fortran programming languages. They can be used to create portable parallel programs utilizing shared memory. The process of parallelization is however not automated, the programmer is responsible for correct usage of the OpenMP APIs and avoidance of race conditions, deadlocks, and other data consistency issues related to the shared memory environment. We do not discuss this kind of parallelization further because it was already deeply discussed in the paper [3].

1.3.2 CUDA

CUDA[10] is a common GPGPU technology. It is an API for general purpose graphics processor unit programming mainly focused on performance through massive parallelism. Programmers are required to have more knowledge of the underlying hardware, than it is usual in CPU-based parallel programming. Parallel programs written in CUDA (for example [11, 12]) are, however, limited to the NVIDIA hardware. We do not discuss this kind of parallelization further because FME algorithm is not suitable for the CUDA execution model (it requires a lot of global synchronization and so on).

1.3.3 MPI

MPI (Message Passing Interface) is a cross-platform standard for parallel processing utilizing distributed memory. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in different computer programming languages such as Fortran, C, C++ and Java. Parallelization using MPI technology is deeply discussed in Sections 2 and 3.

1.4 Related Works

There are several other papers about FME, but most of them are aimed at studying the theoretical properties (e.g. the computational complexity for sparse systems, etc.). As far as we know, there is the only one research paper about FME in the context of the parallelization of this algorithm [3], where the authors present a parallel implementation for a shared memory parallel computer, and several variants for distributed memory parallelization are only sketched.

1.5 Existing Implementations

There are several other implementations of method for solving a system of linear inequalities (not all based on FME) including following products:

- Maple [13]: Maple is a commercial computer algebra system specialized in both symbolic and high-performance numeric mathematics. It provides an extensive suite of visualization tools, has comprehensive linear algebra support, powerful tools for solving optimization problems, and can solve a wide range of systems of equations.
- Maxima [14]: Maxima is a universal symbolic computer algebra system, a descendant of Macsyma CAS developed in the 1960s. Many modern CAS such as Maple or Mathematica were influenced by it. Maxima still has an active user community thanks to its open source nature.
- fm-eliminator package [15]: A somewhat similar solver "fm-eliminator" is open-source and freely available at [15]. The GNU GSL and cdd libraries are required to build it. It is written in C and is purely sequential. Its limitation is that it performs only the elimination and displays the final bounds for one chosen variable. We analyzed its code and discovered that despite the program's name it actually eliminates variables with different algorithm – block elimination using the extreme rays of a dual linear system. The source code also contains the FME, but it is commented out.
- R [16]: The R [16] language and statistical toolbox (version 3.1.0) offers purely sequential FME with its `editrules` package.
- Kessler's code: We could neither download the original code of this algorithm [3] (the project was canceled and deleted) nor run (the original code uses Cray Microtasking technology).

Comparison with these implementations is done in Section 4.3.

2 ANALYSIS AND DESIGN

2.1 Sequential FME

The sequential algorithm executes the sequence: distribute, normalize, record bounds and eliminate for each unknown and then performs back substitution. The distribute inequalities step is important only for the parallel solver and does nothing in the sequential solver.

2.1.1 Sorting Step Elision

We conceived a method to omit the sorting step altogether by having three containers for inequalities of respective categories. Based on the sign of the last coefficient the inequalities are moved to the corresponding container whenever necessary, i.e.:

- when inequalities are read from the input,
- when the last coefficient is truncated,
- when a new inequality is created during the elimination step.

2.1.2 Record Bounds

The record bounds step stores all the inequalities of the positive category as the upper bound and all the inequalities of the negative category as the lower bound for the current variable respectively. Numeric bounds for the current variable cannot be directly expressed at this stage because the bounds for the remaining variables are not known yet, thus all inequalities have to be recorded. No copying is necessary for this step, only references to the inequalities are moved.

2.1.3 Eliminate

In this step, we need to test whether the remaining variables can be chosen arbitrarily – this occurs when the zero category is empty and no inequality sets the lower or upper bound for the current variable. If that is the case the remaining variables can be chosen arbitrarily and we proceed directly to the back substitution. Otherwise all inequalities of the zero category are truncated (the zero coefficient is deleted) and new inequalities are generated by subtracting each inequality from the last recorded lower bound from each inequality from the last recorded upper bound. The last coefficients of the bound inequalities are equal to 1 and can therefore be ignored.

2.1.4 Back Substitute

The back substitution is performed either as the last step when all the unknowns have been eliminated or when arbitrary solution is warranted. In the first case, only inequalities in the form $x_r \leq C$ or $-x_r \leq C$ (where C is an unspecified constant) remain – the lower and upper bound for x_r (i.e. B_1^L and B_1^U) are computed as the maximum or minimum of the right hand sides, respectively. The system does not have any solution if the upper bound is lower than the lower bound ($B_1^U < B_1^L$) or when any of the inequalities of the zero category (in the form $0 \leq C$) does not hold.

If the goal is only one solution then execution of Algorithm 2 is required. Otherwise if the goal is to enumerate the integer bounds then Algorithm 4 must be performed instead.

```
Input: upper and lower bounds $B_1^L \dots B_{m-1}^L$ and $B_1^U \dots B_{m-1}^U$
Output: the complete solution set
Data: $S$ is a stack of triplets ($i$, $v$, $b$) where $i$ is the index of current
      variable, $v$ is the integer value and $b$ is a Boolean value indicating
      whether to start a new interval or continue with previous
push the triplet $(1, B_1^L, \text{false})$ onto $S$
repeat
    $(i, v, b) \leftarrow$ pop the top of $S$
    if $b = \mathit{true}$ then
        $j \leftarrow B_i^L$
    else
        $j \leftarrow i$
    end
    for $j$ to $B_i^U$ do
        $x[i] \leftarrow j$
        if $i = \mathit{unknowns}$ then
            print $x$
        else
            find exact upper and lower bounds $b_U$ and $b_L$ of $x_{i+1}$, using
             $x_1 \dots x_i$
            $B_i^L \leftarrow \lceil b_L \rceil$
            $B_i^U \leftarrow \lfloor b_U \rfloor$
            push $(i, v+1, \text{false})$ onto $S$
            push $(i+1, v, \text{true})$ onto $S$
            break;
        end
    end
until $S \neq \emptyset$;
```

Algorithm 4: Non-recursive version of FME

In the case of arbitrary solution, the remaining variables are set to 0 and start the back substitution for the already eliminated variables.

