Studies in Computational Intelligence 397

Elena Nikolaevskaya
Alexandr Khimich
Tamara Chistyakova

Programming with Multiple Precision

Springer

Studies in Computational Intelligence, Volume 397

Editor-in-Chief
Prof. Janusz Kacprzyk
Systems Research Institute
Polish Academy of Sciences
ul. Newelska 6
01-447 Warsaw
Poland
E-mail: kacprzyk@ibspan.waw.pl

Further volumes of this series can be found on our homepage: springer.com

Vol. 376. Toyohide Watanabe and Lakhmi C. Jain (Eds.)
Innovations in Intelligent Machines – 2, 2011
ISBN 978-3-642-23189-6

Vol. 377. Roger Lee (Ed.)
Software Engineering Research, Management and Applications 2011, 2011
ISBN 978-3-642-23201-5

Vol. 378. János Fodor, Ryszard Klempous, and Carmen Paz Suárez Araujo (Eds.)
Recent Advances in Intelligent Engineering Systems, 2011
ISBN 978-3-642-23228-2

Vol. 379. Ferrante Neri, Carlos Cotta, and Pablo Moscato (Eds.)
Handbook of Memetic Algorithms, 2011
ISBN 978-3-642-23246-6

Vol. 380. Anthony Brabazon, Michael O'Neill, and Dietmar Maringer (Eds.)
Natural Computing in Computational Finance, 2011
ISBN 978-3-642-23335-7

Vol. 381. Radosław Katarzyniak, Tzu-Fu Chiu, Chao-Fu Hong, and Ngoc Thanh Nguyen (Eds.)
Semantic Methods for Knowledge Management and Communication, 2011
ISBN 978-3-642-23417-0

Vol. 382. F.M.T. Brazier, Kees Nieuwenhuis, Gregor Pavlin, Martijn Warnier, and Costin Badica (Eds.)
Intelligent Distributed Computing V, 2011
ISBN 978-3-642-24012-6

Vol. 383. Takayuki Ito, Minjie Zhang, Valentin Robu, Shaheen Fatima, and Tokuro Matsuo (Eds.)
New Trends in Agent-Based Complex Automated Negotiations, 2012
ISBN 978-3-642-24695-1

Vol. 384. Daphna Weinshall, Jörn Anemüller, and Luc van Gool (Eds.)
Detection and Identification of Rare Audiovisual Cues, 2012
ISBN 978-3-642-24033-1

Vol. 385. Alex Graves
Supervised Sequence Labelling with Recurrent Neural Networks, 2012
ISBN 978-3-642-24796-5

Vol. 386. Marek R. Ogiela and Lakhmi C. Jain (Eds.)
Computational Intelligence Paradigms in Advanced Pattern Classification, 2012
ISBN 978-3-642-24048-5

Vol. 387. David Alejandro Pelta, Natalio Krasnogor, Dan Dumitrescu, Camelia Chira, and Rodica Lung (Eds.)
Nature Inspired Cooperative Strategies for Optimization (NICSO 2011), 2011
ISBN 978-3-642-24093-5

Vol. 388. Tiansi Dong
Recognizing Variable Environments, 2012
ISBN 978-3-642-24057-7

Vol. 389. Patricia Melin
Modular Neural Networks and Type-2 Fuzzy Systems for Pattern Recognition, 2012
ISBN 978-3-642-24138-3

Vol. 390. Robert Bembenik, Lukasz Skonieczny, Henryk Rybiński, and Marek Niezgódka (Eds.)
Intelligent Tools for Building a Scientific Information Platform, 2012
ISBN 978-3-642-24808-5

Vol. 391. Herwig Unger, Kyandoghere Kyamaky, and Janusz Kacprzyk (Eds.)
Autonomous Systems: Developments and Trends, 2012
ISBN 978-3-642-24805-4

Vol. 392. Narendra Chauhan, Machavaram Kartikeyan, and Ankush Mittal
Soft Computing Methods for Microwave and Millimeter-Wave Design Problems, 2012
ISBN 978-3-642-25562-5

Vol. 393. Hung T. Nguyen, Vladik Kreinovich, Berlin Wu, and Gang Xiang
Computing Statistics under Interval and Fuzzy Uncertainty, 2012
ISBN 978-3-642-24904-4

Vol. 394. David A. Elizondo, Agusti Solanas, and Antoni Martínez-Ballesté (Eds.)
Computational Intelligence for Privacy and Security, 2012
ISBN 978-3-642-25236-5

Vol. 395. Srikanta Patnaik and Yeon-Mo Yang (Eds.)
Soft Computing Techniques in Vision Science, 2012
ISBN 978-3-642-25506-9

Vol. 396. Marielba Zacarias and José Valente de Oliveira (Eds.)
Human-Computer Interaction: The Agency Perspective, 2012
ISBN 978-3-642-25690-5

Vol. 397. Elena Nikolaevskaya, Alexandr Khimich, and Tamara Chistyakova
Programming with Multiple Precision, 2012
ISBN 978-3-642-25672-1

Elena Nikolaevskaya, Alexandr Khimich, and
Tamara Chistyakova

Programming with Multiple Precision

Springer

Authors
Elena Nikolaevskaya PhD.
V.M. Glushkov Institute of Cybernetics
of NAS of Ukraine
Kyiv
Ukraine

Tamara Chistyakova PhD.
V.M. Glushkov Institute of Cybernetics
of NAS of Ukraine
Kyiv
Ukraine

Dr. Alexandr Khimich
V.M. Glushkov Institute of Cybernetics
of NAS of Ukraine
Kyiv
Ukraine

Additional material to this book can be downloaded from http://extra.springer.com

ISSN 1860-949X
e-ISSN 1860-9503
ISBN 978-3-642-25672-1
e-ISBN 978-3-642-25673-8
DOI 10.1007/978-3-642-25673-8
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2011941765

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Modern needs in solution of qualitatively new scientific and technical problems make it necessary to handle large size of information within a reasonable time. Effective solution of such problems is possible only on parallel computers.

One of the most important uses of computers is their application to solve scientific and engineering problems, most of which ultimately reduced to interim steps in problem solution of computational mathematics. Using computers of parallel architecture makes it possible to increase appreciably the dimension of discrete models. However, as is known, obtaining computer solutions doesn't always have physical meaning. The reasons for this are the following: firstly, the approximate nature of the source data, and the errors which arise from their representation in computer memory, i.e., when translated from decimal to binary; secondly, the rounding errors in the calculations due to the finite size of the computer's discharge grid; thirdly, the error which is the result of the replacement of an infinite iterative process into finite. Therefore, after staging a mathematical problem and putting it into the computer to explore the issues of the correct setting of the computer model of the problem, its condition and authenticity of the results obtained.

Why do we write this book. When we started research the work “Creating parallel algorithms and programs of Computational Mathematics with multiple precision” in V.M. Glushkov Institute of Cybernetics, after analyzing the existing tools to improve accuracy, we decided to use GNU Multiple Precision library. We tried to find some information and examples of using GMP in the Internet. Unfortunately, we didn't manage to find much information about it, several publications were the only finding, but without examples programs. Our department made a considerable amount of work in this field and now we want to publish everything concerning our research in order to help people who work on the arbitrary precision problems and are interested in them.

This book is a real guide for writing programs in C/C++ using the GMP and MPFR libraries to improve the accuracy of the solutions, including in parallel programming. It focuses on solving the problem of achieving reliable solutions and obtaining software algorithmic methods expressed in variable digit. The book includes many examples of the program code with its experiments results. It will be interesting for many scientists, programmers and students, who want to study

programming, improve or expand their level programming. The book is useful for graduates in computer science and mathematics (perhaps too specialized for the most of undergraduates, at least in its present state), researchers in Numerical Analysis, Computer Algebra and developers of the multiple-precision libraries.

This book consists of two parts. The first part presents the main concepts and the basic algorithms of writing a serial program with increased precision. The second part describes an algorithm for the constructing a parallel program using the GMP library.

Part I presents GMP and MPFR library. The example programs with GMP and MPFR are described there as well. At the beginning of Chapter 1 it deals with the necessity to improve the precision, describe real examples that show that the real problem won't reach the reasonable solution because of insufficient accuracy. The main target applications for GMP are cryptography applications and research, the Internet security applications, algebra systems, computational algebra research, etc. GMP is faster than any other bignum library. The advantage of GMP increases with the operand sizes for many operations, since GMP uses asymptotically faster algorithms. Chapter 2 describes methods of work with the GMP library. The considered questions are: the main types of GMP, memory usage, the main function of library, macheps, the way of writing the first program with GMP and so on. All functions have examples. Gauss's method is one of the most effective methods of solving the linear systems with nonsingular matrices of general form. In Chapter 3 the example program for solution of linear equations by Gauss method and its transformation to GMP is fully represented. Then the results obtained with increasing accuracy are analyzed. Chapter 4 describes basic information about MPFR library and the program mentioned above but with using MPFR. Then the comparative results of solution of linear equation, using GMP and MPFR are represented. Since the GMP library provides tools separately for C++, in Chapter 5 the possibility of using GMP in C++ is discussed. The example programs under discussion for matrix multiplication and solving of linear equations by Gauss's method for C++ using `mpf_class` of GMP library.

GMP library provides ability to provide calculations of arbitrary precision in the user programs. Part II presents parallel programming using MPI with GNU Multiple Precision. We demonstrate it in two ways: using MPI_BYTE in communications and using MPIGMP library. Also represented parallel algorithms and programs for solving the system of linear equations by Gauss's method and singular value decomposition with `mpf_t` GMP's type. In Chapter 6 the basics of MPI programming are represented, including the example of the program. Here is demonstrated the example program of matrix multiplication with GMP using MPI_BYTE in communications. For large program it is easy to use MPIGMP library for integration `mpf_t` type of GMP Library into parallel MPI-programs. MPIGMP supports using `mpf_t` variables of GMP and `mpfr_t` variables of MPFR. In Chapter 7 we represent the simple example for understanding the process of this integration. All main functions with examples for working with MPIGMP are

represented here as well. Chapter 8 includes two main parallel algorithms and programs for solving the system of linear equations: Gauss's method and singular value decomposition. These programs were tested on supercomputer and the results of analysis of integration GMP in parallel program were received.

Since the subject is still in a dynamic stage of development, it is important to keep pace with modern literature. In reference to each chapter we give the list of literature, which can help our readers in studying.

List of Contributors

PhD Chistyakova T.V.
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova, 40, Kiev, Ukraine
E-mail: dept150@insyg.kiev.ua

Dr. Khimich A.N.
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova, 40, Kiev, Ukraine
E-mail: khimich_ic@mail.ru

PhD Nikolaevskaya E.A.
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova, 40, Kiev, Ukraine
E-mail: elena_nea@ukr.net

Polyanko V.V.
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova, 40, Kiev, Ukraine
E-mail: polyanko_victor@ukr.net

Tomonori Kouya
Faculty of Comprehensive Informatics at SIST
Kakegawa city, Shizuoka-ken, JAPAN
E-mail: tkouya@cs.sist.ac.jp

Contents

Chapter 1
Why Does the Precision Improve?

Elena A. Nikolaevskaya, Alexandr N. Khimich, and Tamara V. Chistyakova

Abstract. Systems of linear algebraic equations (SLAEs) can arise during the solving of many problems – scientific, engineering, economic etc., for example, in: data processing problems, discretization of linear differential problems by projection difference methods, the solving of linear problems by least squares method; calculation of electric circuits and complicated hydraulic systems. We discuss main definition and problems in solution of SLAEs. Described problems arise through insufficient accuracy. Here you will find examples, that shows why there is a need to improve the precision.

Elena A. Nikolaevskaya ··
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: elena_nea@ukr.net

Alexandr N. Khimich
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: khimich_ic@mail.ru

Tamara V. Chistyakova
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: dept150@insyg.kiev.ua

E. Nikolaevskaya et al.: Programming with Multiple Precision, SCI 397, pp. 3–12.
springerlink.com

Before you begin to learn the GNU MP library, you shall understand why you need this. Most numerical methods for solving systems of linear algebraic equations have one common property. It is actually a computed solution (pseudosolution) that shows the exact opposite in accordance with error analysis [22] for a perturbed problem. These perturbations are very small and are often comparable to the rounding errors of input data. If the input data set with an error (measurements, calculations, etc.), then they usually contain a much larger error than the rounding error. In this case, any attempt to improve the computer solution (pseudosolution) without additional information about the exact problem or input error would be vain.

The situation changes considerably if we consider a mathematical problem with exact input data. Now the criterion of good or bad condition in computer model of the problem depends on the mathematical properties of a computer model of the problem and the mathematical properties of the CPU (computer word length), and appears fundamental to achieve any desired accuracy of computer solutions.

The reliability of the results largely depends on the sensitivity of the tasks to small rounding errors, including the translation of numbers in the decimal system into the computer system of numeration. For example, for a system of linear algebraic equations

$$Ax = b\,, \tag{1.1}$$

where A and b are the following:

$$A = \begin{pmatrix} 0.13485315\,74394464 & 0.18789705\,88235294 & 0.19091176\,47058824 & 0.17792647\,05882353 \\ 0.18789705\,88235294 & 0.262 & 0.265 & 0.247 \\ 0.19091176\,47058824 & 0.265 & 0.281 & 0.266 \\ 0.17792647\,05882353 & 0.247 & 0.266 & 0.255 \end{pmatrix}$$

$b = (0.3516,\ \ 0.4887,\ \ 0.5105,\ \ 0.4818)$,

and the solution is:

$$x = \begin{bmatrix} 0.66621621621616067387980644904305 72\ldots e13 \\ -0.40168918918907235069521662982454 77\ldots e13 \\ -0.16655405405399700518940186397842 41\ldots e13 \\ 0.97972972973027970725749406963011 15\ldots e12 \end{bmatrix}$$

after entering into the computer system we have

$$A_1 x_1 = b_1\,, \tag{1.2}$$

with the matrix A_1 and the right side b_1, having the form:

$$A_1 = \begin{pmatrix} 0.134853157439446397 & 0.187897058823529389 & 0.190911764705882391 & 0.177926470588235297 \\ 0.187897058823529389 & 0.262000000000000010 & 0.265\,000000000000013 & 0.246999999999999999 \\ 0.190911764705882391 & 0.265000000000000013 & 0.281000000000000027 & 0.266000000000000014 \\ 0.177926470588235297 & 0.246999999999999997 & 0.266000000000000014 & 0.255000000000000004 \end{pmatrix}$$

$$b_1 = (0.3516,\ \ 0.4887,\ \ 0.5105,\ \ 0.4818),$$

The exact solution is:

$$x_1 = (3.547...e13,\ \ -2.138...e13,\ \ -8.867...e13,\ \ 5.216...e12).$$

Continuing to study computer systems at double precision by algorithms of methods Bunche and Gauss in the libraries Linpack [5], we obtain the solutions:

$$x_{\text{Banch}} = (2.810...e12,\ \ -1.694...e12,...-7.027...e11,\ \ 4.133...e11)$$
$$x_{\text{Gauss}} = (3.164...e12,\ \ -1.908...e12,...-7.911...e11,\ \ 4.653...e11)$$

which are very distant from both the computer model of problems solution, and from its mathematical solution.

The reasons for getting the incorrect solution are that the estimate of condition number of a matrix *cond(A)*=2.089436217259268e^{16} and hence the `double` precision was not enough to obtain reliable solutions.

In the solving of applied problems of linear algebraic systems with accurate initial data

$$\overline{A}\,\overline{x} = \overline{b}\ , \tag{1.3}$$

arise very rarely.

The most typical statement of the problem is

$$Ax=b \tag{1.4}$$

together with predetermination of appropriate errors in the initial data:

$$\left\|\overline{A} - A\right\| = \left\|\Delta A\right\| \le \varepsilon_A \left\|A\right\|, \quad \left\|\overline{b} - b\right\| = \left\|\Delta b\right\| \le \varepsilon_b \left\|b\right\|, \tag{1.5}$$

where A is a rectangular $m\times n$ matrix and b is a matrix of the right-hand side of the dimension $m\times q$ (m-dimensional matrix), consists in finding such solution x ($n\times q$) matrix, n-dimensional vector), when equation turns into identity. If A is a square and non-singular matrix (i.e. its determinant $|\ A\ | \equiv \det(A) \neq 0$), than solution of system (1.4) exists and is unique.

Non-consistent systems yield no classic solution, however in this case such a solution x can be found that minimizes the Euclidean norm $\|Ax–b\|$. Vector x is called a solution evaluated by least squares method or generalized solution of SLAE [22]. In the general case there exists an infinite set of such generalized solutions. A solution possessing the least norm $\|x\|$ is unique and is called a normal generalized solution (it is also define to as a pseudo-solution).

In so doing it is supposed that structure of matrix of the original problem (1.3) as well as of the perturbed problem (1.4), (1.5) does not change, i.e. if original matrix is symmetric then perturbed matrix remains symmetric, if the original matrix is a banded matrix then perturbed matrix is also a banded matrix.

While studying the mathematical properties of systems of linear algebraic equations with perturbed input data related to computer realization as an approximate model in (1.3), we find out that it is a computer model of the problem. We assume that the error in the original data ΔA, Δb in this case contains the error that occurs while writing the coefficients of the matrix in computer memory or its computing.

When solving SLAEs with approximately given initial data one should consider the entire class of system of equations (1.4), (1.5) possessing sufficiently broad set of normally accessible solutions. Therefore, after the solving of problem (1.4) it is necessary to estimate perturbation of the solution depending on perturbation of the initial data (1.5). Not always the proximity between elements of matrices A and $\overline{A}$

as well as b and $\overline{b}$ ensures the sufficient proximity between solutions. For example, under certain perturbation within the limits of accuracy of predetermination of matrix elements and/or right-hand side of non-consistent accurate system (1.3) the perturbed system (1.4) obtained in computer may turn out to be consistent and, vice versa, consistent SLAEs may turn into a non-consistent system.

Thus, the solution of linear algebraic system with approximately given initial data consists of: investigation of mathematical characteristics of problem (1.4), (1.5); determination of one of admissible solutions for this problem and estimating both inherited and computational errors in the obtained solution [13].

Matrix of full rank within the error of initial data, we assume a matrix that can not change the rank of ΔA change in its elements.

Matrix of full rank within the machine precision, we assume a matrix that can not change the rank when you change the elements within the machine precision.

A computer algorithm for studying the completeness of rank reduced to testing the two relations

$$1.0+\gamma \neq 1.0\,, \qquad \gamma = h^{-1}(A) \tag{1.6}$$

$$\varepsilon_A h(A) < 1\,. \tag{1.7}$$

The first condition (1.6), which is performed in arithmetic floating point means that the matrix has full rank within machine accuracy, and the second (1.7) - that it has full rank within the accuracy of setting initial data.

Under these conditions, solution of the machine problem is unique and stable. Such a machine problem should be considered as well-posed within the accuracy of initial data.

Otherwise, the matrix of the perturbed system may be a matrix without a full rank and, hence, machine model of the problem (1.4) (1.5) should be considered as ill-posed. A key factor in the study of computer model is a criterion for the correctness of the problem. In this regard, a useful fact is that the condition for the study of computer model of the problem (1.6) is the inverse of $h(A)$. Consequently, for large numbers of conditionality does not occur an overflow in order. A loss in order for $1.0/h(A)$ for large numbers of condition is not fatal: engine result assumed to be zero, which leads to the correct conclusion about the loss of rank of the machine problem.

In the analysis the properties of the computer model problems with matrices of not full rank in conditions of approximate initial data, the fundamental role is played by the definition of the rank of a matrix.

The rank of the matrix in the condition of approximate initial data (effective rank or δ-rank) is

$$rank\,(A,\delta) = \min_{\|A-B\|\le\delta} rank\,(B)\,. \tag{1.8}$$

This means that the δ-rank of the matrix is equal to the minimum rank among all matrices in the neighborhood $\|A-B\| \le \delta$.

From [8] that if $r(\delta)$ - the δ -rank of the matrix, than

$$\mu_1 \ge \ldots \ge \mu_{r(\delta)} > \delta \ge \mu_{r(\delta)+1} \ge \ldots \ge \mu_p\,,\ p = \min(m,n)\,.$$

The practical algorithm for finding δ-rank can be defined as follows: find the value of r, equal to the largest value of i, for which there is the inequality

$$\frac{\delta}{\mu_i} < 1, \ \mu_i \neq 0, \ i = 1,2..$$

Using the effective rank of the matrix it is always possible to find a number of stable projection that approximates solution or a projection.

To analyze the rank of the matrix of values within the machine precision value δ can be attributed to machine precision, for example, setting it equal $\text{macheps}\|B\|$.

The program for calculating machine epsilon is presented below:

Listing 1.1. Macheps calculation

```
#include <string>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{
     double eps,eps1,s,d;
     eps=1.0;
     eps1=2.0;
     while (eps1>1.0)
     {
            eps*=0.5;
            eps1=1.0+eps;
     }
            cout << "eps=" << eps;
            cout << endl;
}
```

Correctly posed problem of solving linear algebraic equations, depending on the sensitivity of the solution to the error in the initial data can be divided into good and bad conditioned.

We consider some simple examples of the effect of the error in the initial data on the accuracy of the solutions [13].

Thus, in systems

$$\begin{cases} 2x_1 - x_2 = 0, \\ -x_1 + 2x_2 = 3 \end{cases} \qquad \begin{cases} 2x_1 - x_2 = 0, \\ -x_1 + 2x_2 = 3{,}03 \end{cases}$$

free to members differ among themselves in the third decimal place. The solution of the first one of these is x_1=1, x_2=2, and the second one is x_1 = 1,01, x_2 = 2,02. Error in the third decimal digit of the right side of the latter solution also causes changes in the third decimal and does not substantially change the obtained solution.

On the other hand, we consider system (1.9) and (1.10). Free members of systems of equations are different in the fifth significant digit. Perturbation of a single element in the system (1.9) leads to the large changes in the solution, and the determinant of both of these Systems equals 1.

$$\begin{cases} 100x_1 + 500x_2 = 1700, \\ 15x_1 + 75{,}01x_2 = 255 \end{cases} \qquad \begin{cases} x_1 = 17, \\ x_2 = 0 \end{cases} \tag{1.9}$$

$$\begin{cases} 100x_1 + 500x_2 = 1700, \\ 15x_1 + 75{,}01x_2 = 255{,}03 \end{cases} \qquad \begin{cases} x_1 = 2, \\ x_2 = 3 \end{cases} \tag{1.10}$$

Next, consider the matrix (1.10).

$$A = \begin{pmatrix} 4 & 0 \\ 0 & 0 \end{pmatrix} \qquad A^{+} = \begin{pmatrix} 1/4 & 0 \\ 0 & 0 \end{pmatrix}. \tag{1.11}$$

$$A = \begin{pmatrix} 4 & 0 \\ 0 & 10^{-8} \end{pmatrix} \qquad A^{-1} = \begin{pmatrix} 1/4 & 0 \\ 0 & 10^{8} \end{pmatrix}. \tag{1.12}$$

It is singular and so its pseudoinverse matrix $A^{+} = \begin{pmatrix} 1/4 & 0 \\ 0 & 0 \end{pmatrix}$ exists. After changing one element we obtained the matrix A (1.12). This matrix is nonsingular, and hence there is an inverse matrix $A^{-1} = \begin{pmatrix} 1/4 & 0 \\ 0 & 10^{8} \end{pmatrix}$. And as we can see, small changes in initial data lead to a different solution for any right side of the system.

From the mentioned above it follows that solutions of application problems on your computer may be a situation in which the exact solution of a mathematical problem may not be the solution to the physical model, which is being considered. Implementation of the numerical method on a computer assumes some errors which must be considered while analyzing the machine solution of the problem.

Perturbation of the initial data may change the mathematical properties of the problem to be solved. Therefore, the problem with approximated input data should be viewed as a problem with apriori unknown properties, which must be examined first and then use the appropriate algorithm for solving the problem and assess the reliability of the results.

A similar result can be obtained as a result of errors that occur while entering data into computer problem. In this case the situation is greatly complicated by the fact that most of the mathematical differences between the matrices of full and partial rank exists only in a mathematically perfect world of the real numbers. Since operations on matrices are conducted to rounding, this difference is indefinite. Thus, a nonsingular matrix can become singular in the computer. On

the other hand, it is very likely that in fact the singular matrix due to rounding error will be converted to a close one, but nonsingular.

Thus, in solving correctly posed problems, along with obtaining a single classical solution there is a need to assess the hereditary error or similar of mathematical and physical solutions.

The error in the solution of (1.3), for example, with a square nonsingular matrix, caused by incorrect specification of initial data (1.5), estimated by the formula [13]:

$$\frac{\|x-\bar{x}\|}{\|x\|} \le \frac{\|A\|\|A^{-1}\|}{1-\|\Delta A\|\|A^{-1}\|}(\varepsilon_A + \varepsilon_b) \tag{1.13}$$

subject to $\|\Delta A\|\,\|A^{-1}\| < 1$.

From the formula (1.13) is obvious that the stability of solutions of the system to changes of the initial data largely depends on the value of which is called ***condition number of the matrix*** $\operatorname{cond} A$***:***

$$\operatorname{cond} A = \|A\|\,\|A^{-1}\|. \tag{1.14}$$

If $\operatorname{cond} A$ is small, the matrix linear algebraic equations is called well-conditioned. If $\operatorname{cond} A$ is large, the matrix of this system is called ill-conditioned. Depending on how to introduce the norms of the matrix considering several types of condition numbers of matrices. For example, for symmetric matrices

$$\operatorname{cond} A \equiv \frac{\max\limits_i |\lambda_i|}{\min\limits_i |\lambda_i|},$$

where λ_i, $i = 1,2, \ldots, n$, – eigenvalues of A, for asymmetric matrices

$$\operatorname{cond} A \equiv \sqrt{\frac{\mu_n}{\mu_1}},$$

where μ_n and μ_1 - the largest and smallest eigenvalues of $A^T A$. Some other condition numbers are considered in [8].

The reliability of the mathematical solution is determined not only of a matrix system of linear algebraic equations, but also the accuracy of initial data. As the criterion of conditioning can be considered the value

$$m = \operatorname{cond} A \left[\frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|} \right]$$

It is connects the properties of the matrix system and the error in task initial data. In real problems, it makes sense to consider those systems which estimate the number m uniquely less 1, for example $m \le 0{,}01$. To reduce hereditary error of mathematical solutions needs to reduce the number m or by increasing the accuracy of initial data, or by reformulating the mathematical model in order to reduce the number of a matrix.

For example, typing error in the sixth decimal place of the free term of the first system of (1.10), namely considering the system

$$\begin{cases} 100x_1 + 500x_2 = 1700, \\ 15x_1 + 75{,}01x_1 = 255{,}000003 \end{cases}$$

we get $x_1 = 16{,}9985$, $x_2 = 0{,}0003$, there is sufficiently good approximation to the solution $x_1 = 17$, $x_2 = 0$ unperturbed first system (1.10). It is known that in some cases, for solving scientific and engineering problems on computers, users get machine solutions that have no physical meaning. This, as we have shown is due to error in the initial data, the difference between the properties of discrete and computer models of tasks, as well as the difference between arithmetic and machine arithmetic, etc.

Specificity of a computer model of the problem consists in the fact consist of:

- study the properties of the computer model, by its very nature, should be performed by computer algorithms;
- properties of the computer model of the problem may differ from those of a mathematical problem;
- if the source data of the mathematical problem defined exactly, the criterion of good or bad condition computer model of the problem depends on the mathematical properties of the computer model and arithmetic properties of the computer and there is in principle possible to achieve any desired precision machinery solutions.

The concepts of good and bad due to the matrix closely associated with computing power that particular computer and the length of the mantissa of the computer word. The result is the same system that can qualify for a single length mantissa of machine words like "machine is not conditioned» or almost singular, while for the other – «machines are well conditioned. "

For prediction the length of the mantissa (computer word), which provides the specified accuracy for solutions (collaborative systems) it is possible to use the following empirical rule: the number of correct decimal digits in computing solutions is equal

$$\mu - \alpha , \tag{1.15}$$

where μ - decimal order mantissa floating-point ε, α - decimal order condition number. Such a way, knowing the conditionality of the matrix system and the accuracy of calculations the computer can determine the necessary capacity to produce reliable solutions.

Analysis of the features of realization of computer arithmetic shows that:

- continuum of all real numbers in a computer can be approximated by a finite set of finite convergents (even when you enter the numerical data to rounding errors);
- phenomenon of "machine zero" gives rise to a number of difficulties in the implementation of computational algorithms (any modern computer is the smallest positive number that can be represented in it, and all the numbers smaller in absolute value of this number are replaced by zero);

- arithmetic operations on a computer different from the mathematical: the law of associativity and distributivity does not run on any modern computer, and the laws of commutativity in floating point operations are performed only when the correct procedure to rounding.

Thus, one means of obtaining reliable solutions for ill-conditioned problems is to increase the accuracy of number representation and perform computer operations.

To improve the accuracy of calculations you can use the GMP library [7], a large set of functions which allows you to organize the computational process with different digit capacity.

References

For many readers numerical analysis is studied as an important applied subject. Since the subject is still in a dynamic stage of development, it is important to keep track of recent literature. Therefore we give in reference a more complete overview of the literature than is usually given in textbooks. The presented selection is by no means complete and reflects a subjective choice, but we hope it can serve as a guide for a reader who out of interest wishes to deepen his knowledge. Reviews of most books of interest can be found in Mathematical Reviews as well as in SIAM Review and Mathematics of Computation.

The book [6] is a carefully written introductory text with a wealth of computer exercises and much valuable information in notes after each chapter. More elementary but useful books are [2-4, 8-10, 12, 14-20]. In [21] gives an excellent modern introduction to applied mathematics. In [1, 11] described least squares problem and numerical methods for its solution.

1. Bjorck, A.: Numerical Methods for Least Squares Problems. SIAM, Philadelphia (1996)
2. Blum, E.K.: Numerical Analysis and Computation: Theory and Practice. Addison-Wesley, Reading (1972)
3. Demmel, J.W.: Applied Numerical Linear Algebra. SIAM, Philadelphia (1997)
4. Dongarra, J.J., Duff, I.S., Sorensen, D.C., Van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia (1998)
5. Dongarra, J., Bunch, J.R., Moler, C.B., Stewart, G.W.: LINPACK Users'Guide. SIAM, Philadelphia (1979)
6. Gautschi, W.: Numerical Analysis, an Introduction. Birkhauser, Boston (1997)
7. GMP Library, `http://www.gmplib.org`
8. Golub, G.H., Van Loan, C.F.: Matrix Computations, 2nd edn. Johns Hopkins University Press, Baltimore (1989)
9. Higham, N.J.: Accuracy and Stability of Numerical Algorithms. SIAM, Philadelphia (2002)
10. Horn, R.A., Johnson, C.R.: Matrix Analysis. Cambridge University Press, Cambridge (1985)

11. Lawson, C.L., Hanson, R.J.: Solving Least Squares Problems. SIAM, Philadelphia (1995)
12. Meyer, C.D.: Matrix Analysis and Applied Linear Algebra. SIAM, Philadelphia (2000)
13. Molchanov, I.N.: Machine methods for solving applied problems. Algebra, approximation of functions, ordinary differential equations. Naukova dumka, Kiev (2007)
14. Nash, J.C.: Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation. American Institute of Physics, New York (1990)
15. Noble, B., Daniel, J.W.: Applied Linear Algebra. Prentice-Hall, Englewood Cliffs (1977)
16. Ortega, J.: Numerical Analysis: A Second Course. Academic Press, New York (1972)
17. Ortega, J.M.: Matrix Theory. A Second Course. Plenum Publ., New York (1987)
18. Sacco, Q.R., Saleri, F.: Numerical Mathematics. Springer, New York (2000)
19. Stewart, G.W.: Introduction to Matrix Computations. Academic Press, New York (1973)
20. Stewart, G.W., Sun, J.-G.: Matrix Perturbation Theory. Academic Press, Boston (1990)
21. Strang, G.: Introduction to Applied Mathematics. Wellesley-Cambridge Press, Wellesley (1988)
22. Wilkinson, J.H., Reinsch, C.: Handbook for automatic computation. Linear Algebra. Springer, Berlin (1971)

Chapter 2
About GMP

Elena A. Nikolaevskaya, Alexandr N. Khimich, and Tamara V. Chistyakova

Abstract. We present algorithm for using GMP library in your program. The Main function of GMP for floating point numbers are described here. After learning this Chapter you can start to write your program with multiple precision. Most of described functions are accompanied by examples.

2.1 Introduction

GMP (*GNU MP - GNU multiple precision arithmetic library*) is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers [2]. There is no practical limit to the precision except the

Elena A. Nikolaevskaya
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: elena_nea@ukr.net

Alexandr N. Khimich
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: khimich_ic@mail.ru

Tamara V. Chistyakova
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: dept150@insyg.kiev.ua

E. Nikolaevskaya et al.: Programming with Multiple Precision, SCI 397, pp. 13–30.
springerlink.com

ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface. Library GNU MP was developed for fast work, for both large and small operands. It is intended to provide maximum possible arithmetic for all applications requiring high accuracy then directly supports the basic types of C language. GMP library is fast because it uses integer words as the base type that uses fast algorithms, has optimized assembly code for many types of processors and combines speed with ease and elegance of the execution of operations.

Many applications use just a few hundred bits of precision; but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum.

For updated information on GMP Library and installation, please see the GMP web pages at http://www.gmplib.org/.

2.2 GMP Basics

To perform calculations with the desired accuracy, C or C++ programs are necessary insert the circulation in the respective functions of the GMP library for converting data types and arithmetic operations on them, as well as to connect the file `gmp.h`, which describes the prototypes of these functions.

```
#include <stdio.h>
#include <gmp.h>
```

All programs using GMP must be linked against the '`libgmp`' library. On a typical Unix-like system it can be done with '`-lgmp`', for example

```
gcc myprogram.c -o myprogram -lgmp
```

GMP C++ functions are in a separate '`libgmpxx`' library:

```
gcc++ myprogram.c -o myprogram -lgmp -lgmpxx
```

GMP supports integer, rational numbers, as well as floating point numbers (see Table 2.1.). Types of GMP are applied in the form of single-element arrays of certain structures.

Table 2.1 Types of variables supported by GMP

C	C++
Integer	
`mpz_t x, y[10];`	`mpz_class x, y[10];`

Table 2.1 *(continued)*

C	C++
Rational number	
`mpq_t p, q[8];`	`mpq_class p, q[8];`
Floating point number	
`mpf_t f[12], g;`	`mpf_class f[12], g;`

Integer, defined by the GMP library, means a multiple precision integer. The C data type for such integers is `mpz_t`, and for C++ is `mpz_class`. For example:

```
mpz_t x, y;
struct funk { mpz_t x, y; };
mpz_t vec[5];
```

Rational number, defined by the GMP library, means a multiple precision fraction. The C data type for these fractions is `mpq_t`, and for C++ is `mpq_class`. For example:

```
mpq_t x;
```

Floating- point number or `float` for short is an arbitrary precision mantissa with a limited precision exponent. The C data type for such objects is `mpf_t`, for C++ - `mpf_class`. For example:

```
mpf_t y;
```

We will consider only floating point numbers. A floating point number system (see, for example, [1, 4]) is a subset of the real numbers whose elements have the form

$$y = \pm m \times b^{e-t}.$$

The system F is characterized by four integer parameters:

- the base b (sometimes called the radix),
- the *precision t*, and
- the *exponent range*.

The *mantissa m* is an integer satisfying $0 < m < b^t - 1$.

A floating-point number can have three special values: *Not-a-Number* (`NaN`) or plus or minus Infinity. `NaN` represents an uninitialized object, the result of an invalid operation (like 0 divided by 0), or a value that cannot be determined (like +Infinity minus +Infinity).

The *precision* is the number of bits used to represent the significance of a floating-point number.

Type structure `mpf_t` in GPM is shown in Figure 1. `mpf_t` datatype have parts of extened *sign*, *exponent,* and *mantissa*. The user-defined precision fixes the length of mantissa that the pointers `_mp_d` point to.

