Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

# System for Solving Linear Equation Systems

by

*Luboš Vondra*

A thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

PhD programme: Electrical Engineering and Information Technology
Specialization: Computer Science and Engineering

December 2013

**Thesis Supervisor:**
prof. Ing. Róbert Lórencz, CSc.
Department of Computer Systems
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Praha 6
Czech Republic

# Abstract and contributions

A huge number of problems in various fields of science leads to System of linear equations (SLE) representation. The problems that also lead to a differential equation set should be included here because their solution is obviously obtained by solving a corresponding linear equation system. Due to these facts it is very important to be able to solve SLE accurately and quickly.

The thesis presents a system for solving huge SLEs quickly, numerically stable and indeed accurately to eliminate the problem of generating rounding errors and their propagation through the rest of the computation. This problem is eliminated by solving the SLE by means of Residual Number System (RNS) and Gaussian elimination in modular arithmetic.

The main contributions of the thesis are as follows:

1. System optimizations of operation multiplication with reduction. The special case for many multiplications with one common factor (multiplication of vector by scalar value).

2. System optimizations of solving a SLE by means of RNS, including application of the first mentioned contribution.

3. A new mixed radix conversion algorithm suitable for parallel implementation including the possibility of balancing and sufficient precision determination.

**Keywords:**

Set of Linear Equations, System Optimizations, Residual Arithmetic, Residual Number System, Set of Linear Congruencies, Floating Point Arithmetic, Exact Arithmetic, Mixed Radix Conversion, Parallel Computing, Single Instruction Multiple Data

# Acknowledgements

First of all, I would like to express my gratitude to prof. Ing. Róbert Lórencz, CSc., my thesis supervisor. He has been a constant source of encouragement and insight during my research and provided me with numerous opportunities for professional advancements. His continuous support is gratefully acknowledged. His efforts as thesis supervisor contributed substantially to the quality and completeness of the thesis. I have learned a great deal from them. I have been fortunate to carry out the research for this dissertation under the supervision of prof. Ing. Róbert Lórencz, CSc.

I would like to express my special appreciation to my colleagues from the Applied Numerical Mathematics and Cryptography group, namely Ing. Tomáš Zahradnický, Ph.D., Ing. Jiří Buček and others.

The staff of our department has provided me with a pleasant and flexible environment for my research. Especially, I would like to thank doc. Ing. Miroslav Šnorek, CSc., the head of the department, and the former head of the department, prof. Ing. Pavel Tvrdík, CSc for taking care of my financial support. My work has been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under research program MSM 6840770014.

Many other people influenced my work. I wish to thank my colleagues from the Seznam.cz a.s., namely Ing. Michal Bukovský, Mgr. Jan Klesnil, Filip Volejník and others whose encouragement has always been a valuable source of support.

Finally, my greatest thanks go to my family and friends whose support was of great importance during the times of writing this thesis.

## Dedication

*To my wife Eliška, my sons Martin and Filip, my parents and the rest of my family.*

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# 1 Introduction

## 1.1 Motivation

Finding a solutionto a set of linear equations (SLE) is one of the most common mathematical problems. There are many well known and defined approaches such as various elimination and iteration methods for their solution.

Nevertheless, some SLEs cause considerable problems when one attempts to solve them using the currently easily accessible computational technology. These problems rise from the principle of how numbers are represented in digital devices.

One of the properties of the floating point representation is that it is not able to represent every real number. Moreover, the floating point representation cannot give the full range of rational numbers, but only a small subset of them. Thus, the usage of real numbers and their floating point representation is doubtful and it is complicated to recognize that a loss of accuracy has occurred.

On the other hand, the usage of integer numbers is transparent and computations using integer numbers are easily controlled. Using integer arithmetic is precise and if there is a problem, a flag indicating overflow of the integer range is set.

The most common standard for floating point representation is the IEEE organization standard [18]. When using this standard, the number of bits used for fraction representation is strictly specified. The number of bits is 23 or 52 bits for single or double precision. Knowing these fraction ranges clearly states the rounding errors of computation in such standards.

The range limitations are clear and it is straightforward that it is impossible to break them in hardware. There always has to be a limit in hardware implementation. According to the fraction and exponent limits the range of expressible numbers is given. This range could be extended on software level using a library for multiple precision computing. Such libraries cause a reasonable slowdown of the whole computation. In addition to using these libraries it is still necessary to determine the precision needed for the appropriate problem.

Due to the given problems, it is obvious that floating point arithmetic computations are burdened with plenty of rounding errors. These errors are generated and propagated through the computation, so the result can be frequently inaccurate.

Maybe it seems that those rounding errors are not important but clearly they are propagated through the whole computation. Thus the consequent step of the computation does not have the precise input data and so it is not possible to produce the right result. Moreover, each of the consequent computational steps generate their own rounding errors and these are also propagated through the rest of the computation.

Another way of solving the problem of rounding errors and their propagation is to use another mathematical model for finding a solution. Such a model should be a residual number system (RNS). This model provides the possibility of creating arbitrary large computer word, which leads to preserving all the information from the input data, and generates no rounding errors, because the computations are processed in residual arithmetic, where accuracy is guaranteed. There is also a problem of setting the needed precision but for RNS there is a mechanism for its estimation. This estimation is optimal in comparison with some others which are typically very pessimistic. Another advantage of the RNS is the natural degree of parallelism.

## 1.2 Problem statement

Our aim is to design a distributed system for solving SLEs capable of solving sets with dense coefficient matrices of large dimensions. This system is meant to be usable for clusters of workstations as well as for High Performance Computing (HPC) clusters. The solution process itself is to be error free which means no errors during the computation nor rounding errors.

## 1.3 Contributions of the thesis

There are two main contributions expressed in this thesis.

1. System optimizations of multiplication with reduction, especially the vector form of this operation. This also includes an overall optimization of the whole Gaussian elimination process. Some minor memory demand improvement.

2. The new Mixed Radix Conversion algorithm for distributed environment with possible dynamic weighting and recognition of sufficient precision. This also allows a

completely homogeneous design of application where all the nodes have exactly the same role.

## 1.4 Organization of the thesis

The thesis consists of the seven following chapters.

1. *Introduction*: Describes the motivation for our work and our goals. There is also a list of main contributions of the thesis.
2. *State-of-the-Art*: This chapter maps the current state-of-the-art in solving general dense SLEs exactly.
3. *Our Approach*: Describes the theoretical background of solving a SLE by means of RNS and Gaussian elimination. Presents initial design of the system this thesis is based on. Also presents a complexity analysis of the initial design.
4. *System Optimizations*: Describes the system optimizations performed during the development of the new system, especially multiplication with reduction operation.
5. *Decentralization*: Presents the design of a new decentralized version of the system including partial results on the design of parallel algorithms, such as mixed radix conversion and prime number generation.
6. *Conclusions*: Summarizes the results of our work and concludes the thesis.
7. *Future Work*: Suggests possible topics for further interests.

# 2 State-of-the-Art

The problem of linear equation systems is common in different fields of science. Thus it is not surprising that there are plenty of publications dealing with it.

Some publications focus on different pivoting strategies for different elimination methods. The choice of the pivoting strategy has a direct impact on the result and its accuracy. This is due to the fact that if the elements in the elimination process are well chosen, then there is no need for such a great precision. Numerous pivoting strategies are described in article [9]. Also, paper [29] introduced a new pivoting strategy for Gaussian elimination. But there is no guarantee for the solution to be exact just for a well chosen pivoting strategy. Nevertheless, none of the pivoting strategies should be marked as best for the general linear equation system.

Another often used technique for an ill-conditioned SLE solution is based on various regularization procedures. This technique tries to find a meaningful approximate solution to the problems with an ill-conditioned coefficient matrix. Paper [28] gives a comprehensive description of various regularization techniques. Other methods dealing with regularization of the matrix are expressed in [3, 4, 31].

Paper [30] describes a package for iterative solving of linear equation-systems which are ill-conditioned and have large dimensions. This package contains various iterative methods of SLE solution.

The basics of the method we use are presented in paper [27]. This article describes a system for solving linear equation systems exactly using the residual number system and it gives emphasis on the hardware implementation of the described algorithms.

The hardware implementation of the algorithms described in [27] is expressed in article [25] where the design of this system is presented.

The system described in this thesis is based on facts published in these articles.

There is also a software implementation of this algorithm. It is introduced in paper [16]. The article describes a software implementation of the procedure that solves SLEs by means of RNS. For this implementation, the FORTRAN language was used. The downside of this implementation is its simplicity. It has no support for parallel processing, therefore, it is not suitable for problems of large dimensions. Paper [17] is just a remark to this algorithm.

Another implementation of similar algorithms is described in papers [5–8], where a parallel

system for SLE solution is proposed using the RNS and distribution of individual Set of Linear Congruencies (SLC) solution to separate processors.

The SLE solution algorithm in [5–8] is basically divided into two parts. The first part of the processing is a simultaneous solution of separate SLCs, whereas the other is the mixed radix conversion algorithm itself. The parallel solution of SLCs is algorithmically almost identical to our approach. There is a difference in the knowledge of moduli count used. Doctor Koç presents algorithms where the count of moduli used is predefined. Our system, on the other hand, does not know the moduli count at the beginning of the computation. This count is determined by particular results of the Mixed Radix Conversion (MRC) algorithm which then has to be performed simultaneously with the SLCs solutions. The count of moduli used strictly corresponds with the accuracy of the computation. The more moduli we use the more accuracy we get. Thus we are determining whether the results have a sufficient accuracy and then the computation is finalized. So the algorithm cannot be strictly divided similarly to [5–8].

Articles [5–8] present different approaches to the parallel solution of the mixed radix conversion algorithm. There are some algorithms based on systolic arrays in [5, 8]. And the algorithm suitable for Symmetric multiprocessor architecture with distributed memory are presented in [6, 7]. Those parallel algorithms for the MRC are not suitable for usage in our system, due to the presented necessity of the MRC processed simultaneously with SLCs solution. The algorithm used for moduli count control is described in 3.1.5.

Works [A.7] and [10] deal with primary design of the system. These works describe the basic implementation of the system for SLE solution.

# 3 Our Approach

## 3.1 Theory

### 3.1.1 Outline

The standard approach to solving the SLE in modular arithmetic consists of 4 steps [23,27]. These steps are scaling transformation, SLE reduction into many 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 (RHS) by a constant. The scaling procedure is necessary to ensure that the input SLE is representable within an 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 moduli $q$ is chosen and modular reduction is performed for the entire SLE producing as many SLCs as the number of moduli. A proper choice of moduli is important as modular reduction will often be needed.

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.

4. *Backward transformation.* After we have solved SLCs, we obtain up to so 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) [12, 14, 27].

### 3.1.2 Transformation to integer numbers

Let us presume that in this section the number called the smallest element means the number with the smallest absolute value. To correspond with this, the function *min* also returns the number with the smallest absolute value.

The whole computation is implemented in residual arithmetic, so the first step is transformation into $\mathcal{Z}$. The early implementation used for such a transformation formulae 3.1.

$$a_{ij} = \frac{2^{m+1}}{\min(a_{i1}, a_{i2}, \ldots, a_{in}, y_i)} a_{ij} \qquad 1 \leq i, j \leq n \tag{3.1a}$$

$$y_i = \frac{2^{m+1}}{\min(a_{i1}, a_{i2}, \ldots, a_{in}, y_i)} y_i \qquad 1 \leq i \leq n \tag{3.1b}$$

The first formula in 3.1a fits for elements of coefficient matrix $\mathbf{A}$. The other 3.1b fits for right-hand side vector $\mathbf{y}$. The identifier $m$ represents the mantissa size, $a_{ij}$ represents the elements of the coefficients matrix and $y_i$ represents the RHS vector elements.

The terms 3.1 should be described in a few sentences. Each row is adjusted separately, so the smallest element of the whole row (including the appropriate element of the right-hand side vector) has to be found. Then the whole row is divided by that element. After the division, the smallest element of the row has the value of 1. Finally, all the elements of the row are shifted to the left by the size of fraction ($m$) bits. Now all the elements are integer numbers.

During the experiments we discovered a problem with this part of computation. The division operation is a generator of certain numerical errors that are propagated down the whole computation. Consequently, results are obtained for a different set of linear equations.

This problem is easily solved by using the shift operation instead of a division operation. Numbers are not divided by the smallest element but they are shifted to the right by the number corresponding to the exponent of the smallest element. After this operation the value of the smallest element is in the range $\langle 1, 2)$ and all the other elements in the row are greater than 1. Then these numbers are shifted to the left just in the same way as they were in 3.1. The final formulae are presented in form 3.2.

$$min_i = min_{0 \leq k \leq n}(a_{ik}, y_i) \tag{3.2a}$$

$$a_{ij} = \frac{2^{m+1}}{2^{exp(min_i)}} a_{ij} \qquad 1 \leq i, j \leq n \tag{3.2b}$$

$$y_i = \frac{2^{m+1}}{2^{exp(min_i)}} y_i \qquad 1 \leq i \leq n, \tag{3.2c}$$

In formulae 3.2 the identifier $exp(x)$ represents the exponent of the x value. The $min_i$

represents the element with the smallest absolute value in the $i$-th row. When using formulae 3.2 for the transformation into integer numbers, numerical errors caused by the division operation are avoided.

Using formulae 3.2 for the transformation into integer numbers causes a limitation of the available input range. If there is a need to use the whole original input range, then the range used for the transformation has to have different parameters. The following paragraphs discuss the required ranges.

We should express the relationship for the input range. Let the desired input range has $m$ bits of mantissa, the least acceptable exponent is $e_{min}$ and the largest exponent is $e_{max}$. Then it is necessary to find out the parameters of the range needed for the transformation which are marked as $m_1$, $e_{min1}$, $e_{max1}$. These parameters should be set to allow the transformation of every representable number of the original input range.

$$e_{max1} = e_{max} - e_{min} + m \tag{3.3a}$$

$$e_{min1} = \min(0, e_{min}) \tag{3.3b}$$

$$m_1 = m \tag{3.3c}$$

The relation of the original input range parameters and the parameters of the range needed for an errorless conversion is gathered in formulae 3.3. The relevance of 3.3c is straightforward because during the conversion there are no other operations but a shift operation and there is no need for fraction enlargement.

Formula 3.3b states the relation of the least exponent of the original and the new ranges. The least exponent from the new range is set to minimum of 0 and the original $e_{min}$. This fact rises from the shifting process where numbers are shifted, so they have the exponent equal to 0 and then they are shifted to the left by the size of the fraction.

Finally the greatest exponent which can arise from formulae 3.2 has the value according to 3.3a. This is the case when one row contains both: the element with the smallest possible exponent $e_{min}$ and the greatest one $e_{max}$.

### 3.1.3 Transformation into $\mathcal{Z}_n$

This transformation is performed by **fmod** function, which returns the remainder after division, in floating point representation. This function and its characteristics are described in 3.2.2

### 3.1.4 Gaussian elimination with non-zero pivoting

Gaussian elimination algorithm is commonly used and well understood, so it is described only very briefly. In addition, it is supplied with non-zero pivoting. At the beginning, the form of the expanded coefficient matrix corresponds to formula 3.4.

$$(\mathbf{A}|\mathbf{y}) = \left(\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & y_1 \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} & y_2 \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} & y_3 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} & y_n \end{array}\right) \tag{3.4}$$

The Gaussian elimination is split into two phases. The first one is modification to eliminate some variables in the right equations. More accurately, the first phase consists of transformation of the matrix 3.4 into upper triangular matrix expressed in formula 3.5.

$$\left(\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & y_1 \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \dots & a_{2n}^{(1)} & y_2^{(1)} \\ 0 & 0 & a_{33}^{(2)} & \dots & a_{3n}^{(2)} & y_3^{(2)} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn}^{(n-1)} & y_n^{(n-1)} \end{array}\right) \tag{3.5}$$

The algorithm can be written in a symbolic way by formulae 3.6 and 3.7.

$$a_{ij}^{(0)} = a_{ij} \qquad i = 1, \dots, n \qquad j = 1, \dots, n+1 \tag{3.6}$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)} \quad k = 1, \ldots, n-1 \quad i = k+1, \ldots, n \quad j = k+1, \ldots, n+1 \tag{3.7}$$

Adjusting the expanded coefficient matrix may result into the element $a_{k+1k+1}^{(k)}$ in the $k$-th step of the elimination being equal to zero. And such an element should not be used for the $k+1$-st step of elimination, so it is necessary to modify the matrix. One and the easiest way is non-zero pivoting from [25] where such an element is exchanged with another non-zero element of the matrix. More precisely, row $k$ is swapped with row $l$ where $l > k \wedge a_{lk} \neq 0$ holds. This row swapping implies the change of the determinant sign which is also calculated during the first phase.

Some other pivoting strategies are mentioned in [9] and [29].

Determinant is then the result of the product of elements on the main diagonal. Finally, its sign is set according to the number of row swaps during the elimination. Formula 3.8 expresses the determinant value. Identifier $l$ is the number of row swaps.

$$D = \prod_{i=1}^{dim} a_{ii}^{(i-1)} \cdot (-1)^l \tag{3.8}$$

The second phase of the algorithm just calculates the variables. The value of the variable corresponding to the last column of the expanded coefficient matrix is determined directly by the last row of 3.5. This value is used to gain the value of the variable corresponding to the last but one column. And so the process continues until having values of all variables.

### 3.1.5 RNS and mixed radix representation

The transformation from the RNS back into the integer numbers can be solved by Chinese remainder theorem (CRT). Instead of this classical way, the system uses another method which is based on the relation between RNS representation and mixed radix representation. The reason for the usage of another algorithm is the complexity of the CRT transformation. The following description of the conversion process was taken from [22]. The extension for signed numbers is formed in [27].

We have an n-tuple $\rho = [r_1, r_2, \ldots, r_n]$, where the components are *radices*. Let $R$ is a

product of these radices so $R = \prod_{i=1}^{n} r_i$ holds.

It is straightforward that every number $s$ from the range $0 \leq s < R$ can be expressed in the form corresponding to formula 3.9.

$$s = d_0 + d_1(r_1) + d_2(r_1 r_2) + \ldots + d_{n-1}(r_1 r_2 \ldots r_{n-1}) \tag{3.9}$$

Identifiers $d_0, d_1, \ldots, d_{n-1}$ in equation 3.9 are standard mixed radix digits which have to conform to relation 3.10.

$$0 \leq d_i < r_{i+1} \quad i = 0, 1, \ldots, n-1. \tag{3.10}$$

The mixed radix representation are numbers $d_0, d_1, \ldots, d_{n-1}$ which can be written down in form 3.11.

$$\langle s \rangle_{\rho} = \langle d_0, d_1, \ldots, d_{n-1} \rangle \tag{3.11}$$

The special case of the mixed radix representation is a fixed radix representation with all radices having the same value ($r_1 = r_2 = \cdots = r_n$). For example, when having all radices $r_i = 10$ then, such a representation is the common decimal number system. But a more interesting and relevant case is when radices are equal to moduli of RNS, thus $r_i = m_i$ for $i = 1, 2, \ldots, n$. Then the range of the multi-modulus residual arithmetic and mixed radix representation are equal.

It is necessary to find the relations for the integer number $s$ and its representations in residual number system $|s|_{\beta}$ and mixed radix representation $\langle s \rangle_{\beta}$ now.

The representation $|s|_{\beta}$ in the RNS and its relation to the number $s$ is described in section 3.1.2.

