# SYSTEM OPTIMIZATION OF SOLVING A SET OF LINEAR CONGRUENCIES

Luboš VONDRA, Tomáš ZAHRADNICKÝ, Róbert LÓRENCZ
Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
The Czech Technical University in Prague,
Karlovo náměstí 12, Prague 2, 121 35, Czech Republic,
E-mail address: {vondrl1|zahradt|lorencz}@fel.cvut.cz

**ABSTRACT**

*Solving a set of linear equations is a common problem and often it is transformed into a task of solving a set of linear congruencies. Solving a set of linear congruencies also appears frequently in cryptography and therefore effective solving algorithms are needed. All algorithms solving this task have their bottlenecks in calculating modular reduction as they need to divide. The problem with division or floating point remainder instruction is that these instructions have high latencies and are not pipelined. This paper compares several approaches that can be used to perform modular reduction after multiplication in integer and floating point arithmetic and its purpose is to offer a highly system optimized algorithm for modular multiplication with reduction using SIMD processor extensions.*

***Keywords:*** *Set of Linear Congruencies, Optimization, Residual Arithmetic, SIMD Extensions*

## 1. INTRODUCTION

Solving a set of linear equations (SLE)s is a basic task of linear algebra and for these purposes a computer implementation of floating point arithmetic as defined by the IEEE 754 [5] standard is used. Since precision of this arithmetic may not be sufficient for certain tasks, modular arithmetic is often used to extend it. The entire SLE is brought from floating point set to an integer domain with a transformation and possibly many sets of linear congruencies (SLC)s are solved instead [9]. Gaussian elimination is then used to obtain a solution to each SLC and once solutions to individual SLCs are available, a floating point solution is reconstructed with another transformation. Note there are no intermediate rounding errors committed during the solving process as the entire calculation occurs in an integer set and therefore a choice of the method to solve the SLC can be just Gaussian elimination with modular pivotization [8]. The pivotization is necessary as there is a higher probability that a zero occurs during the elimination process. Other advantage of this approach is that we can easily make the code run in parallel for large SLEs. This advantage is particularly interesting as it is useful to solve SLE or SLCs in a parallel environment on nondedicated clusters rather than special architectures. This paper deals with system optimization of the SLC calculation process, namely modular multiplication with reduction, and compares traditional approach performed in integer arithmetic to an approach performed in floating point arithmetic with and without the use of SIMD vector instructions on Intel x86 type processors.

## 2. SUBJECT

The standard approach of solving an SLE in modular arithmetic consists of 4 steps [7, 9]. These steps are scaling transformation, SLE reduction into SLCs, SLCs solving process, and finally a backward transformation.

1. *Scaling transformation*. This transformation takes the input SLE and performs scaling of its matrix and right hand side by a constant. The scaling procedure is necessary to ensure that the input SLE is representable within the integer set. Scaling can be performed for each row of the input matrix independently or globally for the entire SLE.

2. *SLE reduction into SLCs*. A number of prime number modules $q$ is chosen (typically $> 1000$) and modular reduction is performed for the entire SLE producing as many SLCs as the number of modules. A proper choice of modules is important as modular reduction will be often needed. In order to achieve most speed, modules are typically 15- or 16-bit as a product of two such numbers fits perfectly in a 32-bit computer register and modular reduction can be applied afterwards. The modular reduction process reducing SLE into SLC is done by using either `fmod` or `remainder` function from the "C" math library [1]. The `fmod` function calculates a result in $\mathbb{Z}_m$ and uses the `fprem` instruction [6] for these purposes, while the `remainder` function is useful if we are interested in obtaining a result in a symmetric residue system $\mathbb{S}_m$ [4]. The `remainder` function uses the `fprem1` instruction [6] which conforms to the IEEE standard in contrary to the `fmod` function.

3. *Solving SLCs*. SLCs obtained from the previous step are solved in this step. As they have nothing in common, they can be solved independently. If there is an error during the SLC solving process such as that a multiplicative inverse does not exist, we simply drop the module from the computation and continue working without that particular SLC. We can safely use Gaussian elimination with modular pivotization as there are no rounding errors committed during this step. The modular pivotization differs from full pivotization that it just chooses the first nonzero element before switching appropriate rows and/or columns in the matrix [9]. The most frequent operation during the elimination process is a modular multiplication with reduction which is this paper about.