2.2 Parallelization of FME

In paper [3], several variants for distributed memory parallelization are only sketched. We extend these ideas further, we were inspired by papers [17, 18, 19, 20].

2.2.1 Data Distribution Strategies

The article [3] suggests three data distribution variants:

1. Uniform distribution of inequality categories. Computational load is perfectly balanced. Sorting step of the algorithm causes much communication and elimination step only modest communication.
2. Uniform distribution of inequalities (possibly non-uniform distribution of categories). Computational load is also perfectly balanced. Reduced communication for the sorting step but increased for the elimination step.
3. Cyclical distribution of variables. Imbalanced computational load for the most expensive last $p-1$ iterations. No communication for sorting and elimination steps in exchange for a broadcast for each inequality in the normalization step.

Let us first scrutinize them before we delve into details of the elected strategy.

2.2.2 Uniform Distribution of Categories

The algorithm starts with an arbitrary, equal distribution of all inequalities over p processors. A redistribution of all three inequality categories that achieves (6) must occur at the beginning of each iteration of FME.

$$n_1^{(q)} \approx \frac{n_1}{p}, \quad n_2^{(q)} \approx \frac{n_2 - n_1}{p} \quad \text{and} \quad n_3^{(q)} \approx \frac{s - n_2}{p} \tag{6}$$

where:

- $n_1^{(q)}$ denotes the processor q's share of the n_1 inequalities of the positive category,
- $n_2^{(q)}$ denotes the processor q's share of inequalities of the negative category and
- $n_3^{(q)}$ denotes the processor q's share of inequalities of the zero category.

Incidentally, (6) also implies equal load balance for the normalization step (3).

Maintaining a uniform distribution of inequality categories requires communication of $O(s)$ messages of length r [3]. Moreover, a broadcast of the smaller set of inequalities to all processors is necessitated before generating new inequalities in the elimination step.

2.2.3 Uniform Distribution of Inequalities

The redistribution of inequalities is not necessary before the sorting step. However, for a larger number of processors $n_1^{(q)}$ and $n_2^{(q)}$ cannot be expected to be equal – this would cause a computational load imbalance and [3] states that *all* inequalities have to be broadcast instead of only $\min(n_1, n_2 - n_1)$. This would indeed produce an equal distribution of the s' new inequalities, but it would render the parallelization of the solver pointless. If all processors had all inequalities then they would perform the exact same computation on the same data. This would not mitigate the extreme memory usage by the FME at all.

2.2.4 Cyclical Distribution of Variables

The processor in possession of the coefficients x_r has to broadcast the divisors for all inequalities before the normalization step in every iteration. The load imbalance for the last $p-1$ iterations worsens during the course of the algorithm – this renders the distribution inappropriate for massively parallel computers.

2.2.5 Elected Strategy

We analyzed all three variants and chose to implement the first one. We devised a redistribution algorithm (outlined in Algorithm 5) that determines how many inequalities are to be sent or received as well as by whom. Prior to the algorithm an all-to-all broadcast and subsequent summation of all local counts of inequalities needs to take place:

$$n_1 = \sum_{q=0}^{p-1} n_1^{(q)} \quad \text{and} \quad n_2 = \sum_{q=0}^{p-1} n_2^{(q)}.$$

In MPI, this is done using the `MPI_Allgather` routine. Following this, each processor finds the ideal count by dividing the summations by the number of processors. Next the ideal count is subtracted from the actual counts, resulting in a positive or negative number when the given processor has an excess or short supply of inequalities respectively; this array of differences is then used by the redistribution algorithm.

The algorithm itself is as follows: each processor builds a stack of send or receive operations based on how many inequalities each processor is missing or has excess of. The stack of operations is then executed (it is a no-operation for nodes with ideal inequality counts). Matching of senders and receivers is guaranteed as all the processors execute the same sequence of operations. To be able to track how many inequalities are present in each node, all processors (including those that are neither sending nor receiving) increase or decrease the global inequality counts respectively. To avoid needless communication the nodes only send inequalities if they have more than one extra. This also prevents problems occurring when the inequalities cannot be equally distributed, e.g. when there are 11 inequalities for 6 processors – in this case the ideal inequality count would be one, but 5 processors will have a surplus of one.

The Algorithm 6 from [3] carries out a broadcast of the smaller set of inequalities before generating new in the elimination step. This will incur a communication

```
Input: $d$ is differences from the ideal count of inequalities
Result: Equivalent distribution of inequalities among processors
Data: $S_{Send}$ is a stack of pairs (rank, number of excess inequalities)
$S_{Recv}$ is a stack of pairs (rank, number of missing inequalities)
for int $i \leftarrow 0$ to processor count $p$ do
    if $d[i] > 1$ then
        push the pair $(i, d[i])$ onto $S_{Send}$
    else if $d[i] < 0$ then
        push the pair $(i, -d[i])$ onto $S_{Recv}$
    end
end
Sort both $S_{Send}$ and $S_{Recv}$ in ascending order by number of inequalities
while $S_{Send} \neq \emptyset$ and $S_{Recv} \neq \emptyset$ do
    $s \leftarrow$ reference to the top of $S_{Send}$
    $r \leftarrow$ reference to the top of $S_{Recv}$
    while number of inequalities in both $s$ and $r$ is $> 0$ do
        if $s.rank$ = rank of this processor then
            send an inequality to processor $r$.rank
        else if $r.rank$ = rank of this processor then
            receive an inequality from processor $s$.rank
        end
        --global_counts[$s$.rank]
        ++global_counts[$r$.rank]
        --$s$.number of inequalities
        --$r$.number of inequalities
    end
    if $s$.number of inequalities $= 0$ then
        pop the top off $S_{Send}$
    end
    if $r$.number of inequalities $= 0$ then
        pop the top off $S_{Recv}$
    end
end
clear $S_{Send}$
clear $S_{Recv}$
```