If the number $s$ is known, the process to determine the mixed radix representation $\langle s \rangle_{\beta}$ follows. Let $s = t_1$ and the relation for $d_0$ is expressed in formula 3.12.

$$\begin{aligned} t_1 &= s \\ &= d_0 + m_1 \left[ d_1 + d_2(m_2) + \ldots + d_{n-1}(m_2 \ldots m_{n-1}) \right] \\ &= d_0 + m_1 t_2 \end{aligned} \tag{3.12}$$

The relation 3.13 can be written down as

$$\begin{aligned} |t_1|_{m_1} &= |d_0 + m_1 t_2|_{m_1} \\ &= d_0 \\ &= |s|_{m_1}. \end{aligned} \tag{3.13}$$

From the relations 3.13 and 3.12 it should be clear that there is no need to perform any processing to determine the value of $d_0$ because it is equal to the first residue in RNS representation. Then the relations for $t_2$ and $d_1$ are expressed in formulae 3.14 and 3.15.

$$\begin{aligned} t_2 &= d_1 + m_2 \left[ d_2 + d_3(m_3) + \ldots + d_{n-1}(m_3 \ldots m_{n-1}) \right] \\ &= d_1 + m_2 t_3. \end{aligned} \tag{3.14}$$

$$\begin{aligned} |t_2|_{m_2} &= |d_1 + m_2 t_3|_{m_2} \\ &= d_1 \end{aligned} \tag{3.15}$$

It is easy to extract the recursive form for $d_i$ elements of the mixed radix representation from formulae 3.12, 3.13, 3.14 and 3.15. The initial value is $t_1 = s$. This recursive formula is expressed in 3.16.

$$\begin{aligned} & d_0 = |t_1|_{m_1} \\ & t_1 = s \\ t_{i+1} = \frac{t_i - d_{i-1}}{m_i} \qquad & d_i = |t_{i+1}|_{m_{i+1}} \qquad i = 1, 2, \ldots, n-1. \end{aligned} \tag{3.16}$$

Those relations are only prerequisites for conversion from residual representation to mixed radix representation. This process is based on recursive formula 3.16 for $t_{i+1}$.

If the residual representation is known, the following relations hold:

$$|t_1|_\beta = [d_0, |t_1|_{m_2}, \ldots, |t_1|_{m_n}]$$

$$|d_0|_\beta = \left[|d_0|_{m_1}, |d_0|_{m_2}, \ldots, |d_0|_{m_n}\right].$$

Combining these two relations, we get the following notation:

$$|t_1 - d_0|_\beta = \left[0, \left|z_2^{(1)}\right|_{m_2}, \left|z_3^{(1)}\right|_{m_3}, \ldots \left|z_n^{(1)}\right|_{m_n}\right],$$

where

$$z_i^{(1)} = |t_1|_{m_i} - |d_0|_{m_i} \quad i = 2, 3, \ldots, n.$$

Now we can establish the reduced base vector $\beta_1$.

$$\beta_1 = [m_2, m_3, \ldots, m_n].$$

And the value of $t_1 - d_0$ with base vector $\beta_1$ is

$$|t_1 - d_0|_{\beta_1} = \left[\left|z_2^{(1)}\right|_{m_2}, \left|z_3^{(1)}\right|_{m_3}, \ldots \left|z_n^{(1)}\right|_{m_n}\right].$$

In order to be able to compute the value of $t_2$, a multiplicative inverse $m_1^{-1}(\beta_1)$ has to exist and has to be known. Those inverses exist because all the elements of $\beta_1$ are prime numbers. And the condition for existence of multiplicative inversion $|x^{-1}|_m$ of $x$ is that $x$ and $m$ are relatively prime. Then the inversions can be expressed as:

$$m_1^{-1}(\beta_1) = \left[m_1^{-1}(m_2), m_1^{-1}(m_3), \ldots, m_1^{-1}(m_n)\right].$$

Then the $t_2$ can be expressed in the form:

$$\begin{aligned} |t_2|_{\beta_1} &= \left|\frac{t_1 - d_0}{m_1}\right|_{\beta_1} \\ &= \left[\left|w_2^{(1)}\right|_{m_2}, \left|w_3^{(1)}\right|_{m_3}, \ldots, \left|w_n^{(1)}\right|_{m_n}\right], \end{aligned} \tag{3.17}$$

where

$$w_i^{(1)} = \left|z_i^{(1)}\right|_{m_i} m_1^{-1}(m_i) \quad i = 2, 3, \ldots, n.$$

When using formula 3.16, we get the second element of the mixed radix representation $d_1$.

$$\begin{aligned} \left|w_2^{(1)}\right|_{m_2} &= |t_2|_{m_2} \\ &= d_1 \end{aligned}$$

We can use it in equation 3.17 and then we can write:

$$|t_2|_{\beta_1} = [d_1, |t_2|_{m_3}, \ldots, |t_2|_{m_n}] \tag{3.18}$$

$$|d_1|_{\beta_1} = [|d_1|_{m_2}, |d_1|_{m_3}, \ldots, |d_1|_{m_n}]\,. \tag{3.19}$$

These relations are similar to relations for $d_1$. Repeating the process for the rest, we will get $d_2$, $d_3$, …, $d_{n-1}$.

This algorithm can be easily written in a table where the principle can be understood much more easily. Table 3.1 shows the steps of the algorithm, for example, where $\beta = [13, 11, 7, 5, 17]$ and $|s|_\beta = [4, 2, 4, 2, 5]$. The elements of mixed radix representation corresponding with relation 3.16 are marked out in **bold**.

The above algorithm has to be extended for signed numbers because when used in this fashion, it works only with positive numbers. The extension is based on double conversion, one for the number (residue) itself and the other for its negative value. There, the negative value means the number (residue) subtracted from the corresponding modulus $m_i$. Afterwards it is necessary to determine which of the two results will be used. If the positive result is less than $M/2$, then it is the actual final value, otherwise, if the result gained with negative partial results fulfills the condition of being less than $M/2$, then the negative value of this result is the final value.

Finally, all the elements of the result vector are divided by the determinant to obtain rational number results according to section 3.1.4.

The transformation algorithm is very important for the termination of the whole computation. The process terminates when elements of the mixed radix representation marked in table 3.1 are zero for all the elements of the result vector. This condition has to be fulfilled for at least one of the representations (positive/negative). When it is satisfied, there is no need to extend the range and this means that no other SLCs have to be solved.

This method of process termination does not guarantee the achievement of a sufficient range ($M$) to solve the SLE exactly without any rounding errors. But the probability of

| $\beta$ | $m_1 = 13$ | $m_2 = 11$ | $m_3 = 7$ | $m_4 = 5$ | $m_5 = 17$ | op. |
|---|---|---|---|---|---|---|
| $\|t_1\|_\beta$ | **4** | 2 | 4 | 2 | 5 | |
| $\|d_0\|_\beta$ | 4 | 4 | 4 | 4 | 4 | $-$ |
| $\|t_1 - d_0\|_\beta$ | 0 | 9 | 0 | 3 | 1 | |
| $m_1^{-1}(\beta_1)$ | | 6 | 6 | 2 | 4 | $\cdot$ |
| $\|t_2\|_{\beta_1} = \|\frac{t_1 - d_0}{m_1}\|_{\beta_1}$ | | **10** | 0 | 1 | 4 | |
| $\|d_1\|_{\beta_1}$ | | 10 | 3 | 0 | 10 | $-$ |
| $\|t_2 - d_1\|_{\beta_1}$ | | 0 | 4 | 1 | 11 | |
| $m_2^{-1}(\beta_2)$ | | | 2 | 1 | 14 | $\cdot$ |
| $\|t_3\|_{\beta_2} = \|\frac{t_2 - d_1}{m_2}\|_{\beta_2}$ | | | **1** | 1 | 1 | |
| $\|d_2\|_{\beta_2}$ | | | 1 | 1 | 1 | $-$ |
| $\|t_3 - d_2\|_{\beta_2}$ | | | 0 | 0 | 0 | |
| $m_3^{-1}(\beta_3)$ | | | | 3 | 5 | $\cdot$ |
| $\|t_4\|_{\beta_3} = \|\frac{t_3 - d_2}{m_3}\|_{\beta_3}$ | | | | **0** | 0 | |
| $\|d_3\|_{\beta_3}$ | | | | 0 | 0 | $-$ |
| $\|t_4 - d_3\|_{\beta_3}$ | | | | 0 | 0 | |
| $m_4^{-1}(\beta_4)$ | | | | | 7 | $\cdot$ |
| $\|t_5\|_{\beta_4} = \|\frac{t_4 - d_3}{m_4}\|_{\beta_4}$ | | | | | **0** | |

Table 3.1: Example transformation (RNS to MR)

an insufficient range is very small. The mechanism for computation finishing is described in [24].

## 3.2 Implementation

### 3.2.1 Transformation into integer number space $\mathcal{Z}$

Implementation of the transformation of input data set into an integer number range is just a simple coding of process described in 3.1.2. No notable problems in this field exist.

### 3.2.2 Transformation into $\mathcal{Z}_n$

To gain the remainder after division the **fmod** function from standard C library is used.

During our development and testing we have used the GNU C compiler and the GNU C library as well.

When using the GNU C library implementation this function should be errorfree for the following architectures, according to the GNU C library manual [21]:

- Alpha
- intel x86 architecture including x86_64 architecture.
- IA64 (Itanium)
- Sparc in both 32 bit and 64 bit variants
- PowerPC
- S/390

But in the GNU C compiler, this function is also built-in and thus the compiler is free to use its own implementation under certain conditions. In case of problems the built-in function implementation can be suppressed.

Finally, this function will be translated into the `fprem` instruction [19] for x86 architecture processors.

### 3.2.3 Set of linear congruencies

To solve one set of linear congruencies with modulus $m$ a common Gaussian elimination in residual arithmetic is used. This means that we have to take care of the magnitude of every result of every operation and perform modular reduction if necessary.

### 3.2.4 The calculation of Final results

Final results are gained from the vector of values calculated according to the algorithm expressed in section 3.1.5 using formula 3.9. There is a need to use a multiple precision floating point library. The implementation strictly follows formula 3.9. At the beginning of computation we predict some degree of precision which is used for this computation. Whenever the results are in the range where some precision could be lost, the range used for this computation is extended accordingly. We can state that for every modulus used there is a need to represent all the possible values with full precision (without any loss). Thus when using, for example, ten modules of 31-bit size, it is necessary to use the range with 310 bits of mantissa, because the product of ten 31-bit numbers can be the number with a 310-bit representation.

This form of processing guarantees that full precision of the result is preserved.

Certainly the final result is not expressed with such a precision. There is a default value for mantissa size of the output vector. This default value can be overridden with a concrete desired precision.

### 3.2.5 Final result verification

Due to the fact that the completion of the computation is done basically on probability principles, it is necessary to verify the accuracy of the results. Thus, we perform the test whether the left-hand side (LHS), applying the results, is equal to the RHS in every linear equation. The verification process can be described as formulae 3.20.

$$\begin{gathered} \delta\mathbf{x} = \frac{A\mathbf{x}-\mathbf{y}}{\mathbf{y}} \\ \forall i, 0 \leq i < dim \quad \delta\mathbf{x}_i < t \end{gathered} \tag{3.20}$$

The tolerance $t$ in formulae 3.20 is implicitly set to $10^{-6}$ and it can be easily changed.

### 3.2.6 System Architecture

The application which processes the SLE solution is the core of the whole system. Since its early implementation it has been designed as a parallel application because of its time complexity. The complexity is high because there is a lot of SLCs for every task to solve. This application is designed for parallel clusters using MPI (Message Passing Interface) communication protocol. For more information on MPI see MPI homepage[1].

The application itself is divided into two parts. The first can be described as a **master** process and the other one as a **slave** process. The relationship between these two kinds of processes is demonstrated in figure 3.1.

The computation can be described as a sequence of three phases. The next paragraph describes each phase solely in view of **master**/**slave** process partitioning.

1. This phase should be called **Initialization**. The **master** process allocates the memory needed. Then it reads the input data from the disk and performs the basic conversion of the problem to integer numbers according to section 3.1.2. The **slave** process just allocates memory locations needed.

2. The next phase is the most time consuming part of the computation. **Master** process assigna moduli to **slave** processes (*assign modulus* in figure 3.1) and processes results received from **slave** processes. The way of processing the result is theoretically discussed in section 3.1.5, which also includes sufficient precision resolving. **Slave** processes are responsible for receiving moduli, processing the Gaussian elimination in residual arithmetic with all the received moduli (3.1.4), and providing SLC solutions back to the **master** process.

3. In the last phase, only administrative tasks are performed. **Master** process sends a *stop* message to **slave** processes and stores the final results. All processes clean up their environment.

## 3.3 Complexity

This section concentrates on the complexity of the whole system from different points of view. It is divided into three subsections according to the complexity type. The first section

[1] http://www-unix.mcs.anl.gov/mpi/

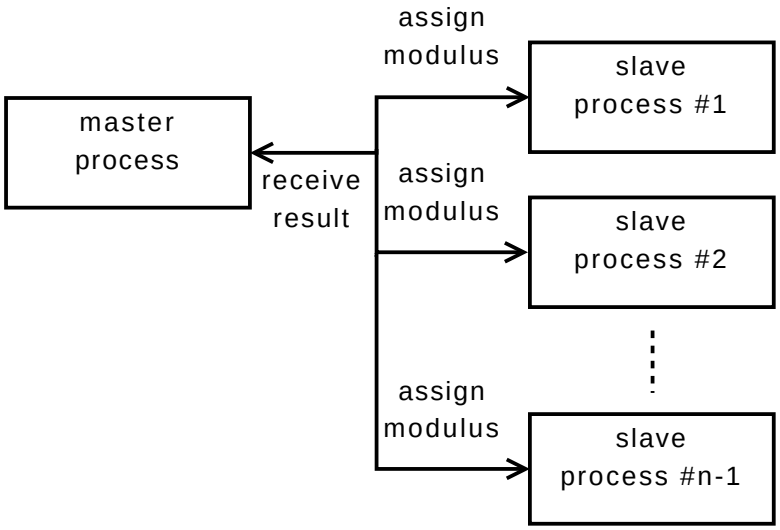


Figure 3.1: Linsolve architecture

3.3.1 deals with the asymptotic time complexity of every single part of the algorithm. The second subsection 3.3.2 is focused on memory consumption and its effects. The last 3.3.3 expresses the communication complexities.

The parameters of the relations vary. Some of them are clearly understood and stated like the number of cluster nodes $p$ and the dimension of the SLE $n$. However, the parameter number of moduli used for computation is not so clear and its value is not known. The only way to find its value is to perform the computation.

The count of moduli used should theoretically correspond to the range required to solve the problem, but in practice it is not easy to determine this value.

### 3.3.1 Time complexity

This section is divided according to the division of the algorithm in section 3.1. It should be remembered that sections 3.3.1.1 and 3.3.1.3, corresponding to sections 3.1.2 and 3.1.5, are performed by the master process (on the main node). Section 3.3.1.2 states the complexity of SLC solution which is handled in slave processes. The difficulty of the whole algorithm is summarized in section 3.3.1.4.

The formulae presented through sections 3.3.1.1 to 3.3.1.3 were taken from [10]. The final complexity of the whole computation stated in 3.3.1.4 can be derived from those relations

and the knowledge of the systems behaviour.

#### 3.3.1.1 Transformation to integer numbers

Transformation to the integer number range is completely determined by the dimension of the problem. The transformation of one value in agreement with the formula 3.2 takes constant amount of time. The time complexity of such a transformation is as follows.

$$t_{toint} = \mathcal{O}(n^2) \tag{3.21}$$

#### 3.3.1.2 SLC

Asymptotic complexity of computation of one set of linear congruencies is given by the complexity of the Gaussian elimination. This algorithm is well known and its complexity is $t_{SLC} = \mathcal{O}(n^3)$.

Furthermore, the Gaussian elimination algorithm is performed for every modulus. The asymptotic complexity for the process of all the eliminations is expressed by formula 3.22.

$$\mathcal{O}(m \cdot n^3) \tag{3.22}$$

In addition, the time needed for the transformation from integer numbers into residue arithmetic should be taken into account. But this conversion has the same asymptotic complexity as the transformation into integer numbers expressed in form 3.21, so it does not affect the relation 3.22.

#### 3.3.1.3 Transformation to real numbers

Time complexity of the transformation from RNS into rational numbers depends on the number of moduli used for the computation. This complexity should be expressed by relation 3.23.

$$\mathcal{O}(m^2 n) \tag{3.23}$$

Relation 3.23 arose from a detailed analysis of the algorithm described in section 3.1.5. When studying table 3.1, we can say that for the transformation of one number it is

necessary to perform $m^2$ computational steps. It is obvious that for the conversion of the whole result vector of dimension $n$ the complexity corresponds to formula 3.23.

In fact , $2n+2$ conversions are performed because of an independent transformation of positive and negative representations and the determinant. Asymptotically, the final relation is truly formula 3.23.

#### 3.3.1.4 Overall complexity

When combining the complexity of the whole computation, all the parameters have to be taken into account. The final complexity of the whole system is not just a trivial combination of the relations expressed in sections 3.3.1.1 to 3.3.1.3. Actually, the final complexity should be written in form 3.24.

$$\mathcal{O}(n^2 + max(m^2 n, m \cdot n^3 \cdot (p-1)^{-1})) \tag{3.24}$$

The first element ($n^2$) of 3.24 represents conversion into integer numbers. The second element of the addition in 3.24 represents the maximum of the time needed for the transformation from the RNS into rational numbers and the SLCs solution itself. The time for SLCs processing is divided by the number of available processors.

We suppose the same hardware configuration for all the nodes used for the computation. If this condition is not fulfilled, then it is impossible to estimate the complexity of the whole computation and the validity of formula 3.24 would be doubtful.

The second part of relation 3.24 shall be described in more details. The meaning of this term is straightforward. When the transformation from the RNS into rational numbers becomes more complicated than the SLCs solution itself, the master process will turn into a bottleneck of the computation. This is due to the fact that the master process will be overloaded and the assignment of the moduli for the slave processes will fail. So, the computational capacity of the slave nodes will not be fully utilized. In this case, the slave nodes will be idle and will wait for the modulus to start the elimination.

This problem would arise in two ways. In the first instance the problem requires too many moduli and in the other where there are too many nodes available. To preserve this

situation, condition 3.25 must be fulfilled.

$$t_{SLC} > (p-1) \cdot t_{bt} \tag{3.25}$$

Figure 3.2 shows the work-flow of the computation for fulfilled relation 3.25. Figure 3.3 brings up an other situation, when relation 3.25 does not hold. Figure 3.2 shows that the slave processes (red colour) are fully utilized when the workload of the master process (blue colour) is low and it is free to assign moduli to them. In figure 3.3 we can observe gaps in slave processes utilization at the end of the computation, when, on the other hand, the master process has a igh workload due to MRC computation.

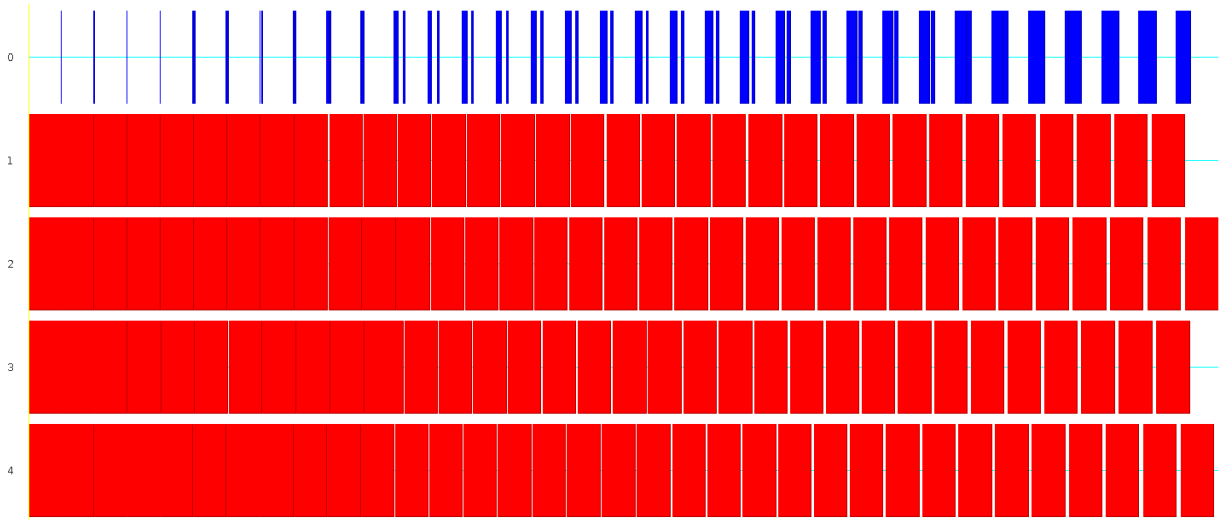


Figure 3.2: An example of the Node workload ($n = 500$, $p = 5$) - The graph of different processes utilization. The blue colour represents the master process and the red colour represents slave processes. The master process is idle almost all the time, while the slave processes are busy.