4. *Backward transformation*. After we have solved SLCs, we obtain up to that many partial solutions that we recombine back into the floating point set yielding an SLE solution. This is done with a backward transformation using a mixed radix conversion (MRC) [4, 8] rather than the Chinese Remainder Theorem (CRT). The backward transformation with MRC is favored over the standard backward transformation based on the CRT as it can be performed in modular arithmetic and does not need a product of many modules which would not fit into a computer register.

## 3. METHODS

Sections 3.1 through 3.5 provide a detailed description of approaches that we have taken. They are all used to find a remainder mod $m$ after finding a product of two integers $a$ and $b$ and correspond to `z = ( a * b ) % m` in the C programming language. This task is the most frequent operation during the SLC solving process as we need to perform a reduction mod $m$ after every operation in order to avoid overflow in a computer register. The remainder mod $m$ is calculated during a modular version of Gaussian elimination with a complexity of $O(n^3q)$, where $n$ stands for an SLC dimension and $q$ for a number of prime number modules that we need for a solution, where we multiply a vector by a scalar variable and then reduce the computed vector by mod $m$. Undoubtedly, this task presents a bottleneck in the code, and therefore a system optimization should be used to obtain a performance improvement in this code.

The two main approaches that we have taken are in integral arithmetic and in floating point arithmetic. The integer arithmetic is a natural choice for reduction purposes as a single `div` instruction calculates both quotient and remainder and we can just pick the remainder. The alternative strategy is to employ the floating point arithmetic. Although, at the first sight, the floating point arithmetic does not look very attractive for the reduction purposes, the paper shows, that the possibility to use SIMD vector extensions to calculate *multiple* reductions with the same module at once is very enticing and several times faster than the original approach using the integer arithmetic.

Section 3.6 deals with the Gaussian elimination and application of the presented approaches for it. In addition the section presents the optimizations for other operations, not only the multiplication with reduction.

The approaches presented in sections 3.1 through 3.3 can be found in papers [10, 11].

### 3.1. Integer Approach

This approach is the traditional one. A product of integers $a$ and $b$ is calculated and a truncated quotient and remainder are obtained by using the `div` instruction. The quotient is simply discarded and the remainder is stored instead. When presented in an assembly language, this approach corresponds to the following code fragment:

```
mov eax, dword ptr [a]          ; load a into eax
mov edx, dword ptr [b]          ; load b into edx
mul                             ; calculate a*b in eax
mov edx, dword ptr [m]          ; load modulus into edx
div                             ; eax=TRUNC(a*b/m), edx=remainder
mov dword ptr [z], edx          ; store remainder
```

### 3.2. Floating Point Approach with fmod/remainder Functions

This is the first and the simplest approach that uses the floating point unit (FPU) instead of the integer unit. The approach loads 3 integers ($a$, $b$, and $m$) onto the floating point stack and uses the `fprem` or `fprem1` instruction afterwards depending on if we wish to obtain a result in $\mathbb{Z}_m$ or in $\mathbb{S}_m$. The following code corresponds to the implementation of the `fmod` function with the `fprem` instruction or `remainder` function with the `fprem1` instruction:

```
fild dword ptr [m]              ; modulus
fild dword ptr [a]              ; a modulus
fild dword ptr [b]              ; b a modulus
fmulp                           ; a*b modulus
fprem/fprem1                    ; calculate remainder
fistp dword ptr [z]             ; store remainder as integer
```