Algorithm 5: Inequality equidistribution algorithm

overhead of $O(s)$ messages of length r. It should, however, be insignificant compared to the computational work $O(rs^2)$ of creating new inequalities for the next iteration of FME. Favorably, both the communication and computation loads are equally distributed over the processors.

```
Input: global counts of inequalities $n_1$ and $n_2$
Result: All $p$ processors having a complete set of inequalities of a category
if $n_1 < n_2$ then
    broadcast local $n_1^{(q)}$ inequalities to all $p$
    receive $n_1 - n_1^{(q)}$ inequalities from other processors
    forall $n_1 - n_1^{(q)}$ inequalities $e$ do
        forall local $n_1^{(q)}$ inequalities $e'$ do
            generate a new inequality $(e, e')$ as per step 6 (elimination)
        end
    end
else
    broadcast local $n_2^{(q)}$ inequalities to all $p$
    receive $n_2 - n_2^{(q)}$ inequalities from other processors
    forall $n_2 - n_2^{(q)}$ inequalities $e$ do
        forall local $n_2^{(q)}$ inequalities $e'$ do
            generate a new inequality $(e, e')$ as per step 6 (elimination)
        end
    end
end
```

Algorithm 6: Parallel inequality generation algorithm

2.2.6 Back Substitution

The bounds computed in all nodes have to be taken into account during the back substitution algorithm. Therefore we extended the back substitution algorithm with an all-to-all broadcast of the bounds at every stage. The global bounds are infinite only if all local bounds are infinite. The resulting extended back substitution is expressed in Algorithm 7.

3 IMPLEMENTATION

3.1 Reading Inequalities

Only the root process (whose rank = 0) reads the input file by calling `DoRead` from the sequential solver. After the first line is read it broadcasts the number of inequalities and unknowns to all other nodes and they compute how many inequalities to expect from the root:

```
Input: local upper and lower bounds $B_1^L \dots B_{m-1}^L$ and $B_1^U \dots B_{m-1}^U$
Output: vector $x$ satisfying the recorded bounds
Data: $L_i^{-\infty}$ and $U_i^{+\infty}$ are Boolean values indicating whether the $i^{\text{th}}$ lower
      bound or upper bound is $-\infty$ or $+\infty$ respectively
$b_L^q$ and $b_U^q$ denote lower and upper from processor $q$
for $i \leftarrow 1$ to $m-1$ do
    $b_L \leftarrow$ local $B_i^L(x_1, \dots, x_i)$
    $b_U \leftarrow$ local $B_i^U(x_1, \dots, x_i)$
    perform an all-to-all broadcast of 4-tuple $(b_L, b_U, L_i^{-\infty}, U_i^{+\infty})$
    $b_L \leftarrow \max_{1 \le q \le p}(b_L^q)$
    $b_U \leftarrow \min_{1 \le q \le p}(b_U^q)$
    if $L_{i,q}^{-\infty}$ = true in all $p$ processors then
        $L_i^{-\infty} \leftarrow$ true
    end
    if $U_{i,q}^{+\infty}$ = true in all $p$ processors then
        $U_i^{+\infty} \leftarrow$ true
    end
    $x_i \leftarrow$ a value chosen from the interval $[b_L, b_U]$
end
return $x$
```

Algorithm 7: Parallel bounds broadcast back substitution algorithm

$$\text{remainder} = \text{inequalities} \mod \text{processors},$$

$$\text{inequalities per node} = \begin{cases} \left\lfloor \dfrac{\text{inequalities}}{\text{processors}} \right\rfloor + 1 & \text{if rank} < \text{remainder}, \\[2ex] \left\lfloor \dfrac{\text{inequalities}}{\text{processors}} \right\rfloor & \text{otherwise.} \end{cases}$$

The root node then keeps some inequalities and distributes the rest among the other nodes.

3.2 Redistributing Categories

Prior to redistribution of the categories an all-to-all broadcast in the form of `MPI_Allgather` is performed so that all nodes know how many inequalities of all categories are present in every node. The numbers are also summed up to compute the global counts.

Afterwards, all three categories are equally redistributed by Algorithm 5. Nodes that have no inequalities after initial redistribution do not participate in the first iteration of the algorithm and pass through to the next iteration where there will be enough inequalities.

3.2.1 Broadcast of a Category

The elected data distribution strategy necessitates a broadcast of the smaller set of inequalities. We experimented with three methods that accomplish this (they are discussed in detail later):

- by matching sends and receives: Every node knows how many inequalities are present in all nodes, thus it can issue an exact number of matching sends and receives of individual inequalities. This is very space efficient because no temporary buffer is needed. However, the network utilization is poor because only one network link is active while sending. For large counts of small inequalities this approach is also inefficient because of the overhead incurred by each send or receive operation.
- by one all-to-all broadcast: The opposite of the previous approach is having a temporary buffer that can hold all inequalities from all nodes. Such a buffer allows the category distribution to be accomplished using just one all-to-all broadcast in the form of `MPI_Allgather`. Network utilization is optimal but this requires an amount of memory so large that it could cause an out of memory error and subsequent termination of the program, particularly during the last iterations of the algorithm.
- by multiple broadcasts: Accumulating all local inequalities into a temporary buffer and performing a `MPI_Bcast` from each node needs extra memory (maximal local inequality count $\times$ inequality size) and time to copy the data, but, as explained in Section 3.4, still very efficiently utilizes the network. The number of broadcasts depends on the number of processors rather than on the amount of inequalities, this is especially advantageous with inequality counts of $\sim 10^5$ and more.

The temporary buffer does not need to be initialized so all we need is so-called scoped array. Then we accumulate the inequalities using `std::copy`, which for `std::valarray` boils down to the most efficient bitwise copy. After the broadcast the inequalities are constructed directly in the category container using the `emplace_back` method of `std::vector`.

We chose this method because it is a reasonable compromise between space and network usage and performs consistently well.

3.3 Gathering Bounds

The all-to-all broadcast of local bounds computed in all nodes by Algorithm 7 occurs for each unknown. An `MPI_Allgather` routine gathers all the values into an array of 4-tuples where the first two members are the numeric bounds and the second two

Boolean values indicating whether the bounds are infinite, i.e. whether the numeric values are actually meaningful. A logical AND of the Boolean values is computed to detect infinite global bounds. The maximal lower bound and minimal upper bound are then found, taking only finite values into account.

3.4 Optimisation of the Communication Efficiency

When some data needs to be broadcast to all processes, there are several available approaches. The simplest way to broadcast inequalities is to perform a sequence of matching `MPI_Send` and `MPI_Recv` operations in all processes. A better solution is offered by the `MPI_Bcast` routine, implemented using a more efficient, tree based algorithm [21], which ensures better utilization of the communication network, as more network links are used every step of the broadcast. We take advantage of this MPI feature when broadcasting the smaller set of inequalities before the elimination step.