When adjusting relation 3.25 we get a relation for the number of processors, the dimension of the problem and the number of moduli used. This relation is expressed in equation 3.26.

$$p < \frac{t_{SLC}}{t_{bt}} + 1 = \frac{\mathcal{O}(n^3)}{\mathcal{O}(m \cdot n)} = \mathcal{O}\left(\frac{n^2}{m}\right) \tag{3.26}$$

Formula 3.26 says that the number of processors can grow as fast as dimension square but, on the other hand, with the growing number of used moduli the count of effectively spent

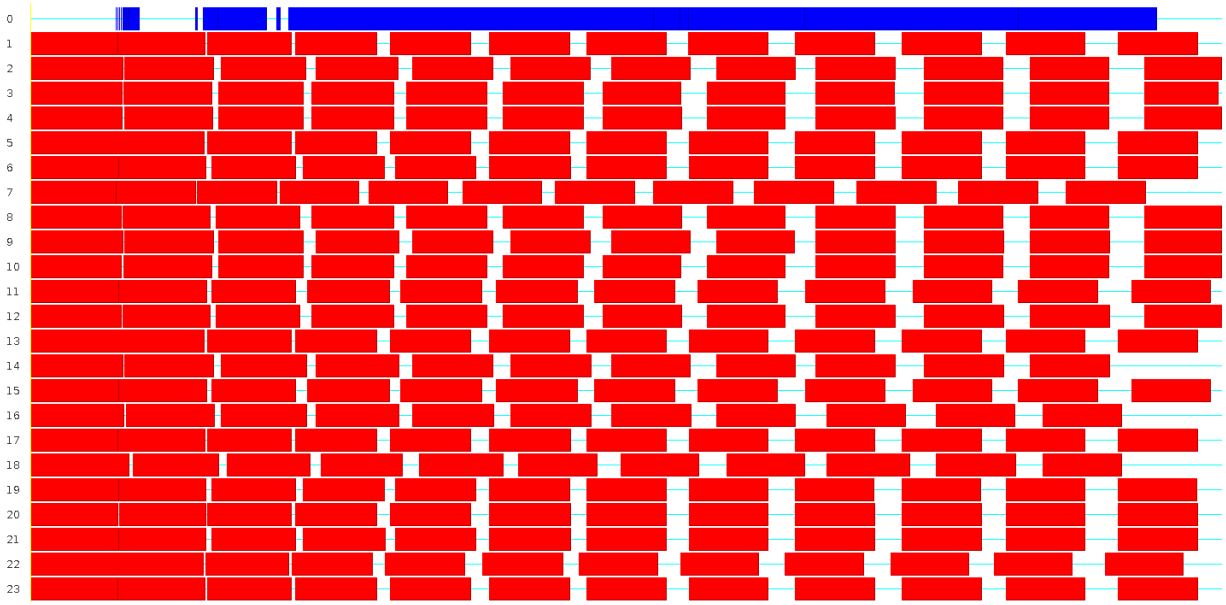


Figure 3.3: An example of the Node workload ($n = 500$, $p = 24$) where starvation is obvious - The meaning of colours remains the same as in 3.2. Master process is busy at later stage of computation and the slave processes starve (the gaps in red bars).

processors decreases.

### 3.3.2 Memory consumption

This section deals with memory consumption of the system. The relations do not cover the entire memory which the system uses. It is just the memory complexity of the algorithm. Furthermore, the system uses some temporary variables and buffers for different purposes. These memory areas are neglected due to their relatively small sizes.

A very simple description of the memory complexity is given in [10]. This section provides a more detailed analysis of memory consumption.

The section is divided in the same way as section 3.3.1. Sections 3.3.2.1 and 3.3.2.3 correspond to the computations performed by the master process and section 3.3.2.2 focuses on the memory consumption of the part processed by slave processes.

Although it is not highly important to state the overall memory consumption because we assume that every process is executed on a single node of the cluster, it is presented in section 3.3.2.4.

#### 3.3.2.1 Transformation to integer numbers

Memory consumption of the part of the algorithm performing the conversion of input data into integer numbers is determined by the size of the input data. So, the memory consumption should be expressed by relation $\mathcal{O}(n^2)$. The statements in section 3.1.2 imply usage of a 80-bit floating point data type because it is necessary to extend the range in comparison with the range of input data. We suppose input data in the range of double precision floating point values defined by literature [18], which means 64 bits in size. The use of the 80-bit floating point data type finally occupies 96 or 128 bits of memory space. It depends on the compiler and hardware architecture. We are working on x86 architecture and using mostly the GNU C compiler where the long double data type has the size of 128 bits.

The final size of memory space needed is expressed by formula 3.27 in bytes. The memory consumed by the input data has to be taken into account too.

$$m_{toint} = \frac{128 + 64}{8} \cdot n^2 = 24 \cdot n^2 \tag{3.27}$$

#### 3.3.2.2 SLC

Memory claims of the Gaussian elimination are also well known as $\mathcal{O}(n^2)$. Unfortunately, it is necessary to guarantee that the needed data will be placed in the physical memory. If this demand is not fulfilled, then the whole will slow down because some parts of the data have to be swapped out of the memory.

Thus, memory consumption has to be expressed more exactly. For the Gaussian elimination itself, 32-bits integer numbers are used. Nevertheless, we need to remember the input data in the form of integer numbers. Those are stored in the variables of the floating point type with 80 bits range. Then the size of memory the SLC solution needs is given by relation 3.28.

$$m_{slc} = \frac{128 + 32}{8} \cdot n \cdot (n + 1) = 20 \cdot n \cdot (n + 1) \tag{3.28}$$

When formula 3.28 is adjusted to the form of formula 3.29 it is possible to give an approximate limit to the dimension of the input problem, that can be processed without any complications. This limit is just a guess because of the system memory requirements,

temporary variables and other auxiliary demands are not included in this limit.

$$\begin{aligned} \frac{m_{slc}}{20} &> n^2 \\ \sqrt{\frac{m_{slc}}{20}} &> n \end{aligned} \tag{3.29}$$

For example, when having a physical memory of 512MB, then the limit for the dimension is approximately given by relation $\sqrt{\frac{512M}{20}} \doteq 5180$.

Experimental results show the fact that this limit is not a strict one. It is caused by the usage of integral scaled input data stored as 80-bit floating point type. These data are only used at the beginning of the SLC solution where the conversion from integer numbers to residual arithmetic is performed. Afterwards only the SLCs data of 32-bit integer type are utilized.

There is a possibility of saving some memory by changing the process of moduli assignment for separate slave processes. The SLC modulus should be assigned in the way, where the master process performs the transformation from the integer number range into residual arithmetic with a given modulus and then sends such a problem to the slave process. This is a way of saving the memory required for remembering the problem in integer numbers and this means saving 128 bits for each element.

But in effect this adaptation increases the communication complexity, because every SLC computation means that the whole task is sent from the master process to the slave process. Hypothetically, when using the 32-bit integer range, then $4B \cdot n^2$ will be sent. For $n = 5000$ it gives $4 \cdot 5000^2 = 100$MB and for $n = 10000$ it gives $4 \cdot 10000^2 = 400$MB.

#### 3.3.2.3 Transformation to real numbers

Expressing memory demands of the transformation from the RNS into real numbers is a little complicated. This is because the multiple precision library is used for the transformation.

Final values are stored as floating point numbers where the size of mantissa is determined by the number of used moduli and the number of their valid bits. The memory demands should be written in the form $\mathcal{O}(m \cdot n)$, too.

More precisely the size of the memory needed should be expressed by the number of bits

used for mantissa. Relation 3.30 then represents the total memory consumption for such elements. Identifier $k$ corresponds to the moduli valid bits count.

$$m_{bt} = k \cdot m \cdot n \tag{3.30}$$

It is straightforward that all the members of the mixed radix representation, called $d_i, i = 0, 1, \ldots, m$ in section 3.1.5, have to be remembered. This gives the $m \cdot 2(n+1)$ elements of 32-bit integer type due to the positive and negative representation computing.

#### 3.3.2.4 Total memory consumption

Memory consumption of the whole system can be expressed as the sum of the consumption of the master process and multiple of the memory consumption of the slave processes. The gained formula is useful when using a single computer where the processes have to share the memory, which obviously was the case of some ofthe tests as well. The total memory consumption is expressed by formula 3.31.

$$\begin{aligned} m_{all} &= k \cdot m \cdot n + m \cdot 2n \cdot 32 + 192n^2 + p(160 \cdot n \cdot (n+1)) \\ &= n^2(192 + 160p) + mn(k + 64) + 160pn \end{aligned} \tag{3.31}$$

### 3.3.3 Communication complexity

Communication complexity is not divided. It is given entirely for the whole system. There are only a few operations the system performs. The first one is the distribution of the problem transformed in the integer number range, then some assignments of the moduli are performed and, finally, the results of individual SLC eliminations are gathered into the master process.

The most complex operation is the distribution of the problem in integer numbers. This operation is performed at the beginning, just after the master process transforms the problem into integer numbers. Its complexity corresponds to the dimension of the problem and therefore it is $\mathcal{O}(n^2)$.

The operation of assigning a modulus is very simple. It only means sending the modulus

for the SLC. The count of moduli is $m$. Then the overall complexity of this operation is $\mathcal{O}(m)$.

The last communication operation is sending the result of the SLC back to the master process. This operation is performed $m$ times, as well. The only difference is that it sends the result vector with dimension $n$. The complexity of this operation is $\mathcal{O}(m \cdot n)$.

The communication complexity of the whole computation is expressed by relation 3.32, where identifier $l$ has the meaning of type size for sending the modulus and the result vector.

$$c = l \cdot n^2 + l \cdot m \cdot n = \mathcal{O}(n^2 + m \cdot n) \tag{3.32}$$

# 4 System Optimizations

## 4.1 Motivation

In spite of parallel implementation, the time needed for solving the SLE is much greater than for standard elimination methods (including some regularization). Thus, we have focused on optimization of time complexity.

The operation which is the most time consuming is the multiplication with reduction. It is due to the fact that modulo reduction after addition or subtraction can be realized by using one comparison and possible subtraction or addition, respectively. But after multiplication, the modulo reduction has to be done by means of division operation, which returns the residuum too. The division operation is very complex and takes a lot of processor cycles. A table of instructions and their latencies can be found in [11].

Nevertheless, the memory demands have been reviewed due to the problems of solving some real data sets and some bad decisions in the system design were discovered and eliminated later in the process.

Although the system has been developed as a linear equation solver for huge and dense matrices, the first real problem to solve was the sparse matrix from the Astronomical Institute of the Academy of Sciences of the Czech Republic. Thus, we have to focus on memory demands again, because the memory limit has been reached for this task. While the symmetric multiprocessing (SMP) is common due to multi-core processors, we decided to use shared memory (SHM) for the problem, where all the processes use this shared storage of the original data. Moreover, shared memory utilization also helps our primary design for dense problems.

### 4.1.1 Assumptions

As we focus on the SLC solution by the Gaussian elimination method, we can state that all the multiplications are performed in form 4.1 the row multiplication operation where each element of the row (vector) is multiplied by the same scalar value.

$$\mathbf{b}' \equiv c \cdot \mathbf{b} \bmod m \tag{4.1}$$

This fact makes some of the following optimizations possible.

## 4.2 Results

Sections 4.2.1.1 to 4.2.1.5 provide a detailed description of approaches that we have taken for multiplication with reduction. They are all used to find a remainder mod $m$ after finding a product of two integers $a$ and $b$ and they 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 an overflow in the computer register. The remainder mod $m$ is calculated during a modular version of Gaussian elimination 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 applied to obtain 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 the quotient and the remainder and we can just pick the remainder. An alternative strategy is to employ the floating point arithmetic. Although, at first sight, the floating point arithmetic does not look very attractive for reduction purposes, the following sections demonstrate that the possibility to use Single Instruction Multiple Data (SIMD) vector extensions to calculate *multiple* reductions with the same module simultaneously is very enticing and several times faster than the original approach using integer arithmetic.

Section 4.2.2.1 deals with the Gaussian elimination and the application of the presented approaches to carry it out. In addition, the section presents optimizations for other operations, not only multiplication with reduction.

The other interesting method for multiplication with reduction optimization is the use of Montgomery domain. This method and its usage for the whole Gaussian elimination is described in 4.2.2.2.

Memory demands of auxiliary data structures are analyzed in section 4.2.3.1. This section also presents a simple inoptimality disposal where memory demands for the specific task have been lowered to 50 percent of the original implementation. Memory demand then corresponds to the theoretical formulae expressed in complexity section 3.3.2.

### 4.2.1 Multiplication with reduction

#### 4.2.1.1 Integer Approach

This approach is 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 the 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
```

#### 4.2.1.2 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 multiplies with `fmulp` and then uses the `fprem` or `fprem1` instruction depending on whether we want to obtain the result in $\mathcal{Z}_n$ or in $\mathcal{S}_n$. 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
```

#### 4.2.1.3 Optimized Floating Point Approach

The problem of the approach described in section 4.2.1.2 is that `fprem` and `fprem1` instructions have a high latency[1]. These latencies are also caused by checks for validity of source operands, a need for division and also by the fact that the floating point divisions are *not* pipelined. The latency of `fprem` is commonly smaller than the latency of `div` but still large enough to do some more work. As we know that input operands are always valid and it is not likely that a floating point exception is to be ever thrown, we can bypass both `fprem/fprem1` and `div` instructions. We can use $m^{-1}$ value which we can precalculate before 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]
```

[1]The latencies of instructions for different processors can be found in [11]

```
;-------------------------------------------------------------------
;  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
```

This approach is senseless if we have a different $m$ for each operation. In our case of SLC solution using Gaussian elimination in residual arithmetic the $m$ is the same value for the whole row and thus for all the modulo reductions in this row multiplication. This is the reason why we can perform one division and then replace the remainder operation with multiplication.

#### 4.2.1.4 Floating Point Approach with MMX™ and SSE2

In fact, there is a necessity for multiple reductions modulo $m$ as we solve SLCs and there we typically multiply a vector (matrix row) with a scalar during Gaussian elimination. In the following $c$ is a scalar, while $\mathbf{b}$ is a vector and we rather calculate $\mathbf{b}' \equiv c\mathbf{b} \bmod m$. The size of operands in this approach is restricted and intermediate products shall not exceed 53-bit mantissa, therefore $c$, $m$, and elements of $\mathbf{b}$ must only be up to 26-bit wide. This limitation does not affect the floating point approach in 4.2.1.2 and 4.2.1.3 thanks to the usage of FPU and its full operational precision of 80 bits. Because of the length of an excessive source code, 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]
```

#### 4.2.1.5 Floating Point Approach with SSE2 by Intel Intrinsics

Since most of the operating systems currently run in the 64-bit mode, there is a need of porting the presented algorithms to the 64-bit architecture. These ports become 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 4.2.1.4 by means of the Intel intrinsic functions instead and let the compiler do its work. This implementation no longer requires any assembly code inlines nor the assembler compiler.

Due to the performed loop unrolling we were able to eliminate the usage of the multimedia extension (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 Streaming SIMD Extensions (SSE) equivalents.

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

### 4.2.2 Gaussian elimination

#### 4.2.2.1 Using SSE2 for Gaussian elimination

All of the optimization performed through sections 4.2.1.1 to 4.2.1.5 has been made to fully optimize the process of the Gaussian elimination in residual arithmetic. Note that 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 to add the row multiple to another row(s) in the matrix. This operation can be written as $\mathbf{v_j} \equiv \mathbf{v_j} - c\mathbf{v_i} \bmod m$, where $\mathbf{v_j}$ and $\mathbf{v_i}$ are rows 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 the reduction mod $m$ is performed after this operation.

We have also used vector processing during backward substitution in the Gaussian elimination process, where SSE extensions are used to calculate a dot product in residual arithmetic of $\mathbf{v_j} \cdot \mathbf{y} \bmod m$, where $\mathbf{v_j}$ 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 4.2.1.5 and Intel intrinsic functions for this implementation.

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

#### 4.2.2.2 Montgomery domain

Montgomery domain is another way of speeding up the modular arithmetic, especially the multiplication with reduction. The principles of the modular multiplication without division in Montgomery domain were first published in [26].

Montgomery found a representation of numbers which allows effective modular multiplication without division and does not affect other algorithms like addition or subtraction. The drawback of this approach is that numbers have to be transformed into the Montgomery domain at the beginning and back in the end. This means that this representation is not suitable for simple single modular multiplication but it fits well when the processing is more complex. The Gaussian elimination is such a process where the benefits of efficient modular multiplication should outweigh the drawback of the initial and final transformations.

The representation of number $a$ in the Montgomery domain with modulus $m$ is given by formula 4.2.

$$\overline{a} \equiv aR \bmod m \tag{4.2}$$

Then operations addition, subtraction and multiplication in the Montgomery domain are defined in formulae 4.3.

$$\begin{aligned} \overline{a} + \overline{b} &\equiv aR + bR \equiv cR \bmod m \\ \overline{a} - \overline{b} &\equiv aR - bR \equiv cR \bmod m \\ \overline{c} \equiv cR \equiv (a \cdot b)R \equiv (aR \cdot bR)R^{-1} &\equiv (\overline{a} \cdot \overline{b})R^{-1} \bmod m \end{aligned} \tag{4.3}$$

We may observe that addition and subtraction are common operations with final reduction. The product of $\overline{a}$ and $\overline{b}$ has to be multiplied by the inverse of $R$ to get the proper Montgomery representation of $\overline{c}$. This operation is called Montgomery reduction. The algorithm of multiplication including the Montgomery reduction is described by algorithm 1.

An essential feature of this algorithm is that it consists of general multiplication and addition, but operations of division and modulo reduction are performed only with $R$ as its operand. Thus we can chose such $R$ that would make these operations cheap. An obvious choice is some power of two for which the division is just a shift operation and the modulo reduction corresponds to the and operation. We always pick $R$ as the smallest

**Algorithm 1** Modular multiplication in Montgomery domain

**Require:** $\overline{a}, \overline{b}$ - The Montgomery representation of operands
**Require:** $R$ - The Montgomery representation multiplier
**Require:** $m$ - The modulus
**Require:** $k$ such that $k = (RR^{-1} - 1)/m \wedge RR^{-1} \equiv 1 \bmod m$
**Ensure:** $\overline{c} \equiv (\overline{a} \cdot \overline{b})R^{-1} \bmod m$ - The Montgomery representation of product of $a$ and $b$

```
1: T ← ā · b̄
2: p ← (T mod R)k mod R
3: t ← (T + pm)/R
4: if t ≥ m then
5:     c̄ ← t − m
6: else
7:     c̄ ← t
8: end if
```

power of two greater then modulus $m$.

$$\begin{gathered} R = 2^x \\ R > m \wedge 2^{x-1} \leq m \end{gathered} \tag{4.4}$$

see Appendix B for amore detailed description of the Montgomery domain and operations of multiplication, addition, subtraction etc. in this representation.

To gain best performance, we used all the knowledge gained from the previous sections mostly 4.2.2.1 and 4.2.1.5 and we have implemented the Gaussian elimination using SSE instructions. For a multiplication operation we can use the 128-bit packed quadword integers to ensure accuracy. For other operations, like addition or subtraction, we can use 128-bit packed doubleword integers, which gives us the opportunity to process even four elements simultaneously instead of two elements for multiplication.

#### 4.2.2.3 Optimizations for sparse matrices

We have also performed some optimizations for sparse matrices. A sparse matrix is a matrix populated primarily with zero elements. This kind of matrices often appears in science when solving partial differential equations. The great number of zero elements opens the possibility for processing only a specific subset of elements in the elimination process.

Truly the elimination algorithm is implemented by stepping one row after another and, from this point of view, in the elimination step $k$, there is no need to process the rows where the $k^{th}$ element of the row is already zero. And also if after the multiplication of the $k^{th}$ row by the multiplicative inverse of its $k^{th}$ element, the $l^{th}$ element is zero, the $l^{th}$ elements of other rows need no processing.