### 3.3. Optimized Floating Point Approach

A problem of the approach described in section 3.2 is that `fprem` and `fprem1` instructions have high latency[1]. These latencies are also caused by checks for validity of source operands, a need for division and also because of the fact that the floating point divisions are not pipelined. As we know that input operands are always valid and there is not likely that a floating exception is to be ever thrown, we can go around both `fprem/fprem1` and `div` instructions. We can use $1/m$ value which we can precalculate in advance of the computation and replace division with multiplication. When we do so, we obtain the following code:

```
;-------------------------------------------------------------------
;  Load the floating point stack and calculate a*b/modulus
;-------------------------------------------------------------------

fild dword ptr [a]              ; a
fimul dword ptr [b]             ; a*b
fld st0                         ; a*b a*b
fmul qword ptr [m_inv]          ; a*b/modulus a*b

;-------------------------------------------------------------------
;  Enforce rounding to integer by adding a rounding constant. Once
;  rounded, remove the constant by subtracting it.
;-------------------------------------------------------------------

fadd qword ptr [mmd_round]
fsub qword ptr [mmd_round]

;-------------------------------------------------------------------
;  Calculate the remainder
;-------------------------------------------------------------------

fimul dword ptr [m]             ; modulus*ROUND(a*b/modulus)
fsubp st1, st0                  ; remainder
fistp dword ptr [mmd_tmp]       ; store remainder as integer...

;-------------------------------------------------------------------
;  Add modulus if the remainder is < 0 or add zero otherwise.
;-------------------------------------------------------------------

mov eax, [mmd_tmp]              ; ...and load it into eax
mov edx, eax                    ; make a copy of eax into edx
sar eax, 31                     ; if eax < 0 then eax = -1 else 0
and eax, dword ptr [mmd_intp]   ; and module's value
add eax, edx                    ; add zero, or module
```

[1] According to [3], latencies of `fprem` and `fprem1` are 16 – 56 cycles depending on lengths of dividend and divisor.

### 3.4. Floating Point Approach with MMX and SSE2

Actually, there is a necessity for multiple reductions modulo $m$ as we solve SLCs and there we typically multiply a vector (matrix column) with a scalar during Gaussian elimination. In the following $a$ is a scalar, while $\mathbf{b}$ is a columnar vector and we rather calculate $\mathbf{z} = a\mathbf{b} \bmod m$. The size of operands in this approach is restricted and intermediate products shall not exceed 53-bit mantissa, therefore $a$, $m$, and elements of $\mathbf{b}$ must only be up to 26-bit wide. Because of an excessive source code length, only the most important portion of the unrolled modular multiplication with reduction is shown:

```
    ;-------------------------------------
    ;  Save the FPU state
    ;-------------------------------------
    fsave    [sse_fpu_state]

    ;-------------------------------------
    ;  Load parameters into registers
    ;-------------------------------------
    mov      ecx, [ebp+16]
    mov      edi, [ebp+12]
    mov      esi, [ebp+8]
    mov      ebx, [mmd_intp]

    ;-------------------------------------
    ;  Is vector of even or odd length?
    ;-------------------------------------
    mov      eax, ecx
    shr      ecx, 1
    and      eax, 1
    mov      [ebp-4], ecx
    mov      [ebp-8], eax
    or       ecx, ecx
    jnz      .loop_2x_sse
    jmp      .pre_loop_1x_sse

    align 32
  .loop_2x_sse:

    ;-------------------------------------
    ;  Process next 2 vector elements
    ;-------------------------------------
    cvtpi2pd xmm0, [esi]
    mulpd    xmm0, [sse_dbl_m]
    movapd   xmm1, xmm0
    movapd   xmm2, [sse_dbl_p]

    mulpd    xmm0, [sse_dbl_pinv]

    addpd    xmm0, [sse_round]
    subpd    xmm0, [sse_round]

    mulpd    xmm0, xmm2

    subpd    xmm1, xmm0

    cvtpd2pi mm0, xmm1

    movq     mm1, mm0
    psrad    mm0, 31
    pand     mm0, [sse_int_p]
    paddd    mm0, mm1

    movq     [edi], mm0

    ;-------------------------------------
    ;  Move to next 2 vector elements
    ;-------------------------------------
    add      esi, 8
    add      edi, 8
    dec      dword [ebp-4]
    jz       .pre_loop_1x_sse
    jmp      .loop_2x_sse
  .pre_loop_1x_sse:

    ;-------------------------------------
    ;  Process 1 vector element
    ;-------------------------------------
    mov      ecx, [ebp-8]
    jecxz    .loop_1x_sse_end
  .loop_1x_sse:

    mov      eax, [esi]
    mul      dword [sse_int_m]
    div      dword [sse_int_p]
    mov      [edi], edx

    ;-------------------------------------
    ;  Move to the next vector element
    ;-------------------------------------
    add      esi, 4
    add      edi, 4
    dec      dword [ebp-8]
    jnz      .loop_1x_sse
  .loop_1x_sse_end:

    ;-------------------------------------
    ;  Restore the FPU state
    ;-------------------------------------
    frstor   [sse_fpu_state]
```

Approach presented in this section can be found in [12].