4 EVALUATION

4.1 Testing Configuration

4.1.1 Input Data for Testing

The input systems were randomly generated by our generator. The inputs are labeled dense and sparse, i.e. the proportion of zero coefficients is 12.5 % and 87.5 %, respectively. The solvers use the `float` data type for all the computation (if our aim is an integer solution, the required changes are discussed in Section 1.2.2). We used different systems of inequalities for various measurements to better accentuate the diverse hardware characteristics of the nodes of the cluster computer.

For the testing of related solvers, we use the following system:

$$\begin{aligned} -x + 2y &\leq 3, \\ x + y &\geq 3, \\ 2x - y &\leq 5. \end{aligned} \tag{7}$$

4.1.2 Software Configuration

The solvers were compiled using the predefined CMake build type "Release", which comprises `-O3 -DNDEBUG` compiler flags. In addition to that, C++11 support is enabled using the switch `-std=c++11`. The validity of results is always verified by re-substitution of the results into the original inequality system. We use the following tools or compilers:

- CMake 2.8 or newer (tested with version 2.8.11.2),
- C++11 compliant compiler (tested on g++ 4.7.3 and 4.8.1),
- MPI-1 compatible MPI implementation (tested on OpenMPI 1.6.4).

4.1.3 Hardware Configuration

The performance tests were carried out on various subsets of the University's computing cluster "Star", composed of

- $6 \times$ IBM BladeCenter LS21, Type 7971, Model 6AG, each with
  - $2 \times$ AMD Opteron 2218 2.6 GHz dual-core CPU (4 cores in total),
  - 8 GB RAM PC2-5300 CL5 ECC DDR2 SDRAM VLP RDIMM,
  - $1\times$73 GB 10 k rpm SAS HDD,
  - $2 \times$ Gigabit Ethernet,
  - $1 \times$ Cisco InfiniBand $4 \times$ HCA ($2 \times$ 10 Gbps ports);
- $6 \times$ IBM BladeCenter LS22, Type 7901, Model 62G, each with
  - $2 \times$ AMD Opteron 6C Model 2435 2.6 GHz/6MB L3 (12 cores in total),
  - 26 GB RAM PC2-6400 CL6 ECC DDR2 800 VLP RDIMM,
  - $1 \times$ 146 GB 10 k rpm SAS HDD,
  - $2 \times$ Gigabit Ethernet,
  - $1 \times$ Cisco InfiniBand $4 \times$ HCA ($2 \times$ 10 Gbps ports);
- Supermicro SuperServer 6027TR-DTQRF Dual Node, each node with
  - $2 \times$ 6-core Intel Xeon E5-2630 2.30 GHz/15 MB L3 (24-cores in total),
  - 32 GB RAM PC3-12800 Dual Rank ECC DDR3 SDRAM,
  - $1 \times$ Western Digital RE4 Enterprise 1.0 TB SATA-II,
  - Intel i350 Dual-Port GbE,
  - Mellanox ConnectX-3 QDR Infiniband 56 Gbps Controller.

4.1.4 Metrics

The most important performance metrics of a parallel algorithm, as found in [22], are parallel time, speed-up and efficiency. Parallel time $T(n,p)$, where n is the problem size and p is the processor count, is defined as the total time between start of computation and the moment when the last processor finishes. This includes both computation and communication time. Parallel speed-up is then defined as

$$S(n,p) = \frac{SU(n)}{T(n,p)}$$

where $SU(n)$ denotes the sequential upper bound on run time. The ideal linear speed-up p is rarely achieved in practice. The speed-up per processor, or parallel efficiency is then defined as

$$E(n,p) = \frac{S(n,p)}{p}.$$

When measuring the performance of a parallel program, speed-up is obtained simply as the elapsed time of the sequential version divided by the elapsed time of the parallel program. Similarly, parallel efficiency is obtained as parallel speed-up divided by number of processors.

4.2 Results

4.2.1 24 Processor Nodes

When measured exclusively on the 24 processor nodes, the impact of communication network was negligible. On the other hand, Hyper-Threading, Intel's proprietary simultaneous multi-threading implementation has significant influence on parallel speed-up. In this super-scalar architecture, each physical processor core has two logical cores that share the arithmetic logic unit, the caches and the memory bus. This does not provide true parallel processing, because a logical core can operate only when the other logical core is stalled, which occurs only due to a cache miss, branch misprediction or data dependency. There are two 6-core Hyper-Threaded Intel Xeon processors present in one node, but only 12 cores are physical out of the 24 that the operating system sees.

Graph in Figure 1 shows the parallel speed-up steadily increasing when more processors are added except in the 24 and 48 processor cases for all systems, the speed-up is lower than expected when the Hyper-Threading cores become active. Parallel efficiency is plotted in graph in Figure 2. The poor speed-up of Hyper-Threading is apparent in the results compiled in Table 1. The solver scales reasonably well up to 12 processes, with more the efficiency drops under 40 %.

4.2.2 12 Processor Nodes

The 12 processor nodes do not have Hyper-Threading and are only partially influenced by the used communication network. Parallel speed-up is pictured in graph in Figure 3, where only the 52×4 system behaves differently when utilizing InfiniBand. Efficiency is plotted in graph in Figure 4 and overall results are listed in Table 2. The scalability on 12 processor nodes is data dependent – the 24×5 and 150×7 systems improve even with large process counts but using more than 36 processes is wasteful for the 52×4 system.

4.2.3 Impact of Communication Network

The impact of communication network is best illustrated when using only the 4-processor nodes, where there is more inter-node communication than with the 12-processor nodes. As can be seen from graphs in Figures 5 and 6, the InfiniBand network significantly improves parallel speed-up and therefore efficiency. The impact of InfiniBand is apparent for 2 nodes (8 processors) and increases when more nodes