### 4.2.3 Memory demands

#### 4.2.3.1 Auxiliary data structures

It is necessary to focus on auxiliary data structures, too. We can express one example of highly impractical allocation of data structures for transformation to real numbers, especially the MRC part without Horner scheme. The theoretical complexity of this part of processing is described in 3.3.2.3. The structure has to be allocated dynamically on the heap, thus we allocate a three-dimensional array for storing the elements of table 3.1 for each element of the result vector and for both the positive and negative representation.

There are huge differences in the size of the additional allocated memory depending on the index order. We have expressed memory consumption for three different index orders in the next paragraphs.The symbols in the following formulae are:

- $p$ The size of pointer type
- $m$ The moduli count
- $n$ The dimension of the result vector
- $u$ The size of one element

**prime, vector, negative** : The worst ordering that we have accidentally chosen at the beginning of the implementation.

$$\begin{aligned} m_{mrc} &= m \cdot (p + (n+1) \cdot (p + 2u)) \\ &= 2m(n+1)u + m(n+2)p \end{aligned} \tag{4.5}$$

**prime, negative, vector** : This ordering is better than the first one.

$$\begin{aligned} m_{mrc} &= m \cdot (p + 2 \cdot (p + (n+1)u)) \\ &= 2m(n+1)u + 3mp \end{aligned} \tag{4.6}$$

**negative, prime, vector** : The best ordering of indices with minimal memory demand overhead.

$$\begin{aligned} m_{mrc} &= 2 \cdot (p + m \cdot (p + (n+1)u)) \\ &= 2m(n+1)u + 2(m+1)p \end{aligned} \tag{4.7}$$

When talking about x86-64 architecture, which is our case, the $p = 64b$ and $u = 32b$. The first factor of all the three formulae represents the theoretical memory demand stated in 3.3.2.3. From 4.5 we can observe that making a bad choice of index order leads to double memory consumption. On the other hand, in 4.7 the overhead of auxiliary data structures is negligible and the smallest of all the three cases.

#### 4.2.3.2 Shared memory

It is not necessary to store the original data of the solved problem separately for each process of computation. Especially, when we know that the SMP is engaged to a high degree. Thus, these data, common to all the processes, should be stored once for each SMP segment.

To engage the shared memory model we have to take several consecutive steps. There is a need to make suresafeguard that only one process on the SMP segment will be responsible for receiving the original problem. We use inter-process communication (IPC) lock to ensure this. Such a process will securely be responsible for the shared memory allocation and its deallocation. Other processes on the same SMP segment only have to prepare theirits own structures to use the shared memory segment. The safeguard and distribution are described by algorithms 2, 3 for the master process (loading and sending the problem) and slave processes, respectively.

From algorithms 2 we can see and 3 that the order of operations **lock** and **barrier** at the beginning is reversed. This ensures that the master process locks the IPC lock at its SMP segment and thus it is responsible for shared memory management. The second operation

**Algorithm 2** Problem distribution for shared memory use (master)

```
1: locked = lock(IPC) {Lock the interprocess lock.}
2: barrier {The global barrier.}
3: sle = load() {Load the input into the shared memory.}
4: lockedv = gather(locked) {Gather the lock flags to the master processes.}
5: for all i such that process i locked IPC and i ≠ 0 do
6:    send(sle) to process i {Send the sle data to all the locked processes.}
7: end for
8: barrier {The global barrier.}
```

**Algorithm 3** Problem distribution for shared memory use (slave)

```
1: barrier {The global barrier.}
2: locked = lock(IPC) {Lock the interprocess lock.}
3: gather(locked) {Gather the lock flags to the master processes.}
4: if locked then
5:    sle = receive() {Receive the sle into the shared memory.}
6:    barrier {The global barrier.}
7: else
8:    barrier {The global barrier.}
9:    sle = loadFromShm() {Load the auxiliary structures for shared memory use.}
10: end if
```

Table 4.1: Vector multiplication with reduction timings

| $n$ | "C" $[s]$ | s. 4.2.1.1 $[s]$ | s. 4.2.1.2 $[s]$ | s. 4.2.1.3 $[s]$ | s. 4.2.1.4 $[s]$ | s. 4.2.1.5 $[s]$ |
|---|---|---|---|---|---|---|
| $10^4$ | 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 |

**barrier** at the end of those algorithms is the safeguard for loading, sending, receiving and the entire necessary SHM management preceding the usage of SHM by non-responsible processes.

## 4.3 Experiments and Evaluation

### 4.3.1 Multiplication with reduction and Gaussian elimination

We have implemented the presented approaches from section 4.2.1.1 through 4.2.1.5 for multiplication with reduction in their vector form. The measurement has been done on a 1.7 GHz Intel machine running Linux. All timings were obtained with the `clock` API (Application Programming Interface). Table 4.1 shows the results obtained for multiplication with reduction $c\mathbf{b} \bmod m$ used during solving SLCs of various vector dimensions $n$ for C language implementation and then for the 5 approaches we have presented:

The columns in table 4.1 are captioned with the appropriate section number. For results from table 4.1 brought into graph 4.1, a significant speedup is visible:

Figure 4.1 plots all 6 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 4.2.1.1 to 4.2.1.5. 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 the instruction level in order to obtain a top performance.

There is an interesting observation that we can safely use the inverse module $1/m$ instead

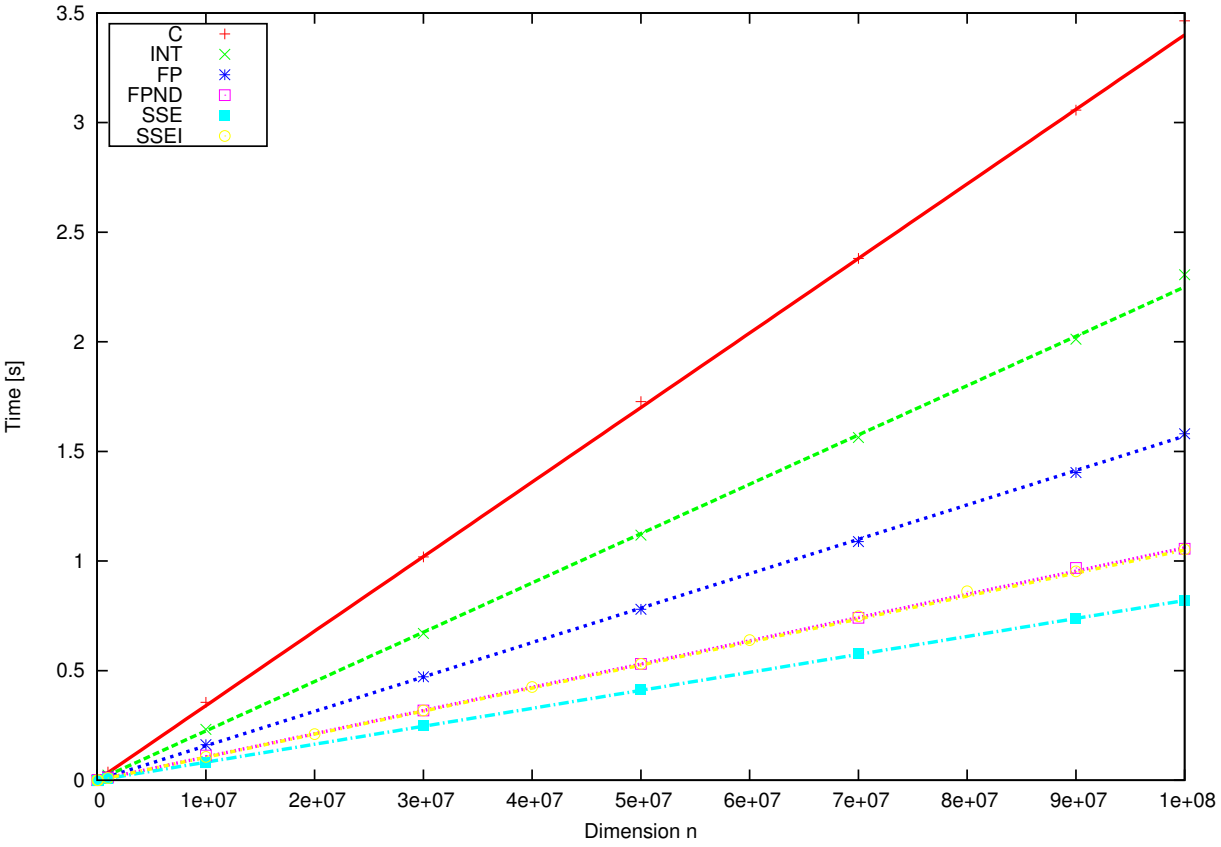


Figure 4.1: Vector multiplication with reduction timings - The meaning of lines is as follows. C - pure C implementation; INT - integer assembler implementation; FP - floating point implementation; FPND - floating point without division; SSE - floating point without division by means of SSE; SSEI - SSE implemented by means of Intel intrinsics

of just $m$ and completely avoid the division. Moreover, the FPU unit is not used in the final approach. The reduction is performed by 2 elements simultaneously with SSE2 instructions with the support of MMX™[3]. The speedup which we obtained in approach 4.2.1.4, when compared to approach 4.2.1.1, tops 4.2 times.

We have to note that the approach from section 4.2.1.5 gives worse results than the approach from section 4.2.1.4, but this slowdown is the cost of higher simplicity and portability.

To present a comparison of the result chosen in the previous paragraph with the new algorithm described in 4.2.2.2, we performed new measurements on the system with Intel® Core™ i5 CPU, type 3210M running at 2.50GHz [4] frequency. The computer ran OS GNU/Linux (Arch distribution) with **gcc** version 4.7.1 compiler and **glibc** version 2.15. All timings were obtained with the `clock` API (Application Programming Interface). Now we only performed measurements for the original C language implementation, the previously chosen approach from section 4.2.1.5 and finally the timings for Montgomery domain algorithm described in 4.2.2.2. The results of these measurements are expressed in table 4.2.

Table 4.2: Vector multiplication timings

| dimension | $T_C$ [s] | $T_{SSE}$ [s] | $T_{MD}$ [s] |
|---|---|---|---|
| $10 \cdot 10^6$ | 0.096 | 0.020 | 0.011 |
| $30 \cdot 10^6$ | 0.292 | 0.059 | 0.031 |
| $50 \cdot 10^6$ | 0.486 | 0.101 | 0.053 |
| $70 \cdot 10^6$ | 0.682 | 0.142 | 0.073 |
| $100 \cdot 10^6$ | 0.974 | 0.202 | 0.105 |

Table 4.3: Gaussian elimination timings

| dimension | $T_C$ [s] | $T_{SSE}$ [s] | $T_{MD}$ [s] |
|---|---|---|---|
| 1000 | 3.453 | 0.667 | 0.466 |
| 1500 | 11.701 | 2.238 | 1.559 |
| 2000 | 27.679 | 5.255 | 3.640 |
| 2500 | 54.054 | 10.213 | 6.994 |
| 3000 | 92.897 | 17.509 | 11.998 |

We can see that the speedup for the approach using floating point arithmetic and the SSE is approximately the same in the new measurement. Also, we have observed that the Montgomery domain utilization made the vector multiplication even twice as fast.

Finally, we have measured Gaussian elimination timings for approach 4.2.2.1 and Montgomery domain implementation from section 4.2.2.2 and compared it to the Gaussian elimination implemented by using pure C. All timings are presented in table 4.3 for various SLCs dimensions $n$. This measurement has been performed on the same Core™ i5 machine.

[3]The MMX™ instructions have been replaced with their 128-bit SSE2 equivalents in approach 4.2.1.5.
[4]Intel® Core™ i5-3210M CPU @ 2.50GHz specification

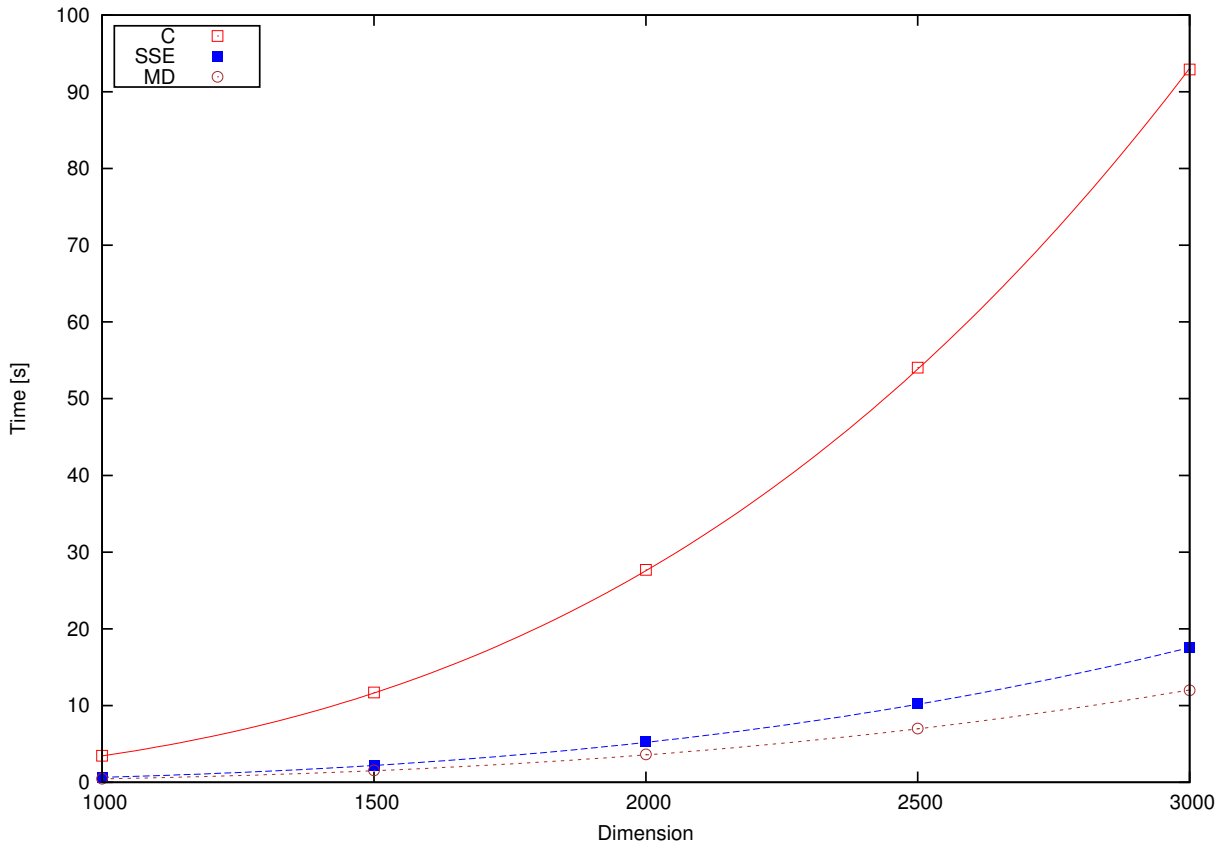


Figure 4.2: Gaussian elimination timings

Data from table 4.3 are expressed in figure 4.2 where we can observe the speedup for the whole single modular elimination.

Though the performance of Gaussian elimination using floating point arithmetic is very good, the Montgomery domain implementation behaves much better.

The optimizations for sparse matrices presented in 4.2.2.3 have a significant impact on the performance. Table 4.4 presents the Gaussian elimination timings for sparse matrix containing non-zero elements just around the main diagonal. More specifically, up to five elements in each row are not equal to zero.

Table 4.4: Gaussian elimination timings for sparse matrices

| dimension | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
|---|---|---|---|---|---|---|---|---|---|
| $T_S$ [s] | 0.005 | 0.012 | 0.024 | 0.040 | 0.063 | 0.091 | 0.129 | 0.171 | 0.206 |

Table 4.5: Latency FP versus Montgomery domain

| FP | | | Montgomery domain | |
|---|---|---|---|---|
| Loading | CVTPI2PD | 4 | MOVD | 1 |
| | | | PUNPCKLQDQ | 1 |
| Processing | MULPD | 5 | PMULUDQ | 5 |
| | MULPD | 5 | PAND | 1 |
| | ADDPD | 3 | PMULUDQ | 5 |
| | SUBPD | 3 | PAND | 1 |
| | MULPD | 5 | PMULUDQ | 5 |
| | SUBPD | 3 | PADDQ | 1 |
| | | | PSRLQ | 1 |
| Storing | CVTPD2PI | 4 | PSHUFD | 1 |
| | | | POR | $0.5 \cdot 1$ |
| | | | PSUBD | $0.5 \cdot 1$ |
| $\sum$ | | 32 | | 23 |

#### 4.3.1.1 Rationale

In this section we try to analyze the reasons for the data presented above. The simplified sequence of instructions for the SSE variants of floating point and Montgomery domain approaches with their latencies are shown in table 4.5

We can see that the summation of the latencies of the Montgomery domain implementation is significantly smaller than the summation for the floating point approach. This fact corresponds to the results presented. Instructions POR and PSUBD have latency 1 but we are counting only a half of this because we need just two repetitions of those two instructions for four repetitions of 4.5.

If we use the SSE4.1, we can replace the set of ADD and SUB in the floating point approach by one ROUNDPD of latency 3. But when using the Advanced Vector eXtension (AVX) we can embrace the 256-bit alternatives for the processing part of the floating point approach, which should take the overall latency under the Montgomery domain implementation. On the other hand, when we have the AVX2 at disposal, we can even use the 256-bit alternatives for the whole Montgomery domain processing, which would probably give the best running time.

### 4.3.2 Memory demands

#### 4.3.2.1 Auxiliary data structures

Here we also present the difference in the index ordering from 4.2.3.1 in a real example. We have made a mistake in choosing bad index ordering at the beginning and thus we had to figure out what caused the memory demand. Figures[5] 4.3 and 4.4 show the memory allocation analysis performed by massif[6] tool of valgrind[7] for the worst 4.5 and the best 4.7 index ordering.