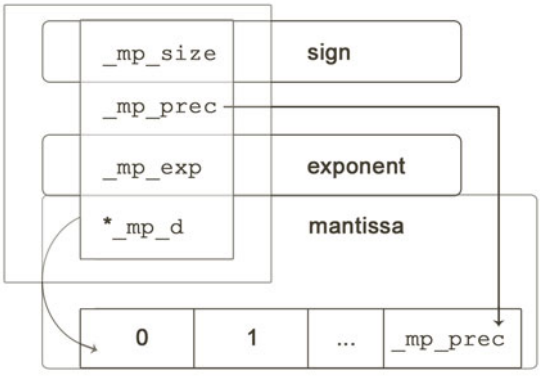

Fig. 2.1 Type structure `mpf_t` in GMP

The floating-point functions accept and return exponents in the C type `mp_exp_t`. Currently this is usually a `long`, but on some systems it is an `int` for efficiency.

A `limb` means the part of a multi-precision number that fits in a single machine word. Normally a limb is 32 or 64 bits. The C data type for a limb is `mp_limb_t`.

Counts of limbs are represented in the C type `mp_size_t`. Currently this is normally a `long`, but on some systems it is an `int` for efficiency.

In general `unsigned long` is used for bit counts and ranges, and `size_t` is used for byte or character counts.

The GMP `mpf_t` type uses a fixed amount of memory that is given at the beginning of the program, which is determined accurately. `mpf_t` variables use a fixed amount of space, determined by the chosen precision and allocated at initialization, so their size does not change. All memory is allocated using malloc and friends by default, but this can be changed.

2.2 Fast Start

Now, start writing programs. Consider a program of adding two variables. In the beginning of the program a library `gmp.h` and `gmpxx.h` would be required for C and C + +, respectively (Table 2.2).

Table 2.2 Organization of GMP-program

C	C++
`#include <stdio.h>`	`#include <stdio.h>`
`#include <stdarg.h>`	`#include <stdarg.h>`
`#include <gmp.h>`	**`#include <gmpxx.h>`**

Table 2.2 *(continued)*

C	C++
`int main( )`	`int main( )`
`{`	`{`
`/* use mp[zqf]_t types and functions here */`	`/* use mp[zqf]_class and functions here */`
`}`	`}`

For better understanding, consider a simple example of adding two variables for the numbers of type `double`.

```
#include <stdio.h>
int main()
{
int i;
double a,b,c;
a=0; b=1;
for (i = 2; i<=10; ++i)
     {
     c=a+b;
     a=b;
     b=c;
     }
printf("b=%4.3e\n", b);
return 0;
}
```

Transform our program using the variable type floating point from GMP library, i.e., `mpf_t` (Listing 2.1).

In the beginning of the program connect the file `gmp.h` and give the required precision (the length of the mantissa):

```
const int precision = 256;
```

declare three floating-point variables *a, b, c:*

```
mpf_t a, b,c;
```

and then, initialize the variables *a, b, c* by using the `mpf_init2()` with already defined 256-bit precision:

```
mpf_init2(a,precision);
```

```
mpf_init2(b,precision);
mpf_init2(c,precision);
```

sets the value of *a* to the double-precision number 0.0 and *b* to the double-precision number 1.0:

```
mpf_set_d(a, 0.0);
mpf_set_d(b, 1.0);
```

prints the value of *b* in base 10, where the third argument 0 means that the number of printed digits is automatically chosen from the precision of *b*:

```
mpf_out_str(stdout,10,0,b);
```

Listing 2.1.

```
#include <stdio.h>
#include <gmp.h>
int main()
{
int i;
mpf_t a, b,c;

// length of the mantissa 256
const int precision = 256;

//Initialization with defined precision 256
mpf_init2(a,precision);
mpf_init2(b,precision);
mpf_init2(c,precision);

mpf_set_d(a, 0.0);
mpf_set_d(b, 1.0);
for (i = 2; i<=10; ++i)
     {
            mpf_add(c, a, b);
            mpf_set(a, b);
            mpf_set(b, c);
            //Clear memory
            mpf_clear(c);
     }
printf("b= ",b);
mpf_out_str(stdout,10,0,b);
printf("\n");
return 0;
```

```
}
```

Note that two lines of print

```
printf("b= ",b);
mpf_out_str (stdout, 10, 0, b);
```

for comfort can be converted to a single row

```
printf("b= ",mpf_out_str (stdout, 10, 0, b));
```

In Chapter 3, we consider C ++ programs with `mpf_class`.

2.3 Main Functions of GMP

In this chapter we review the main functions of GMP library (for details please see [2]).

There are six classes of functions in the GMP library:

1. Functions for signed integer arithmetic, with names beginning with `mpz_`. The associated type is `mpz_t`.
2. Functions for rational number arithmetic, with names beginning with `mpq_`. The associated type is `mpq_t`.
3. Functions for floating-point arithmetic, with names beginning with `mpf_`. The associated type is `mpf_t`
4. Functions compatible with Berkeley MP, such as `itom`, `madd`, and `mult`. The associated type is `MINT`.
5. Fast low-level functions that operate on natural numbers. These are used by the functions in the preceding groups, and you can also call them directly from very time-critical user programs. These functions' names begin with `mpn_`. The associated type is array of `mp_limb_t`.
6. Miscellaneous functions. Functions for setting up custom allocation and functions for generating random numbers.

Each type of variable has its own set of functions. We concentrate on the functions for floating-point numbers.

GMP floating-point numbers are stored in objects of type `mpf_t` and functions operating on them have an `mpf_` prefix. The mantissa of each float has a user-selectable precision, limited only by available memory. Each variable has its own precision, and that can be increased or decreased at any time. The exponent of each `float` is a fixed precision, one machine word on most systems. In the current implementation the exponent is a count of limbs, so for example on a 32-bit system this means a range of roughly $2^{-68719476768}$ to $2^{68719476736}$, or on a 64-bit system this will be greater. Note however `mpf_get_str` can only return an

exponent which fits an `mp_exp_t` and currently `mpf_set_str` does not accept exponents bigger than a `long`.

Each variable keeps a size for the mantissa data actually in use. This means that if a `float` is exactly represented in only a few bits then only those bits will be used in a calculation, even if the selected precision is high. All calculations are performed to the precision of the destination variable. Each function is defined to calculate with “infinite precision” followed by a truncation to the destination precision, but of course the work done is only what is needed to determine a result under that definition.

The precision selected for a variable is a minimum value; GMP may increase it a little to facilitate efficient calculation. Currently this means rounding up to a whole limb, and then sometimes having a further partial limb, depending on the high limb of the mantissa. But applications shouldn’t be concerned by such details.

The mantissa in stored in binary, as might be imagined from the fact precisions are expressed in bits.

In particular results obtained on one computer often differ from the results on a computer with a different word size.

2.3.1 Init and Assign Functions

Before you assign a variable value of GMP, you must initialize it by calling one of the special initialization functions. When you're done with a variable, you need to clean it by using one of the options for this purpose. Variable should be initialized only once, or at least cleared from each initialization. For example,

```
void funk (void)
{
mpf_t x;
int i;
mpf_init (x);
for (i = 1; i < 250; i++)
{
mpf_mul (x, ...);
...
}
mpf_clear (x);
}
```

Consider the initialization function in the GMP.

```
void mpf_set_default_prec(unsigned long int PREC)
```

Set the default precision to be at least PREC bits. All subsequent calls to mpf_init() will use this precision, but previously initialized variables are unaffected.

mp_bitcnt_t mpf_get_default_prec (void)
Return the default precision actually used.

An mpf_t object must be initialized before storing the first value in it. The functions mpf_init and mpf_init2 are used for that purpose.

void mpf_init(mpf_t X)
Initialize X to 0. Normally, a variable should be initialized only once or at least be cleared, using mpf_clear, between initializations. The precision of X is undefined unless a default precision has already been established by a call to mpf_set_default_prec.

void mpf_init2(mpf_t X, unsigned long int PREC)
Initialize X to 0 and set its precision to be at least PREC bits. Normally, a variable should be initialized only once or at least be cleared, using mpf_clear, between initializations.

void mpf_inits(mpf t x, ...)
Initialize a NULL-terminated list of mpf_t variables, and set their values to 0. The precision of the initialized variables is undefined unless a default precision has already been established by a call to mpf_set_default_prec.

void mpf_clear(mpf_t X)
Free the space occupied by X. Make sure to call this function for all mpf_t variables when you are done with them.

void mpf_clears(mpf t x, ...)
Free the space occupied by a NULL-terminated list of mpf_t variables.
Below is an example of initialization and clearing.

```
int main
{
mpf_t x, y;

/* use the default precision */
mpf_init(x);

/* precision at least 128 bits */
mpf_init2(y,128);
...
```

```
/* Clear memory */
mpf_clear (x);
mpf_clear (y);
}
```

The following three functions are useful for changing the precision during a calculation. A typical use would be for adjusting the precision gradually in iterative algorithms such as Newton-Raphson, making the computation precision closely match the actual accurate part of the numbers.

mp_bitcnt_t mpf_get_prec(mpf_t X)
Return the current precision of X, in bits.

void mpf_set_prec(mpf_t X, unsigned long int PREC)
Set the precision of X to be at least PREC bits. The value in X will be truncated to the new precision. This function requires a call to realloc, and so should not be used in a tight loop.

void mpf_set_prec_raw(mpf x rop,unsigned long int PREC)
Set the precision of X to be at least PREC bits, without changing the memory allocated. PREC must be no more than the allocated precision for X, that being the precision when X was initialized, or in the most recent mpf_set_prec.

The value in X is unchanged, and in particular if it had a higher precision than PREC it will retain that higher precision. New values written to X will use the new PREC.

Before calling mpf_clear or the full mpf_set_prec, another mpf_set_prec_raw call must be made to restore X to its original allocated precision. Failing to do so will have unpredictable results.

mpf_get_prec can be used before mpf_set_prec_raw to get the original allocated precision. After mpf_set_prec_raw it reflects the PREC value set.

mpf_set_prec_raw is an efficient way to use an mpf_t variable at different precisions during a calculation, perhaps to gradually increase precision in an iteration, or just to use various different precisions for different purposes during a calculation.

Consider the functions assign:

```
void mpf_set (mpf t x, mpf t y)
void mpf_set_ui (mpf t x, unsigned long int y)
void mpf_set_si (mpf t x, signed long int y)
void mpf_set_d (mpf t x, double y)
void mpf_set_z (mpf t x, mpz t y)
void mpf_set_q (mpf t x, mpq t y)
```

Assign the value x to y. For example, a variable of type mpf_t assigned a double value equal to 1:

```
mpf_set_d (x, 1.0);
```

int mpf_set_str (mpf t x, char *str, int base)

Set the value of x from the string in str. The argument base may be in the ranges from 2 to 62, or from −62 to −2. Negative values are used to specify that the exponent is in decimal.

For example,

```
mpf_set_str (A[4][4], "0.255", 10);
```

void mpf_swap (mpf_t x, mpf_t y)

Swap x and y efficiently. Both the values and the precisions of the two variables are swapped.

For convenience, GMP provides a parallel series of initialize-and-set functions which initialize the output and then store the value there. These functions' names have the form mpf_init_set...

Once the float has been initialized by any of the mpf_init_set... functions, it can be used as the source or destination operand for the ordinary float functions. Do not use an initialize-and-set function on a variable already initialized!

void mpf_init_set(mpf_t *x*, mpf_t *y*)
void mpf_init_set_ui(mpf_t *x*,unsigned long int *y*)
void mpf_init_set_si(mpf_t *x*,signed long int *y*)
void mpf_init_set_d (mpf_t *x*, double *y*)

Initialize x and set its value from y.

The precision of x will be taken from the active default precision, as set by mpf_set_default_prec.

For example, we have code

```
mpf_t x, y;
mpf_init (x);
mpf_init (y);
mpf_set_d (y, 1.0);
mpf_set_d (x, 1.0);
```

If we use mpf_init_set_d() we have next code

```
mpf_t x, y;
mpf_init (y);
mpf_set_d (y, 1.0);
```

```
mpf_init_set_d (x, y);
```

int mpf_init_set_str(mpf_t x, char *str, int base)

Initialize x and set its value from the string in str. See mpf_set_str above for details on the assignment operation.

Note that x is initialized even if an error occurs. (I.e., you have to call mpf_clear for it.)

The precision of x will be taken from the active default precision, as set by mpf_set_default_prec.

2.3.2 Conversion Functions

double mpf_get_d(mpf_t x)

Convert x to a double, truncating if necessary (i.e. rounding toward zero). If the exponent in x is too big or too small to fit a double then the result is system dependent. For too big an infinity is returned when available.

```
mpf_t x;
double y;
mpf_init (x);
mpf_get_d (x);
x=y;
```

double mpf_get_d_2exp(signed long int exp,mpf_t x)

Convert x to a double, truncating if necessary (i.e. rounding toward zero), and with an exponent returned separately.

The return value is in the range $0{,}5 \le |d| < 1$ and the exponent is stored to **exp*. $d*2^{exp}$ is the (truncated) op value. If op is zero, the return is 0.0 and 0 is stored to **exp*. This is similar to the standard C frexp function.

long mpf_get_si(mpf_t x)
unsigned long mpf_get_ui(mpf_t x)

Convert x to a long or unsigned long, truncating any fraction part. If x is too big for the return type, the result is undefined.

char * mpf_get_str(char *str, mp_exp_t *expptr, int base, size_t n_digits, mpf_t x)

Convert x to a string of digits in base base. The base argument may vary from 2 to 62 or from −2 to −36. Up to n_digits digits will be generated. Trailing zeros are not returned. No more digits than can be accurately represented by x are

ever generated. If n_digits is 0 then that accurate maximum number of digits are generated.

For base in the range 2..36, digits and lower-case letters are used; for –2..–36 digits upper-case letters are used; for 37..62 digits, upper-case letters, and lower-case letters (in that significance order) are used.

If str is NULL, the result string is allocated using the current allocation function. The block will be strlen(str)+1 bytes that being exactly enough for the string and null-terminator.

If str is not NULL, it should point to a block of n_digits + 2 bytes that being enough for the mantissa, a possible minus sign, and a null-terminator. When n_digits is 0 to get all significant digits, an application won't be able to know the space required, and str should be NULL in that case.

The generated string is a fraction, with an implicit radix point immediately to the left of the first digit. The applicable exponent is written through the expptr pointer. For example, the number 3.1416 would be written as string "31416" and exponent 1.

When x is zero, an empty string is produced and the exponent returned is 0.

A pointer to the result string is returned, being either the allocated block or the given str.

2.3.3 Arithmetic Functions

As is well known for working with variables of higher precision in C, all arithmetic operations should be carried out using special functions GMP. We consider them in detail.

Addition operation:
void mpf_add (mpf_t z, mpf_t x, mpf_t y)
void mpf_add_ui(mpf_t z,mpf_t x,unsigned long int y)
Set z to x + y. For example,

```
mpf_t x, y, z;
mpf_init (x);
mpf_init (y);

mpf_set_d (x, 1.0);
mpf_set_d (y, 2.0);

/*   z=x+y          */
mpf_add (mpf_t z, mpf_t x, mpf_t y)
…
mpf_clear (x);
```

```
mpf_clear (y);
mpf_clear (z);
```

Subtraction operation:
void mpf_sub(mpf_t z, mpf_t x, mpf_t y)
void mpf_ui_sub(mpf_t z,unsigned long int x, mpf_t y)
void mpf_sub_ui(mpf_t z,mpf_t x,unsigned long int y)
Set `z` to `x-y`. For example,

```
/*   z=x-y          */
mpf_sub (mpf_t z, mpf_t x, mpf_t y)
```

Multiplication operation:
void mpf_mul(mpf_t z, mpf_t x, mpf_t y)
void mpf_mul_ui(mpf_t z, mpf_t x,unsigned long int y)
Set `z` to `x*y`. For example,

```
/*   z=x*y          */
mpf_mul (mpf_t z, mpf_t x, mpf_t y)
```

Division operation:
Division is undefined if the divisor is zero, and passing a zero divisor to the divide functions will make these functions intentionally divide by zero. This lets the user handle arithmetic exceptions in these functions in the same manner as other arithmetic exceptions.

void mpf_div (mpf_t z, mpf_t x, mpf_t y)
void mpf_ui_div(mpf_t z,unsigned long int x,mpf_t y)
void mpf_div_ui(mpf_t z,mpf_t x,unsigned long int y)
Set `z` to `x/y`.
For example, to compute the functions f of type `double`,

```
f=((d[l]-d[k])*(d[l]+d[k])+e[k]*(d[m]/h-e[k]))/d[l]
```

which has one line in C, to work with `mpf_t` need to perform all arithmetic operations individually using the functions described above.

```
mpf_sub( tmp, d[l], d[k] );
mpf_add( tmp2, d[l], d[k] );
mpf_mul( f, tmp, tmp2 );
mpf_sub( tmp, h, e[k] );
mpf_div( tmp, d[m], tmp );
mpf_mul( tmp, e[k], tmp );
mpf_add( f, f, tmp );
mpf_div( f, f, d[l] );
```

The operation of finding the square root:
void mpf_sqrt (mpf_t x, mpf_t y)
void mpf_sqrt_ui (mpf_t x, unsigned long int y)
Set x to the square root of y.

The operation of raising to a power:
void mpf_pow_ui(mpf_t z,mpf_t x,unsigned long int y)
Set z to x raised to the power y.

An assignment of negative values:
void mpf_neg (mpf_t x, mpf_t y)
Set x to −y.

An assignment of the absolute values:
void mpf_abs (mpf_t x, mpf_t y)
Set x to the absolute value of y.

void mpf_mul_2exp(mpf_t z,mpf_t x,unsigned long int y)
Set z to x times 2 raised to y.

void mpf_div_2exp(mpf_t z,mpf_t x,unsigned long int y)
Set z to x divided by 2 raised to y.

2.3.4 Comparison Functions

int mpf_cmp (mpf_t x, mpf_t y)
int mpf_cmp_d (mpf_t x, double y)
int mpf_cmp_ui (mpf_t x, unsigned long int y)
int mpf_cmp_si (mpf_t x, signed long int y)
Compare x and y. Return a positive value if x>y, zero if x=y, and a negative value if x<y.

mpf_cmp_d can be called with an infinity, but results are undefined for a NaN.

For example, consider an example in which the first assigned to the variable y, and then compares the two variables d[k] and y and assigned a negative value for d[k].

```
mpf_set_d(y, 0.0);
     if(mpf_cmp( d[k], y) < 0 )
{
            mpf_neg( d[k], d[k] );
```

```
}
```

int mpf_eq(mpf_t x,mpf_t y,unsigned long int z)

Return non-zero if the first `z` bits of `x` and `y` are equal, zero otherwise. I.e., test if `x` and `y` are approximately equal.

void mpf_reldiff (mpf_t z, mpf_t x, mpf_t y)

Compute the relative difference between `x` and `y` and store the result in `z`. This is $|x-y|/x$.

int mpf_sgn (mpf_t x)

Return +1 if `x>0`, 0 if `x=0`, and -1 if `x<0`.

2.3.5 I/O Functions

Functions that perform input from a stdio stream, and functions that output to a stdio stream. Passing a `NULL` pointer for a stream argument to any of these functions will make them read from stdin and write to stdout, respectively.

When using any of these functions, it is a good idea to include `stdio.h` before `gmp.h`, since that will allow `gmp.h` to define prototypes for these functions.

size_t mpf_out_str (FILE *stream, int base, size_t n_digits, mpf_t x);

Print `x` to `stream`, as a string of digits. Return the number of bytes written, or if an error occurred, return 0.

The mantissa is prefixed with an `0.' and is in the given base, which may vary from 2 to 62 or from −2 to −36. The exponent is always in decimal.

Up to `n_digits` will be printed from the mantissa, except that no more digits than are accurately representable by op will be printed. `n_digits` can be 0 to select that accurate maximum.

size_t mpf_inp_str (mpf_t x, FILE *stream, int base);

Read a string in `base` from `stream`, and put the read `float` in `x`. The argument `base` may be in the ranges from 2 to 36, or from −36 to −2. Negative values are used to specify that the exponent is in decimal. Return the number of bytes read, or if an error occurred, return 0.

GMP library has many another useful function: *Integer function*, *Rational Number function*, *Low-Level function*, *Formatted Output*, *Formatted Input,* and others.

You can know them in more detail on the website http://gmplib.org/.

2.4 Macheps

Rounding is a procedure for choosing the representation of a real number in a floating point number system. For a number system and a rounding procedure, machine epsilon is the maximum relative error of the chosen rounding procedure.

As is known, machine epsilon is the distance between the unit and the nearest machine number on the right:

$$macheps = 2^{1-t},$$

it is the smallest of the machine numbers for which (in the sense of machine arithmetic):

$$1 + macheps > 1.$$

Machine epsilon is determined by the size of the mantissa. For the simple accuracy its value is about 1.19 e-07, and for double it is - 1.11e-16.

The following values of machine epsilon (Table 3) are encountered in practice. Machine epsilon depends on both the kind of floating-point numbers and the kind of rounding. There are several possible rounding modes for use in floating point arithmetic.

Table 2.3 Macheps for different datatype

IEEE 754 – 2008	Common name	C++ data type	Machine epsilon	Value
binary32	single precision	`Float`	2^{-24}	5.96e-08
binary64	double precision	`Double`	2^{-53}	1.11e-16
binary128	quad(ruple) precision	`long double`	2^{-113}	9.63e-35

Knowing the condition number of matrix [5]

$$cond(A) = p * 10^{m}$$

and machine epsilon

$$macheps = k * 10^{-l}$$

you can see the number of correct digits according to the formula:

$$k = l - m. \tag{2.1}$$

Transform the program from Listing 1.1 for the type of `mpf_t` (Listing 2.2).

Listing 2.2. Calculation of machine epsilon with GMP

```
#include <stdio.h>
#include <gmp.h>

main()
{
const int precision = 64;
mpf_t eps, eps1, tmp, tmp1, tmp2;
```

```
mpf_init2(eps, precision);
mpf_init2(eps1, precision);
mpf_init( tmp );
mpf_init( tmp1 );
mpf_init( tmp2 );

mpf_set_d (eps, 1.0);
mpf_set_d (eps1, 2.0);
mpf_set_d( tmp1, 0.5 );
mpf_set_d( tmp2, 1.0 );

while( mpf_cmp( eps1, tmp2 ) > 0 )
            {
            mpf_mul( eps, eps, tmp1 );
            mpf_add( eps1, eps, tmp2 );
            }
            printf( "MEPS= %25.16e",mpf_get_d( eps ) );
            printf("\n");
}
```

Table 2.4 shows the value of machine epsilon, which characterizes the precision of floating-point arithmetic on Inparcom-256 [3] with different digit capacity.

Table 2.4 Macheps with different precision on Inparcom-256

Realization	Length of mantissa	Machine epsilon (macheps)
C++	double (53)	1.110e-16
C++ with GMP	64	2.71050543121376108502e-20
	128	1.46936793852785938496092…e-39
	256	4.31808427754722231269317…e-78

In the next sections we use Formula (2.1) and results of Table 2.4 to obtain the correct solution of linear system Equations (1.1).

References

1. Brent, R., Zimmermann, P.: Modern Computer Arithmetic, Cambridge Monographs on Computational and Applied Mathematics, vol. 18. Cambridge University Press (2010)
2. GMP Library, http://www.gmplib.org/
3. Khimich, A.N., Molchanov, I.N., Mova, V.I., et al.: Numerical program software of intelligent computer Inparcom. Naukova dumka, Kiev (2007)
4. Muller, J.-M., et al.: Handbook of Floating-Point Arithmetic. Birkhäuser, Basel (2009)
5. Nikolaevskaya, E.A., Chistyakova, T.V.: Software and algorithmic methods to improve the accuracy of computer solutions. Cybernetics and Systems Analysis 6, 172–176 (2009)

Chapter 3
Solution of Linear Algebraic Equations by Gauss Method

Elena A. Nikolaevskaya, Alexandr N. Khimich, and Tamara V. Chistyakova

Abstract. Gauss method is one of the most effective methods for solving linear systems with nonsingular matrices of general form. As you know, from Chapter 1 double digits for solution of SLAEs (1.1) was not enough to get the correct solutions of the system. For this case there is a need for increasing precision. It is possible with GMP library. In this Chapter you can find program texts for solution SLAEs (1.1) by Gauss method with double and `mpf_t` type of GMP library. These programs were tested on supercomputer, and the obtained results were analyzed.

Elena A. Nikolaevskaya
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: elena_nea@ukr.net

Alexandr N. Khimich
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: khimich_ic@mail.ru

Tamara V. Chistyakova
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: dept150@insyg.kiev.ua

E. Nikolaevskaya et al.: Programming with Multiple Precision, SCI 397, pp. 31–44.
springerlink.com

3.1 Gauss Method

The Gauss method (a unique division scheme, see also [1, 6, 7, 11, 13-15]) for solving the linear algebraic system

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \qquad\qquad \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n, \end{cases} \tag{3.1}$$

with the matrix A and the right-hand side b in the form

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

can be implemented in the following way. Suppose that $a_{11} \neq 0$; let us divide the first equation of system (3.1) by coefficient a_{11} which is called the leading element of the first step. Then let us multiply the obtained equation in succession by a_{i1}, $i = 2,3, \ldots, n$ and subtract it from the remaining equations $i = 2,3, \ldots, n$ of the system (3.1). As a result, instead of system (3.1) we get its equivalent system

$$\begin{cases} x_1 + a_{12}^{(1)}x_2 + \ldots + a_{1n}^{(1)}x_n = b_1^{(1)}, \\ \qquad a_{22}^{(1)}x_2 + \ldots + a_{2n}^{(1)}x_n = b_2^{(1)}, \\ \qquad\qquad \ldots \\ \qquad a_{n2}^{(1)}x_2 + \ldots + a_{nn}^{(1)}x_n = b_n^{(1)}, \end{cases} \tag{3.2}$$

with the matrix

$$A^{(1)} = \begin{bmatrix} 1 & a_{12}^{(1)} & \ldots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & \ldots & a_{2n}^{(1)} \\ \ldots & \ldots & \ldots & \ldots \\ 0 & a_{n2}^{(1)} & \ldots & a_{nn}^{(1)} \end{bmatrix}.$$

Note that the matrix $A^{(1)}$ can be obtained from $A^{(0)}=A$ by multiplying it by the matrix M_1 of the form

$$M_1 = \begin{bmatrix} \dfrac{1}{a_{11}} & 0 & 0 & \ldots & 0 \\ -\dfrac{a_{21}}{a_{11}} & 1 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ -\dfrac{a_{n1}}{a_{11}} & 0 & 0 & \ldots & 1 \end{bmatrix}, \qquad L_1 = M_1^{-1} = \begin{bmatrix} a_{11} & 0 & 0 & \ldots & 0 \\ a_{21} & 1 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{n1} & 0 & 0 & \ldots & 1 \end{bmatrix}.$$

We treat the system (3.2) in the same manner as system (3.1) by eliminating x_2 from all equations except for the 1st and the 2nd ones. Similarly as for M_1 and L_1 we can write out the matrix M_2, and $L_2 = M_2^{-1}$. At the *k-th* step of such a transformation the *k-th* row of the matrix $A^{(k-1)}$ is divided by its element $a_{kk}^{(k-1)}$, then multiplied in succession by $a_{ik}^{(k-1)}$, $i = k+1, \ldots, n$, and subtracted from all subsequent rows so as to all non-zero elements of the *k-th* column lying below the *k-th* row become zeros. Note, that

$$A^{(k)} = M_k A^{(k-1)}, \quad b^{(k)} = M_k b^{(k-1)}, \quad k = 1, 2, \ldots, n, \tag{3.3}$$

elements of the *k-th* column of the low triangular matrix M_k being expressed by formulas

$$m_i^{(k)} = 0 \text{ , if } i<k, \quad m_k^{(k)} = \frac{1}{a_{kk}^{(k-1)}}, \quad m_i^{(k)} = -\frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \text{ if } i>k. \tag{3.4}$$

Note, that $L_k = M_k^{-1}$.

The low triangular matrix M_k (Fig. 3.1) can be presented in the form

$$M_k = E + (M^{(k)} - e_k)e_k^T, \tag{3.5}$$

where $M^{(k)}$ is the *k-th* column of the matrix M_k; e_k is the *k-th* column of the *n-th* order identity matrix *E*.

$$U_k = \begin{bmatrix} 1 & & & & \times & & \\ & 1 & & & \vdots & & \\ & & \ddots & & \times & & \\ & & & 1 & \times & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ \\ k \\ \\ \\ \end{matrix}$$

$$M_k = \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & & \times & & \\ & & & & \times & 1 & \\ & & & & \times & & \ddots \\ & & & & \vdots & & & 1 \\ & & & & \times & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ \\ k \\ \\ \\ \\ \\ \end{matrix}$$

$$\begin{matrix} & & & & k & & & & \\ & & & & & & & & \end{matrix}$$

Fig. 3.1

From (3.3) we get

$$U = A^{(n)} = M_n A^{(n-1)} = M_n M_{n-1} M_{n-2} \ldots M_1 A^{(0)}.$$

By introducing notations

$$M = M_n M_{n-1} M_{n-2} \ldots M_1, \quad L = L_1 L_2 \ldots L_{n-1} L_n, \tag{3.6}$$

one can write

$$MA = U, \tag{3.7}$$

$$A = LU. \tag{3.8}$$

Thus, an execution of the *n-th* step that leads to the following system of equations

$$\begin{cases} x_1 + a_{12}^{(1)} x_2 + ... + a_{1n}^{(1)} x_n = b_1^{(1)}, \\ \qquad x_2 + ... + a_{2n}^{(2)} x_n = b_2^{(2)}, \\ \qquad\qquad ... \\ \qquad\qquad\qquad x_n = b_n^{(n)}, \end{cases} \tag{3.9}$$

and

$$U = A^{(n)} = \begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & 1 & a_{23}^{(2)} & \dots & a_{2n}^{(2)} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}, L = \begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22}^{(1)} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2}^{(1)} & a_{n3}^{(2)} & \dots & a_{nn}^{(n-1)} \end{bmatrix}. \tag{3.10}$$

This system can be written in the following matrix form:

$$Ux = Mb. \tag{3.11}$$

The transformation of system (3.1) into its equivalent system (3.9) is referred to as the forward elimination, while solving of the triangular system (3.9) is called the backward substitution.

Computational formulas for the Gauss's method are of the following form. The forward elimination, *s-th* step, *s*=1, 2, …, *n*:

$$a_{ik}^{(s)} = \frac{a_{ik}^{(s-1)}}{a_{ss}^{(s-1)}}, \quad b_i^{(s)} = \frac{b_i^{(s-1)}}{a_{ss}^{(s-1)}}, \qquad i = s, \qquad k = s, s+1, \dots, n;$$

$$a_{ik}^{(s)} = a_{ik}^{(s-1)} - \frac{a_{sk}^{(s-1)}}{a_{ss}^{(s-1)}} a_{is}^{(s-1)}, \qquad b_i^{(s)} = b_i^{(s-1)} - b_s^{(s-1)} \frac{a_{is}^{(s-1)}}{a_{ss}^{(s-1)}}$$

$$i = s+1, s+2, \dots, n; \; k = s, s+1, \dots, n.$$

The backward substitution is implemented by formula

$$x_i = b_i^{(i)} - \sum_{k=i+1}^{n} a_{ik}^{(i)} x_k \;, \quad i = n, n-1, ..., 1.$$

From decompositions (3.8), (3.11) it follows that

$$\det A = \det L \det U = \prod_{i=1}^{n} a_{ii}^{(i-1)}, \quad a_{11} = a_{11}^{(0)} \;.$$

This property is used in practice for evaluating a determinant of the matrix *A* by Gauss's method.

Consider a few examples [10].

Example 1. Solve the following linear algebraic system

$$\begin{cases} 5x_1 + x_2 + 2x_3 = 8, \\ 4x_1 + 4x_2 + x_3 = 2, , \\ x_1 - 2x_2 + 3x_3 = 9 \end{cases} \tag{3.12}$$

by the above-described scheme of the Gauss's method.

Solution. After the 1st step we get the following linear algebraic system equivalent to (3.12)

$$\begin{cases} x_1 + 0{,}2x_2 + 0{,}4x_3 = 1{,}6 \\ 3{,}2x_2 - 0{,}6x_3 = -4{,}4 \\ -2{,}2x_2 + 2{,}6x_3 = 7{,}4 \end{cases}$$

The second step yields

$$\begin{cases} x_1 + 0{,}2x_2 + 0{,}4x_3 = 1{,}6, \\ x_2 - 0{,}1875x_3 = -1{,}375, \\ 2{,}1875x_3 = 4{,}375 \end{cases}$$

The forward elimination is completed by the obtaining of the following system:

$$\begin{cases} x_1 + 0{,}2x_2 + 0{,}4x_3 = 1{,}6, \\ x_2 - 0{,}1875x_3 = -1{,}375, \\ x_3 = 2 \end{cases}$$

The backward substitution yields:

$$x_3 = 2, \qquad x_2 = -1{,}375 + 0{,}1875 \cdot 2 = -1, \qquad x_1 = 1.$$

Example 2. Evaluate a determinant of the matrix of system (3.12) by using the unique division scheme of the Gauss's method.

Solution. Using the results of the above example it is not difficult to become convinced that

$$\det A = 5 \cdot 3{,}2 \cdot 2{,}1875 = 35.$$

General results of forward elimination and backward substitution of the Gauss's method scheme described above can be presented in the matrix form as follows. Let U_k, k=1, 2, ..., n be the n-th order matrices of the form

$$U_k = E + \xi^{(k)} e_k^T, \quad U_1 \equiv E, \tag{3.13}$$

where $\xi^{(k)}$ is n-dimensional vector with elements

$$\xi_i^{(k)} = -a_{ik}^{(i)}, \quad i < k; \quad \xi_i^{(k)} = 0, \quad i \geq k \tag{3.14}$$

(see Fig. 3.1).

It is not difficult to become convinced that equality

$$U_1 \ldots U_n U = E \tag{3.15}$$

holds, where U is a matrix in decomposition (3.8), (3.10).

According to (3.15)

$$U^{-1} = U_1 \ldots U_n, \tag{3.16}$$

and, hence, with taking into account (3.6), (3.11), (3.16) the system's solution to be sought can be presented in the form

$$x = U_1 \ldots U_n M_n M_{n-1} \ldots M_1 b. \tag{3.17}$$

Since six different computational schemes correspond to one mathematical operation of two matrices multiplication, therefore with taking into account the

matrix form of Gauss's method representation, one can write out six different computational schemes for this method. For each concrete computer and each programming language it is necessary to use the most economical computational scheme of the Gauss's method.

To apply the unique division scheme it is necessary for all leading elements $a_{ss}^{(s-1)}$, $s = 1, 2, ..., n$ to be nonzeros. Besides, the proximity of leading elements to zero can result in the significant loss in accuracy of the computed solution. In order to avoid the termination of the computational process when dividing by zero or to avoid losses in accuracy when dividing by values close to zero, the Gauss's method schemes are usually employed implementing a choice of largest elements either by row, by column or over the entire matrix. For example, in the Gauss's method scheme with choosing the pivot over the entire matrix, an order of eliminating the unknowns in the given system is determined as follows. At the *s-th* step, $s = 1,2,\ldots n\text{-}1$ from elements of the not yet reduced submatrix (an active matrix) an element with largest modulus is chosen which is called a pivot of the *s-th* step. The unknown associated with this element is eliminated by rule described above. For the sake of convenience in computations, the interchanging of equations and unknowns is done prior to the elimination of this unknown in order to the pivot be situated in the upper left-hand corner of the matrix being transformed. If at the *s-th* step the largest element is chosen only among elements of the *s-th* column (row) such scheme is referred to as a scheme with column (row) pivoting. To solve linear algebraic systems on computer one needs to apply the Gauss's method with some or another choice of the pivot.

The solving of linear algebraic systems by Gauss's method [10] requires n divisions, $\frac{1}{3}\left(n^3 + 3n^2 + n\right)$ multiplications, and $\frac{1}{6}\left(2n^3 + 3n^2 - 5n\right)$ additions. It is necessary to store $n^2 + n + 1$ machine words for the method's implementation.

3.2 Program for Solution System of Linear Equations by Gauss Method Using GMP

Consider the program for solution SLAEs (1.1) by Gauss Method (Listing 3.1).