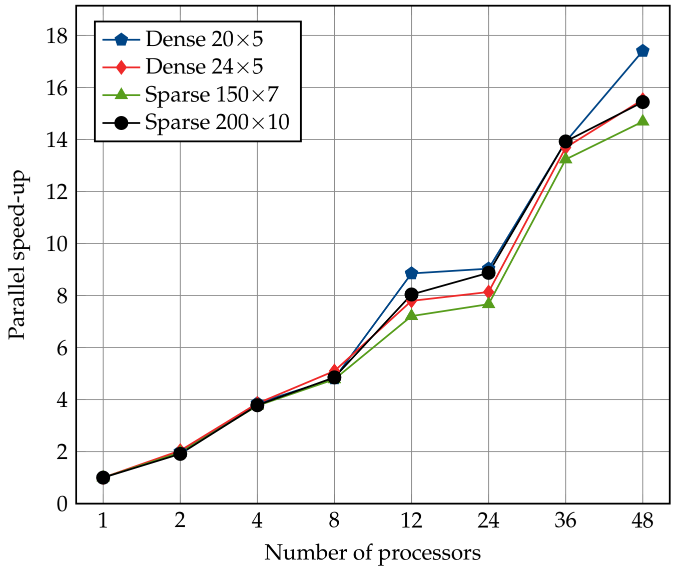


Figure 1. Parallel speed-up on 24 processor nodes

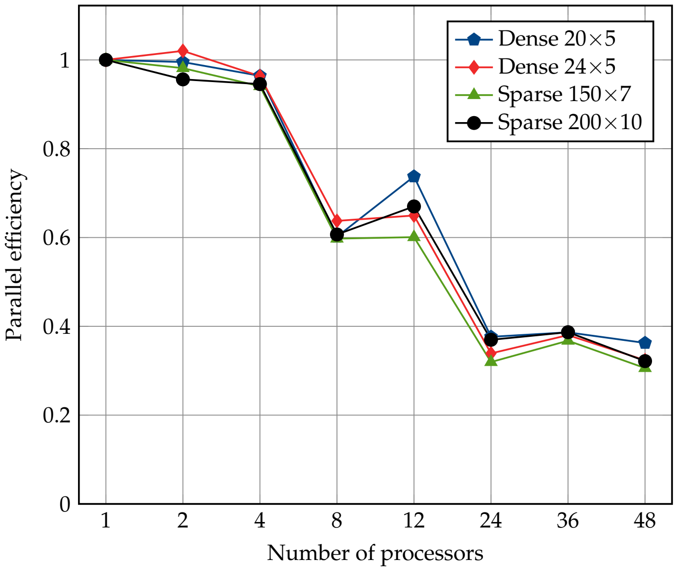


Figure 2. Parallel efficiency on 24 processor nodes

Dense 24 × 5

p	Time [ms]	Speed-Up	Efficiency
1	44 977	1.00	1.00
2	22 041	2.04	1.02
4	11 672	3.85	0.96
8	8 819	5.10	0.64
12	5 770	7.79	0.65
24	5 528	8.14	0.34
36	3 286	13.69	0.38
48	2 896	15.53	0.32

Sparse 150 × 7

p	Time [ms]	Speed-Up	Efficiency
1	37 262	1.00	1.00
2	18 230	2.04	1.02
4	9 961	3.74	0.94
8	7 737	4.82	0.60
12	5 261	7.08	0.59
24	4 976	7.49	0.31
36	2 813	13.25	0.37
48	2 528	14.74	0.31

Dense 20 × 5

p	Time [ms]	Speed-Up	Efficiency
1	29 113	1.00	1.00
2	14 627	1.99	1.00
4	7 549	3.86	0.96
8	6 047	4.81	0.60
12	3 290	8.85	0.74
24	3 223	9.03	0.38
36	2 093	13.91	0.39
48	1 673	17.40	0.36

Sparse 200 × 10

p	Time [ms]	Speed-Up	Efficiency
1	12 356	1.00	1.00
2	6 461	1.91	0.96
4	3 267	3.78	0.95
8	2 545	4.85	0.61
12	1 537	8.04	0.67
24	1 393	8.87	0.37
36	887	13.92	0.39
48	800	15.44	0.32

Table 1. Results on 24 processor nodes

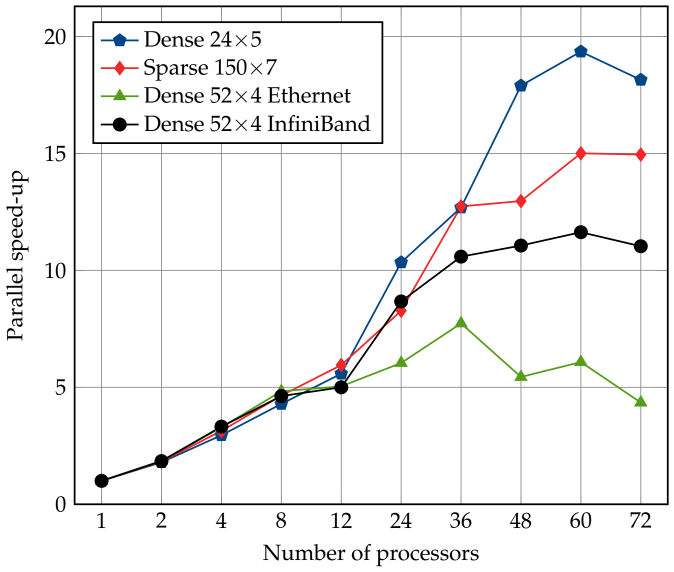


Figure 3. Parallel speed-up on 12 processor nodes

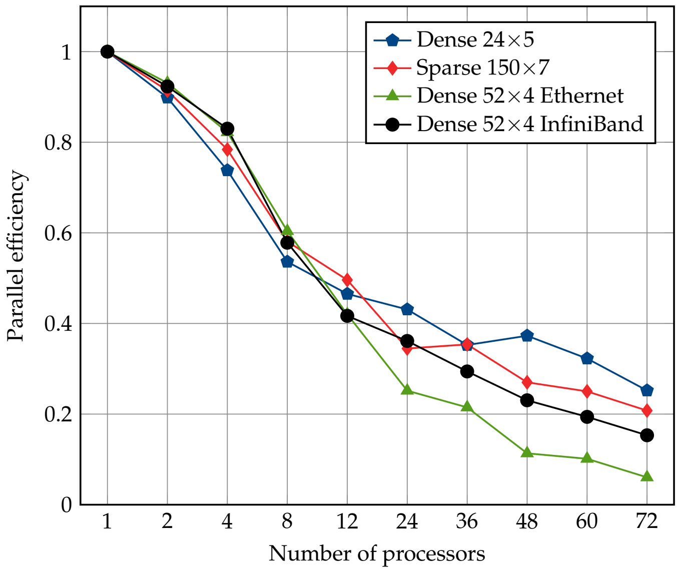


Figure 4. Parallel efficiency on 12 processor nodes

are used. When fully utilizing all 6 nodes, the efficiency almost doubles with a large sparse system of inequalities and more than triples for a smaller dense system. Table 5 shows the exact results.