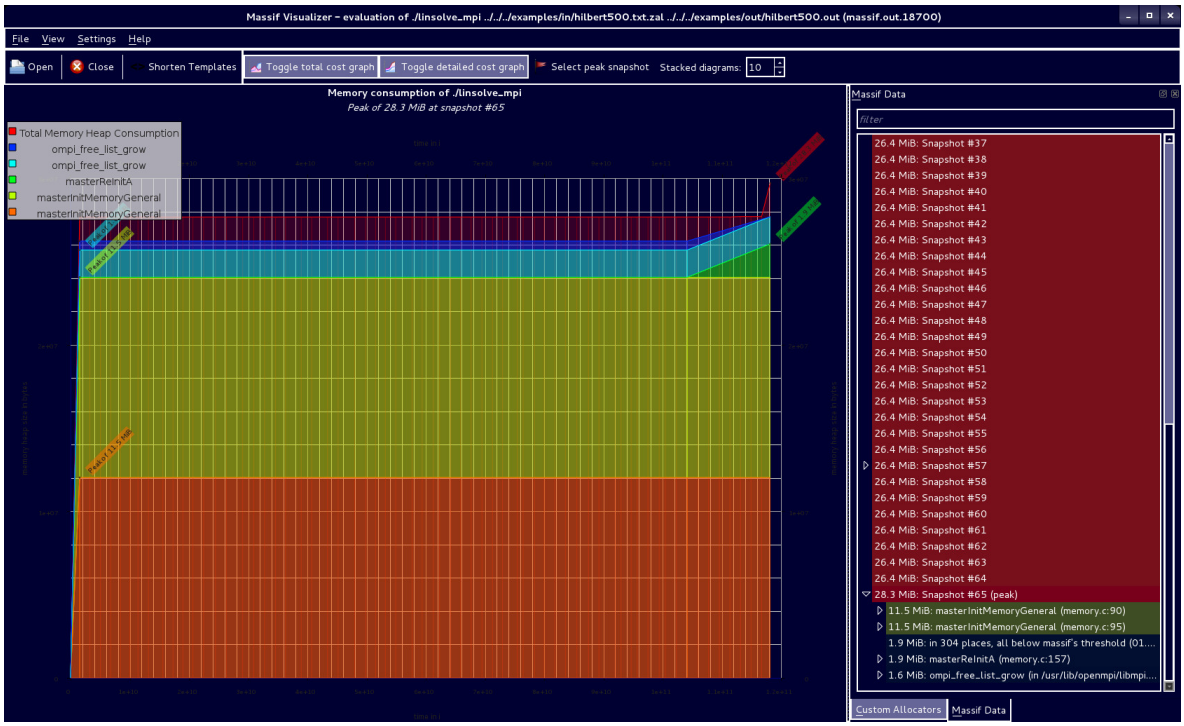


Figure 4.3: Memory demands with a bad index order($n = 500$, $m = 3000$)

We can compare the unrolled snapshots of memory allocated from figures 4.3 and 4.4. It is clear that one of the blocks of size 11.5MB allocated in *masterInitMemoryGeneral* function at 4.3 has disappeared in 4.4. It perfectly fits in the memory demand expressed in 4.5 and 4.7. The first 11.5MB corresponds to the theoretical memory consumption and the second

[5]The figures come from the massif visualizer tool [33]

[6]heap profiler taking the snapshots of allocated memory including the information the parts of which are responsible for most allocations

[7]The framework for dynamic analysis tools [2]

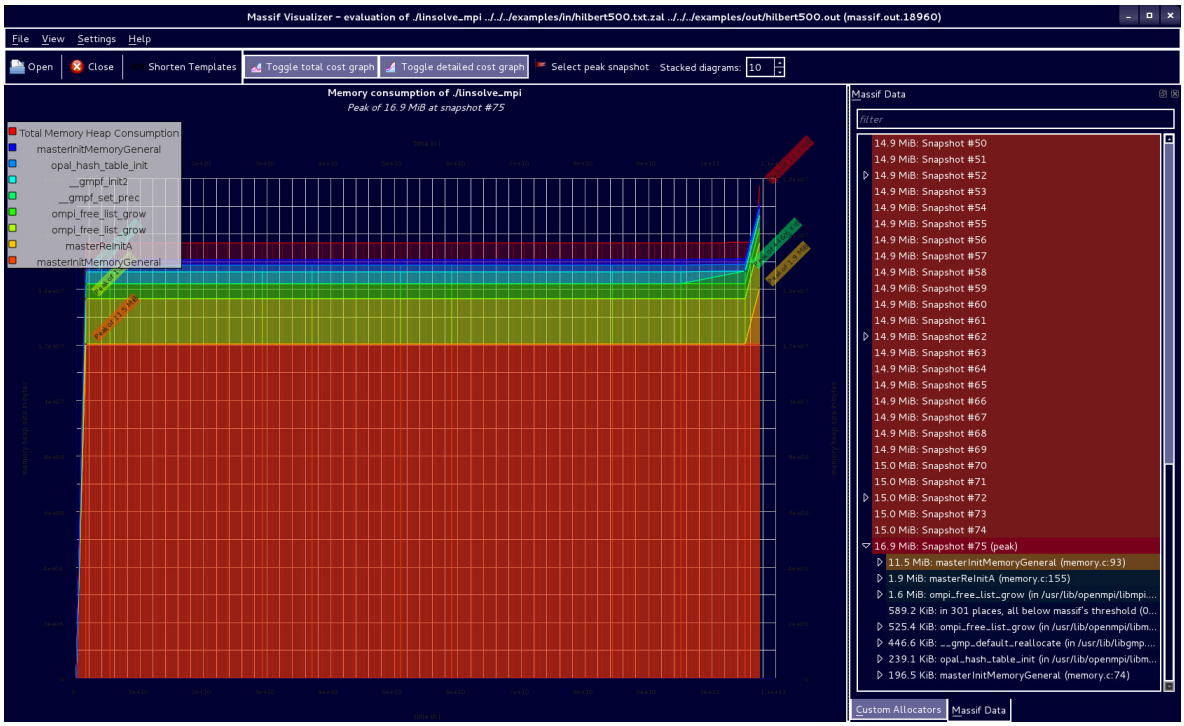


Figure 4.4: Memory demands with a good index order($n = 500$, $m = 3000$)

block of the same size corresponds to the auxiliary structures in 4.5. But for the second case in figure 4.4, the auxiliary structures memory demands are just about 50kB of memory ($(2 \cdot 3001 \cdot 64)/(8 \cdot 1024) \doteq 47kB$). This size is under the chosen threshold and thus is not at all visible.

#### 4.3.2.2 Shared memory

Results of the SHM usage are show in figures 4.5 and 4.6 where the top command output is captured for variants without shared memory usage and with shared memory, respectively.

The significant columns in figures 4.5 and 4.6 are columns marked as **RES** and **SHR**. Where column **RES** gives the amount of non-swapped physical memory a task uses. On the other hand, column **SHR** gives the amount of shared memory available to a task. It is obvious that the non-shared variant uses approximately the same amount of resident memory but as its shared counterpart. The huge difference is in the amount of shared memory where the first variant uses 13MB of memory of which about 5MB are shared between processes, on the other hand, the second variant uses 13MB, too, but all of the

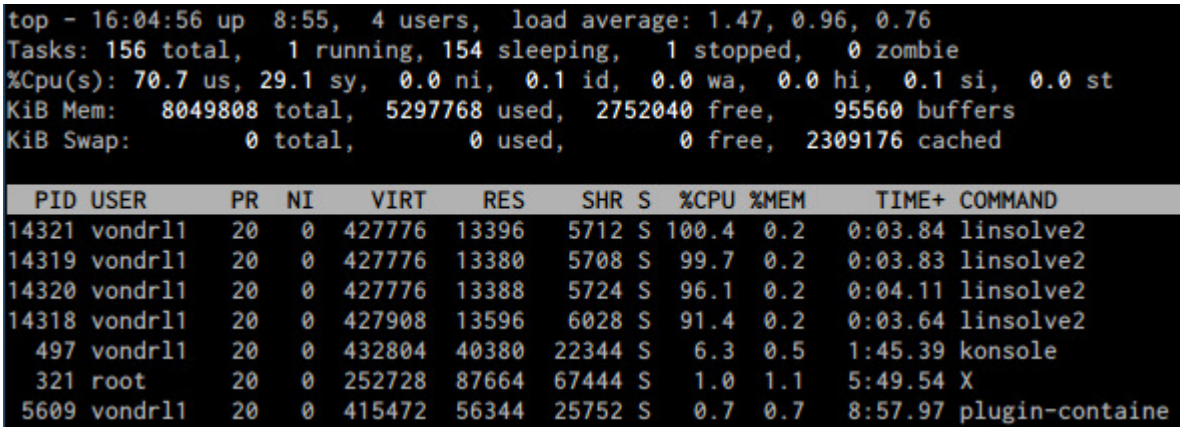


Figure 4.5: Memory demands without shared memory($n = 500$)

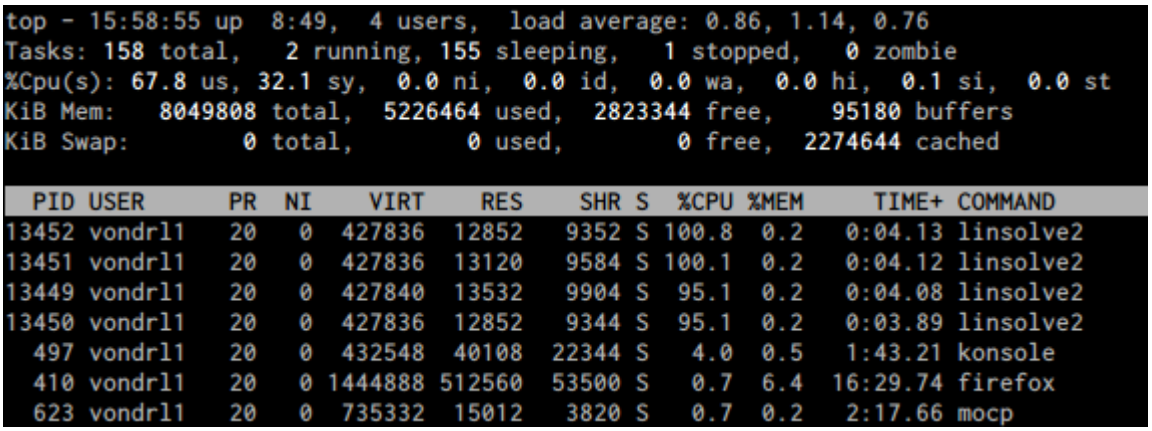


Figure 4.6: Memory demands with shared memory used($n = 500$)

9MB are shared between processes.

More precisely, we can write the following expression $13 \cdot 4 - 9 \cdot 3 < 13 \cdot 4 - 5 \cdot 3$ thus $25 < 37$ and we have observed that by using shared memory we saved approximately 12MB of memory space.

The more profitable the shared memory use is the more cores in one SMP segment we have. To be specific for STAR cluster and, especially for nodes, where there is a processor with twelve cores and hyper-threading, it means that we are running 24 processes on one physical node (SMP segment). Thus we can save 23 instances of problem data.

## 4.4 Summary

This chapter deals with system optimization of the core problem in solving 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 e.g. in Gaussian elimination which is commonly used during solving process. Normally, reduction is performed in the integer unit with the `div` instruction, but because this instruction has a high latency and is not pipelined, we would prefer a way without division.

Another approach would be to perform 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 $m$ we need. These two instructions have also high latencies, and, because we know module $m$ in advance, we can go around using `fprem` and `fprem1` instructions by turning the division by a module into multiplication by its inverse $m^{-1}$. Inverse modules are typically precalculated once just before running the multiplication of the vector for the specific module.

Nevertheless, the process of Gaussian elimination features a multiplication of the entire matrix row $\mathbf{b}$ with the same constant $c$ and therefore a need for modular reduction after multiplication with the same module $m$ arises ($\mathbf{b}' \equiv c\mathbf{b} \bmod m$). This need can be beneficial as we can perform multiple reductions simultaneously with the help of processor features — namely SIMD instructions from the SSE2 processor extension. Taking this approach allows computing two reductions simultaneously and when several reductions are combined together and the code loops are unrolled, a significant speedup is achieved. This

speedup is shown in figure 4.1 and presents more than a four fold enhancement[8].

A completely different way of avoiding division during the multiplication with reduction is the usage of the Montgomery domain, where the operations of addition and subtraction are defined as common but the multiplication with reduction is performed by a set of multiplications, additions and bit operations shift and mask. Table 4.2 shows that the Montgomery domain presents nine-fold enhancement against the original C implementation of vector multiplication.

From section 4.3.1.1 we can observe that those implementations are highly dependent on the architecture used. We have theoretically shown that for some architectures, namely the processor which includes AVX and not AVX2, would run the elimination faster using floating point representation and 256-bit AVX extensions.

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

The special case of sparse matrices is handled and measured too. We observe that the sparse matrices processing timings are disproportional to their dense equivalents.

Also, we have expressed the need to take care of every aspect of algorithm design on the right choice of the indexing order for the dynamically allocated three dimensional array for the transformation back into real numbers (MRC). We have shown that with bad choice of ordering, memory demands could double.

At a later stage we engaged the shared memory concept for storing the original task data. We have shown that it has a significant influence on memory demands. Nowadays, when Intel® produces processors with twelve physical cores and hyper-threading and is about to come with fifteen cores on one chip, which gives thirty threads of execution, it is the right way of reducing memory demands. Advanced Micro Devices produces multi core systems up to sixteen cores, too. Moreover a lot of CPUs can share one memory. We have shown algorithms for safe data distribution and that this concept is valuable for memory saving.

The results presented in this chapter were published in author's papers [A.1, A.2, A.3, A.6].

[8] No influence of communication complexity was accounted.

# 5 Decentralization

## 5.1 Motivation

Architecture of the system has some bad consequences. The processing is split into two strictly separated parts, master and slave. This means that it is strongly unbalanced and it cannot be rebalanced in any way.

Specifically, we present two direct results of the system design.

- The first is expressed in section 3.3.1.4, more specifically, in formula 3.25. The meaning of this formula is that when we have a lot of processes or a lot of moduli used then the master process would become the limiting part for the SLCs processing due to the design of application where the master process, which performs the backward MRC transformation, also handles the assignment of the modulus for the next SLC processing at the slave nodes.
- The next problem of such a tightly bounded architecture is the fact that the memory demands of the master and slave processes can be very different. Here we can get into trouble when we have a homogeneous cluster and the memory demands of master process are much higher than the memory demands of the slave processes. In such a case, the master process memory demands will be limiting the size of the problem in the way of dimension of SLE and the number of moduli used.

From the previous paragraphs we can say that the decentralization of the whole computation could help with balancing the whole problem computation. To decentralize the master process we need to implement its parts in the distributed environment. Thus we need to formulate the distributed version of the prime moduli assignment and the distributed version of MRC algorithm.

## 5.2 Results

### 5.2.1 Prime number generation

The first thing we want to solve is the problem of assigning moduli to the slave processes. It is achieved in a straightforward manner by enabling each process to generate the moduli

for itself alone.

A special requirement for this self generation is to generate a mutually disjoint set of prime numbers, because we do not want to solve the SLC of one moduli more than once.

This means that we are looking for sets $P_i$ for $i = 1, 2, \ldots, p$ such that the following holds.

$$\forall p_j \in P_i : p_j \text{ is prime } \wedge p_j \notin P_k : k \neq i \tag{5.1}$$

We implement the prime number generator simply by scanning integer numbers. Condition 5.1 can be fulfilled by defining *step s* as a number of prime numbers to be skipped. At the beginning, each process makes the *step* equal to the processor's number (index), thus the starting prime number will be different for each of all the processes. Subsequently, the *step* is set equal to the number of processes which ensures $P_i$ sets to be disjoint.

### 5.2.2 Mixed Radix Conversion

Moving Mixed Radix Conversion into distributed environment is not a very complicated task. We can find some parallel implementations in [7, 34]. The problem of those implementations is that they do not fulfill our requirements. First we have to define the desired properties of the algorithm we are looking for.

The point is that we need to process the MRC transformation during the solution process, i. e. concurrently with SLC solving. There are two reasons for such a demand:

- Resolving sufficient precision – at a certain stage of the processing we can say that the precision of the result is already sufficient and stop subsequent processing (another SLCs solving). This process would be meaningless if the MRC transformation was performed at the end of the computation.
- The ability of rebalancing - it would be nice to have the opportunity to rebalance the MRC transformation. This is very important especially for a jheterogeneous parallel environment. As already expressed in 3.1.5, the MRC is responsible for finishing the processing. If there is a huge difference between the nodes performance, the faster nodes in the cluster will process a greater portion of the MRC. If the MRC is still divided into parts of the same size, the decision whether to stop the processing will be suspended until the slow nodes finish the needed MRC parts. As a result, there

could be many SLCs solved by the faster nodes and their solutions will be thrown away without any profit.[1]

When we retain both properties, the algorithms designed in [5, 7, 34] do not perfectly fit. We can certainly use [7, 34] but it can lead to different moduli vector on each node. If we still want to ensure rebalancing, then every member of the transformed result vector has to have its own moduli vector and has to be transformed according to that vector because of result vector elements migration. This is the reason why we have decided to design our own parallel implementation of MRC where we can preserve the moduli vector and ensure the sequence for transformation to be the same for all the elements of the vector.

Our algorithm is based on the same base as the algorithm published in [34]. The basic idea is to divide the problem of finding the representation in $\mathcal{R}$ space when having a RNS representation of $n$ dimensional vector $\mathbf{y}$. We can then perform the MRC separately for each element of $\mathbf{y}$. Thus when we have $p$ processes and a vector of dimension $n$, every process would evaluate MRC transformation for $n/p$ elements of $\mathbf{y}$.

#### 5.2.2.1 Total distributed ordering

The first of our requirements we try to satisfy is the total distributed ordering of the moduli set. To be able to set up the ordering, we have used the principles and mechanisms of ordering the events in distributed systems published in [20]. The events we need to order are the SLC results generation. To get the total ordering we need to order a concurrent result of the same logical time gained on different nodes. This is done by using the ordering according to the moduli of the result. The total ordering $\rightarrow$ is then defined as:

$$\begin{aligned} \mathbf{y_1} \rightarrow \mathbf{y_2} \Leftrightarrow & (C(\mathbf{y_1}) < C(\mathbf{y_2})) \vee \\ & (C(\mathbf{y_1}) = C(\mathbf{y_2}) \wedge M(\mathbf{y_1}) < M(\mathbf{y_2})) \end{aligned} \tag{5.2}$$

where $C(\mathbf{y})$ is a logical clock function meaning the time in which result $\mathbf{y}$ was obtained and $M(\mathbf{y})$ returns the modulus of the $\mathbf{y}$ result. Generally $C(a)$ is the logical time of any event $a$.

**Example 5.2.1.** We will show some example of the ordering we want to get. Figure 5.1 represents the timelines of five processes in the computation. The originating times

[1] This problem does not burden the SLC solution itself. In this part it does not matter whether the specific SLC of specific module is solved.

of the results in the processes are marked with **r** labels. These are indexed with the process number and the sequence number in the process. The messages passed between the processes are indicated by dashed lines.

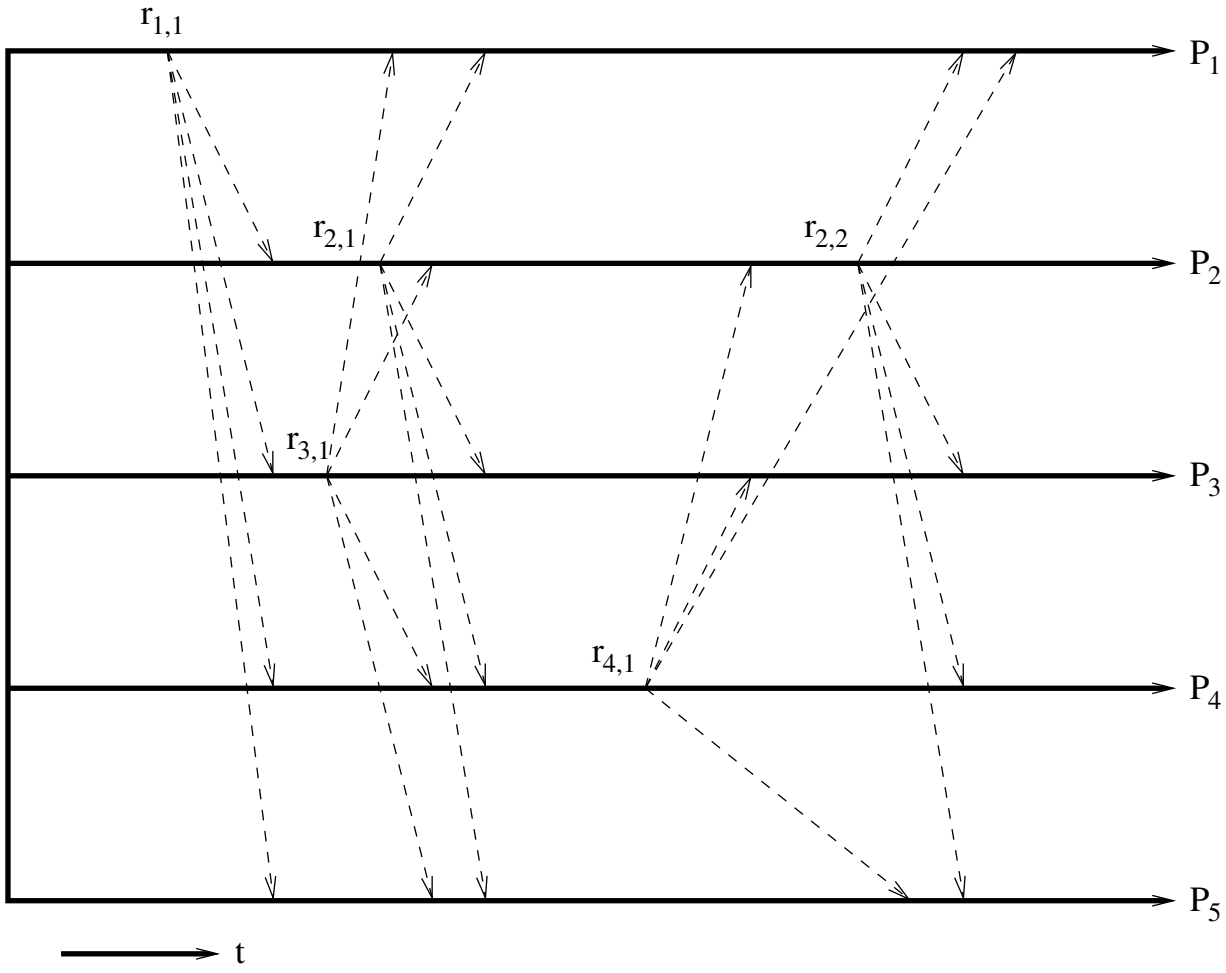


Figure 5.1: Example of results ordering

Table 5.1 expresses the moduli of different results. They are also needed to state the final results ordering.