### 3.5. Floating Point Approach with SSE2 by Intel Intrinsics

Since most of the operating systems currently runs in the 64-bit mode, there is a need of porting the presented algorithms to the 64-bit architecture. These ports became slightly difficult because of minor differences in the assembly languages of IA-32 and the x86-64 architecture, and therefore we have decided to implement the latest approach from section 3.4 by means of the Intel intrinsics functions instead. This implementation no longer requires assembly code inlines nor the assembler compiler.

Due to the performed loop unrolling we were able to eliminate the usage of the MMX technology instructions and we have unrolled the loop to process eight elements per each iteration. Thus the sequence of instructions `psrad`, `pand`, and `paddd` at the end of two elements processing has been superseded by their SSE2 equivalents.

This implementation can be compiled using the GNU C compiler and possibly the Intel C compiler[2]. Both compilers support the Intel intrinsics functions to perform the SSE technology instructions directly from the C source code, and such implementation is portable through IA-32 and x86-64 architectures without any problems.

### 3.6. Using SSE2 for Gaussian elimination

All of the optimization performed through sections 3.1 to 3.5 has been made to fully optimize the process of the Gaussian elimination in residual arithmetic. Note that the multiplication with reduction is not the only possible operation to be vectorized using SIMD instructions since almost all of the processing during the elimination is vector based.

Consequently, we have used SSE instructions also for the addition of the row multiple to another row(s) in the matrix. This operation can be written as $\mathbf{z} = \mathbf{a} - c\mathbf{b} \bmod m$, where $\mathbf{a}$ and $\mathbf{b}$ are columns of the SLC matrix and $c$ is a nonzero scalar. The source code for multiplication just needs to be extended with a load instruction and a subtraction of the elements of the second vector, and a reduction mod $m$ is performed after this operation.

We have also used vector processing during the backward substitution in the Gaussian elimination process, where SSE extensions are used to calculate a dot product mod $m$ of $\mathbf{a} \cdot \mathbf{y} \bmod m$, where $\mathbf{a}$ stands for a row of the SLC matrix, and $\mathbf{y}$ stands for the right hand side vector.

We have used the approach from section 3.5 and Intel intrinsics functions for this implementation.

[2] We have no Intel C compiler at disposal.

## 4. RESULTS

We have implemented all of the previous approaches in their vector form, and applied them on a problem of solving a SLC on 1.7 GHz Linux Intel machine. All timings were obtained with the `clock` API. The following table shows results obtained for multiplication with reduction $a\mathbf{b} \bmod m$ used during solving SLCs of various vector dimensions $n$ for C language and then for all 5 approaches we have presented:

**Table 1** Vector multiplication with reduction timings

| $n$ | "C" $[s]$ | App. 3.1 $[s]$ | App. 3.2 $[s]$ | App. 3.3 $[s]$ | App. 3.4 $[s]$ | App. 3.5 $[s]$ |
|---|---|---|---|---|---|---|
| $10^4$ | 0.000330 | 0.000215 | 0.000145 | 0.000098 | 0.000073 | 0.00001 |
| $10^5$ | 0.003648 | 0.002306 | 0.001484 | 0.001109 | 0.000755 | 0.00010 |
| $10^6$ | 0.037292 | 0.022908 | 0.016579 | 0.011686 | 0.009848 | 0.01003 |
| $10^7$ | 0.355189 | 0.232233 | 0.162232 | 0.112162 | 0.083057 | 0.10602 |
| $10^8$ | 3.464424 | 2.306178 | 1.580290 | 1.056258 | 0.821638 | 1.05307 |

For results from Tab. 1 drawn into a graph, a significant speed up is visible:

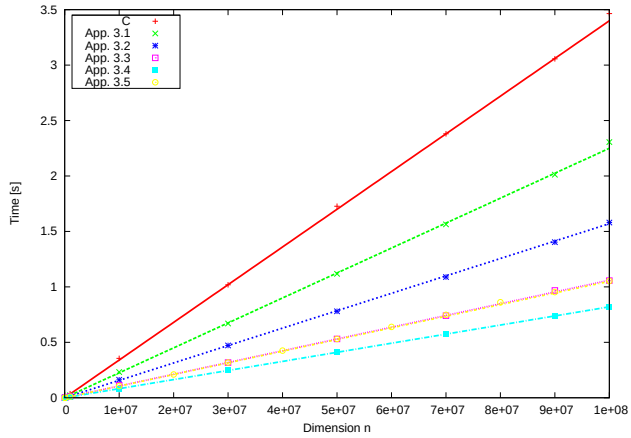


**Fig. 1** Vector multiplication with reduction timings

Fig. 1 plots all 5 approaches we have taken including the plain C implementation (the "C" column) which just uses `( a * b ) % m` followed by 5 approaches described in sections 3.1 through 3.5. It is important to note that timings have been measured for unrolled versions of presented algorithms that calculate vector multiplication with reduction and instructions have been blended with respect to processor architecture. Each approach was optimized separately at instruction level in order to obtain top performance.

There is an interesting observation that we can safely use the inverse module $1/m$ instead of just $m$ and avoid the division at all. Moreover, the FPU unit is not used in the final approach. The reduction is performed by 2 elements at once with SSE2 instructions with a support of MMX[3]. The speed-up which we obtained in approach 3.4, when compared to approach 3.1, tops 4.2 times.

We can observe that the approach from section 3.5 gives worse results than approach from section 3.4, but this slowdown is a cost for higher simplicity and portability.

[3]The MMX dependency has been removed in approach 3.5.

Finally, we have measured a timing for approach 3.6 and compared it to the Gaussian elimination implemented using pure C. Both timings are presented in Tab. 2 for various SLCs dimensions $n$.

**Table 2** Gaussian elimination timings

| $n$ | "C" $[s]$ | App. 3.6 $[s]$ |
|---|---|---|
| 1000 | 13.478000 | 4.592222 |
| 1500 | 45.243000 | 15.435556 |
| 2000 | 107.179000 | 36.250000 |
| 2500 | 209.139000 | 70.493750 |
| 3000 | 361.433000 | 121.618889 |

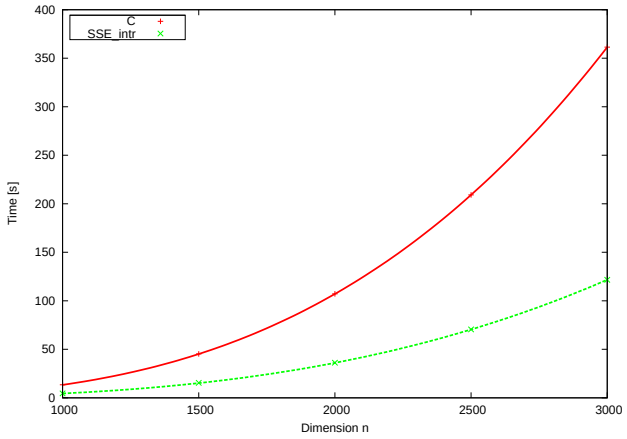


**Fig. 2** The time complexity of results from Tab. 2

## 5. DISCUSSION/CONCLUSIONS

This paper deals with a system optimization of the core problem in solving of a set of linear congruencies, that is a modular reduction after multiplication of two numbers. The reduction is necessary to avoid overflow. This problem appears eg. in Gaussian elimination which is commonly used during the solving process. Normally, the reduction is performed in an integer unit with the `div` instruction, but hence this instruction has high latency and is not pipelined, we would prefer a way without the division.

Another approach would be to perform the modular reduction after multiplication completely in the floating point unit with `fmod` and `remainder` C POSIX functions that correspond to `fprem` or `fprem1` instructions of the Intel architecture set and that calculate the remainder mod $m$ which we need. These two instructions have also high latencies, and, because we know the module $m$ in advance, we can go around using `fprem` and `fprem1` instructions by turning division by a module into multiplication by its inverse $1/m$. Inverse modules are typically precalculated once during an initialization process of a program.

Nevertheless, the process of Gaussian elimination features a multiplication of an entire matrix row $\mathbf{b}$ with the

same constant $a$ and therefore a need for modular reduction after multiplication with the same module $m$ arises ($\mathbf{z} = a\mathbf{b} \bmod m$). This need can be beneficial as we can perform multiple reductions at once with the help of processor features — namely SIMD instructions from the SSE2 processor extension. Taking this approach allows to compute two reductions at once and when several reductions are combined together and the code loops are unrolled, a significant speedup is achieved. This speedup is shown in Fig. 1 and presents more than a four times enhancement[4].