4.2.4 Comparison with Theoretical Expectations

The run time complexity analysis implies the worst-case scenario where $\frac{n^2}{4}$ new inequalities are generated in each iteration. Table 3 illustrates the theoretical growth rate of the inequality count each iteration i. However, as shown by Table 4, randomly generated inputs result in more acceptable numbers. The counts are orders of magnitude lower, especially for sparse systems.

4.2.5 CPU Profiler

The following table lists the five most significant functions in the sequential solver. The `operator delete()` is mostly executed when all the destructors are called at the end of the program, its influence on the computation itself is not significant.

Dense 24 × 5

p	Time [ms]	Speed-Up	Efficiency
1	40 998	1.00	1.00
2	22 814	1.80	0.90
4	13 889	2.95	0.74
8	9 558	4.29	0.54
12	7 342	5.58	0.47
24	3 964	10.34	0.43
36	3 231	12.69	0.35
48	2 290	17.90	0.37
60	2 118	19.36	0.32
72	2 259	18.15	0.25

Sparse 150 × 7

p	Time [ms]	Speed-Up	Efficiency
1	53 648	1.00	1.00
2	29 347	1.83	0.91
4	17 108	3.14	0.78
8	11 542	4.65	0.58
12	9 013	5.95	0.50
24	6 483	8.28	0.34
36	4 212	12.74	0.35
48	4 138	12.97	0.27
60	3 575	15.01	0.25
72	3 588	14.95	0.21

Dense 54 × 4 Ethernet

p	Time [ms]	Speed-Up	Efficiency
1	16 664	1.00	1.00
2	8 950	1.86	0.93
4	5 068	3.29	0.82
8	3 449	4.83	0.60
12	3 305	5.04	0.42
24	2 758	6.04	0.25
36	2 155	7.73	0.21
48	3 062	5.44	0.11
60	2 740	6.08	0.10
72	3 833	4.35	0.06

Dense 54 × 4 InfiniBand

p	Time [ms]	Speed-Up	Efficiency
1	16 664	1.00	1.00
2	9 026	1.85	0.92
4	5 019	3.32	0.83
8	3 602	4.63	0.58
12	3 332	5.00	0.42
24	1 921	8.67	0.36
36	1 574	10.59	0.29
48	1 506	11.06	0.23
60	1 432	11.63	0.19
72	1 510	11.03	0.15

Table 2. Results on 12 processor nodes

i	Inequality Counts					
1	10	20	24	30	40	50
2	25	100	144	225	400	625
3	156	2 500	5 184	12 656	40 000	97 656
4	6 104	1 562 500	6 718 464	4.0×10^7	4.0×10^8	2.4×10^9
5	9 313 226	6.1×10^{11}	1.1×10^{13}	4.0×10^{14}	4.0×10^{16}	1.4×10^{18}
6	2.2×10^{13}	9.3×10^{22}	3.2×10^{25}	4.0×10^{28}	4.0×10^{32}	5.0×10^{35}
7	1.1×10^{26}	2.2×10^{45}	2.5×10^{50}	4.0×10^{56}	4.0×10^{64}	6.4×10^{70}

Table 3. Inequality counts in the theoretical worst-case scenario

Dense Systems				Sparse Systems		
i	Inequality Counts			i'	Inequality Counts	
1	10	20	24	1	150	200
2	21	44	95	6	623 448	562
3	110	343	894	7	482 642 693	1 041
4	2 625	27 423	88 253	8		28 778
5	1 412 964	184 967 954	609 292 552	9		70 737 891

Table 4. Actual inequality counts in randomly generated instances

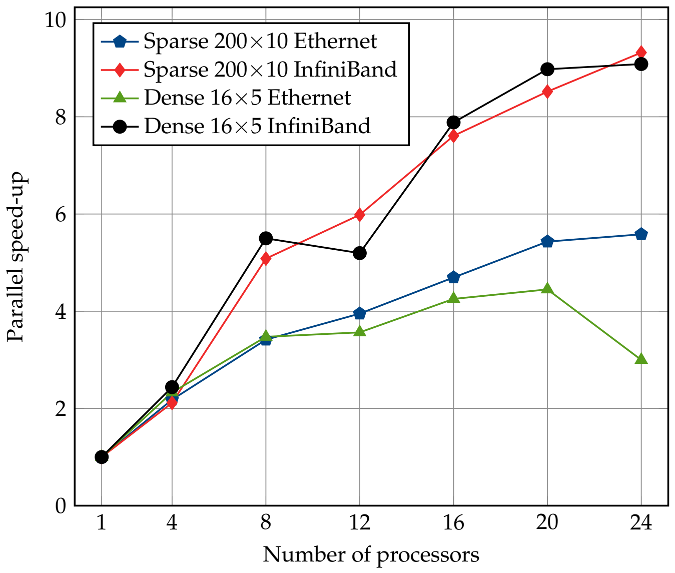


Figure 5. Impact of communication network on speed-up

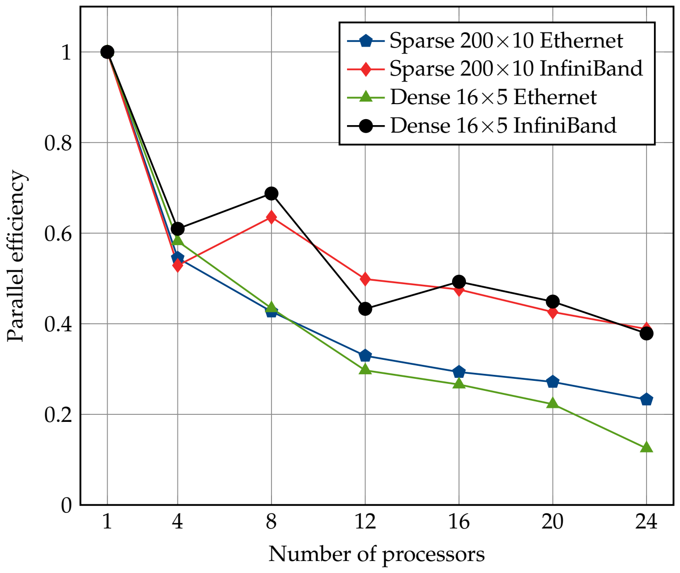


Figure 6. Impact of communication network on parallel efficiency

Sparse 200 × 10

p	Time [ms]	Speed-Up	Efficiency
1	21 415	1.00	1.00
4	9 821	2.18	0.55
8	6 271	3.41	0.43
12	5 420	3.95	0.33
16	4 561	4.69	0.29
20	3 942	5.43	0.27
24	3 836	5.58	0.23