Listing 3.1. Solution of SLAEs by Gauss method

```
#include <stdio.h>
#include <time.h>

int main ()
{
     const int dim = 4;
     long double b[dim+1];
     long double A[dim+1][dim+1];
     long double A1[dim+1][dim+1];
long double b1[dim+1];
     double t1,t2;
```

```
      int i,j,k;
long double d;
      time_t aclock;
      struct tm *newtime;
      printf("Size: %d bits\n\n", sizeof(long double)*8);

  A[1][1] = 0.1348531574394464;
  A[1][2] = 0.1878970588235294;
  A[1][3] = 0.1909117647058824;
  A[1][4] = 0.1779264705882353;
  A[2][1] = 0.1878970588235294;
  A[2][2] = 0.262;
  A[2][3] = 0.265;
  A[2][4] = 0.247;
  A[3][1] = 0.1909117647058824;
  A[3][2] = 0.265;
  A[3][3] = 0.281;
  A[3][4] = 0.266;
  A[4][1] = 0.1779264705882353;
  A[4][2] = 0.247;
  A[4][3] = 0.266;
  A[4][4] = 0.255;
  b[1] = 0.3516;
  b[2] = 0.4887;
  b[3] = 0.5105;
  b[4] = 0.4818;

      for (i=1;i<=dim; i++)
{
             b1[i] = b[i];
      for (j=1;j<=dim;j++)
             A1[i][j] = A[i][j];
}
      t1=time( &aclock );
      for (j=1;j<=dim;j++)
{
             for (i=j;i<=dim; i++)
             {
             d = A[i][j];
             b[i] = b[i]/d;
             for (k=dim;k>=1;k--)
                    A[i][k] = A[i][k]/d;
      }
      for (i=j+1;i<=dim; i++)
```

```
                {
                b[i] -= b[j];
                for (k=dim;k>=1;k--)
                        A[i][k] -= A[j][k];
        }
        }
        for (j=dim; j>=1; j--)
                for (i=1; i<j; i++)
                {
                b[i] -= A[i][j] * b[j];
                for (k=1; k<=dim; k++)
                        A[i][k] -= A[i][j]*A[j][k];
        }
        printf("Result\n");
        for (i=1;i<=dim; i++)
                {
                        printf("\tx[%d] = %.25Lf\n",i,b[i]);
        }
                t2=time( &aclock )-t1;
                printf("\n");
                printf( "The current time are: %le\n", t2);
        return 0;
}
```

The protocol the solution is as follows:

```
Result:

     x[1] = 3546699037390.85696005821228027343750000
     x[2] = -2138450890190.76398253440856933593750000
     x[3] = -886674759347.12280559539794921875000000
     x[4] = 521573387852.11146506667137145996093750
```

As seen, the `double` digits was not enough to get the correct solutions of the system. Transform this program to work with the type of `mpf_t` of GMP library [5]. Consider Listing 3.2.

Listing 3.2. Solution of SLAEs by Gauss method with GMP

```
#include <stdio.h>

// Connect the file gmp.h
#include <gmp.h>

int main ()
{
```

```
//Set precision
const int precision = 128;

const int dim = 4;

// cteate mpf_t variables
mpf_t b[dim+1];
mpf_t A[dim+1][dim+1];
mpf_t A1[dim+1][dim+1];
mpf_t b1[dim+1];
mpf_t d,s;

int i,j,k;

// Init of variables with precision 128
mpf_init2 (d, precision);
mpf_init2 (s, precision);

for (i=1;i<=dim; i++)
{
            for (j=1;j<=dim;j++)
            mpf_init2 (A[i][j], precision);
            mpf_init2 (b[i], precision);
     }
for (i=1;i<=dim; i++)
{
            for (j=1;j<=dim;j++)
            mpf_init2 (A1[i][j], precision);
            mpf_init2 (b1[i], precision);
}
mpf_set_str (b[1], "0.3516", 10);
mpf_set_str (b[2], "0.4887", 10);
mpf_set_str (b[3], "0.5105", 10);
mpf_set_str (b[4], "0.4818", 10);

mpf_set_str (A[1][1], "0.1348531574394464", 10);
mpf_set_str (A[1][2], "0.1878970588235294", 10);
mpf_set_str (A[1][3], "0.1909117647058824", 10);
mpf_set_str (A[1][4], "0.1779264705882353", 10);
mpf_set_str (A[2][1], "0.1878970588235294", 10);
mpf_set_str (A[2][2], "0.262", 10);
mpf_set_str (A[2][3], "0.265", 10);
mpf_set_str (A[2][4], "0.247", 10);
mpf_set_str (A[3][1], "0.1909117647058824", 10);
```

```
mpf_set_str (A[3][2], "0.265", 10);
mpf_set_str (A[3][3], "0.281", 10);
mpf_set_str (A[3][4], "0.266", 10);
mpf_set_str (A[4][1], "0.1779264705882353", 10);
mpf_set_str (A[4][2], "0.247", 10);
mpf_set_str (A[4][3], "0.266", 10);
mpf_set_str (A[4][4], "0.255", 10);

for (i=1;i<=dim; i++)
{
            mpf_set (b1[i], b[i]);
            for (j=1;j<=dim;j++)
            mpf_set (A1[i][j],A[i][j]);
}

//++++++++++++ Gauss ++++++++++++++++++++++++++
// Transform to Upper triangular matrix
for (j=1; j<=dim; j++)
{
            mpf_set( d, A[j][j]);
            mpf_div( b[j], b[j], d);
            for (k=dim; k>=1; k--)
            mpf_div( A[j][k], A[j][k], d );
            for (i=j+1; i<=dim; i++)
            {
            mpf_set( s, A[i][j]);
            mpf_mul( d, b[j], s);
            mpf_sub( b[i], b[i], d);
            for (k=dim; k>=1; k--)
            {
                  mpf_mul( d, A[j][k], s);
                  mpf_sub( A[i][k], A[i][k], d);
      }
            }
     }
// Transform to Identity matrix
     for (j=dim; j>=1; j--)
            for (i=1; i<j; i++)
            {
            mpf_mul( d, A[i][j] ,b[j]);
            mpf_sub( b[i], b[i], d);
            for (k=1; k<=dim; k++)
                  {
                  mpf_mul( d, A[i][j], A[j][k]);
```

```
                    mpf_sub( A[i][k], A[i][k], d);
            }
     }
//++++++++++ End of Gauss ++++++++++++++++++

//Result printout
printf("Result for %d bits\n",precision);
for (i=1;i<=dim; i++)
{
     printf("x[%d] = ",i);
     mpf_out_str (stdout, 10, 120, b[i]);
     printf("\n");
 }
//Free
     for (i=1;i<=dim; i++)
     {
     for (j=1;j<=dim;j++)
            mpf_clear (A[i][j]);
            mpf_clear (b[i]);
 }
     for (i=1;i<=dim; i++)
     {
     for (j=1;j<=dim;j++)
            mpf_clear (A1[i][j]);
            mpf_clear (b1[i]);
 }
     return 0;
}
```

Compiling and running :

```
gcc gaus_mpf.c -o gaus_mpf -lgmp
./gaus_mpf
```

The protocol of the solution is as follows:

```
A[1][1] = 0.1348531574394464e0
A[1][2] = 0.1878970588235294e0
A[1][3] = 0.1909117647058824e0
A[1][4] = 0.1779264705882353e0
A[2][1] = 0.1878970588235294e0
A[2][2] = 0.262e0
A[2][3] = 0.265e0
A[2][4] = 0.247e0
A[3][1] = 0.1909117647058824e0
```

```
A[3][2] = 0.265e0
A[3][3] = 0.281e0
A[3][4] = 0.266e0
A[4][1] = 0.1779264705882353e0
A[4][2] = 0.247e0
A[4][3] = 0.266e0
A[4][4] = 0.255e0
b[1] = 0.3516e0
b[2] = 0.4887e0
b[3] = 0.5105e0
b[4] = 0.4818e0

Result for 64 bits
x[1] = 0.66620670578344609210 8e13
x[2] = -0.40168345495758267654 1e13
x[3] = -0.16655167644581835935 2e13
x[4] = 0.97971574379981708418 5e12
```

Program was tested on Inparcom-256 of V.M. Glushkov Institute of Cybernetics [8, 9]. The condition number of matrix A by formula (1.13) equals

$$\mathrm{cond}(A) = 2.089436217259268e + 016 .$$

Table 2.4 shows that the use of type `double` (length of mantissa equals 53)

$$macheps = 1.110\text{e-}16, \text{ i.e. } l=16,\ m=16.$$

Then we can calculate the number of correct digits by the formula (2.1):

$$k = 16 - 16 = 0 .$$

As seen from the calculations, the solution was not obtained. GMP provides possibility to increase accuracy. Already using length of mantissa 64, we got the right solution.

$$macheps=2.710505431213761085 02\text{e-}20,$$
$$l=20,\ m=16$$
$$k = 20 - 16 = 4 .$$

In the result of testing the exact solution is achieved using GMP library with a length of mantissa 256 (see Table 3.1). With further increase accuracy, the number of correct digits is increasing.

Table 3.1 Solution of SLAEs (1.1) with different precision

Length of mantissa	Solution of (1.1)
C	3.60239e+12
double	-2.17203e+12
(53)	-9.00599e+11
	5.29764e+11

Table 3.1 *(continued)*

Length of mantissa	Solution of (1.1)
C, 64	0.666199107066943565109e13
GMP	-0.401678873378514977738e13
	-0.166549776766692724716e13
	0.979704569216725111902e12
128	0.666216216216160673879806783667710258425 4e13
	-0.401689189189072350695216831583529709773 5e13
	-0.166554054053997005189401947634587399526 6e13
	0.979729729730279707257494561725193725136 2e12
256	0.666216216216160673879806449043057216803065823955356931480210097538868725325 4431e13
	-0.401689189189072350695216629824547728755978173750296346904869751776274086568 2239e13
	-0.166554054053997005189401863978424139235407407550574892875886157773900450639 1711e13
	0.979729729730279707257494069630115723559737931272217917962992860659207387176 8471e12

Thus, using the GMP library functions is obtained a correct solution of the system with an accuracy to sign.

References

In this Chapter we considered our program for solution systems of linear equations by Gauss method. For your interest we described here libraries and packages for solution this problem.

Starting in the 1960s much general purpose software, often collected in large libraries or packages have been developed. Matlab is more used interactive system for matrix computations, with “toolboxes” available for many application areas, e.g., control problems. Many programs and packages are available in the public domain and can be downloaded free. A prime example is LINPACK [2], which contains programs for solving linear systems.

Netlib [12] is a repository of public domain mathematical software, data, address lists, and other useful items for the scientific computing community.

1. Demmel, J.W.: Applied Numerical Linear Algebra. SIAM, Philadelphia (1997)
2. Dongarra, J.J., et al.: LINPACK User’s Guide. SIAM, Philadelphia (1979)
3. Forsythe, G.E., Malcolm, M.A., Moler, C.B.: Computer Methods for Mathematical Computations. Prentice-Hall, Englewood Cliffs (1977)
4. Forsythe, G.E., Moler, C.B.: Computer Solution of Linear Algebraic Systems. Prentice-Hall, Englewood Cliffs (1967)
5. GMP Library, `http://www.gmplib.org/`
6. Golub, G.H., Van Loan, C.F.: Matrix Computations, 2nd edn. Johns Hopkins University Press, Baltimore (1989)
7. Higham, N.J.: Accuracy and Stability of Numerical Algorithms. SIAM, Philadelphia (2002)

8. Inparcom (family of intelligent parallel computers), http://www.geopoisk.com/inparcom/eng/index_eng.htm
9. Khimich, A.N., Molchanov, I.N., Mova, V.I., et al.: Numerical program software of intelligent computer Inparcom. Naukova dumka, Kiev (2007)
10. Molchanov, I.N.: Machine methods for solving applied problems. Algebra, approximation of functions, ordinary differential equations. Naukova dumka, Kiev (2007)
11. Nash, J.C.: Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation. American Institute of Physics, New York (1990)
12. Netlib web-site, http://www.netlib.org
13. Stewart, G.W.: Introduction to Matrix Computations. Academic Press, New York (1973)
14. Stewart, G.W., Sun, J.-G.: Matrix Perturbation Theory. Academic Press, Boston (1990)
15. Strang, G.: Introduction to Applied Mathematics. Wellesley-Cambridge Press, Wellesley (1988)

Chapter 4
What Is MPFR?

Elena A. Nikolaevskaya, Alexandr N. Khimich, and Tamara V. Chistyakova

Abstract. MPFR library (MultiPrecision Float with Rounding) deserves special attention, as the programmer who uses it creates their own floating-point format, indicating the bit order of the number and the mantissa. Thus, you can create formats that are not described in modern standards, but to perform arithmetic operations according to the classical algorithms of floating-point arithmetic. In this chapter we consider main function of MPFR library and program of solution SLAEs (1.1) by Gauss method using `mpfr_t` type of variable. Finally, we analyze obtained results by using GMP and MPFR libraries.

Elena A. Nikolaevskaya·
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: elena_nea@ukr.net

Alexandr N. Khimich
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: khimich_ic@mail.ru

Tamara V. Chistyakova
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: dept150@insyg.kiev.ua

E. Nikolaevskaya et al.: Programming with Multiple Precision, SCI 397, pp. 45–70.
springerlink.com

4.1 Introduction

GNU MPFR [5] is a multiple-precision binary floating-point library, written in C, based on the GNU MP library. It aims to provide a class of floating-point numbers with precise semantics.

The latest version of MPFR and more details are available from ftp://ftp.gnu.org/gnu/mpfr/ or http://www.mpfr.org/.

You can try MPFR online from http://ex-cs.sist.ac.jp/~tkouya/try_mpfr.html (Figure 4.1).

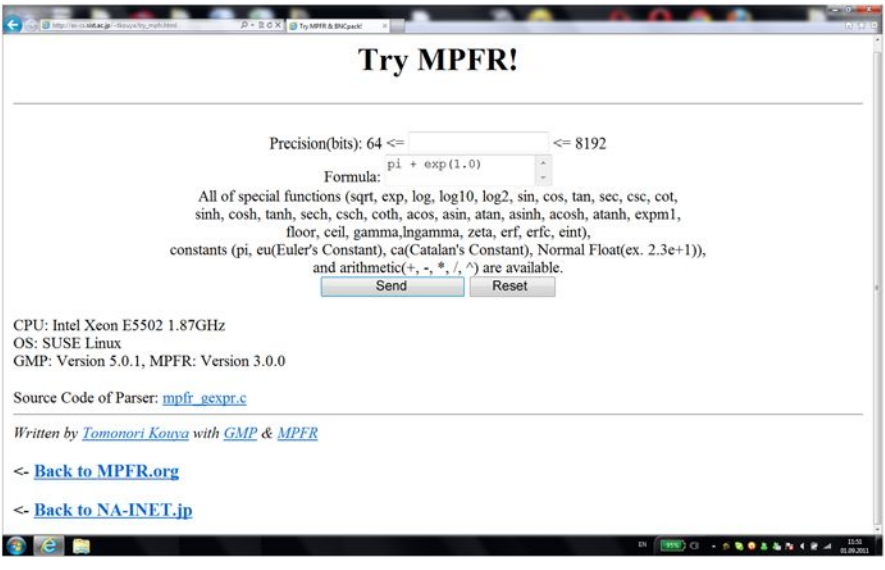

Fig. 4.1 Try MPFR online!

4.2 MPFR Basics

All declarations needed to use MPFR are collected in the include file `mpfr.h`. It is designed to work with both C and C++ compilers. You should include that file in any program using the MPFR library:

```
#include <mpfr.h>
```

Note however that prototypes for MPFR functions with FILE * parameters are provided only if `<stdio.h>` is included too (before `mpfr.h`):

```
#include <stdio.h>
#include <mpfr.h>
```

All programs that use MPFR must link against to both libmpfr and libgmp libraries. On a typical Unix-like system this can be done with '`-lmpfr -lgmp`' (in that order), for example:

```
gcc myprogram.c -lmpfr -lgmp
```

The C data type for floating-point number in MPFR is **`mpfr_t`** (internally defined as an one-element array of a structure, and **`mpfr_ptr`** is the C data type representing a pointer to this structure).

The precision, corresponding C data type is `mpfr_prec_t`. The precision can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`.

MPFR needs to increase the precision internally, in order to provide accurate results (and in particular, correct rounding). Do not attempt to set the precision to any value near `MPFR_PREC_MAX`, otherwise MPFR will abort due to an assertion failure. Moreover, you may reach some memory limit on your platform, in which case the program may abort, crash or have undefined behavior (depending on your C implementation).

The rounding mode specifies the way to round the result of a floating-point operation, in case the exact result can not be represented exactly in the destination significand; the corresponding C data type is `mpfr_rnd_t`.

The structure of the `mpfr_t` datatype is shown in Figure 4.2. Datatype has parts of extened sign, exponent, and mantissa. The user-defined precision fixes the length of mantissa that the pointers `mpfr_d` point to.

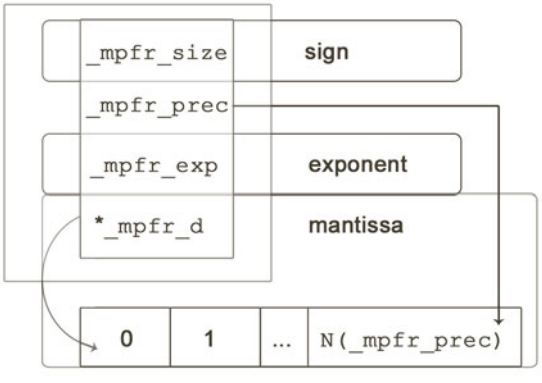

Fig. 4.2 The structure of `mpfr_t` type in MPFR

Due to this structure, multiple precision variables may not be guranteed to be always stored in continous memory areas.

The mpfr_t type is internally defined as a one-element array of a structure, and mpfr_ptr is the C data type representing a pointer to this structure. The mpfr_t type consists of four fields:

- The _mpfr_prec field is used to store the precision of the variable (in bits); this is not less than MPFR_PREC_MIN.
- The _mpfr_sign field is used to store the sign of the variable.
- The _mpfr_exp field stores the exponent. An exponent of 0 means a radix point just above the most significant limb. Non-zero values n are a multiplier 2^n relative to that point. A NaN, an infinity, and a zero are indicated by special values of the exponent field.
- Finally, the _mpfr_d field is a pointer to the limbs, least significant limbs stored first. The number of limbs in use is controlled by _mpfr_prec, namely ceil(_mpfr_prec/mp_bits_per_limb). Non-singular (i.e., different from NaN, Infinity or zero) values always have the most significant bit of the most significant limb set to 1. When the precision does not correspond to a whole number of limbs, the excess bits at the low end of the data are zeros.

This is a free library for arithmetic with an arbitrary digit. It extends the library of functions "mpf", providing the best rounding of the IEEE-754.

Consider in detail how the rounding:

- MPFR_RNDN: rounds to nearest (roundTiesToEven in IEEE 754-2008),
- MPFR_RNDZ: rounds toward zero (roundTowardZero in IEEE 754-2008),
- MPFR_RNDU: rounds toward plus infinity (roundTowardPositive in IEEE 754-2008),
- MPFR_RNDD: rounds toward minus infinity (roundTowardNegative in IEEE 754-2008),
- MPFR_RNDA: rounds away from zero (experimental).

The '*round to nearest'* mode works as in the IEEE 754 standard: in case the number to be rounded lies exactly in the middle of two representable numbers, it is rounded to the one with the least significant bit set to zero.

In our examples we used rounding modes GMP_RNDx. The rounding modes GMP_RNDx were renamed to MPFR_RNDx in MPFR 3.0. However the old names GMP_RNDx have been kept for compatibility (this might change in future versions), using:

```
#define GMP_RNDN MPFR_RNDN
#define GMP_RNDZ MPFR_RNDZ
#define GMP_RNDU MPFR_RNDU
#define GMP_RNDD MPFR_RNDD
```

The floating-point functions expect arguments of type mpfr_t.

The MPFR floating-point functions have an interface that is similar to the GNU MP functions. The function prefix for floating-point operations is mpfr_.

The user has to specify the precision of each variable. A computation that assigns a variable will take place with the precision of the assigned variable; the cost of that computation should not depend on the precision of variables used as input (on average).

The semantics of a calculation in MPFR is specified as follows: Compute the requested operation exactly (with "infinite accuracy"), and round the result to the precision of the destination variable, with the given rounding mode. The MPFR floating-point functions are intended to be a smooth extension of the IEEE 754 arithmetic. The results obtained on a given computer are identical to those obtained on a computer with a different word size, or with a different compiler or operating system.

MPFR does not keep track of the accuracy of a computation. This is left to the user or to a higher layer (for example the MPFI library [4, 8] for interval arithmetic). As a consequence, if two variables are used to store only a few significant bits, and their product is stored in a variable with large precision, then MPFR will still compute the result with full precision.

Most MPFR functions take as first argument the destination variable, as second and following arguments the input variables, as last argument a rounding mode, and have a return value of type `int`, called the ternary value. The value stored in the destination variable is correctly rounded, i.e., MPFR behaves as if it computed the result with an infinite precision, then rounded it to the precision of this variable. The input variables are regarded as exact (in particular, their precision does not affect the result).

MPFR functions may create caches, e.g., when computing constants such as Pi, either because the user has called a function like `mpfr_const_pi` directly or because such a function was called internally by the MPFR library itself to compute some other function.

At any time, the user can free the various caches with `mpfr_free_cache`.

MPFR internal data such as flags, the exponent range, the default precision and rounding mode, and caches (i.e., data that is not accessed via parameters) are either global (if MPFR has not been compiled as thread safe) or per-thread (thread local storage).

4.3 Main Functions

Before you can assign to an MPFR variable, you need to initialize it by calling one of the special initialization functions. When you are done with a variable, you need to clear it out, using one of the functions for that purpose. A variable should only be initialized once, or at least cleared out between each initialization. After a variable has been initialized, it may be assigned to any number of times. For efficiency reasons, avoid initializing and clearing out a variable in loops. Instead, initialize it before entering the loop, and clear it out after the loop has exited. You do not need to be concerned about allocating additional space for MPFR variables,

since any variable has a significand of fixed size. Hence unless you change its precision, or clear and reinitialize it, a floating-point variable will have the same allocated space during all its life.

As a general rule, all MPFR functions expect output arguments are before input arguments. This notation is based on an analogy with the assignment operator. MPFR allows you to use the same variable for both input and output in the same expression. For example, the main function for floating-point multiplication, `mpfr_mul`, can be used like this:

```
mpfr_mul(x, x, x, rnd)
```

This computes the square of `x` with the rounding mode `rnd` and puts the result back in `x`.

Consider a linear algebraic Equation (1.1) and the previous program (Listing 3.1) using the library of MPFR (Listing 4.1).

At the beginning of the program you must include the header files `gmp.h` and `mpfr.h` and link the appropriate libraries. Then define precision (in this program it equals 512 bit) and all `mpfr_t` variables.

Listing 4.1. Solution of Linear System Equations by Gauss method with MPFR

```
#include <stdio.h>
#include <gmp.h>
#include <mpfr.h>

int main ()
{

const int precision = 512;
const int dim = 4;

mpfr_t b[dim+1];
mpfr_t A[dim+1][dim+1];
mpfr_t A1[dim+1][dim+1];
mpfr_t b1[dim+1];
mpfr_t d,s;

int i,j,k;

mpfr_init2 (d, precision);
mpfr_init2 (s, precision);

for (i=1;i<=dim; i++)
{
     for (j=1;j<=dim;j++)
```

```
            mpfr_init2 (A[i][j], precision);
            mpfr_init2 (b[i], precision);
}
for (i=1;i<=dim; i++)
{
     for (j=1;j<=dim;j++)
            mpfr_init2 (A1[i][j], precision);
            mpfr_init2 (b1[i], precision);
}
mpfr_set_str (b[1], "0.3516", 10, GMP_RNDD);
mpfr_set_str (b[2], "0.4887", 10, GMP_RNDD);
mpfr_set_str (b[3], "0.5105", 10, GMP_RNDD);
mpfr_set_str (b[4], "0.4818", 10, GMP_RNDD);

mpfr_set_str(A[1][1],"0.1348531574394464",10,GMP_RNDD);
mpfr_set_str(A[1][2],"0.1878970588235294", 10, GMP_RNDD);
mpfr_set_str(A[1][3],"0.1909117647058824", 10, GMP_RNDD);
mpfr_set_str(A[1][4],"0.1779264705882353", 10, GMP_RNDD);
mpfr_set_str(A[2][1],"0.1878970588235294", 10, GMP_RNDD);
mpfr_set_str(A[2][2],"0.262", 10, GMP_RNDD);
mpfr_set_str(A[2][3],"0.265", 10, GMP_RNDD);
mpfr_set_str(A[2][4],"0.247", 10, GMP_RNDD);
mpfr_set_str(A[3][1],"0.1909117647058824", 10, GMP_RNDD);
mpfr_set_str(A[3][2],"0.265", 10, GMP_RNDD);
mpfr_set_str(A[3][3],"0.281", 10, GMP_RNDD);
mpfr_set_str(A[3][4],"0.266", 10, GMP_RNDD);
mpfr_set_str(A[4][1],"0.1779264705882353", 10, GMP_RNDD);
mpfr_set_str(A[4][2],"0.247", 10, GMP_RNDD);
mpfr_set_str(A[4][3],"0.266", 10, GMP_RNDD);
mpfr_set_str(A[4][4],"0.255", 10, GMP_RNDD);

for (i=1;i<=dim; i++)
{
     mpfr_set (b1[i], b[i],GMP_RNDD);
     for (j=1;j<=dim;j++)
            mpfr_set (A1[i][j],A[i][j],GMP_RNDD);
}

//++++++++++++ Gauss ++++++++++++++++++++++++++
// Transform to Upper triangular matrix
for (j=1; j<=dim; j++)
{
     mpfr_set( d, A[j][j], GMP_RNDD);
     mpfr_div( b[j], b[j], d, GMP_RNDD);
     for (k=dim; k>=1; k--)
```

```
                mpfr_div( A[j][k], A[j][k], d ,GMP_RNDD);
                for (i=j+1; i<=dim; i++)
                {
                        mpfr_set( s, A[i][j], GMP_RNDD);
                        mpfr_mul( d, b[j], s, GMP_RNDD);
                        mpfr_sub( b[i], b[i], d, GMP_RNDD);
                        for (k=dim; k>=1; k--)
                        {
                        mpfr_mul( d, A[j][k], s, GMP_RNDD);
                        mpfr_sub( A[i][k], A[i][k], d, GMP_RNDD);
                        }
                }
    }
    // Transform to Identity matrix
    for (j=dim; j>=1; j--)
         for (i=1; i<j; i++)
         {
                mpfr_mul( d, A[i][j] ,b[j], GMP_RNDD);
                mpfr_sub( b[i], b[i], d, GMP_RNDD);
                for (k=1; k<=dim; k++)
                {
                        mpfr_mul( d, A[i][j], A[j][k], GMP_RNDD);
                        mpfr_sub( A[i][k], A[i][k], d, GMP_RNDD);
                }
         }
    //++++++++++ End of Gauss ++++++++++++++++++
    //Result printout
    printf("Result for %d bits\n",precision);
    for (i=1;i<=dim; i++)
    {
         printf("x[%d] = ",i);
         mpfr_out_str (stdout, 10, 120, b[i], GMP_RNDD);
         printf("\n");
    }
    for (i=1;i<=dim; i++)
    {
         for (j=1;j<=dim;j++)
                mpfr_clears (A[i][j], b[i]);
    }
    for (i=1;i<=dim; i++)
    {
         for (j=1;j<=dim;j++)
                mpfr_clears (A1[i][j], b1[i]);
    }
    mpfr_clears (d,s);
```

```
return 0;
}
```

Consider MPFR below for more details about our program and main functions.

4.3.1 Initialization and Assignment Functions

An `mpfr_t` object must be initialized before storing the first value in it. The functions `mpfr_init` and `mpfr_init2` are used for that purpose.

`void mpfr_init2(mpfr_t x, mpfr_prec_t prec)`

Initialize `x`, set its precision to be exactly prec bits and its value to `NaN`. Normally, a variable should be initialized only once or at least be cleared, using `mpfr_clear`, between initializations. To change the precision of a variable which has already been initialized, use `mpfr_set_prec`. The precision `prec` must be an integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX` (otherwise the behavior is undefined).

For example, in Listing 4.1 all `mpfr_t` objects must be initialized. We can use function `mpfr_init2()`.

```
…
const int precision = 512;
const int dim = 4;

mpfr_t b[dim+1];
mpfr_t A[dim+1][dim+1];
mpfr_t A1[dim+1][dim+1];
mpfr_t b1[dim+1];
mpfr_t d,s;

mpfr_init2 (d, precision);
mpfr_init2 (s, precision);

     for (i=1;i<=dim; i++)
     {
            for (j=1;j<=dim;j++)
                   mpfr_init2 (A[i][j], precision);
                   mpfr_init2 (b[i], precision);
     }
     for (i=1;i<=dim; i++)
     {
     for (j=1;j<=dim;j++)
```

```
            mpfr_init2 (A1[i][j], precision);
            mpfr_init2 (b1[i], precision);
}
…
```

void mpfr_inits2(mpfr_prec_t prec, mpfr_t x, ...)

Initialize all the `mpfr_t` variables of the given variable argument va_list, set their precision to be exactly `prec` bits and their value to `NaN`. See `mpfr_init2` for more details. The va_list is assumed to be composed only of type `mpfr_t` (or equivalently `mpfr_ptr`). It begins from `x`, and ends when it encounters a null pointer (which type must also be `mpfr_ptr`).

We can rewrite previous example with function `mpfr_inits2()` as

```
mpfr_inits2 (precision,s,d);
for (i=1;i<=dim; i++)
{
       for (j=1;j<=dim;j++)
              mpfr_init2(precision,A[i][j],b[i]);
}
for (i=1;i<=dim; i++)
{
       for (j=1;j<=dim;j++)
              mpfr_init2(precision,A1[i][j],b1[i]);
}
```

void mpfr_init (mpfr_t x)

Initialize `x`, set its precision to the default precision, and set its value to `NaN`. The default precision can be changed by a call to `mpfr_set_default_prec`.

void mpfr_inits(mpfr_t x, ...)

Initialize all the `mpfr_t` variables of the given va_list, set their precision to the default precision and their value to `NaN`. See `mpfr_init` for more details.

void mpfr_clear(mpfr_t x)

Free the space occupied by the significand of `x`. Make sure to call this function for all `mpfr_t` variables when you are done with them.

void mpfr_clears(mpfr_t x, ...)

Free the space occupied by all the `mpfr_t` variables.

For example, from listing 4.1 for all `mpfr_t` variables you must to clean memory by functions `mpfr_clear()` or `mpfr_clears()`:

```
…
for (i=1;i<=dim; i++)
{
```

```
    for (j=1;j<=dim;j++)
       mpfr_clears (A[i][j], b[i]);
}
for (i=1;i<=dim; i++)
{
    for (j=1;j<=dim;j++)
       mpfr_clears (A1[i][j], b1[i]);
}
mpfr_clears (d,s);
…
```

Describe `mpfr_clear` for more details. Here is an example of the usage of multiple initialization functions (since `NULL` is not necessarily defined in this context, we use (`mpfr_ptr`) 0 instead, but (`mpfr_ptr`) `NULL` is also correct).

```
{
mpfr_t x, y, z, t;
mpfr_inits2 (256, x, y, z, t, (mpfr_ptr) 0);
...
mpfr_clears (x, y, z, t, (mpfr_ptr) 0);
}
```

`void mpfr_set_default_prec(mpfr_prec_t prec)`

Set the default precision to be exactly `prec` bits, where `prec` can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`. The precision of a variable means the number of bits used to store its significand. All subsequent calls to `mpfr_init` or `mpfr_inits` will use this precision, but previously initialized variables are unaffected. The default precision is set to 53 bits initially.

`mpfr_prec_t mpfr_get_default_prec(void)`

Return the current default MPFR precision in bits.

Here is an example of how to initialize floating-point variables:

```
{
mpfr_t x, y;
const int PRECISION 256;
/* set default precision=256 */
mpfr_set_default_prec(PRECISION);
/* use default precision */
mpfr_init (x);
/* precision exactly 256 bits */
mpfr_init2 (y, 128);
        ...
/* When the program is about to exit, do ... */
```

```
        mpfr_clear (x);
        mpfr_clear (y);
  /* free the cache for constants like pi */
  mpfr_free_cache();
  }
```

The following functions are useful for changing the precision during a calculation. A typical usage would be for adjusting the precision gradually in iterative algorithms, making the computation precision closely match to the actual accurate part of the numbers.

`void mpfr_set_prec(mpfr_t x, mpfr_prec_t prec)`

Reset the precision of `x` to be exactly `prec` bits, and set its value to `NaN`. The previous value stored in `x` is lost. It is equivalent to a call to `mpfr_clear(x)` followed by a call to `mpfr_init2(x, prec)`, but more efficient as no allocation is done in case the current allocated space for the significand of `x` is enough. The precision `prec` can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`. In case you want to keep the previous value stored in `x`, use `mpfr_prec_round` instead.

`mpfr_prec_t mpfr_get_prec(mpfr_t x)`

Return the precision of `x`, i.e., the number of bits used to store its significand.

These functions assign new values to already initialized floats

`int mpfr_set(mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`

Set the value of `x` from `y`, rounded toward the given direction `rnd`. All functions `mpfr_set()` with different type `y` are described in Appendix A (Table A2).

In listing 4.1 function `mpfr_set()`is used to assign `b1[i]= b[i]` and `A1[i][j]=A[i][j]` with rounding to toward minus infinity:

```
for (i=1;i<=dim; i++)
{
     mpfr_set (b1[i], b[i],GMP_RNDD);
     for (j=1;j<=dim;j++)
            mpfr_set (A1[i][j],A[i][j],GMP_RNDD);
}
```

Note, that the input 0 is converted to +0 by `mpfr_set_ui`, `mpfr_set_si`, `mpfr_set_uj`, `mpfr_set_sj`, `mpfr_set_z`, `mpfr_set_q` and `mpfr_set_f`, regardless of the rounding mode.

If you want to store a floating-point constant to a `mpfr_t`, you should use `mpfr_set_str` (or one of the MPFR constant functions, such as `mpfr_const_pi` for Pi) instead of `mpfr_set_flt`, `mpfr_set_d` and

mpfr_set_ld or mpfr_set_decimal64. Otherwise the floating-point constant will be first converted into a reduced-precision (e.g., 53-bit) binary number before MPFR can work with it.

int mpfr_set_str(mpfr_t x, const char *s, int base, mpfr_rnd_t rnd)

Set x to the value of the string s in base base, rounded in the direction rnd. Contrary to mpfr_strtofr, mpfr_set_str requires the whole string to represent a valid floating-point number. This function returns 0 if the entire string up to the final null character is a valid number in base base; otherwise it returns −1, and x may have changed. Note: it is preferable to use mpfr_set_str if one wants to distinguish between an infinite x value coming from an infinite s or from an overflow.

For example, if we have vector b =(0.3516; 0.4887; 0.5105; 0.4818), then we can use function mpfr_set_str(). Here is used rounding to GMP_RNDD, i.e. round toward minus infinity. See below:

```
mpfr_set_str (b[1], "0.3516", 10, GMP_RNDD);
mpfr_set_str (b[2], "0.4887", 10, GMP_RNDD);
mpfr_set_str (b[3], "0.5105", 10, GMP_RNDD);
mpfr_set_str (b[4], "0.4818", 10, GMP_RNDD);
```

int mpfr_strtofr(mpfr_t x, const char *nptr,char **endptr, int base, mpfr_rnd_t rnd)

Read a floating-point number from a string nptr in base base, rounded in the direction rnd; base must be either 0 (to detect the base, as described below) or a number from 2 to 62 (otherwise the behavior is undefined). If nptr starts with valid data, the result is stored in x and *endptr points to the character just after the valid data (if endptr is not a null pointer); otherwise x is set to zero and the value of nptr is stored in the location referenced by endptr (if endptr is not a null pointer). The usual ternary value is returned.

void mpfr_set_nan(mpfr_t x)
void mpfr_set_inf(mpfr_t x, int sign)
void mpfr_set_zero(mpfr_t x, int sign)

Set the variable x to NaN, infinity, or zero, respectively. In mpfr_set_inf or mpfr_set_zero, x is set to plus infinity or plus zero if sign is nonnegative; in mpfr_set_nan, the sign bit of the result is unspecified.

void mpfr_swap(mpfr_t x, mpfr_t y)

Swap the values x and y efficiently. The precisions are exchanged too; in case the precisions are different, mpfr_swap is thus not equivalent to three mpfr_set calls using a third auxiliary variable.

int mpfr_init_set(mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)

Initialize x and set its value from y, rounded in the direction rnd. All functions mpfr_init_set() with different type y are described in Appendix A (Table A3). The precision of x will be taken from the active default precision, as set by mpfr_set_default_prec.