For a SLC solution using the Gaussian elimination with modular pivotization, we have achieved a speedup rate up to 3 times due to a use of SIMD vector extension instructions for multiplication, addition, and all other possible vector processing during the elimination process.

## ACKNOWLEDGEMENT

This research has been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under research program MSM 6840770014 and by the Czech Science Foundation as a project No. 201/06/1039.

## REFERENCES

[1] Apple Computer, Inc: *Libm-292.1 Source Code*, 2008, Mac OS X 10.5.2 source code available online at `http://www.apple.com/opensource`

[2] A broad group of parallel computer users, vendors, and software writers: *The Message Passing Interface (MPI) standard*, A software library for parallel computing, 2007, `http://www-unix.mcs.anl.gov/mpi`

[3] Fog, A.: *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel and AMD CPU's*, Copenhagen University College of Engineering, available online as a part of *Software optimization resources*, `http://www.agner.org/optimize/`

[4] Gregory, R. T., *Error-free computation: Why It Is Needed and Methods For Doing It*, Robert E. Krieger Publishing Company, Huntington, New York, 1980, ISBN 978-0898742404

[5] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute: *IEEE standard for binary floating-point arithmetic*, ser. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, 1985.

[6] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 3A, System Programming Guide, Part 1, 11/2007.

[7] Lórencz, R.: *New Approaches to Computing the Modular Inverse in Hardware*, Habilitation Thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2004.

[8] Lórencz, R., Morháč, M.: *Modular System for Solving Linear Equations Exactly, II. Hardware Realization and Firmware*, Computing and Informatics, vol. 11, no. 4, 1992, 497–507

[9] Morháč, M., Lórencz, R.: *Modular System for Solving Linear Equations Exactly, I. Architecture and Numerical Algorithms*, Computing and Informatics, vol. 11, no. 5, 1992, 351–361

[10] Vondra, L.: *Using Special Instructions for Multiplication with Reduction*, Počítačové architektury a diagnostika 2007. Fakulta aplikovaných věd, Západočeská universita, Plzeň, 2007, 47–52, ISBN 978-80-7043-605-9

[11] Vondra, L.: *Efficient algorithms for modular multiplication*, Proceedings of POSTER 2007, Faculty of Electrical Engineering, The Czech Technical University in Prague, 2007.

[12] Vondra, L., Zahradnický, T., Lórencz, R.: *System Optimization of Solving a Set of Linear Congruencies* Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering, 2008, 344–351, ISBN 978-80-8086-092-9

Received April 30, 2009, accepted July 24, 2009

## BIOGRAPHIES

**Luboš Vondra** was born on 16. 11. 1980 in Vlašim, Czech Republic. In 2005 he graduated (MSc) at the Department of Computer Science and Engineering of the Faculty of Electrical Engineering at The Czech Technical University in Prague. Since 2005 he is a postgraduate student at the same department. His scientific research is focusing on exact solution of linear equation sets in distributed environment especially at heterogenous clusters of workstations.

**Tomáš Zahradnický** was born on 9. 3. 1979, in Prague, Czech Republic. In 2003 he graduated (MSc) at the Department of Computer Science and Engineering of the Faculty of Electrical Engineering at The Czech Technical University in Prague. Since 2004 he is a postgraduate student at the same department and an assistant professor since 2007. His scientific research is focusing on system optimization, parameter extraction, and mathematical modeling.

**Róbert Lórencz** was born on 10. 08. 1957 in Prešov, Slovak Republic. In 1981 he graduated (MSc) at Faculty of Electrical Engineering at The Czech Technical University in Prague. He defended his PhD thesis in 1990. From 1998 he is an assistant professor at the Department of Computer Science and Engineering of the Faculty of Electrical Engineering at The Czech Technical University in Prague. Since 2005 he is an associate professor at the same department. His scientific research is focusing on numerical algorithms for linear algebra, residual arithmetic, error-free computation and algorithms for cryptography.

[4] An influence of communication complexity was not accounted.