Ethernet Sparse 200 × 10 InfiniBand

p	Time [ms]	Speed-Up	Efficiency
1	2 415	1.00	1.00
4	1 126	2.11	0.53
8	4 214	5.08	0.64
12	3 579	5.98	0.50
16	2 815	7.61	0.48
20	2 514	8.52	0.43
24	2 298	9.32	0.39

Dense 16 × 5 Ethernet

P	Time [ms]	Speed-Up	Efficiency
1	16 662	1.00	1.00
4	7 154	2.33	0.58
8	4 796	3.47	0.43
12	4 674	3.57	0.30
16	3 917	4.25	0.27
20	3 745	4.45	0.22
24	5 556	3.00	0.12

Dense 16 × 5 InfiniBand

p	Time [ms]	Speed-Up	Efficiency
1	16 662	1.00	1.00
4	6 834	2.44	0.61
8	3 030	5.50	0.69
12	3 207	5.20	0.43
16	2 113	7.88	0.49
20	1 856	8.98	0.45
24	1 834	9.09	0.38

Table 5. Impact of communication network

`operator delete()`	32.57 %
`SequentialInequalitySolver<>::Eliminate()`	16.17 %
`operator new()`	13.40 %
`tcmalloc::CentralFreeList::ReleaseToSpans()`	10.15 %
`SequentialInequalitySolver<>::Solve()`	9.24 %

We profiled the MPI solver in the same manner and the most significant MPI routine only amounted to 0.45 % of overall execution time.

4.3 Comparison to Other Solvers

The following two solutions (mentioned in Section 1.5) focused mainly on symbolic computation, the next two solutions are numerically oriented. As far as we know, all used implementations of solvers are strictly sequential.

4.3.1 Maple

Solving systems of linear inequalities in Maple is available through its `SolveTools`:

Solving the system (7) gives the following answer:

$$[\{1 \leq x, x \leq 8/3\}, \{y \leq 3/2 + 1/2\,x, -x + 3 \leq y\}],$$

$$[\{x \le 13/3, 8/3 < x\}, \{y \le 3/2 + 1/2\,x, -5 + 2\,x \le y\}].$$

Increasing the size of the input system has a significant impact on running time – producing an analytical answer to a small 5×5 system takes 33 seconds:

```
time[real](LinearMultivariateSystem({
-9*a+9*b-8*c+6*d-e <= 1000000,
-3*a+b+5*c+4*d+0*e <= 1000000,
4*a-10*b+4*c-6*d-2*e <= 1000000,
7*a-10*b-10*c+0*d+8*e <= 1000000,
7*a+4*b+4*c+8*d-7*e <= -1000000},
[a, b, c, d, e]));
33.291
```

Our sequential solver numerically solves this system in a mere 159 μs. Maple fails to solve larger systems – solving a system of 10 inequalities with 5 unknowns in Maple 13 results in an obscure error in intermediate computations after 45 seconds.

4.3.2 Maxima

Maxima can perform Fourier-Motzkin elimination using `fourier_elim`, but it aims for exact symbolic representation of the solution set.

Solving system (7) in Maxima results in the following output:

$$\left[x = \frac{8}{3}, y = \frac{1}{3}\right] \vee \left[x = \frac{13}{3}, y = \frac{11}{3}\right] \vee \left[x = \frac{y+5}{2}, \frac{1}{3} < y, y < \frac{11}{3}\right] \vee$$

$$\vee\, [x = 1, y = 2] \vee \left[x = 3 - y, \frac{1}{3} < y, y < 2\right] \vee \left[x = 2, y - 3, 2 < y, y < \frac{11}{3}\right] \vee$$

$$\vee \left[\max(3 - y, 2\,y - 3) < x, x < \frac{y}{2} + \frac{5}{2}, \frac{1}{3} < y, y < \frac{11}{3}\right].$$

The focus on symbolic computation has a significant drawback – larger systems of inequalities cannot be solved at all. Solving e.g. a system of 10 inequalities with 5 unknowns in Maxima 5.31.2 results in stack overflow error after 5 minutes of computation. In contrast, the resulting solver finds the answer to the same system in about 0.03 seconds.

4.3.3 FM-Eliminator

When we use this code instead of the block elimination, it is incapable of solving even small 16×5 systems in 5 minutes time. Our implementation was able to solve the same systems within seconds.

4.3.4 R

Despite the multi-platform support in R for other operations, package `editrules` (for purely sequential FME) is available only for Windows. When compared to our solver, this solution requests more memory – solving 16×5 systems fails most of the time. When the size of the system allows it to fit in the RAM, this solution and ours present comparable performances. No back substitution is performed and the final result of `eliminate` is often a large amount of inequalities with one coefficient, the bounds are not computed.

5 CONCLUSION

In this paper, we analyze each step of FME then we delve into parallelization strategies of FME. With emphasis on reusing the functionality present in the sequential solver, we carefully design an MPI-based parallel version and give an explanation of used optimizations.

The results are compared to theoretical expectations and show that we succeeded in achieving reasonable speed-up and efficiency with randomly generated data sets in the parallel solver.

6 FUTURE WORKS

Possible future improvements to this work would include designing and implementing an optimized vector representation, capable of both efficient arithmetic operations and of utilizing a custom memory allocation strategy. The memory allocator should take into account that large numbers of tiny vectors are allocated in later phases of FME. Another possible enhancement would be the implementation of type traits for arbitrary precision arithmetic data types, ideally for GNU multi-precision library integer, rational and floating point types.

Acknowledgements

This research has been supported by Czech Science Foundation project No. P103/12/2377 and by CTU internal SGS grant No. SGS15/120/OHK3/1T/18. This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as the Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).

REFERENCES

[1] Farkas, J.: Theorie der Einfachen Ungleichungen. Journal für die Reine und Angewandte Mathematik (Crelle's Journal), Vol. 1902, 1902, No. 124, pp. 1–27, doi:10.1515/crll.1902.124.1.

[2] Schrijver, A.: Theory of Linear and Integer Programming. Wiley Series in Discrete Mathematics and Optimization, John Wiley & Sons, 1998, ISBN 9780471982326.

[3] Kessler, C. W.: Parallel Fourier-Motzkin Elimination. Proceedings 2nd International EURO-PAR Conference, Springer, 1996, pp. 66–71.