For example,

```
mpfr_t x,y;
mpf_t z;
mpf_init_set_d (z, 10.0);
mpfr_init_set_d (x, 5.0,GMP_RNDD);
mpfr_init_set_f (y, z,GMP_RNDD);
```

int mpfr_init_set_str(mpfr_t x, const char *s, int base, mpfr_rnd_t rnd)

Initialize x and set its value from the string s in base base, rounded in the direction rnd. See mpfr_set_str.

4.3.2 Conversion Functions

The next function converts mpfr_t variable op to a float, a double, a long double, a _Decimal64, a long, an unsigned long, an intmax_t, or an uintmax_t, respectively by using the rounding mode rnd.

float mpfr_get_flt(mpfr_t x, mpfr_rnd_t rnd)
double mpfr_get_d(mpfr_t x, mpfr_rnd_t rnd)
long double mpfr_get_ld(mpfr_t x, mpfr_rnd_t rnd)
_Decimal64 mpfr_get_decimal64(mpfr_t x, mpfr_rnd_t rnd)
long mpfr_get_si(mpfr_t x, mpfr_rnd_t rnd)
unsigned long mpfr_get_ui(mpfr_t x, mpfr_rnd_t rnd)
intmax_t mpfr_get_sj(mpfr_t x, mpfr_rnd_t rnd)
uintmax_t mpfr_get_uj(mpfr_t x, mpfr_rnd_t rnd)
double mpfr_get_d_2exp(long *exp, mpfr_t x, mpfr_rnd_t rnd)
long double mpfr_get_ld_2exp(long *exp, mpfr_t x, mpfr_rnd_t rnd)

Return d and set exp (formally, the value is pointed to exp) such that $0{,}5 \le |d| < 1$ and d times 2 raised to exp equals x rounded to double (resp. long double) precision by using the given rounding mode. If x is zero, then a zero of the same sign (or an unsigned zero, if the implementation does not have

signed zeros) is returned, and exp is set to 0. If x is NaN or an infinity, then the corresponding double precision (resp. long double precision) value is returned, and exp is undefined.

mpfr_exp_t mpfr_get_z_2exp(mpz_t x, mpfr_t y)

Put the scaled significand of y (regarded as an integer, with the precision of y) into x, and returns the exponent *exp* (which may be outside the current exponent range) such that y exactly equals x times 2 raised to the power *exp*. If y is zero, the minimal exponent *emin* is returned. If y is NaN or an infinity, the *erange* flag is set, x is set to 0, and the the minimal exponent *emin* is returned. The returned exponent may be less than the minimal exponent *emin* of MPFR numbers in the current exponent range; in case the exponent is not representable in the mpfr_exp_t type, the *erange* flag is set and the minimal value of the mpfr_exp_t type is returned.

int mpfr_get_z(mpz_t x, mpfr_t y, mpfr_rnd_t rnd)

Convert y to a mpz_t, after rounding it with respect to rnd. If y is NaN or an infinity, the *erange* flag is set, x is set to 0, and 0 is returned.

int mpfr_get_f(mpf_t x, mpfr_t y, mpfr_rnd_t rnd)

Convert y to a mpf_t, after rounding it with respect to rnd. The *erange* flag is set if y is NaN or Inf, which does not exist in MPF.

char * mpfr_get_str(char *str, mpfr_exp_t *expptr, int b, size_t n, mpfr_t x, mpfr_rnd_t rnd)

Convert x to a string of digits in base b, with rounding in the direction rnd, where n is either zero or the number of significant digits output in the string; in the latter case, n must be greater or equal to 2. The base may vary from 2 to 62. If the input number is an ordinary number, the exponent is written through the pointer expptr (for input 0, the current minimal exponent is written).

The generated string is a fraction, with an implicit radix point immediately to the left of the first digit. For example, the number −3.1416 would be returned as "−31416" in the string and 1 would be written at expptr. If rnd is to the nearest, and x is exactly in the middle of two consecutive possible outputs, then the one with an even significand is chosen, where both significands are considered with the exponent of x.

If n is zero, the number of digits of the significand is chosen large enough so that re-reading of the printed value with the same precision, assuming both output and input use rounding to the nearest, will recover the original value of x. More precisely, in most cases, the chosen precision of str is the minimal precision m depending only on p = PREC(op) and b that satisfies the above property.

If str is a null pointer, space for the significand is allocated using the current allocation function, and a pointer to the string is returned. To free the returned string, you must use mpfr_free_str.

If `str` is not a null pointer, it should point to a block of storage large enough for the significand.

A pointer to the string is returned, unless there is an error, in which case a null pointer is returned.

`void mpfr_free_str(char *str)`

Free a string allocated by `mpfr_get_str` using the current unallocation function. The block is assumed to be `strlen(str)+1` bytes.

`int mpfr_fits_ulong_p(mpfr_t x, mpfr_rnd_t rnd)`
`int mpfr_fits_slong_p(mpfr_t x, mpfr_rnd_t rnd)`
`int mpfr_fits_uint_p(mpfr_t x, mpfr_rnd_t rnd)`
`int mpfr_fits_sint_p(mpfr_t x, mpfr_rnd_t rnd)`
`int mpfr_fits_ushort_p(mpfr_t x, mpfr_rnd_t rnd)`
`int mpfr_fits_sshort_p(mpfr_t x, mpfr_rnd_t rnd)`
`int mpfr_fits_uintmax_p(mpfr_t x, mpfr_rnd_t rnd)`
`int mpfr_fits_intmax_p(mpfr_t x, mpfr_rnd_t rnd)`

Return non-zero if `x` would fit in the respective C data type, respectively `unsigned long`, `long`, `unsigned int`, `int`, `unsigned short`, `short`, `uintmax_t`, `intmax_t`, when rounded to an integer in the direction `rnd`.

4.3.3 Arithmetic Functions

All arithmetic functions with different types of variables are described in Appendix A (Table A4).

Addition operation:

`int mpfr_add(mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
`int mpfr_add_d(mpfr_t z, mpfr_t x, double y, mpfr_rnd_t rnd)`

Set `z` to `x+y` rounded in the direction `rnd`. For types that have no signed zero, it is considered unsigned (i.e., (+0) + 0 = (+0) and (–0) + 0 = (–0)). The `mpfr_add_d` function assumes that the radix of the `double` type is a power of 2, with a precision at most that declared by the C implementation (macro IEEE_DBL_MANT_DIG, and if not defined 53 bits).

Subtraction operation:

`int mpfr_sub(mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
`int mpfr_sub_d(mpfr_t z, mpfr_t x, double y, mpfr_rnd_t rnd)`

int mpfr_d_sub(mpfr_t z, double x, mpfr_t y, mpfr_rnd_t rnd)

Set z to x-y rounded in the direction rnd. For types that have no signed zero, it is considered unsigned (i.e., (+0) − 0 = (+0), (−0) − 0 = (−0), 0 − (+0) = (−0) and 0 − (−0) = (+0)). The same restrictions as for mpfr_add_d are applied to mpfr_d_sub and mpfr_sub_d.

Multiplication operation:

int mpfr_mul(mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)

Set z to x times y rounded in the direction rnd.

For example, the following fragment of the program computes a lower bound on 1+1/1!+1/2!+...+1/10! using 128-bit precision:

```
…
unsigned int i;
/* declares three floating-point variables x,y,z */
mpfr_t x, y, z;
/* initializes the variable x,y,z with 128-bit precision */
mpfr_inits2 (128,x, y,z);
/* sets the value of x and y to the double-precision number
1.0 rounded toward minus infinity */
mpfr_set_d (x, 1.0, GMP_RNDD);
mpfr_set_d (y, 1.0, GMP_RNDD);
for (i = 1; i <= 10; i++)
{
/* multiplies x in place by the unsigned integer i, where the
result is rounded toward plus infinity */
mpfr_mul_ui (x, x, i, GMP_RNDU);
mpfr_set_d (z, 1.0, GMP_RNDD);
/* divides z by x, rounds the result toward minus infinity and
stores it into z */
mpfr_div (z, z, x, GMP_RNDD);
mpfr_add (y, y, z, GMP_RNDD);
}
…
```

When a result is zero, its sign is the product of the signs of the operands (for types having no signed zero, it is considered positive). The same restrictions as for mpfr_add_d are applied to mpfr_mul_d.

int mpfr_mul_2ui(mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)

int mpfr_mul_2si(mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)

Set z to x times 2 raised to y rounded in the direction rnd. It just increases the exponent by y when z and x are identical.

int mpfr_sqr(mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)
Set x to the square of y rounded in the direction rnd.

Division operation:
int mpfr_div(mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)
int mpfr_d_div(mpfr_t z, double x, mpfr_t y, mpfr_rnd_t rnd)
int mpfr_div_d(mpfr_t z, mpfr_t x, double y, mpfr_rnd_t rnd)
Set z to x/y rounded in the direction rnd.

When a result is zero, its sign is the product of the signs of the operands (for types having no signed zero, it is considered positive). The same restrictions as for mpfr_add_d are applied to mpfr_d_div and mpfr_div_d.

For example, circle from listing 6 below

```
for (j=1; j<=dim; j++)
{
//d=A[j][j]; b[j]=b[j]/d;
     mpfr_set( d, A[j][j], GMP_RNDD);
     mpfr_div( b[j], b[j], d, GMP_RNDD);
     for (k=dim; k>=1; k--)

// A[j][k]=A[j][k]/d;
             mpfr_div( A[j][k], A[j][k], d ,GMP_RNDD);
             for (i=j+1; i<=dim; i++)
             {
// s=A[i][j]; d=b[j]*s; b[i]=b[i]-d;
             mpfr_set( s, A[i][j], GMP_RNDD);
             mpfr_mul( d, b[j], s, GMP_RNDD);
             mpfr_sub( b[i], b[i], d, GMP_RNDD);
             for (k=dim; k>=1; k--)
             {
// d= A[j][k]*s; A[i][k]=A[i][k]-d;
             mpfr_mul( d, A[j][k], s, GMP_RNDD);
             mpfr_sub( A[i][k], A[i][k], d, GMP_RNDD);
             }
     }
}
```

int mpfr_div_2ui(mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)
int mpfr_div_2si(mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)

Set z to x divided by 2 raised to y rounded in the direction rnd. Just decreases the exponent by y when z and x are identical.

The operations of finding the square and cubic root:
int mpfr_sqrt(mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)
int mpfr_sqrt_ui(mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)

Set x to the square root of y rounded in the direction rnd (set x to –0 if y is –0, to be consistent with the IEEE 754 standard). Set x to NaN if y is negative.

int mpfr_rec_sqrt(mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)

Set x to the reciprocal square root of y rounded in the direction rnd. Set x to +Inf if y is Â±0, +0 if y is +Inf, and NaN if y is negative.

int mpfr_cbrt(mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)
int mpfr_root(mpfr_t x, mpfr_t y, unsigned long int k, mpfr_rnd_t rnd)

Set x to the cubic root (resp. the kth root) of y rounded in the direction rnd. For k odd (resp. even) and y negative (including –Inf), set x to a negative number (resp. NaN). The kth root of –0 is defined to be –0, whatever the parity of k.

The operation of raising to a power:
int mpfr_pow(mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)

Set z to x raised to y, rounded in the direction rnd.

An assignment of negative values:
int mpfr_neg(mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)

An assignment of the absolute values:
int mpfr_abs(mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)

Set x to -y and |y| , respectively, rounded in the direction rnd. Just changes or adjusts the sign if x and y are the same variable, otherwise a rounding might occur if the precision of x is less than that of y.

int mpfr_dim(mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)

Set z to the positive difference of x and y, i.e., x-y rounded in the direction rnd if x>y, +0 if x≤y, and NaN if x or y is NaN.

4.3.4 Comparison Functions

All comparison functions with different type variables are described in Appendix A (Table A5).

int mpfr_cmp(mpfr_t x, mpfr_t y)

Compare x and y. Return a positive value if x>y, zero if x = y, and a negative value if x<y. Both x and y are considered to their full own precision, which may differ. If one of the operands is NaN, set the *erange* flag and return zero.

int mpfr_cmp_ui_2exp(mpfr_t x, unsigned long int y, mpfr_exp_t e)
int mpfr_cmp_si_2exp(mpfr_t x, long int y, mpfr_exp_t e)

Compare x and y multiplied by two to the power e. Similar as above.

int mpfr_cmpabs(mpfr_t x, mpfr_t y)

Compare |x| and |y|. Return a positive value if |x|>|y|, zero if |x|=|y|, and a negative value if |x| <|y|. If one of the operands is NaN, set the *erange* flag and return zero.

int mpfr_nan_p(mpfr_t x)
int mpfr_inf_p(mpfr_t x)
int mpfr_number_p(mpfr_t x)
int mpfr_zero_p(mpfr_t x)
int mpfr_regular_p(mpfr_t x)

Return nonzero if x is NaN, respectively an infinity, an ordinary number (i.e., neither NaN nor an infinity), zero, or a regular number (i.e., neither NaN, nor an infinity nor zero). Return zero otherwise.

int mpfr_sgn(mpfr_t x)

Return a positive value if x>0, zero if x=0, and a negative value if x<0. If the operand is NaN, set the *erange* flag and return zero. This is equivalent to mpfr_cmp_ui(x, 0), but more efficient.

int mpfr_greater_p(mpfr_t x, mpfr_t y)
int mpfr_greaterequal_p(mpfr_t x, mpfr_t y)
int mpfr_less_p(mpfr_t x, mpfr_t y)
int mpfr_lessequal_p(mpfr_t x, mpfr_t y)
int mpfr_equal_p(mpfr_t x, mpfr_t y)

Return nonzero if x>y, x≥y, x<y, x≤y, x=y, respectively, and zero otherwise. Those functions return zero whenever x and/or y is NaN.

int mpfr_lessgreater_p(mpfr_t x, mpfr_t y)

Return nonzero if x<y or x>y, zero otherwise (i.e., x and/or y is NaN, or x=y).

int mpfr_unordered_p(mpfr_t x, mpfr_t y)

Return nonzero if x or y is a NaN (i.e., they cannot be compared), zero otherwise.

4.3.5 Input and Output Functions

This section describes functions that perform input from an input/output stream, and functions that output to an input/output stream. Passing a null pointer for a stream to any of these functions will make them read from stdin and write to stdout, respectively.

When using any of these functions, you must include the <stdio.h> standard header before mpfr.h, to allow mpfr.h to define prototypes for these functions.

size_t mpfr_out_str(FILE *stream, int base, size_t n, mpfr_t x, mpfr_rnd_t rnd)

Output x on stream stream, as a string of digits in base base, rounded in the direction rnd. The base may vary from 2 to 62. Prints n significant digits exactly, or if n is 0, enough digits so that x can be read back exactly (see mpfr_get_str). Returns the number of characters written, or if an error occurred, return 0.

For example,

```
printf("Result for %d bits\n",precision);
  for (i=1;i<=dim; i++)
{
    printf("x[%d] = ",i);
    mpfr_out_str (stdout, 10, 120, b[i], GMP_RNDD);
    printf("\n");
  }
```

size_t mpfr_inp_str(mpfr_t y, FILE *stream, int base, mpfr_rnd_t rnd)

Input a string in base base from stream stream, rounded in the direction rnd, and put the read float in y.

This function reads a word (defined as a sequence of characters between whitespace) and parses it using mpfr_set_str. See the documentation of mpfr_strtofr for a detailed description of the valid string formats.

Returns the number of bytes read, or if an error occurred, returns 0.

MPFR has also *Formatted Output Functions* (see more detail in [5]). This class of `mpfr_printf` functions provides formatted output in a similar manner as the standard C `printf`. These functions are defined only if your system supports ISO C variadic functions and the corresponding argument access macros.

For example, the default rounding mode is rounding to nearest. The following three examples are equivalent:

```
mpfr_t x;
mpfr_init (x);
...
mpfr_printf ("%.128Rf", x);
mpfr_printf ("%.128RNf", x);
mpfr_printf ("%.128R*f", MPFR_RNDN, x);
```

4.3.6 Rounding-Related Functions

`void mpfr_set_default_rounding_mode(mpfr_rnd_t rnd)`
Set the default rounding mode to `rnd`. The default rounding mode is to nearest initially.

`mpfr_rnd_t mpfr_get_default_rounding_mode(void)`
Get the default rounding mode.

`int mpfr_prec_round(mpfr_t x, mpfr_prec_t prec, mpfr_rnd_t rnd)`
Round `x` according to `rnd` with precision `prec`, which must be an integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX` (otherwise the behavior is undefined). If `prec` is greater or equal to the precision of `x`, then new space is allocated for the significand, and it is filled with zeros. Otherwise, the significand is rounded to precision `prec` with the given direction. In both cases, the precision of `x` is changed to `prec`.

Here is an example of how to use `mpfr_prec_round` to implement Newton's algorithm to compute the inverse of `a`, assuming `x` is already an approximation to `n` bits:

```
mpfr_set_prec (t, 2 * n);
/* round a to 2n bits */
mpfr_set (t, a, MPFR_RNDN);
/* t is correct to 2n bits */
mpfr_mul (t, t, x, MPFR_RNDN);
/* high n bits cancel with 1 */
mpfr_ui_sub (t, 1, t, MPFR_RNDN);
```

```
/* t is correct to n bits */
mpfr_prec_round (t, n, MPFR_RNDN);
/* t is correct to n bits */
mpfr_mul (t, t, x, MPFR_RNDN);
/* exact */
mpfr_prec_round (x, 2 * n, MPFR_RNDN);
/* x is correct to 2n bits */
mpfr_add (x, x, t, MPFR_RNDN);
```

`int mpfr_can_round(mpfr_t b, mpfr_exp_t err, mpfr_rnd_t rnd1, mpfr_rnd_t rnd2, mpfr_prec_t prec)`

Assuming `b` is an approximation of an unknown number x in the direction `rnd1` with error at most two to the power `E(b)-err` where `E(b)` is the exponent of `b`, returns a non-zero value if one is able to round correctly x to precision `prec` with the direction `rnd2`, and 0 otherwise (including for `NaN` and Inf). This function does not modify its arguments.

If `rnd1` is `MPFR_RNDN`, then the sign of the error is unknown, but its absolute value is the same, so that the possible range is twice as large as with a directed rounding for `rnd1`.

Indeed, if `rnd` is `MPFR_RNDN`, this will check if one can round to `prec+1` bits with a directed rounding: if so, one can surely round to nearest `prec` bits, and in addition one can determine the correct ternary value, which would not be the case when `b` is near from a value exactly representable on `prec` bits.

`mpfr_prec_t mpfr_min_prec(mpfr_t x)`

Return the minimal number of bits required to store the significand of `x`, and 0 for special values, including 0. The returned value can be less than MPFR_PREC_MIN.

The function name is subject to change.

`const char * mpfr_print_rnd_mode(mpfr_rnd_t rnd)`

Return a string (`"MPFR_RNDD"`, `"MPFR_RNDU"`, `"MPFR_RNDN"`, `"MPFR_RNDZ"`, `"MPFR_RNDA"`) corresponding to the rounding mode `rnd`, or a null pointer if `rnd` is an invalid rounding mode.

MPFR library has other functions for you to use: *Special Functions, Input and Output Functions, Integer Related Functions, Miscellaneous Functions, Exception Related Functions,* and others. For more detail please see [5].

4.4 Gauss Method with MPFR

In listing 4.1 we considered program for solution of linear system equations by Gauss method with MPFR. Since the matrix computations are well parallelized, it actually use multiple processors to solve them. So we solved this problem on Inparcom-256 [2, 3] and cluster system SKIT-3 of the Glushkov Institute of Cybernetics of NAS of Ukraine [9].

SKIT-3 is 127-node cluster on multicore processors (75 nodes on the dual-core Intel Xeon 5160, 52 node on quad-core Intel Xeon processor 5345) 6500 Gflops peak performance (in fact, confirmed by the 5317 Gflops), which naturally leads to improved time computing characteristics.

Compilling and running this program:

```
gcc gaus_mpfr.c -o gaus_mpfr -lmpfr -lgmp
./gaus_mpfr
```

The solution of linear algebraic Equations (1.1) using the GMP and MPFR is presented in Table 4.1

Table 4.1 Comparative table of solutions of linear system Equations (1.1) using the GMP and MPFR

Library and prec	Solution
GMP 64	x[1] = 0.666206705783446092108e13
	x[2] = -0.401683454957582676541e13
	x[3] = -0.166551676445818359352e13
	x[4] = 0.979715743799817084185e12
MPFR 64	x[1] = 6.660277599604241456698e12
	x[2] = -4.01575561152525328255e12
	x[3] = -1.665069399900628669l5e12
	x[4] = 9.79452588177725942909e11
GMP 128	x[1] = 0.666216216216160673879800994796992967908le13
	x[2] = -0.401689189189072350695213341235008990527 6e13
	x[3] = -0.166554054053997005189400500416908076791 9e13
	x[4] = 0.979729729730279707257486048680021238326 6e12
MPFR 128	x[1] = 6.662162162161606738798178427262421963000e12
	x[2] = -4.016891891890723506952234995452916133508e12
	x[3] = -1.665540540539970051894047123992203844670e12
	x[4] = 9.797297297302797072575108250465642255839e11
GMP 256	x[1]=0.666216216216160673879806449043057216803065823955356931480210076337566433911780 8e13
	x[2]=-0.401689189189072350695216629824547728755978173750296346904869738993135940274936 3e13
	x[3]= -0.166554054053997005189401863978424139235407407550574892875886152473574877785027e13
	x[4] =0.979729729730279707257494069630115723559737931272217917962992829480821664504459 5e12

Table 4.1 *(continued)*

MPFR 256	x[1] = 6.662162162161606738798064490430572168030658239553569314802100153119523054579635e12
	x[2] = -4.016891891890723506952166298245477287559781737502963469048697021982803628262837e12
	x[3] = -1.665540540539970051894018639784241392354074075505748928758861372171713456670790e12
	x[4] = 9.797297297302797072574940696301157235597379312722179179629927397372714755689207e11

Table 4.2 shows the dependence of results and error of solution of system of linear equations (1.1) from precision.

Table 4.2 Results and error solutions of linear system equations (1.1) using MPFR

Solution	Error (Ax-b)
Precision=128	
X_1=6.662162162161606738798178427262421963000970581959906671384863408036380 8..×10^{12}	3.4×10^{-27}
X_2=-4.016891891890723506952234995452916133507715993963662953671750477369073 4..×10^{12}	2.2×10^{-28}
X_3=-1.665540540539970051894047123992203844669182755438394154734113583593568 8..×10^{12}	3.8×10^{-27}
X_4= 9.797297297302797072575108250465642255839411054081475218723723541458525 6..×10^{12}	7.1×10^{-28}
Precision=256	
X_1= 6.662162162161606738798064490430572168030658239553569314802100153119523 0..×10^{12}	1.1×10^{-65}
X_2=-4.016891891890723506952166298245477287559781737502963469048697021982803 6..×10^{12}	8.8×10^{-67}
X_3=-1.665540540539970051894018639784241392354074075505748928758861372171713 4..×10^{12}	8.2×10^{-66}
X_4= 9.797297297302797072574940696301157235597379312722179179629927397372714 7..×10^{12}	9.0×10^{-67}
Precision=512	
X_1= 6.662162162161606738798064490430572168030658239553569314802100940649855 9..×10^{12}	5.4×10^{-143}
X_2=-4.016891891890723506952166298245477287559781737502963469048697496817269 0..×10^{12}	2.3×10^{-143}
X_3=-1.665540540539970051894018639784241392354074075505748928758861569054296 6..×10^{12}	4.7×10^{-144}
X_4= 9.797297297302797072574940696301157235597379312722179179629928555505557 2..×10^{12}	5.5×10^{-144}

Note that increasing the digit doubled [7], the error is reduced by half almost linearly. The format bit 512-bit is not the limit, it is possible to use the bit in 1024, 2048, and more. For example, the error of solving this problem by using the bit size 40 980 will be equal to 10^{-12324}. Naturally the time taken for these calculations toward classical computer arithmetic is significantly increased, but the maximum bit which is used in the classical floating-point arithmetic 128 bits, which can guarantee the accuracy of solutions of this linear system equations only as 10^{-27}.

References

All publications about MPFR you can find on website http://www.mpfr.org/pub.html. So handbook [1] aims to provide a complete overview of modern floating-point arithmetic, including a detailed treatment of the newly revised (IEEE 754-2008)

standard for floating-point arithmetic. Presented throughout are algorithms for implementing floating-point arithmetic as well as algorithms that use floating-point arithmetic. They are illustrated, whenever possible, by a corresponding program so that the techniques presented can be put directly into practice in actual coding or design. In [6] by Richard Brent and Paul Zimmermann presented algorithms for performing arithmetic, and their implementation on modern computers. It collects in the same document all state-of-the-art algorithms in multiple precision arithmetic (integers, integers modulo n, floating-point numbers).

1. Brent, R., Zimmermann, P.: Cambridge Monographs on Computational and Applied Mathematics, vol. (18). Cambridge University Press (2010)
2. Inparcom (family of intelligent parallel computers), http://geopoisk.com/inparcom/eng/index_eng.htm
3. Khimich, A.N., Molchanov, I.N., Mova, V.I., et al.: Numerical program software of intelligent computer Inparcom. Naukova dumka, Kiev (2007)
4. MPFI Library, http://perso.ens-lyon.fr/nathalie.revol/software.html
5. MPFR Library, http://www.mpfr.org/
6. Muller, J.-M., et al.: Handbook of Floating-Point Arithmetic. Birkhäuser, Basel (2009)
7. Nikolaevskaya, E.A., Chistyakova, T.V.: Software and algorithmic methods to improve the accuracy of computer solutions. Cybernetics and Systems Analysis 6, 172–176 (2009)
8. Revol, N., Rouillier, F.: Motivations for an arbitrary precision interval arithmetic and the MPFI library. Kluwer Academic Publishers (2005), http://perso.ens-lyon.fr/nathalie.revol/publis/RR05.pdf (accessed August 21, 2011)
9. Supercomputer SKIT-3 of V.M. Glushkov Institute of Cybernetics, http://icybcluster.org.ua/index.php?lang_id=3&menu_id=1

Chapter 5
GMP for C++

Elena A. Nikolaevskaya, Alexandr N. Khimich, and Tamara V. Chistyakova

Abstract. This chapter describes the C++ class-based interface for GMP. All GMP C language types and functions can be used in C++ programs, since ‘gmp.h’ has extern "C" qualifiers, but the class interface offers overloaded functions and operators which may be more convenient. Due to the implementation of this interface, a reasonable recent C++ compiler is required, one supporting namespaces, partial specialization of templates and member templates. Matrix multiplication is one of the main operations. So here we consider a program in C++ for matrix

Elena A. Nikolaevskaya
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: elena_nea@ukr.net

Alexandr N. Khimich
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: khimich_ic@mail.ru

Tamara V. Chistyakova
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: dept150@insyg.kiev.ua

E. Nikolaevskaya et al.: Programming with Multiple Precision, SCI 397, pp. 71–96.
springerlink.com

multiplication with the use of mpf_class of GMP library. Also we describe LU-decomposition for solution of SLAEs and its program text with using mpf_class.

5.1 C++ Interface

All the C++ classes and functions are available with

```
#include <gmpxx.h>
```

Programs should be linked with the `libgmpxx` and `libgmp` libraries. For example,

```
g++ mycxxprog.cc -lgmpxx -lgmp
```

The defined classes are

`mpz_class` for integer
`mpq_class` for rational number
`mpf_class` for floating point number

The standard operators and various standard functions are overloaded to allow arithmetic with these classes. It is very easy! For example,

```
int main (void)
{
mpz_class a, b, c;

a = 1234;
b = "-5678";
c = a+b;
cout << "sum is " << c << "\n";
cout << "absolute value is " << abs(c) << "\n";

return 0;
}
```

An important feature of the implementation is that an expression like $a=b+c$ results in a single call to the corresponding `mpz_add()`, without using a temporary for the $b+c$ part. Expressions which by their nature imply intermediate values, like $a=b*c+d*e$, still use temporaries though.

The classes can be easily intermixed in expressions, as can the classes and the standard types can be `long`, `unsigned long` and `double`. Smaller types like `int` or `float` can also be intermixed, since C++ will promote them.

Note that `bool` is not accepted directly, but it must be explicitly cast to an `int` first. This is because C++ will automatically convert any pointer to a `bool`, so if GMP accepted `bool` it would make all sorts of invalid class and pointer combinations compile but almost certainly will not do anything sensible.

Conversions come back from the classes to standard C++ types which aren't done automatically, instead of member functions like `get_si` that are provided.

Also there are no automatic conversions from the classes to the corresponding GMP C types, instead of a reference to the underlying C object which can be obtained with the following functions,

```
mpz_t mpz_class::get_mpz_t ()
mpq_t mpq_class::get_mpq_t ()
mpf_t mpf_class::get_mpf_t ()
```

These can be used to call a C function which does not have a C++ class interface. For example, set a to $b+c$

```
mpf_class a, b, c;
...
mpf_add(a.get_mpf_t(),b.get_mpf_t(), c.get_mpf_t());
```

In the other direction, a class can be initialized from the corresponding GMP C type, or assigned to if an explicit constructor is used. In both cases this makes a copy of the value, it doesn't create any sort of association. For example,

```
mpf_t x;
// ... init and calculate x ...
mpf_class y(x);
mpf_class z;
z = mpf_class (x);
```

When an expression requires the use of temporary intermediate `mpf_class` values, like $f=g*h+x*y$, those temporaries will have the same precision as the destination f. Explicit constructors can use it if this does not suit.

`mpf_class::mpf_class (type x)`
`mpf_class::mpf_class (type x, unsigned long prec)`

Construct an `mpf_class`. Any standard C++ type can be used, except `long long` and `long double`, and any of the GMP C++ classes can be used.

If `prec` is given, the initial precision is that value, in bits. If `prec` is not given, then the initial precision is determined by the type of `x` that is given. An

mpz_class, mpq_class, or C++ builtin type will give the default mpf precision (see Chapter 2). The mpf_class or expression will give the precision of that value. The precision of a binary expression is the higher of the two operands.

```
mpf_class x(1.5);         // default precision
mpf_class x(1.5, 500);    // 500 bits (at least)
mpf_class f(x);           // precision of x
mpf_class f(abs(x));      // precision of x
mpf_class f(-g, 1000);    // 1000 bits (at least)
mpf_class f(x+y);          // greater of precisions
// of x and y
```

void mpf_class::mpf_class (const char *s)
void mpf_class::mpf_class (const char *s, unsigned long prec, int base = 0)
void mpf_class::mpf_class (const string& s)
void mpf_class::mpf_class (const string& s, unsigned long prec, int base = 0)

Construct an mpf_class converted from a string using mpf_set_str (see Chapter 2). If prec is given, the initial precision is that value, in bits. If not, the default mpf precision (see Section 2.3.1) is used.

If the string is not a valid float, an std::invalid_argument exception is thrown. The same applies to operator =.

mpf_class& mpf_class::operator= (type x)

Convert and store the given x value to an mpf_class object. The same types are accepted as for the constructors above.

Note that operator= only stores a new value, it does not copy or change the precision of the destination, instead the value is truncated if necessary. This is the same as mpf_set etc. Note in these particular means for mpf_class a copy constructor is not the same as a default constructor plus assignment.

```
mpf_class x (y);   // x created with precision of y
mpf_class x;  // x created with default precision
x = y;     // value truncated to that precision
```

Applications using templated code may need to be careful about the assumptions the code makes in this area, when working with mpf_class values of various different or non-default precisions. For instance implementations of the standard complex template have been seen in both styles above, though of course complex is normally only actually specified for use with the built-in float types.

mpf_class abs (mpf_class x)
mpf_class ceil (mpf_class x)

```
  int cmp (mpf_class x, type y)
  int cmp (type x, mpf_class y)

  bool mpf_class::fits_sint_p (void)
  bool mpf_class::fits_slong_p (void)
  bool mpf_class::fits_sshort_p (void)
  bool mpf_class::fits_uint_p (void)
  bool mpf_class::fits_ulong_p (void)
  bool mpf_class::fits_ushort_p (void)

  mpf_class floor (mpf_class x)
  mpf_class hypot (mpf_class x, mpf_class y)

  double mpf_class::get_d (void)
  long mpf_class::get_si (void)
  string mpf_class::get_str(mp_exp_t& exp, int base =
10, size_t digits = 0)
  unsigned long mpf_class::get_ui (void)
  int mpf_class::set_str (const char *str, int base)
  int mpf_class::set_str(const string& str, int base)
  int sgn (mpf_class x)
  mpf_class sqrt (mpf_class x)
  mpf_class trunc (mpf_class x)
```

These functions provide a C++ class interface to the corresponding GMP C routines.

`cmp` can be used with any of the classes or the standard C++ types, except `long long` and `long double`.

The accuracy provided by `hypot` is not currently guaranteed.

```
  mp_bitcnt_t mpf_class::get_prec ()
  void mpf_class::set_prec (mp_bitcnt_t prec)
  void mpf_class::set_prec_raw (mp_bitcnt_t prec)
```

Get or set the current precision of an `mpf_class`.

The use of GMP library for C++ takes the less time-consuming. Example of a program of adding two variables in C++ is shown in Listings 5.1. In the left side you can see program without GMP, and in the right side with it. The font rows of program with function of GMP are in bold.

Listing 5.1. Adding the two variables in C + +

```
#include <string>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <stdlib.h>

using namespace std;

//Set precision and type define mpf_class
int main()
{

// Declare and initialize variables
double a,b,c;

// Continuing program
int i;
a=0;
b=1;

for (i = 2; i<=10;
++i)
{
c=a+b;
a=b;
b=c;
}
cout << "b=" << b;
cout << endl;
return 0;
}
```

```
#include <string>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <stdlib.h>
#include <gmpxx.h>
using namespace std;

const int PRECISION=256;
typedef mpf_class BASE_TYPE;
int main()
{
mpf_set_default_prec(PRECISION);
cout << "Precision " <<
mpf_get_default_prec() << endl;

BASE_TYPE a,b,c;
a.set_prec(PRECISION);
b.set_prec(PRECISION);
c.set_prec(PRECISION);

int i;
a=0.0;
b=1.0;

for (i = 2; i<=10; ++i)
{
c=a+b;
a=b;
b=c;
}
cout << "b=" << b;
cout << endl;
return 0;
}
```

5.2 Matrix Multiplication

From the set of matrix operations, one the most fundamental is the matrix-vector multiplication and multiplication of two matrices.

Diversity of computational schemes' construction. If an algorithm for solving a problem is created on the basis of the solution method it has to take into account a specific nature of the problem (for example, whether the matrix is banded or sparsed), the computational schemes must take into account both structure and architecture of computer on which they are to be implemented. For the SISD-architecture (mono-processor computers) one type of computational scheme should be developed, while for the MIMD-computer (computer with multiple instructions and multiple date) another type of computational scheme should be developed, and for vector-pipeline computers - the third type should be developed, and so on.

Let us consider, for example, the operation of multiplication of two square matrices of the n-th order [4].

So, let it be necessary to multiply two square matrices A and B:

$$C = A \times B.$$

It is known that

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \times b_{k,j}, \; i = 1, \ldots, n, \;\; j = 1, \ldots, n. \tag{5.1}$$

Computation by formula (5.1) assumes the utilization of three loops with indices *i, j, k* (inner, middle and outer indices).

Calculations by formula (5.1) suggest the use of three cycles with indices i, j, k (inner, middle and external indexes):

```
C←0
for i=1,…,n
{
for j=1,…,n
{
for k=1,…,n
c_ij ← c_ij + a_ik b_kj
}
}
```

By fixing the inner index and varying indices *i, j, k* one can obtain six versions of computational schemes. So the number of arithmetical operations for each version is equal to $2n^3$.

Computational schemes for *ijk*, *jik* can be graphically represented at Fig. 5.1-5.2.