Table 5.2 shows the events and the times that events occur in a specific processes. The result generation is denoted by $r_{i,j}$ and receiving of the result message is denoted by $Mr_{i,j}$. As we can see, the only conflicting results, meaning results becoming from the same logical time, are $r_{2,1}$ and $r_{3,1}$. If we take into account the moduli of the results with respect to 5.2 the final ordering of the results will be $[r_{1,1}, r_{2,1}, r_{3,1}, r_{4,1}, r_{2,2}]$ because $m_{2,1} < m_{3,1}$.

Table 5.1: Moduli of results of example 5.2.1

| result | modulus |
|---|---|
| $r_{1,1}$ | $m_{1,1} = 11$ |
| $r_{2,1}$ | $m_{2,1} = 13$ |
| $r_{2,2}$ | $m_{2,2} = 31$ |
| $r_{3,1}$ | $m_{3,1} = 17$ |
| $r_{4,1}$ | $m_{4,1} = 19$ |

Table 5.2: Results ordering of example 5.2.1

| $P_1$ | | $P_2$ | | $P_3$ | | $P_4$ | | $P_5$ | |
|---|---|---|---|---|---|---|---|---|---|
| $C_1$ | $e_1$ | $C_2$ | $e_2$ | $C_3$ | $e_3$ | $C_4$ | $e_4$ | $C_5$ | $e_5$ |
| 0 | ⋮ | 0 | ⋮ | 0 | ⋮ | 0 | ⋮ | 0 | ⋮ |
| 1 | $r_{1,1}$ | 0 | ⋮ | 0 | ⋮ | 0 | ⋮ | 0 | ⋮ |
| 1 | ⋮ | 3 | $Mr_{1,1}$ | 3 | $Mr_{1,1}$ | 3 | $Mr_{1,1}$ | 3 | $Mr_{1,1}$ |
| 1 | ⋮ | 4 | $r_{2,1}$ | 4 | $r_{3,1}$ | 3 | ⋮ | 3 | ⋮ |
| 5 | $Mr_{3,1}$ | 5 | $Mr_{3,1}$ | 4 | ⋮ | 5 | $Mr_{3,1}$ | 5 | $Mr_{3,1}$ |
| 6 | $Mr_{2,1}$ | 5 | ⋮ | 5 | $Mr_{2,1}$ | 6 | $Mr_{2,1}$ | 6 | $Mr_{2,1}$ |
| 6 | ⋮ | 5 | ⋮ | 5 | ⋮ | 7 | $r_{4,1}$ | 6 | ⋮ |
| 6 | ⋮ | 8 | $Mr_{4,1}$ | 8 | $Mr_{4,1}$ | 7 | ⋮ | 6 | ⋮ |
| 6 | ⋮ | 9 | $r_{2,2}$ | 8 | ⋮ | 7 | ⋮ | 6 | ⋮ |
| 6 | ⋮ | 9 | ⋮ | 8 | ⋮ | 7 | ⋮ | 8 | $Mr_{4,1}$ |
| 10 | $Mr_{2,2}$ | 9 | ⋮ | 10 | $Mr_{2,2}$ | 10 | $Mr_{2,2}$ | 10 | $Mr_{2,2}$ |
| 11 | $Mr_{4,1}$ | 9 | ⋮ | 10 | ⋮ | 10 | ⋮ | 10 | ⋮ |

Paper [20] describes the "happened before" relation followed by the clock condition. This is out of scope of this thesis. We just denote the implementation rules resulting from these definitions:

1. Each process $P_i$ increments $C_i$ between any two successive events.
2. (a) If $a$ is the sending of message $m$ by process $P_i$, then the message $m$ contains a timestamp $T_m = C_i(a)$.
   (b) Upon receiving the message $m$, process $P_j$ sets $C_j$ greater than or equal to its present value and greater than $T_m$.

Now we can describe the algorithm itself. It consists of a few rules that each process has to follow.

1. When the result is obtained it is broadcast to all the other nodes with actual timestamp $T_i$.
2. When the process receives the result message with timestamp $T_i$ it sends the confirmation message with timestamp $T_i + 1$.
3. All the results with time $T_j$ where $T_j \leq T_i$ can be ordered and processed when the process has received the message with timestamp $T_k$ where $T_k \geq T_i$ from all the other processes.

It is straightforward that when each process follows the presented rules, it leads to the desired total ordering of the SLC results. It is necessary to have logical clocks in each process where the events to be noticed by the clock mechanism are obtaining of the SLC result and the confirmation message, no matter whether the event comes from the process itself or it comes from another process by means of receiving of the appropriate message. Also a mechanism to hold the times of the latest messages received from specific processes is required for each process.

#### 5.2.2.2 Precision Resolving – Round Robin

The basis of precision resolving and processing finalization is stated in 3.1.5. For the early implementation it was the responsibility of the master process to perform the MRC

conversion and also the precision resolving as stated in 3.2.6. It is straightforward that when we do not have the master process and all the processes are "equal", the precision resolving has to be done in a distributed environment like all the other parts of the processing.

As every process is responsible for the MRC conversion of the part of the result vector, all the processes have to make an agreement that all the elements of the mixed radix representation are equal to zero from a certain step. This is achieved by a virtual ring implementation. There are two types of messages described in the following paragraphs.

- *want-finish* message: The *want-finish* message represents the fact that the sending process has achieved a sufficient precision of its part of the result vector. This message carries the information of the step of the whole *MRC* conversion when it was generated. When the message of this type runs through its virtual circle, it means that all the processes have agreed that the *MRC* step carried by this message represents the sufficient precision of the *MRC* conversion.
- *finish* message: This means that all the processes agreed on the sufficient precision and it is possible to finish the processing. It carries the information of which step of the *MRC* conversion should be the final one.

A detailed processing of these messages is described in 5.3.4.3.

## 5.3 Experiments and Evaluation

It is straightforward that the implementation of the new MRC along with decentralization of the whole processing will not be a minor change. So, we have decided to redesign the whole project and subsequently to rewrite it from scratch. The language selected for the implementation of the second version is **C++**. It has been chosen for the possibility of Object oriented design and it is still a compiled language and offers sufficient performance.

In the following two sections we present the results for the problems drawn in 5.2. The next sections describe the overall design of the new implementation of the system and its complexity analysis.

### 5.3.1 Prime number generation

The new implementation for the prime number (moduli) generation has a disadvantage. Specifically, it has to find all the primes and skip some of them. Thus the process of finding the next prime is done more than once. Instead of one pass through the prime generation there will be $p$ steps performed (one for each of the processes). We decided to use this implementation because in comparison with the rest of the computation the prime generation is negligible in time.

There is another aspect of the prime number generation. A fear of a sufficient number of primes can arise. For a homogeneous environment the problem does not occur because all the nodes acquire approximately the same count of primes. This is analogous to the original implementation. The situation is different in a heterogeneous environment where the processes could run with a very different speed on each node. It can lead to unbalanced acquiring of prime numbers and the concerns are eligible.

To defend the procedure we will express the number of primes available and state that the count of primes is sufficient.

According to the Prime Number Theorem (PNT) (can be found in [13]) the number of prime numbers less than $x$ is

$$\pi(x) \sim \frac{x}{\ln x} \tag{5.3}$$

Because we use the 26 bit moduli (meaning the numbers with 26 valid bits) we can write

$$p_{cnt} = \pi(2^{26}) - \pi(2^{25}) = \frac{2^{26}}{\ln 2^{26}} - \frac{2^{25}}{\ln 2^{25}} = 1787402 \tag{5.4}$$

This means that for one hundred processes we still have almost 18 thousand prime numbers for each. This is the foundation of our claim that even after the prime numbers space split, the computation will not run out of primes.

Moreover, when running on a highly heterogeneous cluster there is still the possibility of employing some negotiation for passing a part of the prime number set of slower nodes to the faster ones.

Table 5.3: MRC conversion timings

| dim | linsolve | | | new | | | SNB | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 2 | 3 | 4 | 2 | 3 | 4 |
| 10 | 0.0098 | 0.0130 | 0.0166 | 0.0220 | 0.0202 | 0.0228 | 0.0333 | 0.0243 | 0.0187 |
| 20 | 0.0177 | 0.0180 | 0.0270 | 0.0187 | 0.0165 | 0.0222 | 0.0590 | 0.0090 | 0.0176 |
| 50 | 0.0300 | 0.0517 | 0.0680 | 0.0178 | 0.0206 | 0.0240 | 0.0155 | 0.0094 | 0.0148 |
| 100 | 0.0758 | 0.1433 | 0.1789 | 0.0326 | 0.0292 | 0.0373 | 0.0278 | 0.0284 | 0.0338 |
| 200 | 0.2544 | 0.4591 | 0.5351 | 0.0904 | 0.0949 | 0.0966 | 0.0780 | 0.0879 | 0.1010 |
| 500 | 2.5301 | 4.1442 | 6.0595 | 0.8321 | 0.8546 | 0.8744 | 0.7551 | 0.9488 | 0.9903 |

### 5.3.2 Mixed Radix Conversion

The mixed radix conversion algorithm presented in 5.2.2 has been implemented as an inseparable part of the whole system. Nevertheless, to test the performance of this algorithm some changes took place in a way how the SLC solutions are obtained. They are not computed for the performance test but just loaded from the previous run of the solver. This adjustment has been done in the original implementation, too. As a reference algorithm we have taken the algorithm called Single Node Broadcast (SNB) from [7] which we have implemented.

Then a set of tests has been passed for different dimensions of vector (SLE). All the measurements have been repeated 10 times to exclude the external influences of the other system components. The results of the measurements are summarized in table 5.3 and presented in figure 5.2 as well. The measurements have been performed on the Intel® Core™2 Duo CPU T9400 running at 2.53GHz frequency which has two physical cores.

Table 5.3 shows the average timings for the test runs. There have been measurements for two, three and four processes and different dimensions. We have to bear in mind that there were just two physical cores which led to the results presented. For the original implementation it means that the master process has to share the processor time with other processes, which means a significant slowdown of the whole processing. For the other implementations the slowdown is not so considerable due to the fact that every process participates on the MRC conversion, thus sharing the processor time is not a significant drawback. Nevertheless, figure 5.2 represents just two process variants of the tests.

From figure 5.2 and table 5.3 we can see that the implementation of the algorithm presented

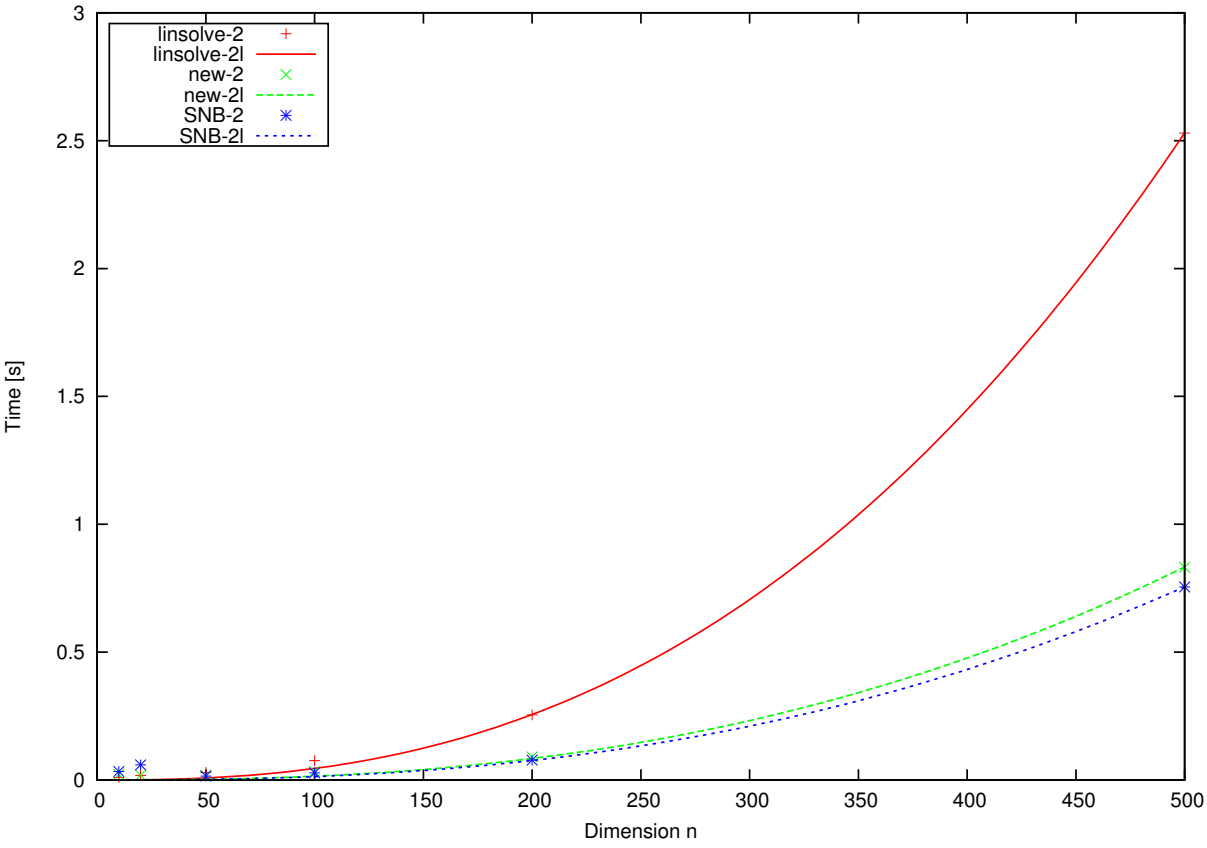


Figure 5.2: MRC conversion

in 5.2.2 is a little bit slower than the SNB implementation but still comparable. We have to say that the overhead of our new implementation is not negligible. As stated, this implementation arises from the whole system implementation with small modifications. It means that we are still using the architecture described in 5.3.4 including several threads and the communication scheme. On the contrary, the SNB was implemented directly for the tests in the simplest way.

### 5.3.3 System Architecture

The whole system is designed as almost homogeneous. The only exceptions where the processes are not equal are:

- loading the task including transformation into integer numbers.
- result verification
- storing the results

The reason of this inconsistency is that the performance improvement will be negligible. We can also assume that the task is available at one node locally and for other computational nodes it would be loaded throughout the network anyway.

Thus we can still talk about **master** and **slave** processes where the **master** process loads the task at the beginning, transforms it into the integer number range $\mathcal{Z}$ and then broadcasts the task in this form to all the other nodes while the **slave** is just receiving the task. At the end of the computation the **master** receives all the partial results and performs their verification followed by storing the results, while **slaves** just send their part of the result vector to the **master**. We can see that this is a reasonable degree of asymmetry.

### 5.3.4 Process architecture

As we decided to change the implementation language we also decided to involve the threading mechanisms. Each of the processes has several "independent" threads. Two of them perform computations directly and the other one is designated for receiving all the different messages from other processes.

The communication between processes is split into two different types of messages:

- A direct message addressed to the specific node. Often these messages are broadcast to all the processes
- The virtual ring messages, where the messages are passed across the virtual ring made from processes by sorting them according to their number.

The type of individual messages is given in section 5.3.4.3, where the different types of messages are described.

Figure 5.3 shows the overall new architecture of the system and all its processes.

In the following sections we present the control flow of each kind of thread from figure 5.3, that is involved in finding the solution itself more deeply.

#### 5.3.4.1 SLC solver thread

The main task of this thread is to solve the SLCs repeatedly.

**Algorithm 4** SLC solver thread

```
1: while not finished do
2:    while cachedResults() > cResults do
3:       wait()
4:    end while
5:    mod = getNextPrime()
6:    slc = toMod(sle, mod)
7:    result = solve(slc)
8:    send(result)
9: end while
```

Algorithm 4 is straightforward but to clarify we describe it in a few sentences. The thread is running in an "endless" while loop where it checks whether there are a lot of results cached (results still unprocessed by the MRC thread). If so, the thread waits on the condition variable which would be set at the moment when the count of cached results lowers under the specified *cResults* value. This value is configurable easily. If the count of cached results is lower then *cResults* the thread continues to the SLC solution itself. First, the thread obtains the next prime. Then it performs the transformation from the $\mathcal{Z}$ to $\mathcal{Z}_n$ representation. Subsequently, the obtained SLC is solved through Gaussian elimination and finally, the result is sent (broadcast) to all the other nodes.

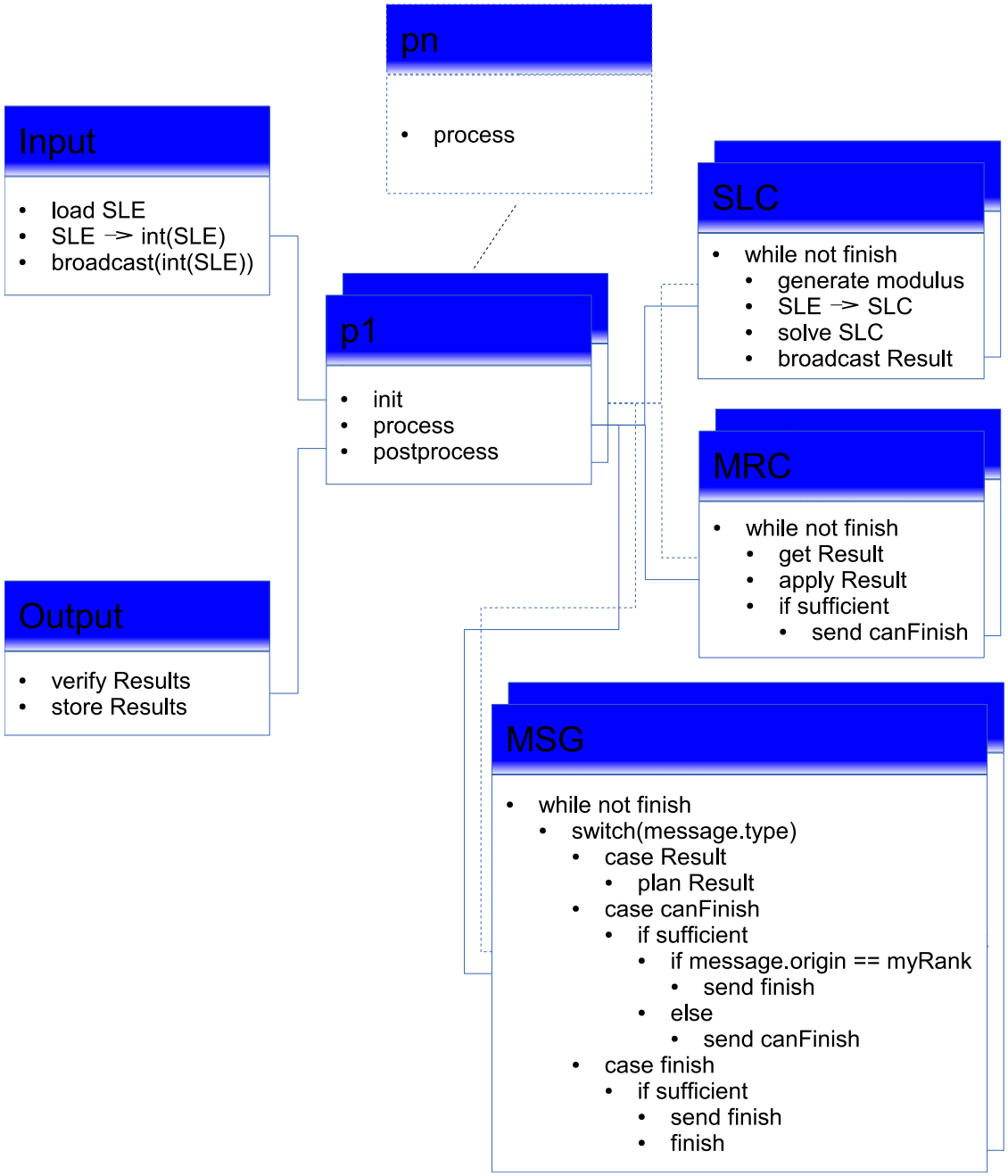


Figure 5.3: The process architecture - The system consists of $n$ processes. Only one process performs the input (loading and initial transformation of SLE) and output (verification and storing the results). All the processes includes SLC solver, MRC thread and message processing thread (MSG). The SLC solver performs the solution of single SLCs. The MRC thread performs the mixed radix conversion for the appropriate part of the result vector. The message processing thread handles all the received messages and takes appropriate actions.