[4] Dantzig, G. B.—Eaves, B. C.: Fourier-Motzkin Elimination and Its Dual. Journal of Combinatorial Theory, Series A, Vol. 14, 1973, No. 3, pp. 288–297, ISSN 0097-3165, doi:10.1016/0097-3165(73)90004-6. `http://www.sciencedirect.com/science/article/pii/0097316573900046`.

[5] Korovin, K.—Tsiskaridze, N.—Voronkov, A.: Conflict Resolution. In: Gent, I. P. (Ed.): 15th International Conference on Principles and Practice of Constraint Programming (CP 2009). Springer, Lecture Notes in Computer Science, Vol. 5732, 2009, pp. 509–523.

[6] Korovin, K.—Tsiskaridze, N.—Voronkov, A.: Implementing Conflict Resolution. Perspectives of Systems Informatics, 8th International Andrei Ershov Memorial Conference (PSI 2011), Novosibirsk, Russia, June 27–July 1, 2011, Revised Selected Papers, 2011, pp. 362–376, doi:10.1007/978-3-642-29709-0_31.

[7] Xue, J.: Loop Tiling for Parallelism. Kluwer Academic Publishers, Norwell, MA, USA, 2000, ISBN 0-7923-7933-0.

[8] Weispfenning, V.: Parametric Linear and Quadratic Optimization by Elimination. Mip-9404, Fakultät für Mathematik und Informatik, Universität Passau, 1994.

[9] OpenMP Architecture Review Board. OpenMP Application Program Interface. online, 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[10] Sanders, J.—Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 2010, ISBN 978-0131387683.

[11] Šimeček, I.: A New Approach for Indexing Powder Diffraction Data Suitable for GPGPU Execution. In: Snasel, V., Abraham, A., Corchado, E. S. (Eds.): Soft Computing Models in Industrial and Environmental Applications. Springer Berlin Heidelberg, Advances in Intelligent Systems and Computing, Vol. 188, 2013, pp. 409–416, ISBN 978-3-642-32921-0, doi:10.1007/978-3-642-32922-7_42.

[12] Šimeček, I.—Rohlíček, J.—Zahradnický, T. et al.: A New Parallel and GPU Version of a TREOR-Based Algorithm for Indexing Powder Diffraction Data. Journal of Applied Crystallography, Vol. 48, 2015, No. 1, pp. 166–170, doi:10.1107/S1600576714026466.

[13] What is Maple: Product Features. `http://www.maplesoft.com/products/Maple/features/`, Accessed: 2014-04-21.

[14] Maxima, a Computer Algebra System. `http://maxima.sourceforge.net/`, Accessed: 2014-04-21.

[15] fm-eliminator – Google Project Hosting. `https://code.google.com/p/fm-eliminator/`, Accessed: 2014-04-21.

[16] The R Project for Statistical Computing. `http://www.r-project.org/`, Accessed: 2014-04-21.

[17] Limongelli, C.—Pirastu, R.: *p*-Adic Arithmetic and Parallel Symbolic Computation: An Implementation for Solving Linear Systems over Rationals. Computers and Artificial Intelligence, Vol. 15, 1996, No. 1, pp. 35–62, ISSN 1335-9150.

[18] Che, Y.—Zhang, L. et al.: Optimization of a Parallel CFD Code and Its Performance Evaluation on Tianhe-1A. Computing and Informatics, Vol. 33, 2014, No. 6, pp. 1377–1399, ISSN 1335-9150.

[19] Šimeček, I.—Langr, D.—Tvrdík, P.: Space Efficient Formats for Structure of Sparse Matrices Based on Tree Structures. 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), September 2013, pp. 344–351, doi:10.1109/SYNASC.2013.52.

[20] Langr, D.—Šimeček, I.—Tvrdík, P. et al.: Parallel Data Acquisition for Visualization of Very Large Sparse Matrices. 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientic Computing (SYNASC), September 2013, pp. 336–343, doi:10.1109/SYNASC.2013.51.

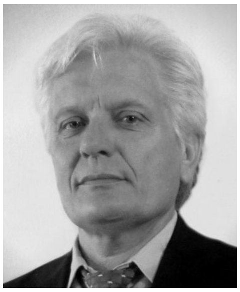
[21] Fagg, G.—Bosilca, G.—Pješivac-Grbović, J. et al.: Tuned: An Open MPI Collective Communications Component. In: Kacsuk, P., Fahringer, T., Nemeth, Z. (Eds.): Distributed and Parallel Systems. From Cluster to Grid Computing, Part II. Springer US, 2007, pp. 65–72, ISBN 978-0-387-69857-1.

[22] Tvrdík, P.: Parallel Algorithms and Computing. Third Edition. CTU Publishing House, 2005, ISBN 80-01-02824-0.

Ivan Šimeček is a lecturer and computer scientist at the Faculty of Information Technology (FIT), Czech Technical University in Prague, and a member of its Parallel Computing Group. His research activities include efficient design and architecture-dependent implementations of algorithms and related data structures in multi/many-threaded environments. He primarily focuses on sparse-matrix computations with the respect to cache behaviour, applied crystallography, parallel numerical linear algebra, and other compute-intensive problems.

Richard Fritsch graduated from the Faculty of Information Technology (FIT) of the Czech Technical University in Prague, he is a former Ph.D. student. Now he develops the VLAB system simulation environment in ASTC company (Australia). He primarily focuses on design, development and optimization of automotive navigation software in C, C++, and Java, cryptography, scripting in various languages, unit testing, development of profiling tools for multiple platforms.

Daniel Langr is Research Associate at the Faculty of Information Technology, Czech Technical University in Prague, Czech Republic, where he is a member of the Parallel and Distributed Computing Research Group. His main research interests include algorithms and data structures for large sparse matrices, efficient implementation of codes for large-scale HPC systems, and generic programming and template metaprogramming in C++.



Róbert Lórencz graduated from the Faculty of Electrical Engineering of the Czech Technical University in Prague in 1981. He received his Ph.D. degree in 1990 from the Institute of Measurement and Measuring Methods, Slovak Academy of Sciences in Bratislava. Currently he is Full Professor at the Faculty of Information Technology of the Czech Technical University in Prague. His research interests are cryptography and arithmetic units for cryptography primitives, various cryptoanalysis methods of block and stream ciphers. Another topic of his interest is alternative arithmetic for numerical computation.