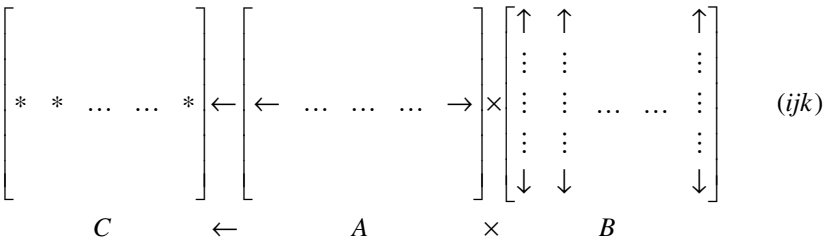

Fig. 5.1 Computational scheme of matrix multiplications for *ijk*

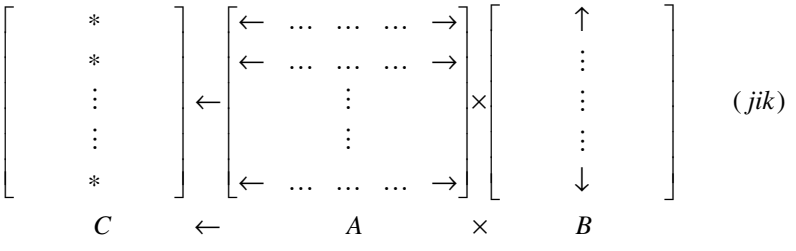

Fig. 5.2 Computational scheme of matrix multiplications for *jik*

Both algorithms are represented in the form of an inner product. Inner loop evaluates the element of the matrix *C*. In the middle loop of the computational scheme *ijk* the *i-th* row of the matrix *A* is multiplied consequently by all columns of *B*. In the computational scheme *ijk* all rows of *A* are multiplied one-by-one by the *j-th* column of the matrix *B*.

The *kij* and *kji* computational schemes are represented in the form of outer products: the *k-th* row of *B* is multiplied by *i-th* element of the *k-th* row of the matrix *A* and added to the *i-th* row of the matrix *C*. Similarly, the *k-th* column of the matrix *A* is multiplied by the *j-th* element of the *k-th* of the matrix *B* and added to the *j-th* column of the matrix *C*. The principle operation that is performed in the inner loop of both computational schemes is the vector's multiplication by scalar and summation of the result with other vector. Each execution of the middle loop (i.e., one step of the outer loop) results changes of all the elements of the matrix *C* being evaluated (see Fig. 5.3-5.4):

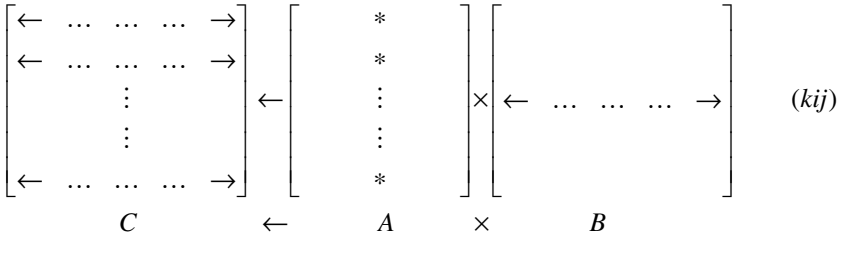

Fig. 5.3 Computational scheme of matrix multiplications for *kij*

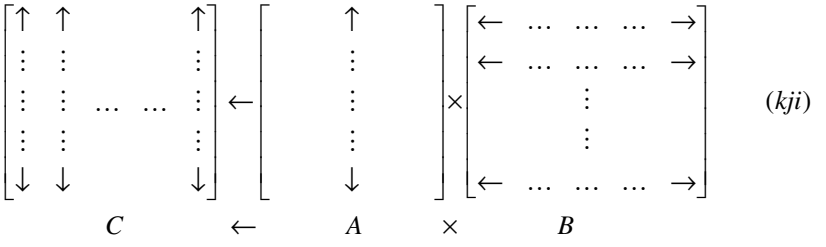

Fig. 5.4 Computational scheme of matrix multiplications for *kji*

And, finally, in the *ikj* and *jki* computational schemes every execution of the middle loop (one step of the outer loop) results in the multiplication of all the vectors by scalar and their summation with the same vector of the matrix *C* (Fig. 5.5-5.6).

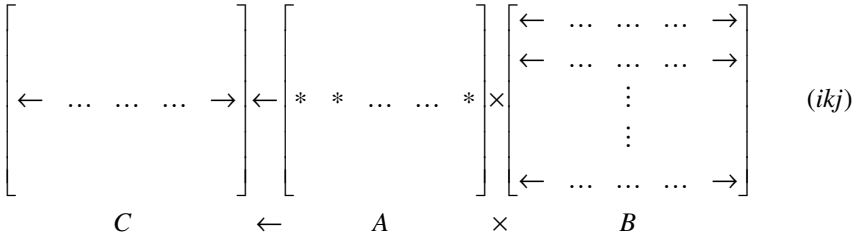

Fig. 5.5 Computational scheme of matrix multiplications for *ikj*

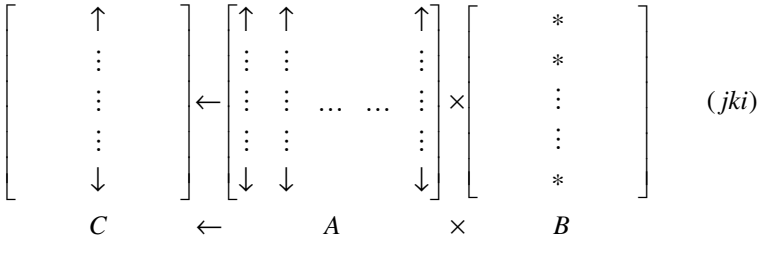

Fig. 5.6 Computational scheme of matrix multiplications for *jki*

In the *ikj* computational scheme the *k-th* row of the matrix *B* is multiplied by the *k-th* element of the *i-th* row of the matrix *A* and added to the *i-th* row of the matrix *C*. In the *jki* computational scheme the *k-th* column of the matrix *A* is multiplied by the *k-th* element of the *j-th* column of the matrix *B,* the result is added to the *j-th* column of *C*. In both computational schemes only elements of one row (one column) of the matrix *C* change during the execution of the middle loop.

Times required for the implementation of these six computational schemes on the same computer can significantly differ from each other. In programming and implementing them on a computer one should take into account peculiarities of the following: the computer (its structure and architecture), operating systems, programming languages, standard program libraries, etc., as well as the specific nature of problems are being solved (order of matrices, requirements to the accuracy of computation, execution time and so on).

Time required for the multiplication of two matrices of the orders n=1000 and n=1500 on the AMD- Athlon (tm) 64 computer (the clock frequency is 1.81 GHz) is given in Table 5.1. Programs are written in C. From the table one can see that the most economical computational scheme is *kij*.

Table 5.1 Time characteristics for the matrix multiplication problem

Computational scheme for the matrix multiplication	Execution time (in sec.)	
	n = 1000	n = 1500
ijk	21	98
ikj	14	48
jik	17	66
jki	14	48
kij	12	46
kji	15	49

Program for multiplication of two square matrices is shown in Listing 5.2.

Listing 5.2. Multiplication of two square matrices

c_mult_double.cpp

```
#include <string>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
#include "other/nr.h"
#include <stdio.h>
#include <time.h>
#include "c_funk_d.h"
using namespace std;

typedef double BASE_TYPE;
```

```
typedef NRVec<BASE_TYPE> Vec1d;

int main(int argc, char **argv)
{
     int n = 1000, i, j;
Vec1d matrixa(n*n), matrixb(n*n), matrixc(n*n);
     timeval t1, t2;

/* Initialization of matrices */
     init_matrixa(matrixa, n);
     init_matrixb(matrixb, n);

     gettimeofday(&t1, NULL);

/* Matrix multiplication */
     matrix_mult_matrix(matrixa, matrixb, matrixc, n);

     gettimeofday(&t2, NULL);

cout << endl << "Problem: matrix_mult_matrix" << endl;
     cout << endl << "Matrix C:" << endl;
     cout << "n = " << n << endl;

     for (i = 0; i < 4; i++)
     {
            for (j = 0; j < 4; j++)
            {
            cout.precision(100);
            cout << setw(12) << matrixc[i * n + j] << endl;
            }
     }
     cout << "Total time of solving " << t2.tv_usec -
t1.tv_usec << " msec" << endl;

return 0;
}

template<typename T>
void init_matrixa (NRVec<T> &matrixa, int n)
{
     int i, j;
     for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                   matrixa[i * n + j] = std::max(i,j);
```

```
}

template<typename T>
void init_matrixb (NRVec<T> &matrixb, int n)
{
     int i, j;
     for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                   matrixb[i * n + j] = 1.;
}

template<typename T>
void matrix_mult_matrix(NRVec<T> &a, NRVec<T> &b, NRVec<T> &c,
int n)
{
int i, j, k;

for (i = 0;i < n; i++)
{
            for (j = 0;j < n;j++)
            c[i * n + j]=0.0;
                   for (j = 0;j < n; j++)
            {
            for (k = 0;k < n;k++)
            c[i * n + j] += a[i*n + k]*b[k * n + j];
                   }
     }
     }
```

Transform the previous program of multiplication of two matrices for working with arbitrary precision in the calculations, using `mpf_class` of the GMP library (Listing 5.3).

For this purpose at first one need to specify the header file `gmpxx.h`, then identify `mpf_class` and specify the accuracy `PRECISION=64`:

```
#include <gmpxx.h>

const int PRECISION=64;
typedef mpf_class BASE_TYPE;
typedef NRVec<BASE_TYPE> Vec1d;
```

Then in the `main` program you must set the precision of the above using

```
mpf_set_default_prec(PRECISION);
```

and set the current precision for all the variables:

```
for(i=0; i < n*n; i++)
{
         matrixa[i].set_prec(PRECISION);
         matrixb[i].set_prec(PRECISION);
         matrixc[i].set_prec(PRECISION);
     }
```

Output of the resulting matrix with precosion has the form

```
for (i = 0; i < 4; i++)
{
             for (j = 0; j < 4; j++)
             {
             cout.precision(100);
             cout << setw(12) << matrixc[i * n + j] << endl;
             }
}
```

Among the rest of the programs using `mpf_class` in fact there is no difference between the programs using a `double`, which makes it convenient to use GMP in C++.

Listing 5.3 Multiplication of two square matrices with mpf_class

c_mult_gmp.cpp

```
#include <string>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
#include "other/nr.h"
#include <stdio.h>
#include <time.h>
#include "c_funk_gmp.h"
#include <gmpxx.h>

using namespace std;
```

```
  const int PRECISION=64;
  typedef mpf_class BASE_TYPE;
  typedef NRVec<BASE_TYPE> Vec1d;

  int main(int argc, char **argv)
  {
  int n = 1000, i, j;
  Vec1d matrixa(n*n), matrixb(n*n), matrixc(n*n);
  timeval t1, t2;

       mpf_set_default_prec(PRECISION);
  cout << "Precision " << mpf_get_default_prec() << endl;

  for(i=0; i < n*n; i++)
  {
              matrixa[i].set_prec(PRECISION);
              matrixb[i].set_prec(PRECISION);
              matrixc[i].set_prec(PRECISION);
       }
  /* Initialization of matrices */
       init_matrixa(matrixa, n);
       init_matrixb(matrixb, n);

       gettimeofday(&t1, NULL);

  /* Matrix multiplication */
       matrix_mult_matrix(matrixa, matrixb, matrixc, n);

       gettimeofday(&t2, NULL);
  cout << endl << "Problem: matrix_mult_matrix" << endl;
  cout << endl << "Matrix C:" << endl;
  cout << "n = " << n << endl;

  for (i = 0; i < 4; i++)
  {
              for (j = 0; j < 4; j++)
              {
              cout.precision(100);
              cout << setw(12) << matrixc[i * n + j] << endl;
              }
       }
  cout<<"Total  time  of  solving"<<t2.tv_usec-t1.tv_usec<<"msec"
<<endl;
```

```
  return 0;
  }

  template<typename T>
  void init_matrixa (NRVec<T> &matrixa, int n)
  {
       int i, j;
       for (i = 0; i < n; i++)
              for (j = 0; j < n; j++)
                     matrixa[i * n + j] = std::max(i,j);
  }

  template<typename T>
  void init_matrixb (NRVec<T> &matrixb, int n)
  {
       int i, j;
       for (i = 0; i < n; i++)
              for (j = 0; j < n; j++)
                     matrixb[i * n + j] = 1.;
  }

  template<typename T>
  void matrix_mult_matrix(NRVec<T> &a, NRVec<T> &b, NRVec<T> &c,
int n)
  {
  int i, j, k;

  for (i = 0;i < n; i++)
       {
              for (j = 0;j < n;j++)
              c[i * n + j]=0.0;
              for (j = 0;j < n; j++)
              {
              for (k = 0;k < n;k++)
              c[i * n + j] += a[i*n + k]*b[k * n + j];
                     }
       }
  }
```

Time required for computing the product of two matrices, is presented in Table 5.2. As can be seen, the use of GMP library requires more time than the `double` digit. However, in most practical problems such accuracy is a must.

Table 5.2 Comparison of time multiplication of two matrices

Matrix size	C++	C++, GMP
	`double` (53)	64
1000	17.31	174.9
2000	228.69	1000.1

5.3 Again about Gauss Method

Algebraic basis for the Gauss elimination is a statement that for square non-linear matrix A of order n there exists unique lower triangular matrix L with units on the principal diagonal and unique upper triangular matrix U such that $LU = A$ and $\det(A) = u_{11}u_{22}\ldots u_{nn}$ (see also [2, 4, 6, 7]).

During the solving of system with non-singular matrix by Gauss method the following three subproblems can be separated:

1) LU-decomposition of system's matrix

$$A = LU; \tag{5.2}$$

2) the solving of SLAEs with lower triangular matrix решение (this problem is often referred to as forward substitution):

$$Ly = b; \tag{5.3}$$

3) the solving of SLAEs with upper triangular matrix (backward substitution):

$$Ux = y. \tag{5.4}$$

Let us have at our disposal a computer whose hardware or software enables to carry out (with little loss in performance) the evaluation of a scalar product in double precision by accumulating and rounding-off only the final result. In this case for solving the linear algebraic system $Ax = b$ with non-symmetric general non-singular matrix A which can be fully stored in the operating memory of a computer, it is advisable to apply a method which is based on the direct decomposition $A = LU$, when L is a low triangular and U is an upper triangular matrices:

$$Ax = LUx = b. \tag{5.5}$$

This decomposition, if it exists, is unique within the accuracy of the choice of diagonal elements of matrices L and U, namely, one can set either all $l_{ii} = 1$ or all $u_{ii} = 1$. Consider in more detail such a case where U is an upper triangular matrix with units on the main diagonal. Formulas for the evaluation of elements l_{ij} and u_{ij} can easily be driven by equating corresponding elements to each other in the matrix equality

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & \dots & u_{1n} \\ 0 & 1 & \dots & u_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix}. \quad (5.6)$$

Such a decomposition corresponds to that obtained in the unique division scheme of the Gauss's method.

Elements of matrices L and U at the *r-th* step $r = 1, 2, ..., n$ of the computational process are evaluated in the following order: at first, elements of the *r-th* column of the matrix L:

$$l_{ir} = a_{ir} - \sum_{k=1}^{r-1} l_{ik} u_{kr}, \qquad i = r, ..., n, \quad (5.7)$$

further – elements of the *r-th* row of the matrix **U**

$$u_{ri} = (a_{ri} - \sum_{k=1}^{r-1} l_{rk} u_{ki}) \frac{1}{l_{rr}}, \; i = r + 1, ..., n. \quad (5.8)$$

Consider now (5.5). By setting $Ux = y$, we get the equation

$$Ly = b \quad (5.9)$$

with a low triangular matrix solution that can be evaluated by formulas

$$y_i = (b_i - \sum_{k=1}^{i-1} l_{ik} y_k) \frac{1}{l_{ii}}, \; i = 1, 2, ..., n. \quad (5.10)$$

The unknown x can be found from the system

$$Ux = y \quad (5.11)$$

with upper triangular matrix by evaluating

$$x_i = y_i - \sum_{k=i+1}^{n} u_{ik} x_k, \; i = n, n\text{-}1, ..., 1. \quad (5.12)$$

The LU-decomposition scheme described above can be implemented only in cases where all its elements $l_{ii} \neq 0$. Besides, the proximity of these elements to zero can result in the considerable loss in the accuracy when computing a solution. To avoid this, the solving of linear algebraic system by the LU-decomposition method should be carried out by choosing the pivot with the largest modulus, for example, in column. Toward this end the following operations are performed at each *r-th* step. Elements l_{ir} (see (5.7)) are evaluated and moved into positions of a_{ir}, $i = r, ..., n$. If $\max_{i \geq r} |l_{ir}| = |l_{pv}|$, then rows r and p are interchanged, i.e. the row index p is replaced by r for all the elements, and vice versa. In so doing numbers of rows being interchanged should be memorized. Elements u_{ri} are evaluated by formulas (5.8) and moved into positions of a_{ri}, $i = r+1, ..., n$.

To increase the accuracy of computations it is advisable to evaluate all the scalar products in formulas (5.7), (5.8), (5.10) and (5.12) by accumulating and rounding-off only the final result. An implementation of the LU-decomposition on the computer does not provide the evaluation of a scalar product by accumulating and

rounding-off only the final result which has no advantages over the method of Gaussian elimination.

LU-decomposition of system's matrix (5.2) consists of (n–1) steps. At the s-th (s = 1, 2, ..., n–1) step a diagonal block of order n–s+1 is transformed and located in lower right-hand corner of the matrix $A^{(s-1)}$ is. Element $a_{ss}^{(s-1)}$ is referred as to a pivotal element of the s-th step. For the successful implementation of Gauss method all pivotal elements should be nonzero. Besides, the proximity of pivotal elements to zero can result in large error in the computed solution. In order to ensure the fulfillment of conditions $\left|a_{ss}^{(s-1)}\right| \geq \varepsilon > 0$ for all s = 1, 2, ..., n–1 the largest modulus (pivotal) elements should be chosen at each step either in the entire block of matrix $A^{(s-1)}$ being transformed or in the first row of this block or in its first column. By permutation of rows and/or columns of matrix $A^{(s-1)}$ the pivotal element is placed on the position of the element $a_{ss}^{(s-1)}$.

The solving of a linear algebraic system by LU-decomposition method [4] requires n divisions (if values of $1/l_{ii}$ rather than l_{ii} are stored), $\frac{1}{6}\left(2n^3 - 3n^2 + n\right)$ multiplications and $\frac{1}{6}\left(2n^3 - 3n^2 + n\right)$ additions. The implementation of the method requires $\left(n^2 + n + 1\right)$ machine words (without counting machine words in the program).

Since it is known that a determinant of a product of two square matrices is equal to the product of determinants of matrices that are being multiplied, from (5.6) it immediately follows that:

$$\det A = \det L \ \det U = (-1)^s \prod_{i=1}^{n} u_{ii}, \tag{5.13}$$

where s is the number of permutations of rows in the process of solving.

Example. Solve the linear algebraic system (3.12) by LU –decomposition method.

Solution. Elements of matrices L and U are evaluated by formulas (5.7), (5.8). As a result we get

$$L = \begin{pmatrix} 5 & 0 & 0 \\ 4 & 3{,}2 & 0 \\ 1 & -2{,}2 & 2{,}1875 \end{pmatrix}, \qquad U = \begin{pmatrix} 1 & 0{,}2 & 0{,}4 \\ 0 & 1 & -0{,}1875 \\ 0 & 0 & 1 \end{pmatrix}.$$

By using formulas (5.10) we find a solution to the linear algebraic system (5.9)

$$y_1 = 1{,}6, \qquad y_2 = -1{,}375, \quad y_3 = 2,$$

and by formulas (5.13) - a solution to system (5.11)

$$x_3 = 2, \qquad x_2 = -1, \qquad x_1 = 1.$$

Consider a program for solving of SLAEs by LU decomposition in C++ with GMP (Listing 5.4). For solving of SLAEs using LU decomposition of a matrix

transform function `ludcmp()` and `lubksb()` from [6], using `mpf_class` of GMP.

Listing 5.4 Solution of SLAEs by LU decomposition with mpf_class

LU_mpf.cpp

```
#include <string>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
#include "other/nr.h"
#include "lud.h"
#include "Matr.h"
#include <gmpxx.h>

using namespace std;
const int PRECISION=64;
typedef mpf_class BASE_TYPE;
typedef NRMat<BASE_TYPE> Mat2d;
typedef NRVec<BASE_TYPE> Vec1d;
const int MATR = 1;
const int FIXED = 2;
void show_usage();

int main(int argc, char *argv[])
{
mpf_set_default_prec(PRECISION);
cout << "Precision " << mpf_get_default_prec() << endl;

int i,j,k,l;
     string txt;
     double d;

if(argc != 3)
{
            show_usage();
            exit(-1);
}
int mat_t = atoi(argv[1]);
int n = atoi(argv[2]);
```

```
  if(n == 0)
  {
  cerr << "ERROR: Incorrect matrix size" << std::endl;
       show_usage();
       exit(-1);
  }
  if(mat_t != MATR && mat_t != FIXED)
  {
  cerr << "ERROR: unknown matrix init function. Correct one is 1
or 2." << std::endl;
  show_usage();
  exit(-1);
  }
       if(mat_t == FIXED)
  {
  cerr << "Supplied matrix size is ignored. Matrix size = 4\n";
  n = 4;
  }
  timeval tim1, tim2;

  Vec_INT indx(n),jndx(n);
  Mat2d A(n,n), A1(n,n);
  Vec1d b(n), b1(n), y(n), z(n);

       for(i=0; i < n; i++)
  {
              for(j=0; j < n; j++)
              {
                     A[i][j].set_prec(PRECISION);
                     A1[i][j].set_prec(PRECISION);
              }
              b[i].set_prec(PRECISION);
              b1[i].set_prec(PRECISION);
              y[i].set_prec(PRECISION);
              z[i].set_prec(PRECISION);
       }
       if(mat_t == MATR)
       {
              init_matrices1(15, A, b);
       init_matrices1(15, A1, b);
       }
       else
  {
              init_matrices(A, b);
```

```
                init_matrices(A1, b);
        }

   gettimeofday(&tim1, NULL);

   // Perform the decomposition
   ludcmp(A,indx,d);
   lubksb(A,indx,b);

   gettimeofday(&tim2, NULL);

   cout << endl << "Solution vector" << endl;
   for (l = 0; l < n; l++)
                cout << setw(12) << b[l] << "   ";
                cout << endl;

                cout << "Total time of solving " << tim2.tv_usec -
tim1.tv_usec << " msec" << endl;

   return 0;
   }

   void show_usage()
   {
   cout << "Usage: lud MATRIX SIZE " << endl;
   }
```

Lud.h

```
   #include <cmath>
   #include <gmpxx.h>
   #include "other/nr.h"
   using namespace std;

   template<typename T>
   void ludcmp(NRMat<T> &a, Vec_O_INT &indx, DP &d)
   {
        const T TINY=1.0e-99;
        int i,j,k, imax;
        T big,dum,sum,temp;

        int n=a.nrows();
        NRVec<T> vv(n);
        d=1.0;
```

```
        for (i=0;i<n;i++) {
                big=0.0;
                for (j=0;j<n;j++)
                if ((temp=abs(a[i][j])) > big)
                        big=temp;
                if (big == T(0.0))
    NR::nrerror("Singular matrix in routine ludcmp");
                vv[i]=T(1.0)/big;
        }
        for (j=0;j<n;j++) {
                for (i=0;i<j;i++) {
                        sum=a[i][j];
                        for (k=0;k<i;k++)
                        sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
                }
                big=0.0;
                for (i=j;i<n;i++) {
                        sum=a[i][j];
                        for (k=0;k<j;k++)
                        sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
                        if ((dum=vv[i]*abs(sum)) >= big) {
                                big=dum;
                                imax=i;
                        }
                }
                if (j != imax) {
                        for (k=0;k<n;k++) {
                                dum=a[imax][k];
                                a[imax][k]=a[j][k];
                                a[j][k]=dum;
                        }
                        d = -d;
                        vv[imax]=vv[j];
                }
                indx[j]=imax;
                if (a[j][j] == T(0.0))
                    a[j][j]=TINY;
                if (j != n-1) {
                        dum=1.0/(a[j][j]);
                        for (i=j+1;i<n;i++)
                            a[i][j] *= dum;
                }
```

```
      }
  }

template<typename T>
void lubksb(NRMat<T> &a, Vec_O_INT &indx, NRVec<T> &b)
{
      int i,ii=0,ip,j;
      T sum;
      int n=a.nrows();

      for (i=0;i<n;i++) {
             ip=indx[i];
             sum=b[ip];
             b[ip]=b[i];
             if (ii != 0)
                    for (j=ii-1;j<i;j++)
                           sum -= a[i][j]*b[j];
             else if (sum != 0.0)
                   ii=i+1;
                    b[i]=sum;
      }
      for (i=n-1;i>=0;i--) {
             sum=b[i];
             for (j=i+1;j<n;j++)
                           sum -= a[i][j]*b[j];
                           b[i]=sum/a[i][i];
      }
}
```

Matr.h

```
#include"other/nr.h"
#include<exception>
#include<algorithm>
#include <gmpxx.h>

template<typename T>
void init_matrices1(int ind,NRMat<T> &A,NRVec<T> &B)
{
  int k,j;
    for(k=0; k < A.nrows(); k++) {
      for(j=0; j < A.ncols(); j++)
           switch(ind)     {
                        case 1:
```

```
                              A[k][j]=1+std::max(k,j);
                              break;
        }
     }
     for(k=0; k < B.size(); k++) {
              switch(ind)     {
                           case 1:
                              B[k]= k == 0 ? 1 : 0;
                              break;
        }
   }
  }

  template<typename T>
  void init_matrices(NRMat<T> &A, NRVec<T> &B)
  {
    A[0][0] = "0.1348531574394464";
    A[0][1] = "0.1878970588235294";
    A[0][2] = "0.1909117647058824";
    A[0][3] = "0.1779264705882353";
    A[1][0] = "0.1878970588235294";
    A[1][1] = "0.262";
    A[1][2] = "0.265";
    A[1][3] = "0.247";
    A[2][0] = "0.1909117647058824";
    A[2][1] = "0.265";
    A[2][2] = "0.281";
    A[2][3] = "0.266";
    A[3][0] = "0.1779264705882353";
    A[3][1] = "0.247";
    A[3][2] = "0.266";
    A[3][3] = "0.255";

    B[0] = "0.3516";
    B[1] = "0.4887";
    B[2] = "0.5105";
    B[3] = "0.4818";
   }
```

Program computes solution of system of linear equations with two different sets of initial data:

1. for a system of linear algebraic equations with matrices

$$a_{ij} = 1+\max(i, j),\ i, j = 1, 2, \ldots, n,$$
$$b_1 = 1,\ b_i = 0,\ \ i = 2, 3, \ldots, n.$$

2. for a system of linear algebraic equations (1.1), i.e.

$$A = \begin{pmatrix} 0.13485315\,74394464 & 0.18789705\,88235294 & 0.19091176\,47058824 & 0.17792647\,05882353 \\ 0.18789705\,88235294 & 0.262 & 0.265 & 0.247 \\ 0.19091176\,47058824 & 0.265 & 0.281 & 0.266 \\ 0.17792647\,05882353 & 0.247 & 0.266 & 0.255 \end{pmatrix}$$

$b = (0.3516,\ \ 0.4887,\ \ 0.5105,\ \ 0.4818)$.

With solving linear algebraic equations, the classical method of computing the solutions obtained do not correspond to reality, so the correct solution of linear algebraic equations can be obtained by using libraries GMP or MPFR. Precision of calculations, using the GMP library is directly dependent on the bit format which is used. SLAE is solved on the bit formats 64, 128 and 256 bits (see Table 5.3 and Fig. 5.7). This bit of GMP is not the limit, there is the possibility of using the bit 1024, 2048 and more. For example, the error of solving the linear, using the bit size is 40980 10^{-12324}. Naturally time taken on such calculations with respect to the classical computer arithmetic is significantly increased.

Table 5.3 Time for solving of SLAEs by LU method using different precision

Matrix size	C++	++, GMP		
	`double` (53)	64	128	256
200	0.04	0.75	0.84	1.04
1000	6.55	96.44	107.7	133.78
4500	688.94	7320.06	8050.12	10104.56

Right solution is achieved using GMP library with a length of mantissa 256 [5].

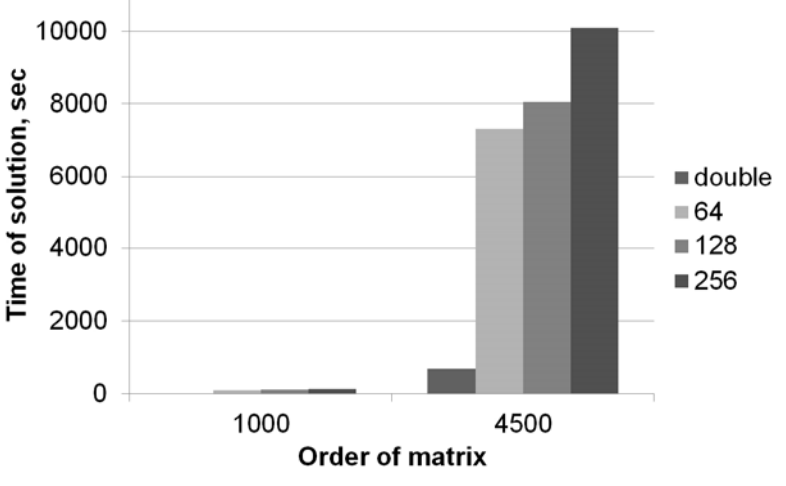

Fig. 5.7 Time of solving of SLAEs by LU method with mpf_class

The time required for the execution of programs with increasing the discharge is much higher than when using the `double` digits. So it makes sense to use the GMP library in parallel programs using MPI on multiprocessor computers.

References

1. GMP Library, `http://www.gmplib.org/`
2. Golub, G.H., Van Loan, C.F.: Matrix Computations, 2nd edn. Johns Hopkins University Press, Baltimore (1989)
3. Khimich, A.N., Molchanov, I.N., Mova, V.I., et al.: Numerical program software of intelligent computer Inparcom. Naukova dumka, Kiev (2007)
4. Molchanov, I.N.: Machine methods for solving applied problems. Algebra, approximation of functions, ordinary differential equations. Naukova dumka, Kiev (2007)
5. Nikolaevskaya, E.A., Chistyakova, T.V.: Software and algorithmic methods to improve the accuracy of computer solutions. Cybernetics and Systems Analysis 6, 172–176 (2009)
6. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C The Art of Scientific Computing. Cambridge University Press, Cambridge (1992)
7. Stewart, G.W.: Introduction to Matrix Computations. Academic Press, New York (1973)

Chapter 6
Parallel Calculations Using MPI

Elena A. Nikolaevskaya, Alexandr N. Khimich, and Tamara V. Chistyakova*

Abstract. One of the most important uses of computers is their use for solving scientific and engineering problems, most of which finally reduced or have their intermediate stage solution of problems in computational mathematics. MPI interface for a long time is the unofficial standard for building distributed computing systems. We consider organization and the main functions of MPI program. Chapter 6 contains parallel MPI-program for matrix multiplications. In order to increase the accuracy in parallel programs functions GMP and MPFR should be used. The easiest way to implement GMP in a parallel program with MPI is to use it in communications instead `mpf_t` or `mpf_class` of GMP library MPI-type MPI_BYTE.

Elena A. Nikolaevskaya
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: elena_nea@ukr.net

Alexandr N. Khimich
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: khimich_ic@mail.ru

Tamara V. Chistyakova
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: dept150@insyg.kiev.ua

E. Nikolaevskaya et al.: Programming with Multiple Precision, SCI 397, pp. 99–121.
springerlink.com

6.1 Introduction

The use of computers of parallel architecture makes it possible to generate much more dimensions of discrete models. However, as we know, obtained computer solutions do not always have physical meaning. The reasons for this are: firstly, the approximate character of the initial data, and errors arising from their representation in computer memory, i.e. when translated from decimal to binary, and secondly, the rounding errors in calculations resulting from the finite size of the mesh bit computer in the third error, resulting from the replacement of an infinite iterative process of final. Therefore, after setting a mathematical problem and entering into the computer one needs to explore well-posed questions of the problem of computer models, its condition, and authenticity of the results.

GMP Library provides a possibility of the calculations with arbitrary precision in the user programs. However, the GMP library does not provide its facilities that can be used in parallel computing.

MPI (Message passing interface) is a language-independent communication protocol that is used to program parallel computers. Both point-to-point and collective communication are supported. MPI [5] is a message-passing application programmer interface, together with protocol and semantic specifications, its features must behave in any implementation. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model that is used in high-performance computing today.

MPI provides the programmer a single mechanism of interaction processes inside the parallel task executing independently of computer architecture (single processor, multiprocessor with shared or separate memory), relative position of the processes (on the same physical processor or different), and the API of the operating system. A program that uses MPI, easily tweaked and extended to other platforms, often by simply recompiling the source code.

MPI is a well-standardized mechanism to build programs by model messaging. There are standard "binding" MPI for C / C + +, Fortran 77/90. There are also free and commercial implementations for almost all HPC platforms, as well as for the high-performance cluster systems based on the nodes that are running by Unix, Linux, and Windows. Currently, MPI is the most widely used and a dynamic interface of the class.

See more details about MPI standard at website http://www.mpi-forum.org and http://www-unix.mcs.anl.gov/mpi/

MPI and parallel programming cover a list of publications. There are [1, 3-6, 9-12]., The best MPI reference is the handbook [5]. Book [3] introduces the basic concepts of parallel and vector computing in the context of introduction to numerical methods. It contains chapters on parallel and vector matrix multiplication and solution of linear systems by direct and iterative methods.

At present, the standard has several popular versions: version 1.3 (shortly called MPI–1), which emphasizes message passing and has a static runtime environment, and MPI–2.2 (shortly MPI–2), which includes new features such as

parallel I/O, dynamic process management and remote memory operation. MPI–2 [5] has over 500 functions and provides language bindings for ANSI C, ANSI Fortran (Fortran90), and ANSI C++. Object interoperability was also added to allow for easier mixed-language message passing programming. A side effect of MPI–2 standardization (completed in 1996) was clarification of the MPI–1 standard, creating the MPI–1.2. Note that MPI–2 is mostly a superset of MPI–1, although some functions have been deprecated. MPI–1.3 programs still work under MPI implementations compliant with the MPI–2 standard.

MPI is often compared with PVM, which is a popular distributed environment and message passing system developed in 1989, and which was one of the systems that motivated the need for standard parallel message passing. Threaded shared-memory programming models (such as Pthreads and OpenMP) and message passing programming (MPI/PVM) can be considered as complementary programming approaches, and can occasionally be seen together in applications, e.g. in servers with multiple large shared-memory nodes.

When you run the task c MPI creates a group of p processes. The group is identified by an integer descriptor (communicator). Inside, the group processes are numbered from 0 to p-1. During solution of the problem original group (it was named `MPI_COMM_WORLD`) can be divided into subgroups, and the subgroups can be combined into a new group, which has its communicator. Thus, the process can simultaneously belong to multiple groups of processes. Each process is available in your number `myProc` of any group of which it is.

The behavior of all processess described by one and the same program. Interprocess communication is programmed explicitly using the library of MPI, which dictates the standard of programming. Quasi-simultaneous launch of the original group of processes is performed by the operating system. Thus p is determined by request of the user, and not by the number of available processors!

If we formulate short, MPI is a library of functions providing interaction of parallel processes through a mechanism of communication. It is quite heavy and complicated library, consisting of about 130 functions, which include:

- init function and closing MPI processes;
- the functions that implement the communication operations such as point to point;
- the functions that implement the collective operations;
- the functions for working with process groups and communicators;
- the functions for working with data structures;
- the functions of formation of topology processes.

Set of functions of the MPI library goes far beyond the range of functions, the minimum that is required to support a mechanism for transferring messages in computational problems of low and medium complexity. However, the complexity of this library should not scare users because, ultimately, all this set of functions is designed to facilitate the development of efficient parallel programs. In the end, the user has the right to decide what tools to use in the arsenal provided, and what

does not. In principle, any parallel program can be written using only 6 MPI functions, but rather a complete and convenient programming environment is a set of 24 functions.

6.2 Basics of MPI-Program

MPI is a set of utilities and library functions that allow you to create and run applications that run on parallel computing facilities of various natures.

The names of all functions, data types, constants, etc., relating to the MPI-library, start with the prefix `MPI_` and described in the header file `mpi.h`. All functions (except `MPI_Wtime` and `MPI_Wtick`) have a return type `int` and return an error code or `MPI_SUCCESS` in case of success. See Table 6.1.

Before the first call any MPI-function calls the `MPI_Init`. Its prototype is:

```
int MPI_Init (int *argc, char ***argv);
```

where `argc`, and `argv` point to the number of arguments to the program and the argument vector, respectively (this is address of arguments of function of the main program). Many MPI implementations require that the process before calling `MPI_Init` did not do anything that would change his status, such as opening or reading / writing files, including standard input and output.