#### 5.3.4.2 MRC thread

This thread is responsible for a mixed radix conversion including the initiation of the computation finalization.

**Algorithm 5** Mixed Radix Conversion thread

```
1: while (not finished) or (precision not sufficient) do
2:   res = getNextResult()
3:   if res then
4:     canFinish = mrcStep(res)
5:     if canFinish then
6:       send(wantFinishMessage)
7:     end if
8:   end if
9: end while
```

The steps of algorithm 5 should be described in more details. The MRC thread runs in an "endless" while loop as the SLC solver thread does. Line number 2 checks for the result that was obtained in the "past" and thus can be processed in MRC. If such a result exists, the execution thread will perform one MRC step described theoretically in 3.1.5. As the result of this step we get the information whether the particular results have a sufficient precision, and if they do, the *want-finish* message is sent.

#### 5.3.4.3 Message processor

This thread is simply responsible for receiving all the messages and processing them. The processing is different for each type of message. The following list describes the most important messages and their processing.

- *result message*: This is the message containing a result for some SLC. Processing this message means storing the result and confirming its receiving.
- *confirm message*: The *confirm* message is bound closely to the result message and it says that the process received the result. There is no additional processing for a message of this type.
- *want-finish* message: The *want-finish* message is sent whenever the **MRC** process draws the conclusion that the precision is sufficient for its part of the result vector.

After receiving such a message, the sufficiency of precision for the receiving process is checked. If the receiving process has a sufficient precision, too, and the message does not originate in this process (meaning it has not passed the whole virtual ring), the message is forwarded into the next process in the virtual ring. A special case is, when the precision of the receiving process is declared sufficient, but for the later step of the *MRC* conversion. In this case the message is replaced with receiving process's own *want-finish* message containing its finalization step. If the message comes from the receiving process, it means that all the nodes have agreed on finishing the computation and a *finish* message is sent. When the precision is not at all sufficient for the receiving process, the message is dropped.

- *finish* message: This message is sent after all the processes agree on sufficiency of precision. It is just forwarded in the virtual ring and the final step for the *MRC* conversion is set.

There is a check for results with timestamp that has already passed with every message receiving because it always generates a tick for logical clock thus it always results in a clock step (it can be more than one step) and every message also has the timestamp which means that the sending process would never send the message with timestamp smaller than received. Thus there could be new results to be passed into the MRC processing.

#### 5.3.4.4 Process architecture summary

As it was described in the previous section, the process consists of three "independent" threads of execution. The SLC solving thread which autonomously solves the SLCs for different moduli. The second one is the thread performing the MRC including the initiation of the execution finalization. The last thread handles all the received messages.

At the beginning of the second implementation we did not have the conditional wait in the SLC thread. The initially proposed solution of results cumulation was to set the lower priority using the **nice** to the SLC solver thread which should ensure that the MRC would be prioritized. But when solving the real problem for the Astronomical Institute of the Academy of Sciences of the Czech Republic we quickly hit the point from where the MRC processing was more complex then the SLC solution and we finished up with up to two times more SLCs solved than was necessary. Thus the necessity to control the SLC solver thread arose and we brought the conditional wait into the algorithm.

### 5.3.5 Complexity

This section focuses on complexity of the new implementation from different points of view. Its structure is similar to the structure of 3.3.

#### 5.3.5.1 Time complexity

Time complexity of almost all the parts of the processing is the same as in 3.3.1. The only difference is in the generation of prime numbers where the new implementation performs the generation independently on all the nodes. The time complexity analysis is done for a homogeneous distributed environment.

**Transformation to real numbers** The transformation to real numbers using the MRC conversion algorithm presented in 5.2.2 has the same overall complexity as expressed in 3.3.1.3 by formula 3.23. On the other hand, the MRC conversion is processed in parallel where every process performs its proportional part. Then the complexity can be expressed by formula 5.5.

$$\mathcal{O}(m^2 n \cdot p^{-1}) \tag{5.5}$$

**The whole computation** The complexity for the whole computation differs much the original implementation. Because all the processes are similar there will be no difference among them, thus, there will be no condition on processor or moduli count to avoid starvation of slave processes. The complexity of the whole computation is expressed by formulae 5.6.

$$\mathcal{O}\left(n^2 + \frac{m^2 n + m \cdot n^3}{p}\right) \tag{5.6}$$

The first element ($n^2$) of 5.6 represents the conversion into integer numbers as in 3.24. The fraction then represents the sum of the MRC conversion and the SLC solving where those parts are performed in parallel, thus divided by the number of processors.

#### 5.3.5.2 Memory consumption

The memory demands of the processes have changed due to the distribution of the MRC conversion algorithm. In this case, the data used for the MRC are also distributed through the processes.

The demands described in sections 3.3.2.1 and 3.3.2.2 have not changed but we state once more that at this time they fit to all the processes. The total memory consumption expressed for performing the computation on a single computer expressed in 3.3.2.4 still holds, too.

**Transformation to real numbers** Additionally the MRC conversion demands are appropriate to the size of the processed part of the vector. This part is also allocated in all the processes. The memory demands are the same as expressed in formulae 3.30 but when split to the $p$ processors, we obtain formula 5.7.

$$m_{bt} = k \cdot m \cdot n \cdot p^{-1} \tag{5.7}$$

Also it is necessary to count on the mixed radix coefficients for both negative and positive representations which gives $m \cdot 2(n+1) \cdot p^{-1}$. When we use 32-bit type for moduli, the mixed radix coefficients are also 32-bit wide.

#### 5.3.5.3 Communication complexity

The communication is more complex than in the original case described in 3.3.3. It can be split into several parts. The first part is the task distribution which, of course, has the same complexity. The second part is the distribution of the results. This part has the same complexity too, even it is not so obvious. And the last part is the agreement protocol for processing finalization. This part has not been present in the previous design at all.

**Task distribution** As we have said, there is no difference in this part of the algorithm. Thus the complexity is the same as in 3.3.3 specifically $\mathcal{O}(n^2)$.

**Results distribution** The complexity of results distribution is almost the same as in 3.3.3. It is due to the fact that the result is split into separate parts and every part is sent to the appropriate node which is responsible for this MRC part. More than that, we will save the sending of a part belonging to the originating process which is not sent through the network. Therefore the complexity is $\mathcal{O}(m \cdot (1 - \left\lfloor \frac{n}{p} \right\rfloor)n)$.

**Finalization agreement** We present the complexity for several situations. The worst case is when all the processes will send both (*want-finish* and *finish*) messages and each of these messages runs almost the whole virtual circle. For the *want-finish* message, the longest run can be of $2(p-1)$ length because this type of message can be replaced by the brand new message of the receiving process which has to go through the whole circle again. For the *finish* message the longest run is clearly $p-1$. We can see that $3(p-1)p$ messages of constant size (messages of those types carry only a modulus) are sent. On the other hand, the best case is when one process sends the *want-finish* message which passes the whole circle and in consequence the *finish* message is sent and it also passes the circle. The complexity of this case is $2p$.

**Rationale** In the following paragraphs we defend the statement that the communication complexity is relatively low in comparison to the computation itself. Table 5.4 presents the size and theoretical time consumed by sending the set of linear equations as the distribution of the task and the size and sending time for one result vector of one SLC.

Table 5.4: SLC and result vector sizes and timings for different bandwidth networks

| dim | $SLC_{size}$[Mb] | $T_{100Mb/s}$ [s] | $T_{1Gb/s}$ [s] | $RES_{size}$[Mb] | $T_{100Mb/s}$ [s] | $T_{1Gb/s}$ [s] |
|---|---|---|---|---|---|---|
| 1000 | 122 | 1.221 | 0.119 | 0.030 | 3.05E-4 | 2.98E-5 |
| 1500 | 274 | 2.747 | 0.268 | 0.046 | 4.58E-4 | 4.47E-5 |
| 2000 | 488 | 4.883 | 0.477 | 0.061 | 6.10E-4 | 5.96E-5 |
| 2500 | 762 | 7.629 | 0.745 | 0.076 | 7.63E-4 | 7.45E-5 |
| 3000 | 1098 | 10.986 | 1.073 | 0.092 | 9.16E-4 | 8.94E-5 |
| 12000 | 17578 | 175.8 | 17.17 | 0.366 | 3.66E-3 | 3.58E-4 |

The communication passes in three phases described in the preceding paragraphs. For the first phase (the task distribution) the communication complexity would be a defining one for its overall complexity. But the second phase, where the SLCs are solved and the results

are sent among processes, we can compare the timings for sending the result from table 5.4 with timings for solving the SLC from table 4.3, the communication complexity would be negligible.

With the growing dimension of the SLE the time needed for solving one SLC grows with third power of dimension whilst the result vector size grows only as fast as the dimension. Thus, the only thing which could increase the communication complexity is the growing number of processes. But, for example, when having 100 processes and solving the SLE of dimension 1000 we get 100 results at a given time, thus every half a second and we have another half of second to send them but the time for sending one result, on the standard $100Mb/s$ network is 0.0003, thus for 100 result the time needed is 0.003 which obviously is less than the SLC timing ($0.003 < 0.466$).

### 5.3.6 Architecture impact

The two problems described in section 5.1 are solved by the new architecture of the system. As of memory demands, it is obvious that when the architecture is homogeneous the memory demands are the same on all the nodes. In the following charts we demonstrate the elimination of slave nodes starvation phenomenon.

Chart 3.3 shows the process utilization for the problem of dimension 500 solved by the use of 24 nodes (processes). We can see the gaps for the slave processes (red color) caused by the overloading of the master process (blue color).

The results of the distribution of the MRC process through all the computational nodes meaning the use of architecture presented in section 5.3.4 are presented in chart 5.4. This chart shows the same 500 dimension problem solution on the same 24 processes but using the new designed system. To make it clearer we have preserved the colors to the operations, meaning that the SLC solving is red color, MRC is blue. For completeness we have also logged the utilization of the message processor thread, using green color.

As we can see, there are no gaps in the screen to give a better image so we present the detail of a specific time close to the end of the computation for new architecture in figure 5.5.

By comparison of figures 3.3 and 5.5 we can observe that the new architecture is much better in usage of the whole cluster, especially, for data where the one master node architecture leads to starvation of slave processes due to the overload of the master node with MRC

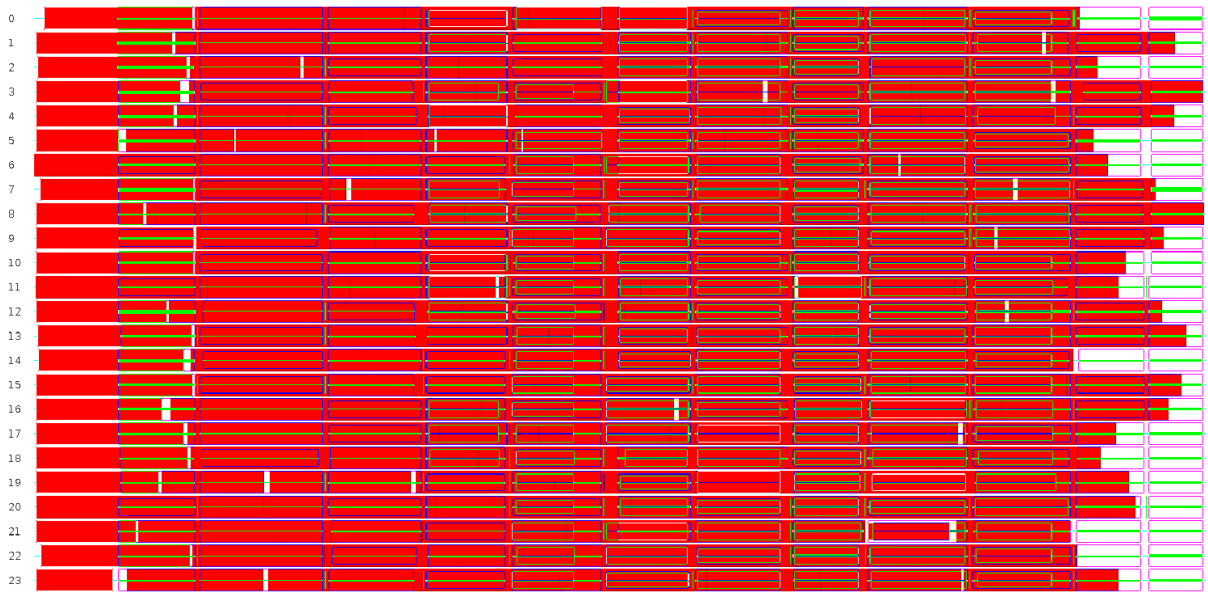


Figure 5.4: An example of the node workload for new architecture ($n = 500, p = 24$) - The meaning of colours remains the same as in 3.2. Message processing is marked with green colour.

process. There is no visible starvation in ongoing computation.

### 5.3.7 Scalability

In this section we present the results achieved by MPI use and with distribution of the MRC process to all nodes. Table 5.5 shows timings for solution of SLEs of different dimensions using different number of computational nodes. The results presented here were obtained on the **STAR** cluster[2].

The timings of linsolve run from table 5.5 are presented in logarithmic scale at figure 5.6.

From table 5.5 and figure 5.6 we can observe that the system scales well. For small dimension the running time can grow when we use more processes. The reason for such behaviour can be the increased complexity of algorithms used, especially, the agreement on computation finalization can take more time when exposing more processes. The measurement shows such behaviour for dimensions 100 and 200. The conclusion can be that we devote some time to problems of small dimensions but we gain a significant speedup

[2]More information about the cluster are available at [1]

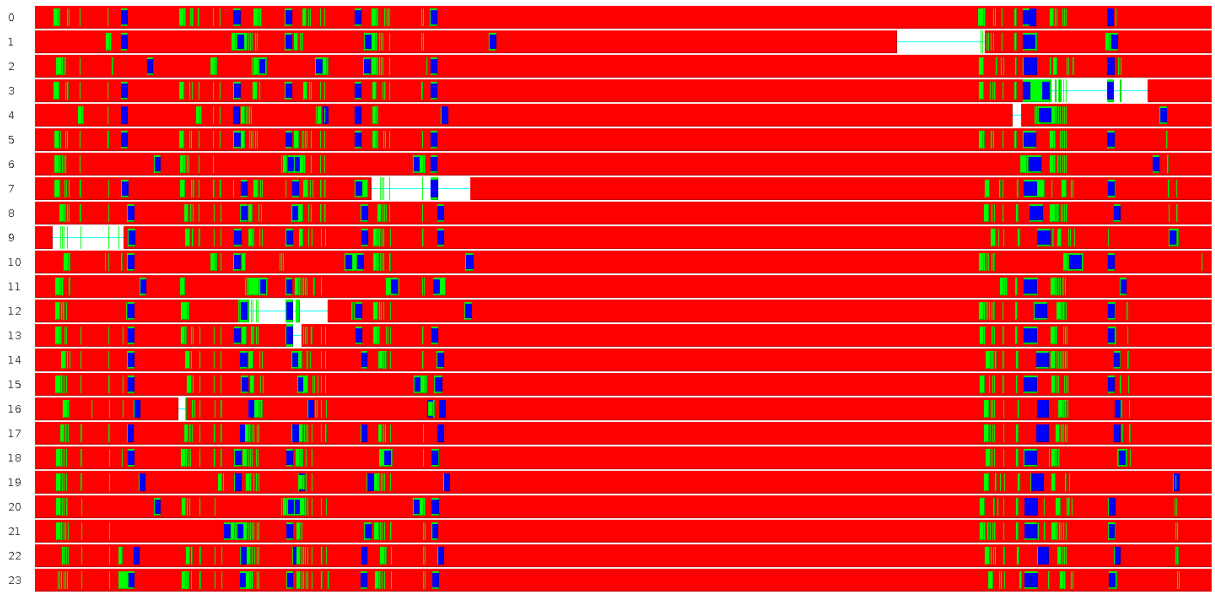


Figure 5.5: An example of the node workload for new architecture ( $n = 500, p = 24$) - A detail close to the end. The meaning of colours remains the same as in 5.4. Even in the detailed view the nodes are busy almost all the time.

for problems of large dimensions.

## 5.4 Summary

This chapter deals with the design of a new homogeneous system for solving linear equation systems. We have developed an algorithm for the MRC in a distributed environment which fits our needs. This algorithm is able to compete with other distributed MRC algorithms and, in addition, it can be performed during the SLC solution. Thus it can help us state the sufficient precision of the results and stop the processing.

By developing homogeneous architecture we have eliminated the problem of starvation of slave processes when the master is overloaded which is described in 3.3.1.4. Also the processes are now well balanced from the memory demands point of view.

Results contained in this chapter have not been prviously published.

Table 5.5: Timings for parallel run SLE solution T[s]

| p | dimension | | | | | |
|---|---|---|---|---|---|---|
| | 100 | 200 | 500 | 1000 | 2000 | 5000 |
| 2 | 0.082 | 0.430 | 12.465 | N/A | N/A | N/A |
| 4 | 0.065 | 0.274 | 11.429 | N/A | N/A | N/A |
| 8 | 0.070 | 0.278 | 6.787 | 97.556 | N/A | N/A |
| 12 | 0.100 | 0.226 | 5.222 | 71.944 | 1188.25 | N/A |
| 24 | 0.243 | 0.348 | 3.079 | 37.360 | 635.63 | 27263.42 |
| 48 | 0.630 | 0.712 | 2.362 | 22.148 | 342.12 | 13993.17 |

# 6 Application

In the past year we have been cooperating with the Astronomical Institute of the Academy of Sciences of the Czech Republic, namely with Dr. Miroslav Bárta and Jan Skála. We are solving a system of equations provided by them.

They are studying plasma multi-scale processes in solar plasma by magnetohydrodynamic (MHD) simulations, namely magnetic field reconnection and its multi-scale nature. The MHD set of partial differential equations (PDEs) is solved by the Least-Squares Finite Element Method (LSFEM) from which a system of linear equations arises. The LSFEM is able to solve an overdetermined system of PDEs, which allows to include the Gauss's law for magnetism to MHD equations and keep the simulation 'divergence free' (no magnetic charge). Unfortunately, it leads to an ill-conditioned set of equations at a later stage of system evolution. The resulting system of equations is solved by iterative solver – Conjugate Gradient Method (CGM).

The simulation results shows a possible problem with precision of CGM solution at places where the current density (curl of magnetic field) become high. We want to clarify this question by solving this system by means of the RNS method and checking the precision of CGM. The LSFEM implementation of the MHD numerical solver is described in [32].