The calls

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

set size to the number of processes in the group specified by comm and the call

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

sets rank to the rank of the calling process within the group (from 0 up to n-1 where n is the size of the group).

When you are finished with the MPI-functions you need to call `MPI_Finalize`. Its prototype is:

```
int MPI_ Finalize (void);
```

Table 6.1

Functions	Parameters
`int MPI_Init(int *argc, char **argv[])`	`*argc` - argument from main()
Initializes MPI environment.	`**argv[]` - argument from main()

Table 6.1 *(continued)*

Functions	Parameters
`int MPI_Finalize(void)` Terminates MPI execution environment	None
`int MPI_Comm_rank(MPI_Comm comm, int *rank)` Determines rank of process in communicator.	`comm` - communicator `*rank` - rank (returned)
`int MPI_Comm_size(MPI_Comm comm, int *size)` Determines size of group associated with communicator.	`comm` - communicator `*size` - size of group (returned)
`double MPI_Wtime(void)` Returns elapsed time from some point in past, in seconds.	None

Here's an MPI multi-process "Hello World":

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv)
{
int i, myrank, nprocs;
double a = 0, b = 1.1, c = 0.90;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

printf("Hello world! This is process %d out of %d\n",myrank,
nprocs);
if (myrank == 0)
printf("Some processes are more equal than others.");
MPI_Finalize();
}
```

which is compiled and executed on the SP-2 at Ames by

```
mpicc -O3 example.c –o ex -lm
mpiruni -np 2 ./ex
```

and produces (on the standard output)

```
0:Hello world! This is process 0 out of 2
1:Hello world! This is process 1 out of 2
0:Some processes are more equal than others.
```

Another important thing to know is about the MPI wall clock timer:

```
double MPI_Wtime()
```

which returns the time in seconds from some unspecified point in the past.

6.1.1 Main MPI-Functions

The following is a collection of MPI routines (please, see Table 6.2, 6.3) that is sufficient for most programs in the text.

A very large number of routines are provided in MPI. The routines described here are divided into basic point-to-point message passing and collective message passing.

MPI defines various datatypes for `MPI_Datatype`, mostly with corresponding C datatypes, including `MPI_CHAR (signed char), MPI_INT (signed int), MPI_FLOAT (float), MPI_DOUBLE (double).`

The complete set of routines and additional details can be found in [4-6].

Point-to-Point Message Passing. A number of important MPI functions involve communication between two specific processes.

Point-to-point operations, as these are called, are particularly useful in patterned or irregular communication, for example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other processors between calculation steps, or a master-slave architecture in which the master sends new task data to a slave whenever the previous task is completed.

MPI-1 specifies mechanisms for both blocking and non-blocking point-to-point communication mechanisms, as well as the so-called 'ready-send' mechanism whereby a send request can be made only when the matching receive request has already been made.

Table 6.2 Point-to-Point communications

Functions	Parameters
`int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)` Sends message (blocking).	`*buf` - send buffer `count` - number of entries in buffer `datatype` - data type of entries `dest` - destination process rank `tag` - message tag `comm` – communicator

Table 6.2 *(continued)*

Functions	Parameters
`int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)` Receives message (blocking).	`*buf` - receive buffer (loaded) `count` - max number of entries in buffer `datatype` - data type of entries `source` - source process rank `tag` - message tag `comm` - communicator `*status` - status (returned)

`int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,MPI_Request *request)`
Starts a non-blocking send.
 	
Related:
`MPI_Ibsend()`
Starts a nonblocking buffered send
`MPI_Irsend()`
Starts a nonblocking ready send
`MPI_Issend()`
Starts a non-blocking synchronous send
	`*buf` - send buffer
	`count` - number of buffer elements
	`datatype` - data type of elements
	`dest` - destination rank
	`tag` - message tag
	`comm` - communicator
	`*request` - request handle (returned)

`int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)` Begins a non-blocking receive.	`*buf` - receive buffer address (loaded) `count` - number of buffer elements `datatype` - data type of elements `source` - source rank `tag` - message tag `comm` - communicator `*request` - request handle (returned)

Table 6.2 *(continued)*

Functions	Parameters
`int MPI_Wait(MPI_Request *request, MPI_Status *status)` Waits for a MPI send or receive to complete and then returns.	`*request` - request handle `*status` - status (same as return status of `MPI_recv()` if waiting for this.
Related: `MPI_Waitall()` Wait for all processes to complete	

`MPI_Waitany()`
Wait for any process to complete
`MPI_Waitsome()`
Wait for some processes to complete	

`int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)` Tests for completion of a nonblocking operation.	`*request` - request handle `*flag` - true if operation completed (returned) `*status` - status (returned)
`int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)` Blocking test for a message (without receiving message).	`source` - source process rank `tag` - message tag `comm` - communicator `*status` - status (returned)
`int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Comm *status)` Nonblocking test for a message (without receiving message).	`source` - source process rank `tag` - message tag `comm` - communicator `*flag` - true if there is a message (returned) `*status` - status (returned)

A popular example is `MPI_Send`, which allows one specified process to send a message to a second specified process.

The call

```
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,int tag, MPI_Comm comm)
```

sends `count` items of data of type `datatype` starting at the location `buf`. In all message passing systems, the processes have identifiers of some kind. In MPI, the process is identified by its `rank`, an integer. The data is sent to the processes whose

rank dest. tag is an integer used by the programmer to allow the receiver to select from among several arriving messages in the MPI_Recv. Finally, comm is something called a communicator, which is essentially a subset of the processes. Usually, message passing occurs within a single subset. The subset MPI_COMM_WORLD consists of all the processes in a single parallel job, and is predefined.

A receive call matching the send above is

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int
source,int tag, MPI_Comm comm, MPI_Status *status)
```

buf is where the data is placed once it arrives. count, an input argument, is the size of the buffer; the message is truncated if it is longer than the buffer. Of course, the receive has to be executed by the correct destination process, as specified in the dest part of the send, for it to match. source must be the rank of the sending process. The communicator and the tag must be matched. So must the datatype.

The purpose of the datatype field is to allow MPI to be used with heterogeneous hardware. A process running on a little-endian machine may communicate integers to another process on a big-endian machine; MPI converts them automatically. The same holds for different floating-point formats. Type conversion, however, is not supported: an integer must be sent to an integer, a double to a double, etc.

Suppose the producer and consumer transact business in two word integer packets. The producer is process 0 and the consumer is process 1. Then the send would look like this:

```
int outgoing[2];
MPI_Send(outgoing, 2, MPI_INT, 1 100, MPI_COMM_WORLD)
```

and the receive like this:

```
MPI_Status stat;int incoming[2];
MPI_Recv(incoming, 2, MPI_INT, 0 100, MPI_COMM_WORLD, &stat)
```

Group Routines. Collective functions involve communication among all processes in a process group (which can mean the entire process pool or a program-defined subset). A typical function is the MPI_Bcast call (short for "broadcast"). This function takes data from one node and sends it to all processes in the process group. A reverse operation is the MPI_Reduce call, which takes data from all processes in a group, performs an operation (such as summing), and stores the results on one node. Reduce is often useful at the beginning or at the end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result.

Other operations perform more sophisticated tasks, such as MPI_Alltoall which rearranges n items of data processor such that the nth node gets the nth item of data from each.

Table 6.3 Collective communications

Functions	Parameters
`int MPI_Barrier(MPI_Comm comm)` Blocks process until all processes have called it.	`comm` - communicator
`int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)` Broadcasts message from root process to all processes in comm and itself.	`*buf` - message buffer (loaded) `count` - number of entries in buffer `datatype` - data type of buffer `root` - rank of root

`int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount,`
`MPI_Datatype recvtype, MPI_Comm comm)`
Sends data from all processes to all processes.
 	
Related:
`MPI_Alltoallv()`
Sends data to all processes, with displacement
	`*sendbuf` - send buffer
	`sendcount` - number of send buffer elements
	`sendtype` - data type of send elements
	`*recvbuf` - receive buffer (loaded)
	`recvcount` - number of elements each receive
	`recvtype` - data type of receive elements
	`comm` – communicator

`int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount,`
`MPI_Datatype recvtype, int root, MPI_Comm comm)`
Gathers values for group of processes.
 	
Related:
`MPI_Allgather()`
Gather values and distribute to all
`MPI_Gatherv()`
Gather values into specified locations
`MPI_Allgatherv()`
Gather values into specified locations and distributes to all
`MPI_Gatherv()` and `MPI_Allgatherv()` require additional parameter: `*displs` – array of displacements, after `recvcount`.
	`*sendbuf` - send buffer
	`sendcount` - number of send buffer elements
	`sendtype` - data type of send elements
	`*recvbuf` - receive buffer (loaded)
	`recvcount` - number of elements each receive
	`recvtype` - data type of receive elements
	`root` - rank of receiving process
	`comm` – communicator

Table 6.3 *(continued)*

Functions	Parameters

`int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
Scatters a buffer from root in parts to group of processes.
 	
Related:
`MPI_Scatterv()`
Scatters a buffer in specified parts to group of processes.
`MPI_Reduce_scatter()`
Combines values and scatter results.
	`*sendbuf` - send buffer
	`sendcount` - number of elements send, each process
	`sendtype` - data type of elements
	`*recvbuf` - receive buffer (loaded)
	`recvcount` - number of recv buffer elements
	`recvtype` - type of recv elements
	`root` - root process rank
	`comm` – communicator

`int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root,MPI_Comm comm)`
Combines values on all processes to single value.
 	
Related:
`MPI_Allreduce()`
Combine values to single value and return to all.
	`*sendbuf` - send buffer address
	`*recvbuf` - receive buffer address
	`count` - number of send buffer elements
	`datatype` - data type of send elements
	`op` - reduce operation.
	Several operations, including
	`MPI_MAX` Maximum
	`MPI_MIN` Minimum
	`MPI_SUM` Sum
	`MPI_PROD` Product
	`root` - root process rank for result
	`comm` - communicator

Here's a sum operation, on `doubles`. The variable `sum` on process `root` gets the sum of the variables `x` on all the processes.

```
  double x, sum;int root, count = 1;
  MPI_Reduce(&x,  &sum,  count,  MPI_DOUBLE,  MPI_SUM,  root,
MPI_COMM_WORLD);
```

The fifth argument specifies the operation; other possibilities are `MPI_MAX`, `MPI_LAND`, `MPI_BOR`, and other, which specify maximum, logical AND, and bitwise OR, for example.

The semantics of the collective communication calls are subtle to this extent: nothing happens except that a process stops when it reaches such a call, until all processes in the specified group reach it. Then the reduction operation occurs and the result is placed in the `sum` variable on the root processor. Thus, reductions provide what is called a barrier synchronization.

There are quite a few collective communication operations provided by MPI, all of them are useful and important. We will use several in the assignment. To mention a few, `MPI_Bcast` broadcasts a vector from one process to the rest of its process group; `MPI_Scatter` sends different data from one process to each process in its a group; `MPI_Gather` is the inverse of a scatter: one process receives and concatenates data from all processes in its group; `MPI_Allgather` is like a gather followed by a broadcast: all processes receive the concatenation of data that are initially distributed among them; `MPI_Reduce_scatter` is like reduce followed by scatter: the result of the reduction ends up distributed among the process group. Finally, `MPI_Alltoall` implements a very general communication in which each process has a separate message to send to each member of the process group.

Often the process group in a collective communication is some subset of all the processors. In a typical situation, we may view the processes as forming a grid, let us say a 2d grid, for example. We may want to do a reduction operation within rows of the process grid. For this purpose, you can use `MPI_Reduce`, with a separate communicator for each row.

6.2.2 Matrix Multiplication with MPI

Matrix-vector and matrix–matrix multiplication are the base for many macro-operations of computational mathematics, such as iterative methods for solving systems of linear algebraic equations.

Let us consider the problem of calculating the product of two square matrices. To calculate the product of two matrices $C = A \times B$ as follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \ , \ i, j = 1, \ldots, n,$$

where n is order of the matrices, and we use a block algorithm.

Listing 6.1 shows an example program for matrix multiplication in C + + using MPI.

In the main function each process initializes subsystem MPI with `MPI_Init`, gets his number `my_rank` and total number of processes `p` by `MPI_Comm_rank` and `MPI_Comm_size`.

Rows of A and B with numbers

`n*my_rank/=p,…,n*(my_rank+1)/p-1`

are stored in each of the processes, where n – number of rows and columns of A matrix, my_rank - current process, p - required number of processes, and l - cycles meet the following calculation in the program.

Because these blocks are exchanged between all the processes, the memory of each is allocated under the block with a maximum number of component `max_rows=n/p+n%p`.

Thus, in each process should be stored `max_rows=n/p+mod(n/p)` rows of the two original matrices *A* and *B* (`mod` - remainder from division operation of one integer to another).

Listing 6.1 Parallel multiplication of two matrices with MPI

m_mult_m_d.cpp

```
#include <string>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include "other/nr.h"
#include "matrix.h"
#include <math.h>

// Connecting the MPI header file
#include "mpi.h"

using namespace std;
typedef double BASE_TYPE;
typedef NRVec<BASE_TYPE> Vec1d;

// main program
int main(int argc, char **argv)
{
// Declaring variables
int my_rank;        /* rank of process */
int p;              /* total number of processes */
int n;              /* order of matrices */
double t;           /* execution time of all program*/

int first_row, last_row, rows, i, j;
int max_rows;
Vec1d matrixa(max_rows*n),matrixb(max_rows*n);
Vec1d matrixc(n*n);

/* Get started with MPI */
```

```
     MPI_Init(&argc,&argv);

  /* Get the process number my_rank*/
     MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  /* Get the total number of processes running */
     MPI_Comm_size(MPI_COMM_WORLD,&p);

  n = 1000;
  if (my_rank==0)
              {
  printf("Quantity of the equations %d\nQuantity of used proces-
sors %d\n\n",n,p);
              }
  /* The first row of the matrix that takes part in calculations
in the process my_rank*/
       first_row = n * my_rank;
       first_row /= p;

  /* The last row of the matrix that takes part in calculations
in the process my_rank*/
       last_row = n * (my_rank + 1);
       last_row = last_row / p - 1;

  /* Number of rows of matrices involved in calculations in the
process my_rank*/
       rows = last_row - first_row + 1;

  /* Maximum number of rows in the process my_rank*/
       max_rows = n / p + n % p;

  /* Initialization of the array of the matrix A in current pro-
cessor, sets the block of the matrix in each process my_rank */
       init_matrixa(matrixa, n, first_row, last_row);
  /* Initialization of the array of the matrix B in current pro-
cessor, sets the block of the matrix in each process my_rank */
       init_matrixb(matrixb, n, first_row, last_row);

       if (my_rank == 0)
       {
  cout << endl << "Problem: matrix_mult_matrix" << endl;
       cout << "n = " << n << endl;
  }
  /* Synchronization of all processes*/
```

```
        if (p > 1)
        MPI_Barrier (MPI_COMM_WORLD);

   /* Start time multiplying */
        t = MPI_Wtime();

   /*  Multiplication  matrix  A  by  matrix  B  in  each  process
my_rank,function matrix_mult_matrix */
   matrix_mult_matrix(matrixa,matrixb,matrixc,n,my_rank,p);
   /* Synchronization of all processes */
        if (p > 1)
        MPI_Barrier (MPI_COMM_WORLD);

   /* Time execution  */
        t = MPI_Wtime() - t;
        if (my_rank == 0)
        {
                for ( i = 0; i < 4; i++)
                {
                        for (j = 0; j < 4; j++)
                        {
                        cout.precision(100);
                        cout<<setw(12)<<  matrixc[i  *  n  +  j]  <<
endl;
                        }
                }
                printf("\nTime : %.10f",t);
        }
   /* End of program */
        MPI_Finalize ();
        return 0;
   }

   /* Function initialization of the matrix A */
   template<typename T>
   void  init_matrixa(NRVec<T>  &matrixa,int  n,int  first_row,int
last_row)
   {
        int i, j;
        for (i = first_row; i <= last_row; i++)
                for (j = 0; j < n; j++)
                matrixa[i * n + j] = 2.0;
   }
```

```
  /* Function initialization of the matrix B */
  template<typename T>
  void init_matrixb(NRVec<T> &matrixb,int n,int first_row,int
last_row)
  {
       int i, j;

       for (i = first_row; i <= last_row; i++)
              for (j = 0; j < n; j++)
                     matrixb[i * n + j] = 1.0;
  }
  /* Multiplication matrix A by matrix B, C = AB */
  template<typename T>
  void matrix_mult_matrix(NRVec<T> &a,NRVec<T> &b,NRVec<T>
&c,int n, int my_rank,int p)
  {
       int i, j, k, l, m, np, len;
       int max_rows;
       int last_row, first_row, rows;
       int first_row_k, last_row_k, rows_k;
       int dest, source, tag = 0;
       double s;
       MPI_Status status;

  /* Compute the maximum number of rows in the process */
       max_rows = n / p + n % p;
  /* Calculate the sender and recipient dest for the current
process */
       source = (my_rank + 1) % p;
       if(my_rank == 0)
              dest = p - 1;
       else
              dest = my_rank - 1;

  /* The first row of matrix, that takes part in calculations in
my_rank */
       first_row = n * my_rank;
       first_row /= p;

  /* The last row of matrix, that takes part in calculations
my_rank*/
       last_row = n * (my_rank + 1);
       last_row  = last_row / p - 1;
```

```
  /* Number of rows of the matrix participating in calculations
in my_rank*/
       rows = last_row - first_row + 1;
  /* Zero out of result */
       len = (last_row - first_row + 1) * n; /* длина c*/
       for (i = 0; i < len; i++)
              c[i ] = 0.0;

  /* Cycle by blocks */
              for (l = 0; l < p; l++)
              {

  /* Current number block for strip of the matrix B in the
process k */
                     k = (my_rank + l) % p;

  /* First row of block for strip of the matrix B in the process
k */
                     first_row_k = n * k;
                     first_row_k /= p;

  /* Last row of block for strip of the matrix B in the process
k */
                     last_row_k = n * (k + 1);
                     last_row_k  = last_row_k / p - 1;
  /* Number of rows of the matrix participating in calculations
in the process k */
                     rows_k = last_row_k - first_row_k + 1;

  /* Multiplying rectangular block of the matrix A in row
first_row ... last_row and columns first_row_k ... last_row_k on
the vector, the corresponding components first_row_k ...
last_row_k   (i -  i          j -  m        j -  k)   */

              for (i = 0; i < rows; i++)
              {
                     for (m = 0; m < n; m++)
                     {
                     for (s = 0., j = 0; j < rows_k; j++)
                            s+= a[i*n+j+first_row_k]*b[j*n+m];
                            c[i*n+m]+=s;
                     }
              }
```

```
  MPI_Sendrecv_replace(&b,sizeof(b),MPI_DOUBLE,dest,tag,source,t
ag, MPI_COMM_WORLD,&status);
              }
  }
```

Function `matrix_mult_matrix ()` gets the pointers to the rows of *A*, *B,* and *C* with indices `n*my_rank/=p,...,n*(my_rank +1)/p-1`, and calculates the indicated components of the matrix *C* - the product of two square matrices *A* and *B*.

Denote $A=(a_{ij})$, $B=(b_{ij})$, $C=(c_{ij})$, $i,j=0,1,\ldots,n-1$, m=`my_rank`.

Function `matrix_mult_matrix()` in the cycle for `l=0,1,...p-1` computes a matrix that is the product of the block matrix *A*, which is on rows

`nm/p, …, n(m+1)/p-1` (6.1)

and colums

`(nm/p+l)(mod p),…,(n(m+1)/p-1+l)(mod p)` (6.2)

and block of matrix *B*, which is located in the rows with numbers (6.2) and columns

`1, …., n.` (6.3)

This matrix is added to the block of the matrix *C*, which is located in the rows with numbers (6.1) and columns with indices (6.3).

After the cirle execution over l strip of matrix B, which was created by the rows with numbers (6.2), is sent to the process number `(m-1)(mod p)`, and in its place is located strip of matrix B, obtained from the process with a number `(m +1)(mod p)`, i.e. implemented the cyclic sending, available to each process of the matrix *B*.

6.3 Using MPI_BYTE in Parallel Programming with GMP

There are a lot of practical problems that require large orders of matrices. In order to reduce the time that is wasted on solving these problems one must use parallel programming.

To increase the accuracy in parallel programs functions of GMP and MPFR should be used. The easiest way to implement GMP [2] in a parallel program with MPI is to use it in communications instead of `mpf_t` or `mpf_class` of GMP library MPI-type `MPI_BYTE`.

The use of `MPI_BYTE` means that the contents of the corresponding array should not be a subject of any changes - and at the receiver and on the sending side the array will have the same length and the same binary representation. `MPI_BYTE` describes bytes, which is defined as 8 binary digits. `MPI_BYTE` guarantees the number that is in the range (0 .. 255).

Consider the earlier example of multiplying of two square matrices combined using `MPI_BYTE` and `mpf_class` (Listing 6.2).

Listing 6.2 Parallel multiplication of matrices with GMP

m_mult_m_gmp.cpp

```
#include <string>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include "matrixg.h"
#include "mpi.h"
#include <math.h>
#include <gmpxx.h>

using namespace std;

const int PRECISION=64;
typedef mpf_class BASE_TYPE;
typedef NRVec<BASE_TYPE> Vec1d;

int main(int argc, char **argv)
{
mpf_set_default_prec(PRECISION);
cout << "Precision " << mpf_get_default_prec() << endl;

int my_rank;
int p;
int n;
double t;
int first_row, last_row, rows, i, j;
int max_rows;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);

/*   …
continuing the program as in the Listing 14
            …
*/

MPI_Finalize ();
```

```
  return 0;
  }

  /* Multiplication matrix A by matrix B, C = AB */
  template<typename T>
  void matrix_mult_matrix(NRVec<T> &a, NRVec<T> &b, NRVec<T>
&c,int n, int my_rank, int p)
  {
       int i, j, k, l, m, np, len;
       int max_rows;
       int last_row, first_row, rows;
       int first_row_k, last_row_k, rows_k;
       int dest, source, tag = 0;
       double s;
       MPI_Status status;
       ….      …..     …..

  /* Multiplying rectangular block of the matrix A in row
first_row ... last_row and columns first_row_k ... last_row_k on
the vector, the corresponding components first_row_k ...
last_row_k     (i -  i         j -  m        j -  k)*/
       for (i = 0; i < rows; i++)
       {
              for (m = 0; m < n; m++)
              {
                     for (j = 0; j < rows_k; j++)
                     c[i*n+m]+=a[i*n+j+first_row_k]*b[j*n+m];

              }
       }
  /* Sending strip of matrix B to process dest and obtain it
from the source*/
       if (p > 1)

  MPI_Sendrecv_replace (&b, sizeof (b), MPI_BYTE, dest,      tag,
source, tag, MPI_COMM_WORLD, &status);
       }
       }
```

As it is seen from the listing, the program differs from the previous only by a few lines:

1. Description `mpf_class` and set precision by using `mpf_set_default_prec (PRECISION)` of GMP library

```
#include <gmpxx.h>
…
const int PRECISION=64;
typedef mpf_class BASE_TYPE;
typedef NRVec<BASE_TYPE> Vec1d;
…
mpf_set_default_prec(PRECISION);
cout << "Precision " << mpf_get_default_prec() << endl;
```

2. Use in communications `MPI_BYTE`

```
MPI_Sendrecv_replace (&b, sizeof (b), MPI_BYTE, dest,      tag,
source, tag, MPI_COMM_WORLD, &status);
```

Programs tested on Inparcom-256 [7, 8]. Table 6.4 and Fig. 6.1 provide a comparative analysis of run-time using a parallel C++ program without using the functions of GMP and by using the functions GMP. Total time which is taken to parallel computation product of two matrices is presented in Table 6.4.

Table 6.4 Comparison of the time (sec) of the parallel multiplication of two matrices for various numbers of processors

Order of matrices	Number of process	C++, MPI	C++, GMP, MPI
		double (53)	64
1000	1	17,31	174,9
	4	0,5	57
	8	0,22	30
	16	0, 10	15,5
	32	0,05	5,8
2000	1	228,69	1000,1
	4	14,8	468,6
	8	3,33	242,2
	16	1,86	124,6
	32	1,2	64,9
3000	4	66,8	1631
	8	29,2	824
	16	19,3	410,2
	32	11,1	218, 3

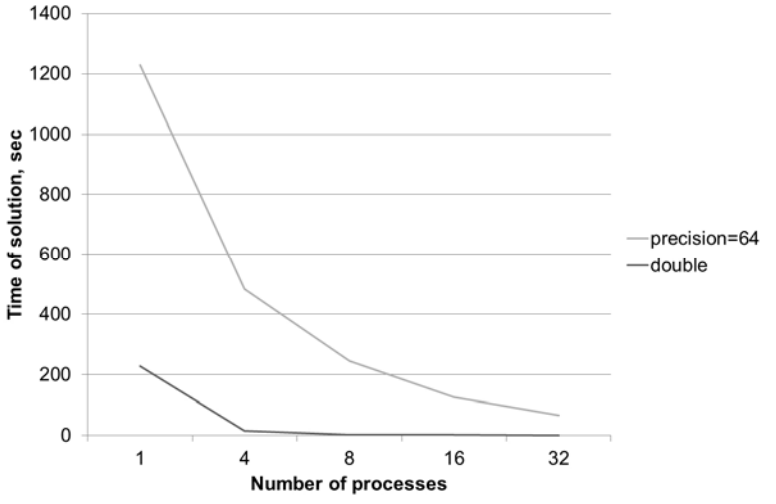

Fig. 6.1 The time required to multiply two matrices using MPI and mpf_class for order N=2000.

The use of MPI_BYTE is the easiest way for simple programs, but not a panacea for complex and large programs. In the next chapter we will consider other, more universal variants of using the arbitrary precision GMP library in parallel programs with MPI.

References

1. Dongarra, J.J., Duff, I.S., Sorensen, D.C., Van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia (1998)
2. GMP library, `http://www.gmplib.org`
3. Golub, G.H., Ortega, J.M.: Scientific Computing. An Introduction with Parallel Computing. Academic Press (1993)
4. Gropp, W., Huss–Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI — The Complete Reference: The MPI-2 Extensions, vol. 2. MIT Press, Cambridge (1998)
5. Gropp, W., Lusk, E., Skjellum, A.: Using MPI Portable Parallel Programming with the Message-Passing Interface, 2nd edn. MIT Press, Cambridge (1999)
6. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2 Advanced Features of the Message Passing Interface. MIT Press, Cambridge (1999)
7. Inparcom (family of intelligent parallel computers), `http://www.geopoisk.com/inparcom/eng/index_eng.htm`
8. Khimich, A.N., Molchanov, I.N., Mova, V.I., et al.: Numerical program software of intelligent computer Inparcom. Naukova dumka, Kiev (2007)
9. MPI-forum, `http://www.mpi-forum.org`

10. Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann Publication (1996)
11. Saad, Y., Schultz, M.H.: Data communication in parallel architectures. Parallel Comput. 11, 131–150 (1989)
12. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI — The Complete Reference: The MPI Core, vol. 1. MIT Press, Cambridge (1998)

Chapter 7
MPIBNCpack Library

Elena A. Nikolaevskaya, Tomonori Kouya,
Alexandr N. Khimich, and Tamara V. Chistyakova *

Abstract. The algorithm of joint using GMP (MPFR) library and MPI interface in parallel programming is described. The most suitable way to implement GMP in a parallel program with MPI is the usage of MPIGMP library, which can transport some multiple presicion variables of MPFR and GMP by packing them into one memory area. MPIGMP is a part of MPIBNCpack.

Elena A. Nikolaevskaya
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: elena_nea@ukr.net

Tomonori Kouya
Faculty of Comprehensive Informatics at SIST
Kakegawa city, Shizuoka-ken, JAPAN
e-mail: tkouya@cs.sist.ac.jp

Alexandr N. Khimich
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: khimich_ic@mail.ru

Tamara V. Chistyakova
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: dept150@insyg.kiev.ua

E. Nikolaevskaya et al.: Programming with Multiple Precision, SCI 397, pp. 123–134.
springerlink.com

7.1 Introduction

MPIBNC pack [5] supports the usage of `mpf_t` variables of GMP [1] and `mpfr_t` variable MPFR [3]. They are based on GMP 4.1.2 and MPFR 2.4.1 or later, that might be changed in future.

The library is available at http://na-inet.jp/na/bnc/

The main idea is that because multiple precision variables may not be guranteed to be always stored in continous memory areas, therefore, multiple precision data must be packed in MPIBNCpack before transfering among processes.

MPIGMP library collects the minimum set of functions necessary for these works, and MPIBNCpack is based on BNCpack for serial computation and MPIGMP for parallel computation with MPI [4]. Figure 7.1 shows the structure of software layers around MPIBNCpack.

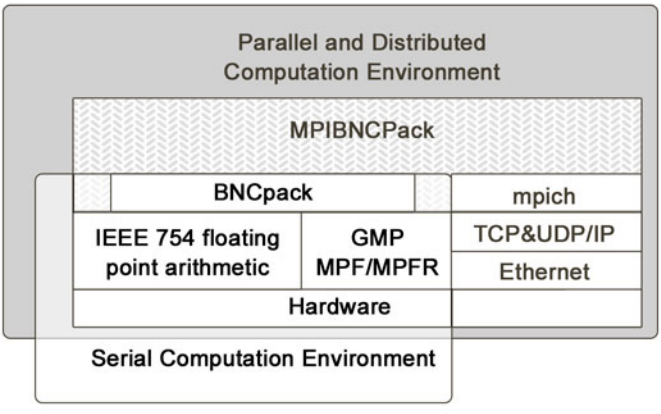

Fig. 7.1 The location of MPIBNCpack in software layers

7.2 Fast Start - 2

Consider example of calculating function $f(a) = 4/(1 + a^2)$ and number π. Listing 7.1, which demonstrates that with MPIGMP is shown below.

Listing 7.1

`cpi-gmp.c`

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
```

```
#include "mpi_gmp.h"
#define MPF_PREC 128

// f(a) = 4/(1 + a^2)
void mpf_f(mpf_t ret, mpf_t a)
{
     mpf_t tmp;
     mpf_init(tmp);
     mpf_mul(tmp, a, a);
     mpf_add_ui(tmp, tmp, 1UL);
     mpf_ui_div(ret, 4UL, tmp);
     mpf_clear(tmp);
     return;
}

int main(int argc, char *argv[])
{
     int done = 0, n, myid, numprocs, i;
     double startwtime, endwtime;
     int  namelen;
     char processor_name[MPI_MAX_PROCESSOR_NAME];

     mpf_t mpf_mypi, mpf_pi, mpf_h, mpf_sum, mpf_x, mpf_fret;
     mpf_t mpf_tmp;
     void * packed_mpf_mypi, * packed_mpf_pi;
     int mpf_size, pos;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
     MPI_Get_processor_name(processor_name, &namelen);

     fprintf(stdout,"Process %d of %d on %s\n",myid, numprocs,
processor_name);

     /* set presicion */
     mpf_set_default_prec(MPF_PREC);

     /* typedef and commit to mpich */
     commit_mpf(&(MPI_MPF), MPF_PREC, MPI_COMM_WORLD);

     /* если нужно, например MPI_SUM для работы с mpf_t  */
     create_mpf_op(&(MPI_MPF_SUM),_mpi_mpf_add,MPI_COMM_WORLD);

     if(myid == 0)
```

```
        {
                printf("----- Start (MPF or MPFR) -----\n");
                n = 16384;
                startwtime = MPI_Wtime();
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* Init */
        mpf_init(mpf_h);
        mpf_init(mpf_x);
        mpf_init(mpf_pi);
        mpf_init(mpf_sum);
        mpf_init(mpf_mypi);
        mpf_init(mpf_tmp);
        mpf_init(mpf_fret);

        mpf_set_ui(mpf_h, n);
        mpf_ui_div(mpf_h, 1UL, mpf_h);
        mpf_set_ui(mpf_sum, 0UL);

        // mpf_tmp = 0.5
        mpf_set_ui(mpf_tmp, 1UL);
        mpf_div_ui(mpf_tmp, mpf_tmp, 2UL);

        for (i = myid + 1; i <= n; i += numprocs)
        {
                mpf_set_ui(mpf_x, (unsigned long)i);
                mpf_sub(mpf_x, mpf_x, mpf_tmp);
                mpf_mul(mpf_x, mpf_x, mpf_h);
                mpf_f(mpf_fret, mpf_x);
                mpf_add(mpf_sum, mpf_sum, mpf_fret);
        }

        mpf_mul(mpf_mypi, mpf_sum, mpf_h);
        ….
/* Allocate buffer */
        packed_mpf_mypi= allocbuf_mpf(mpf_get_prec(mpf_mypi), 1);
        packed_mpf_pi = allocbuf_mpf(mpf_get_prec(mpf_pi), 1);

/* Pack */
        pack_mpf(mpf_mypi, 1, packed_mpf_mypi);
        pack_mpf(mpf_pi, 1, packed_mpf_pi);

        //MPI_Send/Reduce
```

```
        MPI_Reduce(packed_mpf_mypi,packed_mpf_pi,1,MPI_MPF,MPI_MP
F_SUM,0,MPI_COMM_WORLD);

        /* Unpack */
  unpack_mpf(packed_mpf_pi, mpf_pi, 1);

        /* Print */
        if (myid == 0)
        {
                endwtime = MPI_Wtime();
        printf("mpf_pi :");
        mpf_out_str(stdout, 10, 0, mpf_pi);
        printf("\n");
        printf("mpf_h :");
        mpf_out_str(stdout, 10, 0, mpf_h);
        printf("\n");
        printf("wall clock time = %f\n", endwtime - startwtime);
        printf("----- End (MPF or MPFR) -----\n");
        }
  // Clear
        mpf_clear(mpf_mypi);
        mpf_clear(mpf_sum);
        mpf_clear(mpf_x);
        mpf_clear(mpf_h);
        mpf_clear(mpf_pi);
        mpf_clear(mpf_fret);
        mpf_clear(mpf_tmp);
        …
  // Free typedef
        free_mpf(&(MPI_MPF));
        free_mpf_op(&(MPI_MPF_SUM));

        MPI_Finalize();
  }
```

With the help of library's functions mantissa of any length can easy organize the delivery .

With the help of functions of the library can easily organize delivery of the variables of any length mantissa.

Organization of programs and communications, using functions of the library MPIGMP are as follows:

1. Include the following header files and link the appropriate libraries

```
#include <stdio.h>
#include <math.h>
```

```
#include "mpi.h"
#include "bnc.h"
#include "mpi_gmp.h"
#include "mpi_bnc.h"
```

2. Initialize MPI functonarity by `MPI_Init`. After that, indicate a communicator (like `MPI COMM WORLD`) and then get the numbers of PEs (stored in `numprocs`) and the rank (stored in `myid`) on which it is being executed.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Get_processor_name(processor_name, &namelen);
```

3. For defining mutiple precision datatype available in the current communicator, indicate the precision `MPF_PREC` (in binary). In example it is of 256 bits. In addition, define the operation of MPI Reduce (in our case it is `MPI_MPF_SUM`) and etc. The current MPIGMP does not support others except for summation.

```
#define MPF_PREC 256
mpf_set_default_prec(MPF_PREC);
commit_mpf(&(MPI_MPF), MPF_PREC, MPI_COMM_WORLD);
create_mpf_op(&(MPI_MPF_SUM),_mpi_mpf_add,MPI_COMM_WORLD);
```

4. Describe your calculation running on each process and then get started to communicate in the way as follows:

a) Allocate a buffer enough to pack multiple precision variables with `allocbuf_mpf` function.

```
packed_mpf_mypi=allocbuf_mpf(mpf_get_prec(mpf_mypi),1);
packed_mpf_pi=allocbuf_mpf(mpf_get_prec(mpf_pi),1);
```

b) Pack them into the buffer previously allocated.

```
pack_mpf(mpf_mypi, 1, packed_mpf_mypi);
pack_mpf(mpf_pi, 1, packed_mpf_pi);
```

c) Get started to send/receive the packed data in the buffer.

```
MPI_Reduce(packed_mpf_mypi,packed_mpf_pi,1,MPI_MPF,MPI_MPF_SUM
,0,MPI_COMM_WORLD);
```

d) When completed, unpack them into the original multiple precision variables by `unpack_mpf` function.

```
unpack_mpf(packed_mpf_pi, mpf_pi, 1);
```

5. Finalize MPI, and free the allocated memory areas to store multiple precision datatypes and summation operation previously defined.

```
free_mpf(&(MPI_MPF));
free_mpf_op(&(MPI_MPF_SUM));
MPI_Finalize();
```

Fig. 7.2 shows organization of sending and receiving of data type `mpfr_t`, described above.

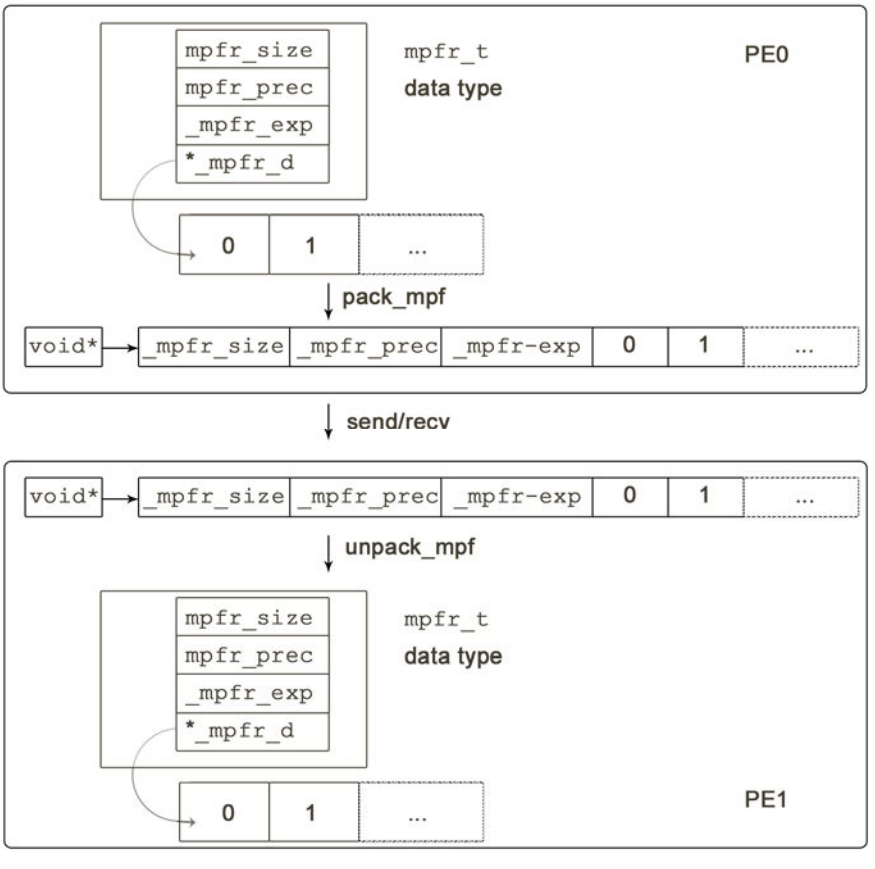

Fig. 7.2 Sending and receiving one mpfr_t variable

7.3 MPIGMP Functions

Consider the basic functions of the library MPIGMP [2]. Library MPIGMP includes a work with all types of variables libraries GMP and MPFR. We consider only the case for floating point numbers.

Type `mpf_t` and the new MPI-operation for the numbers with high accuracy like below:

```
MPI_Datatype gmp_mpf;
#define MPI_MPF gmp_mpf

MPI_Op gmp_mpf_add;
#define MPI_MPF_SUM gmp_mpf_add
```

7.3.1 Function commit_mpf()

Function `commit_mpf()` defines the data type of MPI-arbitrary-precision floating-point `mpi_mpf_t`, available in the communicator comm.

Prototype:

`void commit_mpf(MPI Datatype *mpi_mpf_t, unsigned long prec, MPI Comm comm)`

For example, function `commit_mpf()` defines MPI multiple precision floating-point datatype `MPI_MPF` with precision `MPF_PREC` available in the communicator `comm`.

```
…
#define MPF_PREC 128
…
mpf_set_default_prec(MPF_PREC);
commit_mpf(&(MPI_MPF), MPF_PREC, MPI_COMM_WORLD);
…
```

7.3.2 Function create_mpf_op()

The Function is described in the header file `mpigmp.h`, creates MPI-operation `mpi_mpf_op` in increased accuracy for collective communication that is available in the communicator `comm`.

Prototype:

void create_mpf_op(MPI Op *mpi_mpf_op, void (*func)(void *,void *,int *,MPI Datatype *), MPI Comm comm)

For example, function `create_mpf_op()` creates summation function `MPI_MPF_SUM` for `mpf_t` variable like `MPI_SUM` for `double`:

```
create_mpf_op(&(MPI_MPF_SUM),_mpi_mpf_add,MPI_COMM_WORLD);
```

The function **mpi_mpf_add()** is the summation function for MPI collective communication.

Prototype:

void mpi_mpf_add(void *in, void *ret, int *len, MPI Datatype *datatype)

7.3.3 Memmory Allocation Functions

To select enough buffer to pack a few variables with accuracy **prec** you must use the function `allocbuf_mpf ()`.

Prototype:

void *allocbuf_mpf(unsigned long prec, int incount)

Function returns a pointer to the allocated buffer in which incount multiple precision floating-point numbers of prec bit-length will be stored.

For example:

```
packed_mpf_mypi=allocbuf_mpf(mpf_get_prec(mpf_mypi), 1);
```

size_t get_bufsize_mpf(mpf_t `fdata`**, int** `incount`**)**

Calculate and return the size of buffer that is able to store incount multiple precision variables `fdata`.

7.3.4 Free Functions

At the end of each program the memory must be cleaned. Function for cleaning memory from the MPI-type with arbitrary precision `mpi_mpf_t` is as follows:

Prototype:

void free_mpf(MPI Datatype *mpi_mpf_t)

Free MPI multiple precision floating-point datatype `mpi_mpf_t`.

Function for cleaning memory from the MPI-operation `mpi_mpf_op` for collective communication is as follows:

Prototype:

`void free_mpf_op(MPI Op *mpi_mpf_op)`

Free MPI operation `mpi_mpf_op` for collective communication.

For example, these function are cleaning the memory from MPI multiple precision floating-point datatype `MPI_MPF` and MPI-operation `MPI_MPF_SUM`.

```
free_mpf(&(MPI_MPF));
free_mpf_op(&(MPI_MPF_SUM));
```

7.3.5 Functions pack_mpf()and unpack_mpf()

As previously mentioned, before sending the data on the processes they must be packed. Function `pack_mpf()` packs data into arbitrary precision `mpf_t` buffer `buf`.

Prototype:

`void pack_mpf(mpf_t a, int incount, void *buf)`

Pack the `incount` multiple precision variables a into the buffer `buf`.

After sending the data on the processes they must be unpacked. Function `unpack_mpf()` unpacks the data stored in the buffer in the original data `mpf_t` accuracy.

Prototype:

`void unpack_mpf(void *buf, mpf_t ret, int count)`

Unpack the data stored in the buffer `buf` into `count` multiple precision floatingpoint variables starting at `ret`.

Below is a fragment from the listing 19, which uses the function `allocbuf_mpf()`, `pack_mpf()` and `unpack_mpf()`:

```
…
/* Allocate buffer */
packed_mpf_mypi=allocbuf_mpf(mpf_get_prec(mpf_mypi), 1);
packed_mpf_pi=allocbuf_mpf(mpf_get_prec(mpf_pi), 1);

/* Pack */
     pack_mpf(mpf_mypi, 1, packed_mpf_mypi);
```

```
        pack_mpf(mpf_pi, 1, packed_mpf_pi);

        //MPI_Send/Reduce
  MPI_Reduce(packed_mpf_mypi,packed_mpf_pi,1,MPI_MPF,MPI_MPF_SUM
,0,MPI_COMM_WORLD);

        /* Unpack */
  unpack_mpf(packed_mpf_pi, mpf_pi, 1);
```

We considered the main functions to work together MPI and GMP for `mpf_t` type. MPIGMP library has also the same functions for `mpf_z` and `mpf_q` types.

MPIGMP library has set of *Functions for Send/Receive/Collective Communications*. For example, function `_mpi_bcast_mpfvector()` for Broadcast vector of type of `mpf_t` from process `P0` to other is presented below.

```
  void _mpi_bcast_mpfvector(MPFVector vec, MPI_Comm comm)
  {
        long int i, dim;
        unsigned long prec;
        void *buf;
        int procs, num_procs;
        MPI_Status st;

        MPI_Comm_size(comm, &num_procs);
        MPI_Comm_rank(comm, &procs);

        if(procs == 0)
        {
               dim = vec->dim;
               prec = vec->prec;
        }

        MPI_Bcast(&dim, 1, MPI_LONG, 0, comm);
        MPI_Bcast(&prec, 1, MPI_UNSIGNED_LONG, 0, comm);
        vec->dim = dim;
        vec->prec = prec;

        /* P0 -> Pi */

        if(dim <= 0)
               return;

        buf = allocbuf_mpf(prec, dim);

  #ifndef MPICH2
        if(procs == 0)
        {
               for(i = 1; i < num_procs; i++)
               {
                      pack_mpf(get_mpfvector_i(vec, 0),dim, buf);
                      MPI_Send(buf,  dim,MPI_MPF,i, 0,comm);
               }
```

```
        }
        else
        {
                MPI_Recv(buf, dim,MPI_MPF,0, 0, comm, &st);
                unpack_mpf(buf, get_mpfvector_i(vec, 0), dim);
        }
#else
        if(procs == 0)
                pack_mpf(get_mpfvector_i(vec, 0), dim, buf);

        MPI_Bcast(buf, dim, MPI_MPF, 0, comm);
        unpack_mpf(buf, get_mpfvector_i(vec, 0), dim);
#endif

        free(buf);
}
```

All functions are available in MPIBNCPack [2, 5]. There are: *Complex number arithmetic functions, Basic linear computation functions, Functions of Krylov subspace method, Functions of Durand-Kerner-Aberth Method, Numerical Integrator functions, Functions for Send/Receive/Collective Communication*.

In the next Chapter we we'll be consider parallel MPI-programs for solution SLAEs with arbitrary precision using MPIGMP library.

References

1. GMP Library, http://www.gmplib.org/
2. Kouya, T.: A Brief Introduction to MPIGMP & MPIBNCpack (2008), http://na-inet.jp/na/bnc/brief_intro_mpibncpack.pdf (accessed August 21, 2011)
3. MPFR Library, http://www.mpfr.org/
4. MPI-forum, http://www.mpi-forum.org
5. MPIGMP Library, http://na-inet.jp/na/bnc

Chapter 8
Parallel Method for Solution SLAE with Multiple Precision

Elena A. Nikolaevskaya, Alexandr N. Khimich, Tamara V. Chistyakova, and Victor V. Polyanko

Abstract. We want our readers to have more examples of programs using GMP and MPFR library. Chapter 8 consists of two parallel algorithms and appropriate parallel programs for solving SLAE by LU-decomposition and SVD method. These are parallel MPI-programs using MPIGMP and GMP library. Considered programs were tested on supercomputer with different numbers of processes for different orders of matrices of the problem.

Elena A. Nikolaevskaya
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: elena_nea@ukr.net

Alexandr N. Khimich
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: khimich_ic@mail.ru

Tamara V. Chistyakova
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: dept150@insyg.kiev.ua

Victor V. Polyanko
Department of Numerical Methods and Computer Modeling,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Glushkova,40, Kiev, Ukraine
e-mail: polyanko_victor@ukr.net

E. Nikolaevskaya et al.: Programming with Multiple Precision, SCI 397, pp. 135–209.
springerlink.com

8.1 LU-Decomposition

Block algorithm for solving LS enables to reduce implementation of algorithms to perform operations in processes over separate blocks of matrices which they are broken into instead of performing operations over separate elements of matrices. As a rule, dimensions of blocks are determined taking into account the by volume of computers' cache memory. This provides high speed of matrix-vector operations.

During solving LS $Ax=b$ algorithm for LU-factorization reduces matrix A to the form $A=PLU$, where P – is a matrix of permutations, L is a lower triangular matrix (with units on principal diagonal) while U is an upper triangular matrix. This enables to replace the solving of one system (1.3) with general matrix by the solving of two LS $Ly=b$ and $Ux=y$ with triangular matrices.

Let us assume that n-th order matrices A, L and U are to be divided into square blocks of order s.

At the k-th step of algorithm ($k = 1, 2, \ldots$) let us represent a sub-matrix $A^{(k)}$ (diagonal block of matrix A) of order $r = n-(k-1)s$ which contains the last r rows and r columns of the matrix A in the formula

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = P\begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} = P\begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix},$$

where block A_{11} is of size $s\times s$, block A_{12} is $s\times(r-s)$, block A_{21} is $(r-s)\times s$ and block A_{22} is $(r-s)\times(r-s)$.

Let us carry out a sequence of LU transformations (see Section 5.4) on the part of matrix (Fig. 8.1) consisting of blocks A_{11} and A_{21}:

$$l_{im} = \frac{a_{im}}{a_{mm}}, \quad l_{ij} = a_{ij} - l_{im}a_{mj}, \tag{8.1}$$

where $i = \overline{1,s}$, $m = \overline{1,s}$, $j = \overline{m+1,r}$.

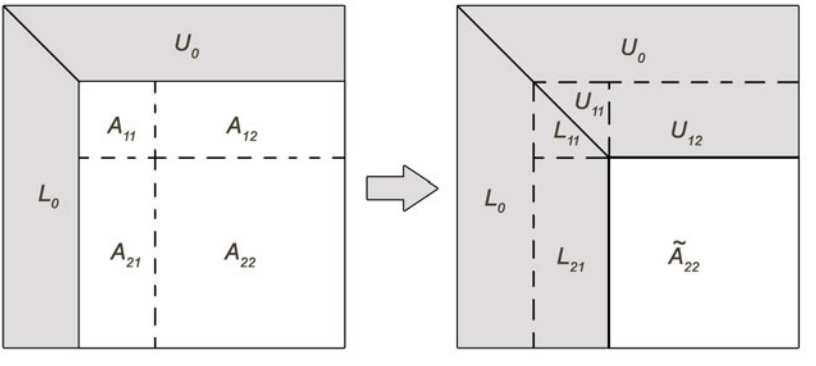

Fig. 8.1 Scheme of the implementation of block algorithm for LU-decomposition

As a result we get matrices L_{11}, L_{21}. Further let us find from the unknown matrices U_{11} and $\tilde{A}_{22}$ (the latter is formed on the place of A_{22}):

$$U_{12} \leftarrow (L_{11})^{-1} A_{12}, \tag{8.2}$$

$$\tilde{A}_{22} \leftarrow A_{22} - L_{21}U_{12} = L_{22}U_{22}. \tag{8.3}$$

Further, the value of k is increased by 1 and computations are repeated for the new value of k, the matrix $\tilde{A}_{22}$ being considered as a sub-matrix $A^{(k)}$.

It is easily seen that total number of arithmetic operations required for the block LU-decomposition (5.2) of square matrix by LU-decomposition is estimated, as in the case of non-block version, by value $O_1 \approx 2n^3/3 + O(n^2)$.

8.1.1 Block Parallel Algorithm of LU-Decomposition

Two-dimensional block-cyclic distribution of matrix elements between processes is used in the solving of LS with dense non-singular matrices by parallel block algorithms.

For the distribution of original n-th order matrix A between p processes it should be represented in the form

$$A = \begin{pmatrix} A_{11}^* & A_{12}^* & \dots & A_{1p}^* \\ A_{21}^* & A_{22}^* & \dots & A_{2p}^* \\ \dots & \dots & \dots & \dots \\ A_{q1}^* & A_{q2}^* & \dots & A_{qp}^* \end{pmatrix},$$

where

$$A_{ij}^* = \begin{pmatrix} A_{ij} & A_{ij+p} & \dots & A_{ij+(p-1)p} \\ A_{i+qj} & A_{i+qj+p} & \dots & A_{i+qj+(p-1)p} \\ \dots & \dots & \dots & \dots \\ A_{i+(q-1)qj} & A_{i+(q-1)qj+p} & \dots & A_{i+(q-1)qj+(p-1)p} \end{pmatrix}.$$

Here A_{ij} is a block of elements of the matrix A of order s.

Matrix of the system is distributed between processes so that $A_{ij}^* \in P_{ij}$, where i and j are Cartesian coordinates of process on the two-dimensional $p \times q$ grid. Let us illustrate such distribution by the following example.

Let n=8, s=2, p=2, q=2, then for the matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{18} \\ a_{21} & a_{22} & \dots & a_{28} \\ \dots & \dots & \dots & \dots \\ a_{81} & a_{82} & \dots & a_{88} \end{pmatrix}$$

its block representation has a form:

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix},$$

where

$$A_{ij} = \begin{pmatrix} a_{2i-1,2j-1} & a_{2i-1,2j} \\ a_{2i,2j-1} & a_{2i,2j} \end{pmatrix}.$$

In this case block-cyclic distribution for the matrix A takes the form

$$A = \begin{pmatrix} A_{11}^{*} & A_{12}^{*} \\ A_{21}^{*} & A_{22}^{*} \end{pmatrix},$$

where

$$A_{11}^{*} = \begin{pmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{pmatrix}, \; A_{12}^{*} = \begin{pmatrix} A_{12} & A_{14} \\ A_{32} & A_{34} \end{pmatrix}, \; A_{21}^{*} = \begin{pmatrix} A_{21} & A_{23} \\ A_{41} & A_{43} \end{pmatrix}, \quad A_{22}^{*} = \begin{pmatrix} A_{22} & A_{24} \\ A_{42} & A_{44} \end{pmatrix}.$$

Thus, each process P_{ij} contains elements of the matrix A_{ij}^{*} as well as corresponding blocks of matrix of the right-hand sides.

Each vector of right-hand sides is distributed cyclically by $p \times q$ blocks between processes of the first column of processes' grid. Hence, process P_{k1} contains elements numbered ks, $ks+1$, $ks+2,\ldots,ks+s-1$, $(k+p)s$, $(k+p)s+1$, $(k+p)s+2,\ldots$, $(k+p)s+s-1,\ldots$, where s is the number of blocks equal to its counterpart in the matrix distribution. In general, let us assume that $\frac{n}{pqs}$ is an integer.

Two-dimensional block-cyclic parallel algorithm for LU-factorization [2, 12, 13]. Let us consider the implementation of this algorithm at the k-th step. Taking into account the fact that algorithm is determined by formulas (8.1), (8.2), let us draw the reader's attention to the data exchanges between processes. A process which contains block A_{11} will be called a leading process of the k-th step. Then algorithm is implemented as follows:

1) in the s-step cycle the LU transformation over blocks A_{11} and A_{12} is performed, i.e. maximum element in sought in the leading column; exchange by elements of the leading row and row containing maximum elements is carried out; the leading and all subsequent rows are transformed according to (8.1); as a result of transformation, blocks A_{11} and A_{12} are located on the place of blocks L_{11} and L_{12}, respectively;

2) the matrix $(L_{11})^{-1}$ is evaluated (since L_{11} is completely contained in the leading process, $(L_{11})^{-1}$ is evaluated within the limits of one process); then $(L_{11})^{-1}$ is broadcasted to all processes of the corresponding row of the process' grid, i. e. if P_{ij} is the leading process then $(L_{11})^{-1}$ is sent to processes P_{il}, $l = \overline{1, p}$;

3) processes independently on each other evaluate a product of the obtained block $(L_{11})^{-1}$ by blocks A_{12} distributed to them according to (8.2); as a result elements of A_{12} are replaced by elements of U_{12};

4) blocks L_{21} are broadcasted horizontally over the rows of the processes' grid, while U_{12} are broadcasted vertically over columns. After this, a transformation of the corresponding blocks of matrix A_{22} according to (8.3) takes place in every process.

Coefficients of acceleration and efficiency of algorithms will be determined according to formulas $S_{pq} = \frac{T_1}{T_{pq}}$, $E_{pq} = \frac{S}{pq}$, where pq is the number of processes; T_1 is the time required for the performance of algorithm on one process; T_{pq} is the time required for the performance of algorithm on the grid ($p \times q$) of processes. To determine T_{pq} let us make use of the formula $T_{pq} = Nt + Mt_o + Qt_c$, where N is the number of arithmetic operations (addition and multiplication); M is the number of exchanges; Q is the number of inter-processes synchronizations.

For two-dimensional block-cyclic parallel algorithm for LU-factorization the number of arithmetic operations is mainly determined by stages 3 and 4 of the algorithm and is estimated by value $N \approx 2n^3/(3pq)$.

To evaluate coefficients of acceleration and efficiency for this algorithm it is necessary to determine basic components for the amount of processes performed at the k-th step of algorithm. The broadcasting of matrices $(L_{11})^{-1}$, L_{21}, U_{12} requires $(p-1)s^2$ and $((p-1)(n-ks)/q+(q-1)(n-ks)/p)s$ transfers of elements. Then after k steps we get $M \approx n^2(p+q)(p+q-1)/(2pq)$. In doing so at each step $Q = 4n(p+q-2)/s$ synchronizations is performed. Thus, we have

$$T_{pq} = \frac{2n^3 t}{3pq} + \frac{n^2(p+q)(p+q-1)}{2pq} t_o + \frac{4n(p+q-2)}{s} t_c .$$

Hence, for coefficients of acceleration and efficiency we get:

$$S_{pq} = \frac{T_1}{T_{pq}} \approx pq\left(1 + \frac{3(p+q)(p+q-1)}{4n}\tau_o + \frac{6pq(p+q-2)}{sn^2}\tau_c\right)^{-1},$$

$$E_{pq} = \frac{S}{pq} \approx \left(1 + \frac{3(p+q)(p+q-1)}{4n}\tau_o + \frac{6pq(p+q-2)}{sn^2}\tau_c\right)^{-1}.$$

Consider function for solution of SLAEs by LU decomposition.

Listing 8.1 Parallel program for solution SLAEs by LU decomposition

```
//main program
int main(int argc,char*argv[])
{
int n;//dimention of main matrix 0<n
int r;//quantity of right parts
int *P;//permutation vector
int info;//exit code
int dims[2];//dimentions of process mesh
int coord[2];//coordinates of process
MPI_Status status;
int myid;//number of current process
```

```
  int size;//number of processes
  int nx;//number of matrix columns in process
  int ny;//number of matrix rows in process
  int s;//size of matrix block
  int i, j;//temporary values
  double *e, *e1;//estimates
  double time;//time
  double ea, eb;//maximum relative errors of matrix and right
parts
  double *cond;//condition number of the matrix
  int * PP;// temporary array
  int * P1;//temporary array
  int kod;//indicate is it nesseccary to show on screen input
data
  char*fname1;// file with matrix
  char*fname2;// file with right part
  int modef;  // 0 - matrix from formula,
              // 1 - matrix and right partfrom file,
              // 2 - matrix and right part from two files,
              // 3 - all data from one file.
  int model;  // 0 - english
              // 1 - russian
              // 2 - germany
              // 3 - ukrainian
  int ndim;//number of dimentions
  int periods[2];//periods of mesh
  int reorder;//reorder of mesh
  int cart[2];//array for mesh division
  MPI_Comm comm_cart;//communicator with decart topology
  MPI_Comm comm_cart_x;//communicator with columns of processes
  MPI_Comm comm_cart_y;//communicator with rows of processes
  double *A;  //main matrix n*n
  double *B;  //matrix of right part
  double * Block;//temp array
  double * Col;//temp array
  double * U12;//temp array
  double * L21;//temp array
  double * AC;//copy of A
  double * BC;//copy of B
  double * A1;//array for gathering A
  double * B1;//array for gathering B

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
  ndim = 2;
  reorder = 1;
  dims[0] = 1;
  //define number of processes in each dimention of mesh
  for(i = 1; i * i <= size; i++)
  if(size==(size/i)*i)
  dims[0]=i;
  dims[1]=size/dims[0];
  periods[0]=1;periods[1]=1;
  MPI_Cart_create(MPI_COMM_WORLD,ndim,dims,periods,reorder,&comm
_cart);
  MPI_Cart_coords(comm_cart,myid,ndim,coord);
  cart[0]=1;
  cart[1]=0;
  MPI_Cart_sub(comm_cart,cart,&comm_cart_x);
  cart[0]=0;
  cart[1]=1;
  MPI_Cart_sub(comm_cart,cart,&comm_cart_y);
  //here you can input parameters of the task
  info=0;
  modef=0;
  model=0;
  kod=1;         // 1 - print
                 // 2 - not print parameters of the task
  fname1="A15_100abc";
  fname2="A15_100b";
  n = 1000;
  nx=n;
  ny=n;
  r=1;
  s = 10;
  //read command line
  if(argc==3)
  {
        fname1 = argv[1];
        model = atoi(argv[2]);
  }
  if(modef == 3)
  ReadAndBcastParametrs(fname1, myid, size, &n, &r, &ea, &eb);
  //calculate number of matrix elements, which will be storaged
in the current process
  i=n/s;
  nx=i/dims[0];
  j=i-nx*dims[0];
```

```
if(coord[0]<j)
      nx++;
      nx*=s;
if(coord[0]==j)
      nx+=n%s;
      ny=i/dims[1];
      j=i-ny*dims[1];
if(coord[1]<j)
      ny++;
      ny*=s;
if(coord[1]==j)
      ny+=n%s;
//print first part of protocol
if(myid==0) {
      PrintStartTime(fname1);
      ShowData(n,ea,eb,model,1,r);
}
//allocate memory for main and temporary arrays
if((size>1))
{
      if(coord[0]==0)
      {
              A1=(double*)malloc(n*n*sizeof(double));
              B1=(double*)malloc(n*r*sizeof(double));
              P1=(int*)malloc(n*r*sizeof(int));
      }
      else
      {
              A1=(double*)malloc(sizeof(double));
              B1=(double*)malloc(sizeof(double));
              P1=(int*)malloc(sizeof(int));
      }
}
else
{
      A1=(double*)malloc(nx*ny*sizeof(double));
      B1=(double*)malloc(sizeof(double));
      P1=(int*)malloc(sizeof(int));
}
PP=(int*)malloc(2*size*sizeof(int));
U12=(double*)malloc(nx*s*sizeof(double));
L21=(double*)malloc(ny*s*sizeof(double));
A=(double*)malloc(nx*ny*sizeof(double));
B=(double*)malloc(ny*r*sizeof(double));
```

```
  AC=(double*)malloc(nx*ny*sizeof(double));
  BC=(double*)malloc(ny*r*sizeof(double));
  P=(int*)malloc(ny*sizeof(int));
  Block=(double*)malloc(s*s*sizeof(double));
  Col=(double*)malloc(4*(nx+1)*sizeof(double));
  //check for errors
  if((A==NULL)||(B==NULL)||(AC==NULL)||(BC==NULL)||(P==NULL)||(B
lock==NULL)||(Col==NULL)||(U12==NULL)||(L21==NULL)||(PP==NULL)||(
P1==NULL)||(A1==NULL)||(B1==NULL))
  info=-2;
  if(size>1)
        info = GatherErrorInfo(PP, info);
  if(info==0)
  {
  BuildAndScatterMatrixes(modef,fname1, fname2, myid, size, n,
r, A, B, A1, B1, nx, ny, s, comm_cart_x, comm_cart_y, dims,
coord);
        if(size > 1)
                info = GatherErrorInfo(PP, info);
  }
  if(info==0)
  {
        MPI_Barrier(MPI_COMM_WORLD);
        time = MPI_Wtime();
        //run main program, which find solution of the system
and errors
        info = GausSolver(A, B, P, Block, Col, U12, L21, A1, B1,
AC, BC, P1, PP, n, r, nx, ny, s, comm_cart_x, comm_cart_y, dims,
coord,info, ea, eb, cond, e, e1);
        MPI_Barrier(MPI_COMM_WORLD);
        time = MPI_Wtime()-time;
  }

  //if solution is successful, gather result and print on screen
  if(info>=0)
  {
        if(size>1)
        {
                if(coord[0] == 0)
                GatherFromAllDB(B, B1, PP, s, n, r, comm_cart_y,
dims[1], coord[1]);
                if(coord[1] + coord[0] == 0)
                {
                        ShowSolution(B1,n,r,r,model);
```

```
                        WriteFile("result.out",n,r,B1);
                }
        }
        else
        {
                ShowSolution(B,n,r,r,model);
                WriteFile("result.out",n,r,B);
        }
        if(myid==0)
                ShowAnalysis(model, info, e[0], e1[0], cond[0]);
  }
  if(myid==0)
  {
        ShowError(model,info);
        printf("\n #result=%d \n",info);
        if(model==2)
                printf("\n   Zeit:   %f   \n   Die   Nummer   des
Prozessores: %d \n \n ",time, size);
        else
                printf("\n Time: %f \n Proc number: %d \n \n
",time, size);
  }
  //free arrays and exit MPI
  free(A);
  free(B);
  free(P);
  free(Block);
  free(Col);
  free(U12);
  free(L21);
  free(PP);
  free(A1);
  free(B1);
  free(P1);
  MPI_Finalize();
  return info;
  }
```

//Function for solution SLAE by Gauss method

```
int DenseSolver(double *A, double *B, int *P, double *Block,
double *Col, double*U12, double*L21, double *A1, double *B1, double
*AC, double *BC, int *P1, int *PP, int n, int r, int nx, int ny,
int s, MPI_Comm comm_cart_x, MPI_Comm comm_cart_y,int *dims, int
*coord,int info, double ea, double eb, double *cond, double *e,
double *e1)
```

```
  {
  int i, j;
  //make copies of A and B
  for(i = 0; i < ny * nx; i++)
        AC[i] = A[i];
  for(i = 0; i < ny * r; i++)
        BC[i] = B[i];
  //make decomposition of main matrix and find condition number
  info                                                    =
CalculateConditionNumber(A,P,Block,Col,U12,L21,n,nx,ny,s,comm_car
t_x,comm_cart_y,dims,coord,cond, info, PP, P1,B1);
  if(info < 0)
        return info;
  //if successfully, use permutation vector for changing order
of elements of B
  if(coord[0] == 0)
  {
        if(dims[1]==1)
                SwapB(B,P,r,n);
        else
        {
  GatherFromAllIB(P,P1,PP,s,n,1,comm_cart_y,dims[1],coord[1]);
  GatherFromAllDB(B,B1,PP,s,n,r,comm_cart_y,dims[1],coord[1]);
  if(coord[1] == 0)
        SwapB(B1,P1,r,n);
        SendToAllDB(B,B1,PP,s,n,r,comm_cart_y,dims[1],coord[1]);
        }
  }
  //find solution
  for(i=0;i<r;i++)
        info = CalculateSolution(A, &B[i*ny], Col, Block, r, n,
nx, ny, s, 0, comm_cart_x, comm_cart_y,dims, coord, info);

  //find estimates of errors
        info = SolutionAnalysis(A, BC, B, A1, B1, P, Col, Block,
PP, P1, AC, r, n, nx, ny, s, ea, eb, cond, e, e1, comm_cart_x,
comm_cart_y, dims, coord, info);

  return info;
  }

  //main function of matrix factorization
  int MatrixFactorization(double *A, int *P,double *Block,
double *Col, double*U12, double*L21, int n, int nx, int ny, int
```

```
s, MPI_Comm comm_cart_x, MPI_Comm comm_cart_y,int *dims, int
*coord,int info, int*PP)
  {
  int i,j,k,k1,k2,k3,j1=10; //temp
  int kx,ky; //coords of left upper corner of the leading block
  //initialization of P
  for(i=0;i<ny;i++)
       P[i] = (coord[1] + i/s*dims[1])*s + i%s;
  //main cycle
  for(k=0;k<(n-1)/s;k++)
  if(info==0)
  {
       kx=(k/dims[0]+(coord[0]<k%dims[0]))*s;
       ky=(k/dims[1]+(coord[1]<k%dims[1]))*s;
       //modification of the leading block
  info =
BlockFactorization(A,P,Col,n,s,nx,ny,s,k,comm_cart_x,comm_
cart_y,dims,coord,info, PP);
  if(info < 0)
       return info;
       //modification of L11
       if(k%dims[0]==coord[0])
       {
              if(k%dims[1]==coord[1])
                     BlockL11(A,Block,kx,ky,nx,ny,s);
                     kx+=s;
       }
       //calculation of U12
       if(k%dims[1]==coord[1])
       {
              if(dims[0]>1)
       MPI_Bcast(Block,s*s, MPI_DOUBLE,k%dims[0],comm_cart_x);
                     BlockU12(A,Block,Col,kx,ky,nx,ny,s,k);
                     ky+=s;
       }
       //sending U12 in vertical direction
       if(k%dims[1]==coord[1])
              for(i=0;i<s;i++)
              {
                     k3=(i+ky-s)*nx;
                     for(j=kx;j<nx;j++)
                            U12[i+j*s]=A[k3+j];
              }
       if(dims[1]>1)
```

```
            MPI_Bcast(&U12[kx*s],(nx-kx)*s,
MPI_DOUBLE,k%dims[1],comm_cart_y);

        //sending L21 in horizontal direction
        if(k%dims[0]==coord[0])
            for(i=ky;i<ny;i++)
            {
                k2=i*s;
                k3=i*nx+kx-s;
                for(j=0;j<s;j++)
                    L21[k2+j]=A[k3+j];
            }
        if(dims[0]>1)
            MPI_Bcast(&L21[ky*s],(ny-ky)*s,
MPI_DOUBLE,k%dims[0],comm_cart_x);
        //modification of the rest matrix (block A22)
        BlockA22(A,U12,L21,kx,ky,nx,ny,s,k);
    }
    k=(n-1)/s;
    k1=n-k*s;
    //modification of the last block (which has rows less then s)
    info                                                    =
BlockFactorization(A,P,Col,n,k1,nx,ny,s,k,comm_cart_x,comm_cart_y
,dims,coord,info, PP);
    return info;
    }
```

8.1.2 *Parallel Program for Solution SLAEs by LU-Decomposition with Multiple Precision*

Consider an example of solving system of linear algebraic equations

$$Ax=b \tag{8.4}$$

where matrix A is square, symmetric and nonsingular. Its elements a_{ij} are set by the formula:

$$A = \left\{a_{ij}\right\}_1^n,\ a_{ij} = n - \max(i, j) + 1, \qquad i, j = 1, 2, \ldots, n,$$

where n – order of matrix. The right part has the view like vector-columns of the identity matrix. Suppose, for example, the right part is given by a vector, whose elements have values:

$$b = \left\{b_i\right\}_1^n,\ b_1 = 1,\ b_i = 0, \qquad i = 2, 3, \ldots, n.$$

Exact solution of the system with this right side looks like:

$$x_1 = 1, x_2 = -1, x_i = 0, \quad i = 3, 4, \ldots, n.$$

Consider program for solving linear system equations by method of singular value decomposition of matrix. Since the listing of the program takes up much

space, we consider only the basic functions of the program, others are available on the disk that is attached to the book.

Listing 8.2 presents the program `GaussSolver.c`. Program `GaussSolver` is designed to solve systems of linear algebraic equations of the form $AX = B$ with a dense nonsingular matrix A and multiple right sides B by the LU-decomposition with partial selection of main element with the distributed data in environment of parallel programming MPI.

In addition, the program calculates:

- Evaluation of the condition number of matrix A;
- Evaluation of inherited and computational errors of results of the solution.

The solution of the problem is provided in the presence of at least one CPU and required memory. In each process of the first ([*n*/*s*]-*q x* [*n*/*s*]-p*)* processes will save elements *q*= ([[*n*/*s*]/ *sizeq*]+1)*s* rows and p= ([[*n*/*s*]/ *sizep*]+1)*s* columns, and *q*= [[*n*/*s*]/*sizeq*]*s* rows and *q*= [[*n*/*s*]/*sizep*]*s* column - in other processes, where *n* is the order of the matrix of the system, *sizeq* and *sizep* are dimensions of a grid of processes, which solve the problem. Matrices *A*, *B* are distributed by two-dimensional row-cyclic method (distributed *s*-consecutive located rows of matrices and *s*-consecutive located columns).