Recently, we have been able to solve the problem of the dimension of approximately twelve thousand. We are optimizing the code for solving problems of greater dimensions. As the system has been designed for solving dense matrices there are several possibilities for further optimization of sparse matrix processing. We have to focus on memory optimizations and to optimization of the MRC process. The results for the currently solved systems show

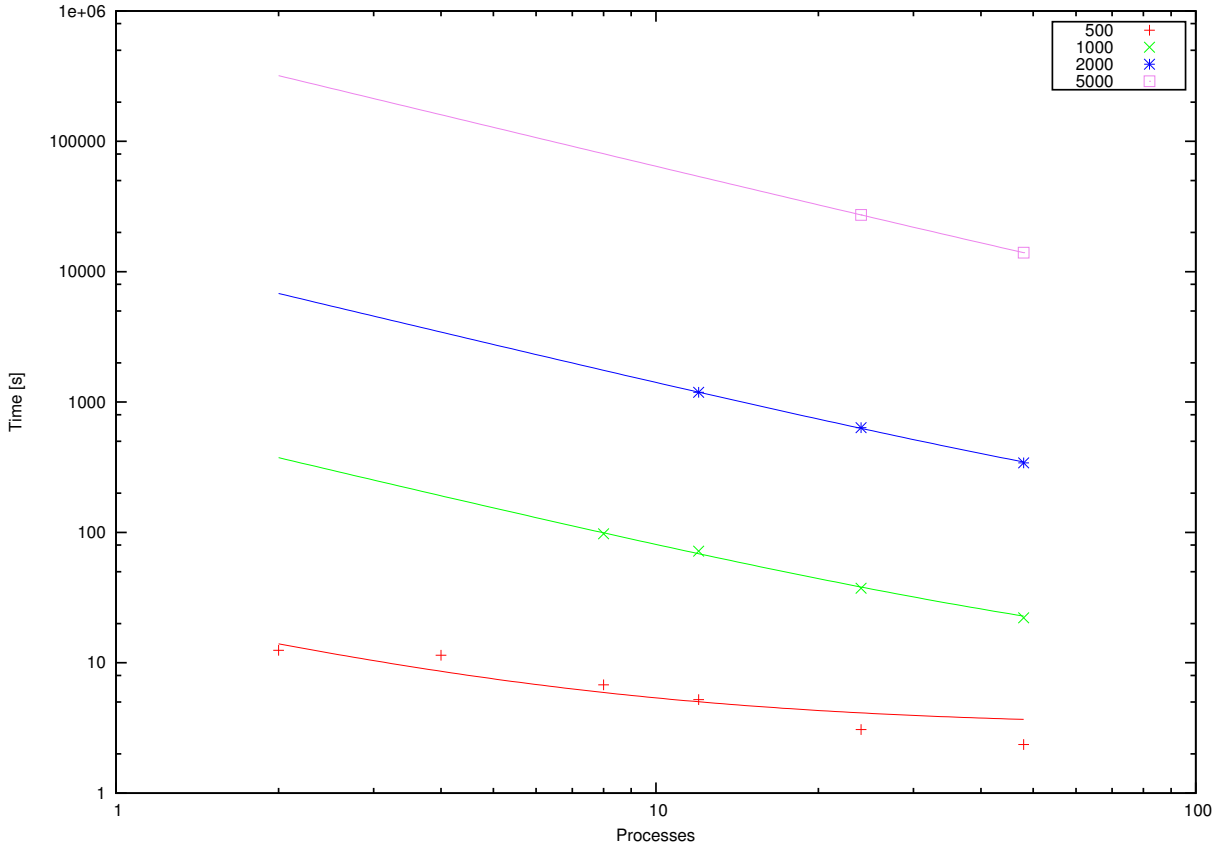


Figure 5.6: The linsolve parallel run timings in dependency on process count for different dimensions of SLE

that our system gives a slightly better accuracy but the data obtained by the CGM are also sufficiently accurate.

**Note 6.0.1.** *We have solved the problem of dimension* 12243. *This solution took* 211640 *seconds (*2.5 *days) and ran on 48 nodes of* **STAR** *cluster. The solver used* 46010 *moduli to gain accurate results, where the solution of one SLC took one second (It corresponds to table 4.4). Thus, when using 48 nodes the solution of separate SLCs should take a thousand seconds. We can see that the MRC process complexity gets to front and the SLC solution is almost negligible.*

# 7 Conclusions

This thesis describes a consecutive development of the system for solving linear equation systems. The system was originally developed as a diploma project. Further, the project was optimized in the way of space and time complexity. Those optimizations were published in several papers [A.1, A.2, A.3, A.6] and are presented in chapter 4. Afterwards, some problems in the system architecture were discovered and they had to be solved. The problems were the starvation of the slave processes solving SLCs when the master process is overloaded. Another problem with the master-slave architecture was the imbalance in memory demands. The solution of those problems was a completely different architecture of the system which is homogeneous and thus the imbalance disappeared and the problem of starvation was eliminated due to the distribution of the MRC process. The new architecture, including its detailed analysis, is presented in Chapter 5.

We have used various optimization techniques over the time. The first type was the usage of the machine code oriented language. The floating point unit was engaged consequently for multiplication with reduction. This seems unusual but gives a better performance. The next reasonable step was the usage of SIMD instruction extension of common processors which gives a significant speedup, as well. Finally we have chosen a different way of avoiding the expensive division operation the Montgomery domain representation, which gives an even better performance. From the memory demands point of view, the system has been redesigned to eliminate the redundant demands of auxiliary structures. The main contribution of this part is the multiplication with reduction using SSE2 floating point instructions giving the second best performance. A better implementation is only the Montgomery domain using the SSE2 integer instructions.

The architecture of the system has been completely changed. The new design is “entirely” homogeneous. This design gives more flexibility in usage of resources, eliminates problems related to the heterogeneous architecture and gives the possibility of a better future balancing. The outcome is a more complex design, which was difficult to realize and reveals errors made during the realization. The main contribution of this part is a new distributed algorithm for a mixed radix conversion with dynamic moduli ordering suitable for usage during the SLC elimination processing with sufficient results precision determination taken into account.

# 8 Future Work

We suggest the following themes for future work:

- There is a space for optimization of the processing for sparse matrices. We have already faced the problem of solving the SLE where the matrix of coefficients is sparse. In this case, the memory demands become limiting because we store a huge amount of zero elements of the coefficient matrix. Some of the special sparse matrix storing formats should be used for sparse matrices, like, for example, Compressed sparse row (CSR).

- The processing of the partial algorithms using General-purpose computing on graphics processing units (GPGPU). The Graphics processing units (GPU) are nowadays capable of massive multiprocessing, which could be advantageous and lead to a better performance. Our research group has already made some progress in this matter [15].

- The new implementation described in Chapter 5 leads to the usage of more than one SLC solver thread in the process. The opportunity of adding some solver threads should be investigated and if at all possible tested, too.

- Chapter 5 describes the MRC algorithm designed with view to the rebalancing possibility. Unfortunately, the rebalancing itself has not been developed yet. This could be another direction of future research. Also, there is a possibility of further optimization of this algorithm.

- Another possible step for the future is the optimization of the mixed radix conversion process itself. We have observed that for some tasks the complexity of mixed radix conversion overgrows the complexity of the SLC solving process.

# 9 Bibliography

[1] *STAR cluster web page*. http://star.fit.cvut.cz/.

[2] *Valgrind Documentation*. Valgrind official webpage.

[3] C. Brezinski, M. Redivo-Zaglia, G. Rodriguez, and S. Seatzu. Extrapolation techniques for ill-conditioned linear systems. *Numer. Math.*, 81(1):1–29, 1998.

[4] C. Brezinski, M. Redivo-Zaglia, G. Rodriguez, and S. Seatzu. Multi-parameter regularization techniques for ill-conditioned linear systems. *Numer. Math.*, 94(2):203–228, 2003.

[5] Çetin Kaya Koç. A parallel algorithm for exact solution of linear equations via congruence technique. *Computers and Mathematics with Applications*, 23(12):13–24, July 1992.

[6] Çetin Kaya Koç, A. Guvenc, and B. Bakkaloglu. Exact solution of linear equations on distributed-memory multiprocessors. In *Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics*, pages 1339–1341, July 1994.

[7] Çetin Kaya Koç, A. Guvenc, and B. Bakkaloglu. Exact solution of linear equations on distributed-memory multiprocessors. *Parallel Algorithms and Applications*, 3:135–143, 1994.

[8] Çetin Kaya Koç and R. M. Piedra. A parallel algorithm for exact solution of linear equations. In *Proceedings of International Conference on Parallel Processing*, pages 1–8, August 1991.

[9] A. DAX. Partial pivoting strategies for symmetric gaussian elimination. *Mathematical Programming*, 22(1):288–303, December 1982. DOI 10.1007/BF01581044.

[10] K. Dlouhý. Řešení rozsáhlých a špatně podmíněných SLR na výpočetním svazku "STAR": I. systémové řešení. Master's thesis, Czech Technical University - Faculty of Electrical Engineering, Department of Computer Science, 2005.

[11] A. Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel and AMD CPU's*, 2008. http://www.agner.org/optimize/.

[12] H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EL-8:140147, June 1959.

[13] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[14] R. Gregory. *Error-free computation: why it is needed and methods for doing it*. R. E. Krieger, 1980.

[15] J. Hladík and I. Šimeček. Modular arithmetic for solving linear equations on the gpu. In *Seminar on Numerical Analysis*, pages 68–70, January 2012.

[16] J. A. Howell. Algorithm 406: exact solution of linear equations using residue arithmetic [f4]. *Commun. ACM*, 14(3):180–184, 1971.

[17] J. A. Howell. Remark on algorithm 406. *Commun. ACM*, 16(5):311, 1973.

[18] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)*, 1985. http://grouper.ieee.org/groups/754.

[19] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[21] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper. *The GNU C Library Reference Manual*. Free Software Foundation, Inc., September 2007. http://www.gnu.org/software/libc/manual/.

[22] R. Lórencz. *Aplikovaná Numerika a Kryptografie*. Vydavatelství ČVUT, 2004.

[23] R. Lórencz. *New Approaches to Computing the Modular Inverse in Hardware*. Habilitation thesis, Faculty of Electrical Engineering, Czech Technical Universtity in Prague, 2004.

[24] R. Lórencz and P. Mikšík. The residue number system. *IRE Transactions on Electronic Computers*, EL-8:140147, June 1959.

[25] R. Lórencz and M. Morháč. Modular system for solving linear equations exactly, ii. hardware realization and firmware. *Computing and Informatics*, 11(5):497–507, 1992. ISSN 1335–9150.

[26] P. L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(170):519–521, 1985.

[27] M. Morháč and R. Lórencz. Modular system for solving linear equations exactly, i. architecture and numerical algorithms. *Computing and Informatics*, 11(4):351–361, 1992. ISSN 1335–9150.

[28] A. Neumaier. Solving ill-conditioned and singular linear systems: A tutorial on regularization. *SIAM Review*, 40(3):636–666, 1998.

[29] M. Olschowka and A. Neumaier. A new pivoting strategy for Gaussian elimination. *Linear Algebra and its Applications*, 240(1–3):131–151, 1996.

[30] C. Pommerell and W. Fichtner. Pils: an iterative linear solver package for ill-conditioned systems. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 588–599, New York, NY, USA, 1991. ACM.

[31] G. Rodriguez, S. Seatzu, and D. Theis. A new technique for ill-conditioned linear systems. *Numer. Algorithms*, 33(1-4):433–442, 2003. International Conference on Numerical Algorithms, Vol. I (Marrakesh, 2001).

[32] J. Skala and M. Barta. Lsfem implementation of mhd numerical solver. *ArXiv e-prints*, June 2012. http://adsabs.harvard.edu/abs/2012arXiv1206.2730S.

[33] M. Wolff. *Massif Visualizer*. KDE e.V. non-profit organization representing KDE Project. https://projects.kde.org/projects/extragear/sdk/massif-visualizer.

[34] T. Zahradnický. *MOSFET Parameter Extraction Optimization*. PhD thesis, Faculty of Electrical Engineering, Czech Technical Universtity in Prague, December 2010. http://users.fit.cvut.cz/ zahradt/PhDThesis.pdf.

# 10 Refereed Publications of the Author

[A.1] L. Vondra, T. Zahradnický, R. Lórencz System Optimization of Solving a Set of Linear Congruencies. *Acta Electrotechnica et Informatica* 9(3):1–8, 2009 ISSN 1335-8243

[A.2] L. Vondra, T. Zahradnický, R. Lórencz System Optimization of Solving a Set of Linear Congruencies. In *Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering*, pages 344–351, Košice, Slovakia, 2008. Department of Computers and Informatics of FEI, Technical University Košice. ISBN 978-80-8086-092-9.

[A.3] L. Vondra. Using Special Instructions for Multiplication with Reduction. In *Počítačové architektury a diagnostika 2007*, pages 47–52, Plzeň, Czech Republic, 2007. Západočeská universita, Fakulta aplikovaných věd. ISBN 978-80-7043-605-9.

[A.4] L. Vondra. Solving Linear Equation Systems Exactly. In *Počítačové architektúry a diagnostika*, pages 95–100, Bratislava, Slovakia, 2006. Ústav informatiky SAV. ISBN 80-969202-2-7.

# 11 Unrefereed Publications of the Author

[A.5] L. Vondra, R. Lórencz System for solving linear equation systems. In *Seminar on Numerical Analysis*, pages 171–174, Liberec, Czech Republic, 2012. Technical University of Liberec. ISBN 978-80-7372-821-2.

[A.6] L. Vondra. Efficient algorithms for modular multiplication. In *POSTER 2007* [CD-ROM]. Prague: CTU, Faculty of Electrical Engineering, 2007 4 pages

[A.7] L. Vondra. Řešení rozsáhlých a špatně podmíněných SLR na výpočetním svazku "STAR": II. Řešení soustav lineárních kongruencí, Master's thesis, Czech Technical University - Faculty of Electrical Engineering, Department of Computer Science, 2005

# A List of Abbreviations

| | |
|---|---|
| SLE | Set of Linear Equations |
| RHS | Right-Hand Side |
| LHS | Left-Hand Side |
| RNS | Residual Number System |
| SLC | Set of Linear Congruencies |
| MRC | Mixed Radix Conversion |
| CRT | Chinese Remainder Theorem |
| PNT | Prime Number Theorem |
| CPU | Central Processing Unit |
| FPU | Floating Point Unit |
| GPU | Graphics Processing Unit |
| GPGPU | General-Purpose computation on Graphics Processing Units |
| SIMD | Single Instruction Multiple Data |
| SSE | Streaming SIMD Extension |
| SSE2 | Streaming SIMD Extension 2 |
| MMX™ | "MultiMedia eXtension" |
| MPI | Message Passing Interface |
| API | Application Programming Interface |
| HPC | High Performance Computing |
| SNB | Single Node Broadcast |
| SMP | Symmetric MultiProcessing |
| SHM | SHared Memory |
| IPC | Inter-Process Communication |
| HT | Hyper-Threading |
| AVX | Advanced Vector eXtensions |
| AVX2 | Advanced Vector eXtensions 2 |
| MHD | MagnetoHydroDynamic |
| LSFEM | Least-Squares Finite Element Method |
| CGM | Conjugate Gradient Method |
| PDE | partial differential equation |

# B Montgomery domain

## B.1 Introduction

The Montgomery domain is a representation of integer numbers focused on efficient processing of the multiplication with reduction. It was introduced By Peter Montgomery in his article [26]. This representation does not affect the efficiency of other operations.

We are trying to find an alternative to the common multiplication with reduction B.1 which would not use the division operation.

$$c \equiv a \cdot b \bmod n. \tag{B.1}$$

Because numbers have to be converted to and from the Montgomery domain, a single modular multiplication performed using a Montgomery reduction is actually slightly less efficient than a common approach. However, for more complex processing like modular exponentiation or, in our case, Gaussian elimination, the conversion should be made once at the start and once at the end of the whole process. In this case, the greater speed of the Montgomery steps should outweigh the need for extra conversions.

## B.2 Formal statement

We have modulus $m$ which is a positive integer ($m > 1$). We have radix $R$ which is coprime to $m$ ($\gcd(m, R) = 1$) such that $R > m$. Let $R^{-1}$ be the multiplicative inverse modulo $m$ of $R$. Then for $T$ such that $0 \leq T < Rm$ the Montgomery reduction of $T$ modulo $m$ with respect to $R$ is defined by following formula:

$$T \cdot R^{-1} \bmod m. \tag{B.2}$$

The algorithm for getting Montgomery reduction is much more efficient then common remainder modulo $m$ operation using division.

## B.3 Rationale

We wish to calculate $c$ such that B.1 holds. Rather than working directly with $a$ and $b$, we define the residue (common residue multiplied by radix $R$).

$$\begin{aligned} \overline{a} &\equiv aR \bmod m \\ \overline{b} &\equiv bR \bmod m \end{aligned} \tag{B.3}$$

The number $R$ has to be greater than and coprime to the modulus $m$. It is chosen so that the operations of division and remainder are easy to perform. An obvious choice is a power of two for machine processing. Then these operations correspond to the shift right and bitwise mask, respectively. When the $R$ is chosen as power of two it is sufficient when $m$ is odd to fulfill the relative primality.

For such a representation the operations addition and subtraction does not change:

$$xR + yR \equiv zR \bmod m \iff x + y \equiv z \bmod m. \tag{B.4}$$

It is very important not to affect the efficiency of other operations like addition and subtraction. It would not be beneficial if we had to transform the data during computations back from and into Montgomery domain due to the overhead of such transformations.

To define the multiplication operation we need the modular inverse of $R$, $R^{-1}$ such that

$$\begin{aligned} R \cdot R^{-1} &\equiv 1 \bmod m \\ R \cdot R^{-1} &= km + 1 \\ k &= \frac{R \cdot R^{-1} - 1}{m} \end{aligned} \tag{B.5}$$

where $k$ is a positive integer and its value can be expressed from the last formula in B.5.

To find $c$ such that formula B.1 holds we have to find its representation in the Montgomery domain $\overline{c} \equiv cR \bmod m$ :

$$\overline{c} \equiv cR \equiv (a \cdot b)R \equiv (aR \cdot bR)R^{-1} \equiv (\overline{a} \cdot \overline{b})R^{-1} \bmod m \tag{B.6}$$

Formula B.6 states that to gain the $\overline{c}$ we perform the ordinary multiplication and then we

perform the Montgomery reduction $T \cdot R^{-1} \bmod m$.

To get the $c$ we transform $\overline{c}$ from the Montgomery domain by another Montgomery reduction operation.

$$c \equiv \overline{c}R^{-1} \bmod m. \tag{B.7}$$

This rationale leads to the following algorithm.

## B.4 Description of Algorithm

The Montgomery reduction algorithm 6 calculates $T \cdot R^{-1} \bmod m$.

**Algorithm 6** Montgomery reduction

**Require:** $m$ such that $m > 1$; $R$ such that $R > m$ and $\gcd(m, R) = 1$; $k$ such that $k = (RR^{-1} - 1)/m \wedge RR^{-1} \equiv 1 \bmod m$

**Ensure:** For $T$ such that $0 \leq T < R \cdot m$ return $T \cdot R^{-1} \bmod m$

```
1: p ← (T mod R)k mod R
2: t ← (T + pm)/R
3: if t ≥ m then
4:     return t − m
5: else
6:     return t
7: end if
```

We can observe that algorithm 6 does not contain remainder modulo $m$ operation. There are only multiplication, addition, subtraction, division by $R$ and remainder modulo $R$ operations. When the radix $R$ is well chosen the last operations become the bitshift right and bit mask and those are inexpensive as well as the other mentioned operations.

To validate algorithm 6 observe the following

- $pm \equiv Tkm \bmod R$. By the definition of $R^{-1}$ and $k$ B.5, $km + 1$ is a multiple of $R$, so $T(km + 1) \equiv Tkm + T \equiv 0 \bmod R$ thus $Tkm \equiv -T \bmod R$. Therefore, $(T + pm) \equiv 0 \bmod R$; in other words, $(T + pm)$ is divisible by $R$ without remainder and so $t$ is an integer.
- Moreover, $tR = (T + pm) \equiv T \bmod m$; then, $t$ fulfills required relation B.2.

- When $0 \leq T < Rm$ then $t < 2m$ because $p < R \Rightarrow pm < Rm$, thus $T + pm < 2Rm$. Consequently the final result of algorithm 6 is always less than $m$.

This method is less efficient for calculation of a simple multiplication with reduction alone, due to necessary transformations into the Montgomery representation and back. But when the processing is more complex and requires more than just one multiplication, like modular exponentiation or Gaussian elimination, in our case, the efficiency of single Montgomery reduction comes to front. We can say that this method is suitable for processing where the benefits of Montgomery reduction outweigh the overhead of transformations.