Listing 8.2. Parallel program for solution SLAEs by LU decomposition

```
//main program
int main(int argc,char*argv[])
{
int n;//dimention of main matrix 0<n
int r;//quantity of right parts
int *P;//permutation vector
int info;//exit code
int dims[2];//dimentions of process mesh
int coord[2];//coordinates of process
MPI_Status status;
int myid;//number of current process
int size;//number of processes
int nx;//number of matrix columns in process
int ny;//number of matrix rows in process
int s;//size of matrix block
int i, j;//temporary values
mpf_t *e, *e1;//estimates
double time;//time
mpf_t ea, eb;//maximum relative errors of matrix and right
parts
mpf_t *cond;//condition number of the matrix
int * PP;// temporary array
int * P1;//temporary array
```

```
  int kod;//indicate is it nesseccary to show on screen input
data
  char*fname1;// file with matrix
  char*fname2;// file with right part
  int modef;  // 0 - matrix from formula,
              // 1 - matrix and right partfrom file,
              // 2 - matrix and right part from two files,
              // 3 - all data from one file.
  int model;  // 0 - english
              // 1 - russian
              // 2 - germany
              // 3 - ukrainian
  int ndim;   //number of dimentions
  int periods[2];//periods of mesh
  int reorder;//reorder of mesh
  int cart[2];//array for mesh division
  MPI_Comm comm_cart;//communicator with decart topology
  MPI_Comm comm_cart_x;//communicator with columns of processes
  MPI_Comm comm_cart_y;//communicator with rows of processes
  mpf_t *A;//main matrix n*n
  mpf_t *B;//matrix of right part
  mpf_t * Block;//temp array
  mpf_t * Col;//temp array
  mpf_t * U12;//temp array
  mpf_t * L21;//temp array
  mpf_t * AC;//copy of A
  mpf_t * BC;//copy of B
  mpf_t * A1;//array for gathering A
  mpf_t * B1;//array for gathering B

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  ndim = 2;
  reorder = 1;
  dims[0] = 1;
  //define number of processes in each dimention of mesh
  for(i = 1; i * i <= size; i++)
  if(size==(size/i)*i)
       dims[0]=i;
  dims[1]=size/dims[0];
  periods[0]=1;periods[1]=1;
  MPI_Cart_create(MPI_COMM_WORLD,ndim,dims,periods,reorder,&comm
_cart);
```

```
MPI_Cart_coords(comm_cart,myid,ndim,coord);
cart[0]=1;
cart[1]=0;
MPI_Cart_sub(comm_cart,cart,&comm_cart_x);
cart[0]=0;
cart[1]=1;
MPI_Cart_sub(comm_cart,cart,&comm_cart_y);

//init mpf operations
mpf_set_default_prec(MPF_PREC);
commit_mpf(&(MPI_MPF), MPF_PREC, MPI_COMM_WORLD);
create_mpf_op(&(MPI_MPF_SUM), _mpi_mpf_add, MPI_COMM_WORLD);
create_mpf_op(&(MPI_MPF_MAX), _mpi_mpf_max, MPI_COMM_WORLD);
commit_mpf(&(MPI_MPF), MPF_PREC, comm_cart_x);
create_mpf_op(&(MPI_MPF_SUM), _mpi_mpf_add, comm_cart_x);
create_mpf_op(&(MPI_MPF_MAX), _mpi_mpf_max, comm_cart_x);
commit_mpf(&(MPI_MPF), MPF_PREC, comm_cart_y);
create_mpf_op(&(MPI_MPF_SUM), _mpi_mpf_add, comm_cart_y);
create_mpf_op(&(MPI_MPF_MAX), _mpi_mpf_max, comm_cart_y);

e = ALLOC_ARRAY_MPF(MPF_PREC, 1);
e1 = ALLOC_ARRAY_MPF(MPF_PREC, 1);
cond = ALLOC_ARRAY_MPF(MPF_PREC, 1);

mpf_init(cond[0]);
mpf_init(e1[0]);
mpf_init(e[0]);
mpf_init(ea);
mpf_init(eb);
//here you can input parameters of the task
info=0;
modef=0;
model=0;
kod=1;      // 1 - print
            // 2 - not print parameters of the task
fname1="A15_100abc";
fname2="A15_100b";
n = 1000;
nx=n;
ny=n;
r=1;
mpf_init(ea);
mpf_init(eb);
s = 10;
```

```
  //read command line
  if(argc==3){
       fname1 = argv[1];
       model = atoi(argv[2]);
  }
  if(modef == 3)
  ReadAndBcastParametrs(fname1, myid, size, &n, &r, &ea, &eb);

  //calculate number of matrix elements, which will be storaged
in the current process
  i=n/s;
  nx=i/dims[0];
  j=i-nx*dims[0];
  if(coord[0]<j)
       nx++;
       nx*=s;
  if(coord[0]==j)
       nx+=n%s;
       ny=i/dims[1];
       j=i-ny*dims[1];
       if(coord[1]<j)
       ny++;
       ny*=s;
  if(coord[1]==j)
       ny+=n%s;
  //print first part of protocol
  if(myid==0)
  {
       PrintStartTime(fname1);
       ShowData(n,ea,eb,model,1,r);
  }
  //allocate memory for main and temporary arrays
  if(size>1)
  {
       if(coord[0]==0)
       {
              A1 = ALLOC_ARRAY_MPF( MPF_PREC, n * n);
              B1 = ALLOC_ARRAY_MPF( MPF_PREC, n * r);
              P1=(int*)malloc(n*r*sizeof(int));
              for(i = 0; i < n * r; i++)
                     mpf_init(B1[i]);
              for(i = 0; i < n * n; i++)
                     mpf_init(A1[i]);
       }
```

```
        else
        {
                A1 = ALLOC_ARRAY_MPF( MPF_PREC, 1);
                B1 = ALLOC_ARRAY_MPF( MPF_PREC, 1);
                P1=(int*)malloc(sizeof(int));
                mpf_init(A1[0]);
                mpf_init(B1[0]);
        }
}
else
{
        A1 = ALLOC_ARRAY_MPF( MPF_PREC, nx * ny);
        B1 = ALLOC_ARRAY_MPF( MPF_PREC, 1);
        P1=(int*)malloc(sizeof(int));
        for(i = 0; i < nx * ny; i++)
                mpf_init(A1[i]);
                mpf_init(B1[0]);
}
PP=(int*)malloc(2*size*sizeof(int));
P=(int*)malloc(ny*sizeof(int));
A = ALLOC_ARRAY_MPF( MPF_PREC, nx * ny);
B = ALLOC_ARRAY_MPF( MPF_PREC, ny * r);
U12 = ALLOC_ARRAY_MPF( MPF_PREC, nx * s);
L21 = ALLOC_ARRAY_MPF( MPF_PREC, ny * s);
AC = ALLOC_ARRAY_MPF( MPF_PREC, nx * ny);
BC = ALLOC_ARRAY_MPF( MPF_PREC, ny * r);
Col = ALLOC_ARRAY_MPF( MPF_PREC, 4 * (nx + 1));
Block = ALLOC_ARRAY_MPF( MPF_PREC, s * s);
//init values of mpf-arrays
for(i = 0; i < nx * ny; i++)
{
        mpf_init(A[i]);
        mpf_init(AC[i]);
}
for(i = 0; i < ny * r; i++)
{
        mpf_init(B[i]);
        mpf_init(BC[i]);
}
for(i = 0; i < nx * s; i++)
        mpf_init(U12[i]);
for(i = 0; i < ny * s; i++)
        mpf_init(L21[i]);
for(i = 0; i < 4 * (nx + 1); i++)
```

```
        mpf_init(Col[i]);
    for(i = 0; i < s * s; i++)
        mpf_init(Block[i]);
    //check for errors
    if((A==NULL)||(B==NULL)||(AC==NULL)||(BC==NULL)||(P==NULL)||(B
lock==NULL)||(Col==NULL)||(U12==NULL)||(L21==NULL)||(PP==NULL)||(
P1==NULL)||(A1==NULL)||(B1==NULL))
    info=-2;
    if(size>1)
        info = GatherErrorInfo(PP, info);
    if(info==0)
    {
    BuildAndScatterMatrixes(modef,fname1, fname2, myid, size, n,
r, A, B, A1, B1, nx, ny, s, comm_cart_x, comm_cart_y, dims,
coord);
    if(size > 1)
        info = GatherErrorInfo(PP, info);
    }
    if(info==0)
    {
    MPI_Barrier(MPI_COMM_WORLD);
    time = MPI_Wtime();
    //run main program, which find solution of the system and er-
rors
    info = GausSolver(A, B, P, Block, Col, U12, L21, A1, B1, AC,
BC, P1, PP, n, r, nx, ny, s, comm_cart_x, comm_cart_y, dims,
coord,info, ea, eb, cond, e, e1);
    MPI_Barrier(MPI_COMM_WORLD);
    time = MPI_Wtime()-time;
    }

    //if solution is successful, gather result and print on screen
    if(info>=0)
    {
        if(size>1)
        {
                if(coord[0] == 0)
    GatherFromAllDB(B,B1,PP,s,n,r,comm_cart_y,dims[1],coord[1]);
                if(coord[1] + coord[0] == 0)
                {
                        ShowSolution(B1,n,r,r,model);
                        WriteFile("result.out",n,r,B1);
                }
        }
```

```
        else
        {
                ShowSolution(B,n,r,r,model);
                WriteFile("result.out",n,r,B);
        }
        if(myid==0)
                ShowAnalysis(model, info, e[0], e1[0], cond[0]);
  }
  if(myid==0)
  {
        ShowError(model,info);
        printf("\n #result=%d \n",info);
        if(model==2)
                printf("\n Zeit: %f \n Die Nummer des Prozessores:
%d \n \n ",time, size);
        else
                printf("\n  Time:  %f  \n  Proc  number:  %d  \n  \n
",time, size);
  }

  //free arrays and exit MPI
  if(0 != A)FREE_ARRAY_MPF(A);
  if(0 != B)FREE_ARRAY_MPF(B);
  if(0 != Block)FREE_ARRAY_MPF(Block);
  if(0 != Col)FREE_ARRAY_MPF(Col);
  if(0 != U12)FREE_ARRAY_MPF(U12);
  if(0 != L21)FREE_ARRAY_MPF(L21);
  if(0 != A1)FREE_ARRAY_MPF(A1);
  if(0 != B1)FREE_ARRAY_MPF(B1);
  if(0 != AC)FREE_ARRAY_MPF(AC);
  if(0 != BC)FREE_ARRAY_MPF(BC);
  free(P);
  free(PP);
  free(P1);
  free_mpf(&(MPI_MPF));
  free_mpf_op(&(MPI_MPF_SUM));
  MPI_Finalize();
  return info;
  };

  //Function for solution of SLAEs by Gauss method with GMP
  int GausSolver(mpf_t *A, mpf_t *B, int *P, mpf_t *Block, mpf_t
*Col, mpf_t *U12, mpf_t *L21, mpf_t *A1, mpf_t *B1, mpf_t *AC,
mpf_t *BC, int *P1, int *PP, int n, int r, int nx, int ny, int s,
```

```
MPI_Comm comm_cart_x, MPI_Comm comm_cart_y,int *dims, int
*coord,int info, mpf_t ea, mpf_t eb, mpf_t *cond, mpf_t *e, mpf_t
*e1)
  {
  int i, j;
  //make copies of A and B
       for(i = 0; i < ny * nx; i++)
              mpf_set(AC[i],A[i]);
       for(i = 0; i < ny * r; i++)
              mpf_set(BC[i],B[i]);

  //make decomposition of main matrix and find condition number
       info                                                    =
CalculateConditionNumber(A,P,Block,Col,U12,L21,n,nx,ny,s,comm_car
t_x,comm_cart_y,dims,coord,cond, info, PP, P1,B1);
       if(info < 0)
              return info;

  //if successfully, use permutation vector for changing order
of elements of B
       if(coord[0] == 0)
       {
              if(dims[1]==1)
                     SwapB(B,P,r,n);
              else
              {
  GatherFromAllIB(P,P1,PP,s,n,1,comm_cart_y,dims[1],coord[1]);
  GatherFromAllDB(B,B1,PP,s,n,r,comm_cart_y,dims[1],coord[1]);
              if(coord[1] == 0)
                     SwapB(B1,P1,r,n);
                     SendToAllDB(B,B1,PP,
s,n,r,comm_cart_y,dims[1],coord[1]);
              }
       }

  //find solution
       for(i=0;i<r;i++)
              info = CalculateSolution(A, &B[i*ny], Col, Block,
r, n, nx, ny, s, 0, comm_cart_x, comm_cart_y,dims, coord, info);

  //find estimates of errors
              info = SolutionAnalysis(A, BC, B, A1, B1, P, Col,
Block, PP, P1, AC, r, n, nx, ny, s, ea, eb, cond, e, e1,
comm_cart_x, comm_cart_y, dims, coord, info);
```

```
        return info;
    }

    //main function of matrix factorization
    int MatrixFactorization(mpf_t *A, int *P,mpf_t *Block, mpf_t
*Col, mpf_t *U12, mpf_t *L21, int n, int nx, int ny, int s,
MPI_Comm comm_cart_x, MPI_Comm comm_cart_y,int *dims, int
*coord,int info, int*PP)
    {
    int i,j,k;//temp
    int k1,k2,k3;
    int j1=10;
    int kx,ky;//coords of left upper corner of the leading block

        //initialization of P
        for(i=0;i<ny;i++)
                P[i] = (coord[1] + i/s*dims[1])*s + i%s;
    //main cycle
        for(k=0;k<(n-1)/s;k++)
        if(info==0)
        {
                kx=(k/dims[0]+(coord[0]<k%dims[0]))*s;
                ky=(k/dims[1]+(coord[1]<k%dims[1]))*s;
                //modification of the leading block
                info                =                BlockFactoriza-
tion(A,P,Col,n,s,nx,ny,s,k,comm_cart_x,comm_cart_y,dims,coord,inf
o, PP);
                if(info < 0)
                        return info;
                //modification of L11
                if(k%dims[0]==coord[0])
                {
                        if(k%dims[1]==coord[1])
                                BlockL11(A,Block,kx,ky,nx,ny,s);
                                kx+=s;
                }
                //calculation of U12
                if(k%dims[1]==coord[1])
                {
                        if(dims[0]>1)
                        MPF_Bcast(Block,s*s,  k%dims[0],  coord[0],
comm_cart_x);
                        BlockU12(A,Block,Col,kx,ky,nx,ny,s,k);
                        ky+=s;
```

```
                }
                //sending U12 in vertical direction
                if(k%dims[1]==coord[1])
                        for(i=0;i<s;i++)
                        {
                                k3=(i+ky-s)*nx;
                                for(j=kx;j<nx;j++)
                                mpf_set(U12[i+j*s], A[k3+j]);
                        }
                if(dims[1]>1)
                        MPF_Bcast(&U12[kx*s],(nx-kx)*s,  k%dims[1],
coord[1], comm_cart_y);
                //sending L21 in horizontal direction
                if(k%dims[0]==coord[0])
                        for(i=ky;i<ny;i++)
                        {
                                k2=i*s;
                                k3=i*nx+kx-s;
                                for(j=0;j<s;j++)
                                        mpf_set(L21[k2+j], A[k3+j]);
                        }
                if(dims[0]>1)
                        MPF_Bcast(&L21[ky*s],(ny-ky)*s,  k%dims[0],
coord[0], comm_cart_x);
                //modification of the rest matrix (block A22)
                BlockA22(A,U12,L21,kx,ky,nx,ny,s,k);
  }
        k=(n-1)/s;
        k1=n-k*s;
        //modification of the last block (which has rows less
then s)
        info                    =                    BlockFactoriza-
tion(A,P,Col,n,k1,nx,ny,s,k,comm_cart_x,comm_cart_y,dims,coord,in
fo, PP);
        return info;
  }
```

For the program created special functions for MPI-communications with type of MPI_MPF of MPIGMP library. For example, there are: MPF_Bcast, MPF_Reduce, MPF_Allreduce, MPF_Send, MPF_Recv, MPF_Gatherv, MPF_Allgather, MPF_Scatterv. Please see for more details below.

```
  void MPF_Bcast(mpf_t *buf, int count, int source, int rank,
MPI_Comm comm)
  {
```

```
        void *packed = allocbuf_mpf(MPF_PREC, count);
        if(rank == source)
               pack_mpf(buf[0], count, packed);
        MPI_Bcast(packed, count, MPI_MPF, source, comm);
        if(rank != source)
               unpack_mpf(packed, buf[0], count);
        free(packed);
  }

  void MPF_Reduce(mpf_t *In, mpf_t *Out, MPI_Op op, MPI_Comm
comm) {
        void *packed_in, *packed_out;
        packed_in = allocbuf_mpf(MPF_PREC, 1);
        packed_out = allocbuf_mpf(MPF_PREC, 1);
        pack_mpf(In[0], 1, packed_in);
        pack_mpf(In[0], 1, packed_out);
        MPI_Reduce(packed_in,packed_out,1,MPI_MPF, op, 0, comm);
        unpack_mpf(packed_out, Out[0], 1);
        free(packed_out);
        free(packed_in);
  }

  void MPF_Allreduce(mpf_t *In, mpf_t *Out, MPI_Op op, MPI_Comm
comm)
  {
        void *packed_in, *packed_out;
        packed_in = allocbuf_mpf(MPF_PREC, 1);
        packed_out = allocbuf_mpf(MPF_PREC, 1);
        pack_mpf(In[0], 1, packed_in);
        pack_mpf(In[0], 1, packed_out);
        MPI_Allreduce(packed_in,packed_out,1, MPI_MPF, op, comm);
        unpack_mpf(packed_out, Out[0], 1);
        free(packed_out);
        free(packed_in);
  }

  void MPF_Send(mpf_t*buf, int count, int dest_proc, MPI_Comm
comm) {
        void *packed_buf;
        packed_buf = allocbuf_mpf(MPF_PREC, count);
        pack_mpf(buf[0], count, packed_buf);
        MPI_Send(packed_buf, count, MPI_MPF, 0, dest_proc, comm);
        free(packed_buf);
  }

  void MPF_Recv(mpf_t*buf, int count, int dest_proc, MPI_Comm
comm) {
        MPI_Status status;
        void *packed_buf;
        packed_buf = allocbuf_mpf(MPF_PREC, count);
        MPI_Recv(packed_buf, count, MPI_MPF, 0, dest_proc, comm,
&status);
        unpack_mpf(packed_buf, buf[0], count);
        free(packed_buf);
  }
```

```
  void MPF_Gatherv(mpf_t*part,int scount, mpf_t*all, int*rcount,
int*displs, int dest_proc, int proc_count, int rank, MPI_Comm
comm)
  {
       void *packed_part, *packed_all;
       int temp = rcount[0], i;
       for(i = 1; i < proc_count; i++)
              temp += rcount[i];
       packed_part = allocbuf_mpf(MPF_PREC, scount);
       pack_mpf(part[0], scount, packed_part);
       packed_all = allocbuf_mpf(MPF_PREC, temp);
       if(rank == dest_proc) pack_mpf(all[0], temp, packed_all);
       for(i = proc_count - 1; i >= 0; i--)
              displs[i] -= displs[0];
       MPI_Gatherv(packed_part,scount,    MPI_MPF,    packed_all,
rcount, displs, MPI_MPF, dest_proc, comm);
       if(rank == dest_proc)
              unpack_mpf(packed_all, all[0], temp);
       free(packed_part);
       free(packed_all);
  }

  void     MPF_Allgather(mpf_t*part,int     scount,mpf_t*all,int
rcount,int proc_count, MPI_Comm comm)
  {
       void *packed_part, *packed_all;
       packed_part = allocbuf_mpf(MPF_PREC, scount);
       packed_all = allocbuf_mpf(MPF_PREC, rcount * proc_count);
       pack_mpf(part[0], scount, packed_part);
       pack_mpf(part[0], 1, packed_all);
       MPI_Allgather(packed_part,scount,   MPI_MPF,   packed_all,
rcount, MPI_MPF, comm);
       unpack_mpf(packed_all, all[0], rcount * proc_count);
       free(packed_part);
       free(packed_all);
  }

  void     MPF_Scatterv(mpf_t*all,int     *scount,     int*displs,
mpf_t*part, int rcount, int dest_proc, int proc_count, int rank,
MPI_Comm comm)
  {
       void *packed_part, *packed_all;
       int temp = scount[0], i;
       for(i = 1; i < proc_count; i++)
              temp += scount[i];
       packed_part = allocbuf_mpf(MPF_PREC, rcount);
       packed_all = allocbuf_mpf(MPF_PREC, temp);
       if(rank == dest_proc)
              pack_mpf(all[0], temp, packed_all);
       pack_mpf(part[0], rcount, packed_part);
       for(i = proc_count - 1; i >= 0; i--)
              displs[i] -= displs[0];
       MPI_Scatterv(packed_part,scount,      displs,      MPI_MPF,
packed_all, rcount, MPI_MPF, dest_proc, comm);
       if(rcount > 0)
              unpack_mpf(packed_part, part[0], rcount);
```

```
        free(packed_part);
        free(packed_all);
}
```

In the subprogramme `GausSolver` uses the following computational subprograms:

- `MatrixFactorization` calculates LU-decomposition of dense nonsingular matrix with partial selection of main element by the column.
- `CalculateSolution` finds solution of system of linear equations $AX=B$ with a dense nonsingular matrix, using LU-decomposition, which is obtained by `plgefa`.
- `CalculateConditionNumber` calculates estimates of the condition number of matrice of a system.

In addition to this, functions such as BLAS, implementing basic operations over vectors with the the unfolding of cycles: `CycleA11`, `CycleA11F`, `CycleA12`, `CycleA22`, function for calculating estimates of solutions `SolutionAnalysis`, `GatherFromAllI`, `GatherFromAllD`, `SendToAllD`, `Swap` are used.

The program checks the correspondence of algorithm of solution to properties of current problem.

If `info = -101`, then the matrix of the machine-singular and the classical solutions do not exist;

if `info = -105`, then the solution is obtained, but its accuracy is not guaranteed because within the accuracy of definition of elements of the matrix may be singular, needs to have initial data system with greater precision;;

if `info = -2`, then to solve the problem of low memory;

if `info = - 3`, then error parameter input; you must verify that the input data set for the program.

By including procedures for selecting the lead element of one of the components of solution have the deviation from the exact value:

$$x_1 = 1, x_2 = -1, x_i = 0, \qquad i = 3, 4, \ldots, n.$$

In addition to finding a solution, the problem of SLAEs were analyzed for the problem, because of the data set approximately (accurate setting of the coefficients and right parts is 0). In the case when the matrix order n = 1000 evaluation of condition number equals 4.00000+008, hereditary error is 1.33205e-009, and is calculated - 1.85886e-010.

The protocols of executed programs with `double` and increased digit are presented in Listings 8.3 and 8.4.

Listing 8.3 The protocol of the program DenseSolver with double precision

```
P R O B L E M:
         solving of the linear algebraic system
         with a general matrix
```

```
       D a t a :
           - number of matrix's rows                  = 1000
           - number of matrix's columns               = 1000
           - number of the right-hand side
                    of the systems                    = 1
           - maximum relative error
                   of the matrix elements         = 0.00000e+00
           - maximum relative error
             of elements of the right-hand sides   = 0.00000e+00

      P r o c e s s   o f   i n v e s t i g a t i n g   a n d
s o l v i n g

       M E T H O D:
           Gauss elimination with partial pivoting

   R E S U L T S :

   Solution (first and last 5 components) for right part number
1:

  9.9999999999997757e-01
  -1.0000000000000013e+00
  1.1357581541903890e-13
  -1.1357581541902924e-13
  6.3445847911892908e-25

  -4.4322184810851527e-16
  -2.2605733685481399e-27
  2.2161092405651848e-16
  -1.1080546202825937e-16
  0.0000000000000000e+00

   SOLUTION IS OBTAINED IN FILE result.out

       E s t i m a t e s :
         - inherited error in the solution : 8.89067e-10
         - computational error in the solution : 2.68341e-11

       P r o p e r t i e s  :
        - estimate of condition number of the matrix 2.00200e+06

   #result=0
```

```
 Time: 0.299037
 Proc number: 4
```

Listing 8.4 The protocol of the program GausSolver with mpf_t precision

```
P R O B L E M:
         solving of the linear algebraic system
         with a general matrix

      D a t a :
         - number of matrix's rows                  = 1000
         - number of matrix's columns               = 1000
         - number of the right-hand side
                  of the systems                    = 1
         - maximum relative error
           of the matrix elements = 2.71050543121376108502e-20
         - maximum relative error
of elements of the right-hand sides = 2.71050543121376108502e-
20
         - MPF precision                            = 64

P r o c e s s  o f  i n v e s t i g a t i n g  a n d  s o
l v i n g

M E T H O D:
         Gauss elimination with partial pivoting

 R E S U L T S :

 Solution (first and last 5 components) for right part number
1:

1.00000000000000000119
-1.00000000000000000033
-1.206715480780127464l8e-33
5.42101086242539221189e-20
-5.42101086242657575603e-20

-4.23516473627257127811e-22
2.23346222543638576294e-35
2.11758236813606229375e-22
-1.05879118406803114699e-22
0

 SOLUTION IS OBTAINED IN FILE result.out
```

```
  E s t i m a t e s :
  - inherited error in the solution: 1.0140146018031602677 7e-13
  -      computational      error      in      the      solution:
1.744313534603118772 00e-15

  P r o p e r t i e s  :
  - estimate of condition number of the matrix  1.891028172e6
   #result=0

   Time: 1.318378
   Proc number: 8
```

Testing program performed on Inparcom-256 [10, 11], with precision `double` (53), 64 and 128 bit. Results of testing program is presented are Tables 8.1, 8.2.

Table 8.1 Time solution (sec) of problem (8.4) with GMP and MPI for N=1000

Prec	1	2	4	8
double	0,86	0,52	0,33	0,27
64	4,76	2,68	1,73	1,19
128	4,9	2,79	1,79	1,27

Table 8.2 Time solution (sec) of problem (8.4) with GMP and MPI for N=2000

Prec	1	2	4	8
double	6,35	3,47	1,98	1,23
64	33,67	17,38	10,16	6,62
128	34.96	17.99	10,82	6,81

Figure 8.2 presents dependence of time of the solution of linear algebraic equations (8.4) for different order of matrices from required precision `MPF_PREC` (double (53), 64 and 128 bit) using 8 processes.

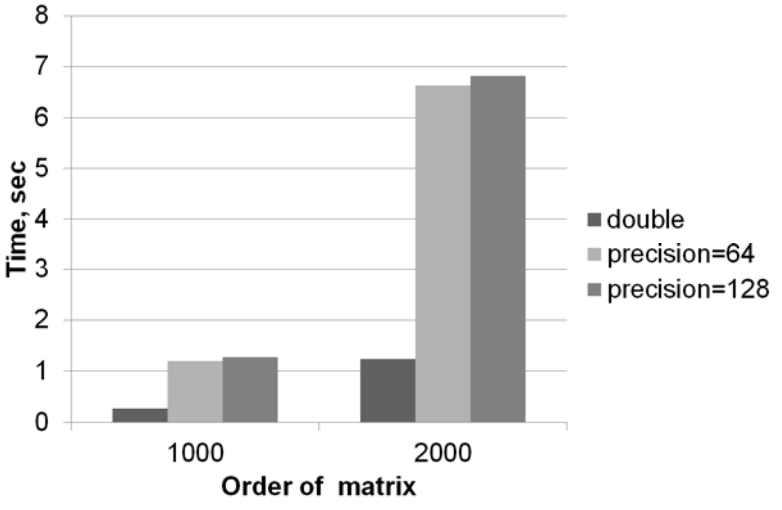

Fig. 8.2. Dependence of time of solution of problem (8.4) with different order of matrices from required accuracy `MPF_PREC` using 8 processes

8. 2 Singular Value Decomposition

The singular value decomposition (SVD) of a matrix $A \in R^{mxn}$ is a matrix decomposition of great theoretical and practical importance for the treatment of least squares problems. It provides a diagonal form of A under an orthogonal equivalence transformation. The history of this matrix decomposition goes back more than a century. See a very interesting survey of the early history of the SVD by Steward [19]. However, only recently has the SVD been used as much as it should be. Now it is a main tool in numerous application areas such as signal and image processing, control theory, pattern recognition, time-series analysis, etc.

Consider a linear algebraic equation (1.3) with a rectangular matrix of arbitrary rank in size $m \times n$. In general, this problem is incorrect and the system is in many cases - inconsistent. For these linear systems can obtain a solution by the method of least squares (generalized solution), which for the joint and correct the problem is identical with the classical solution.

For the obtaining of generalized solution with minimum norm the singular value decomposition of the matrix can be employed:

$$A = U \Sigma V^T,$$

where U, V are matrices of left- and right-hand singular vectors, respectively, $(U^T U = V^T V = V V^T = I_n)$, $\Sigma = \mathrm{diag}[\sigma_1, \sigma_2, \ldots, \sigma_{\min\{m,n\}}]$ is diagonal matrix of singular values, and I_n is an identity matrix of the n-th order. At that, if the rank of the matrix A, $r < \min\{m,n\}$ then $\sigma_{r+1} = \ldots = \sigma_{\min\{m,n\}} = 0$. Denote $\sigma_i^{\#} = \{1/\sigma_i, \sigma_i > 0\ ;0, \sigma_i = 0\}$, $\Sigma^{\#} = \mathrm{diag}[\sigma_1^{\#}, \sigma_2^{\#}, \ldots, \sigma_{\min\{m,n\}}^{\#}]$ a pseudoinverse matrix to

Σ. Then a generalized solution of the original problem possessing the minimum norm can be evaluated by formula

$$\hat{x} = V\Sigma^{\#}c\,, \quad \text{where } c = U^T b. \tag{8.5}$$

A stable algorithm for SVD was first outlined by Golub and Kahan [8]. They suggested that the matrix A first should be reduced to bidiagonal form by Householder transformation of a Lanczos process. The singular values and vectors can then be computed as eigenvalues and eigenvectors of a spend tridiagonal matrix, by a method based on Sturn sequences. Later, Golub [7] gave an adaptation of the QR algorithm for computing the SVD of the bidiagonal matrix, and described a simplified interpretation of this process due to Wilkinson. The final form of the QR algorithm for computing SVD was given by Golub and Reinsch [9].

In the evaluation of generalized solution (8.5) to the SLAEs with rectangular arbitrary rank matrix of dimension $m \times n$ ($m \geq n$) by means of singular value decomposition of the following five subproblems can be separated:

1) reduction of the original matrix to upper two-diagonal form $A^{(n-1)} \equiv \begin{pmatrix} J^{(0)} \\ \hline 0 \end{pmatrix}$, where $J^{(0)}$ is a square upper two-diagonal matrix of order n in which only $j^{(0)}_{k,k}$, $j^{(0)}_{k,k+1}$ are non-zero elements;

1) singular-value decomposition of the upper two-diagonal matrix $J^{(0)}$, i. e. the reduction of $J^{(0)}$ to two-diagonal matrix Σ;

2) forming of matrix of the right-hand singular vectors V;

3) forming of matrix (vector) c from (8.5);

4) evaluation of the generalized solution (8.5).

If $m < n$, then singular value decomposition of the transposed matrix $A^T = V \Sigma U^T$ is carried out.

In the Golub-Reinsch method, the first step in computing SVD is to reduce A to upper bidiagonal form. This reduction can be performed using a finite sequence of Householder transformations from left and right as follows.

For the reduction of rectangular matrix $A^{(0)}$ of dimension $m_1 \times n_1$ ($m_1 = \max\{m, n\}$, $n_1 = \min\{m, n\}$, $A^{(0)} \equiv A$ if $m \geq n$ if $A^{(0)} \equiv A^T$ if $m < n$) to upper two-diagonal form by Householder's transformation (see, for example, [1, 3, 15, 16, 20, 21]) in so doing $n_1 - 1$ two-sided elementary reflection transformations are required:

$$A^{(i)} = P^{(i)} A^{(i-1)} Q^{(i)} \quad (i = 1, 2, \ldots, n_1 - 1), \tag{8.6}$$

where orthogonal matrices are equal to $P^{(i)} = I - s_i u_i u_i^T$, $Q^{(i)} = I - t_i v_i v_i^T$ while vectors u_i, v_i and multipliers s_i, t_i are determined so that for every $i = 1, 2, \ldots, n_1 - 1$ the conditions

$$s_i u_i^T u_i = t_i v_i^T v_i = 2, \tag{8.7}$$

$$a^{(i)}_{k,j} = a^{(i)}_{j,k} = 0 \quad (k = i+1, \ldots, m_1; \quad j = i+2, \ldots, n_1). \tag{8.8}$$

should be hold.

After the first step we have:

$$A^{(1)} = P^{(0)}A^{(0)}Q^{(0)} = \begin{pmatrix} p_1 & e_2 & 0 & \dots & 0 \\ 0 & \overline{a}_{22} & \overline{a}_{23} & \dots & \overline{a}_{2n} \\ 0 & \overline{a}_{32} & \overline{a}_{33} & \dots & \overline{a}_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \overline{a}_{m2} & \overline{a}_{m2} & \dots & \overline{a}_{mn} \end{pmatrix}$$

Here $P^{(0)}$ is chosen so that $P^{(0)}A^{(0)}$ has zero in the first column under the main diagonal. $Q^{(0)}$ is then chosen to zero the last *n-2* elements in the first row. The first column is not obviously touched by $Q^{(0)}$, which only affects the last *n-1* columns. All the next steps are similar.

To decompose upper two-diagonal matrix $J^{(0)}$ by singular values one can employ *QR*-algorithm with implicit shift. This algorithm consists in (see, for example, [21]) construction of sequence of upper two-diagonal matrices $J^{(0)}, J^{(1)}, \dots, J^{(N)}, \dots$, that converges to the diagonal matrix Σ. Transition from $J^{(k-1)}$ to $J^{(k)}$ is performed by means of series of two-sided planar Given's rotations:

$$J^{(k)} = S^{(k)T}J^{(k-1)}T^{(k)}, \tag{8.9}$$

where $S^{(k)} = S_2^{(k)}S_3^{(k)}\cdots S_n^{(k)}$, $T^{(k)} = T_2^{(k)}T_3^{(k)}\cdots T_n^{(k)}$, while $S_j^{(k)}, T_j^{(k)}$ are elementary singular value decomposition of two-diagonal matrix

$$J^{(0)} = G\tilde{\Sigma}H^T, \tag{8.10}$$

here $\tilde{\Sigma} \equiv J^{(N)}$ and

$$G = S^{(1)}S^{(2)\dots}S^{(N)}, \quad H = T^{(1)}T^{(2)\dots}T^{(N)}. \tag{8.11}$$

The forming of matrix of right-(or left)-hand singular vectors is carried out by means of accumulating both of the elementary reflection transformations

$$Q = Q^{(1)}Q^{(2)}\cdots Q^{(n-1)} \text{ or } P = P^{(1)}P^{(2)}\cdots P^{(n-1)} \tag{8.12}$$

and rotation transformations (8.11). Then columns of the matrix $V = QH$ are approximate right-hand singular vectors of the original matrix, while columns of the matrix $U = PG$ are left-hand singular vectors.

The matrix (vector) c from (8.5) is formed by analogy –at first, according to formulas

$$b^{(i)} = P^{(i)}b^{(i-1)} \ (m \ge n) \text{ or } b^{(i)} = Q^{(i)}b^{(i-1)} \ (m < n), \tag{8.13}$$

$i = 1, 2, \dots, n_1-1$ $b^{(n-1)}$ or $b^{(m-1)}$ is formed, and then

$$c = G^T b^{(n-1)} \text{ or } c = H^T b^{(m-1)}, \tag{8.14}$$

is evaluated.

While evaluating generalized solution (8.5) one should properly determine the matrix rank under conditions of approximately given initial data of the problem (1.3), (1.5). To do this one may make use of the definition of the efficient rank (1.8). A singular value σ_k for which $\sigma_k \le \varepsilon$, where ε is maximum value of off-diagonal elements of matrix $J^{(N)}$. It is considered to be zero singular value within machine precision.

If (1.3), (1.5) are homogenous system, i. e. $b \equiv 0$ then there non-trivial solutions are right-hand singular vectors corresponding to zero singular values (within machine precision).

8.2.1 Parallel Algorithm of SVD

Two-sided transformations (8.6) under conditions (8.7), (8.8) can be written as follows [17]:

$$A^{(i)} = A^{(i-1)} + u_i y_i^T + z_i v_i^T \quad (i = 1, 2, \ldots, n-1) \tag{8.15}$$

Where

$$u_i = \left(0, \ldots, 0, a_{i,i}^{(i-1)} - d_i, a_{i,i+1}^{(i-1)}, \ldots, a_{i,m}^{(i-1)}\right)^T,$$

$$v_i = g_i - f_{i+1} e_{i+1}, \quad y_i = w_i + c_i v_i, \quad z_i = t_i x_i - c_i u_i,$$

$$g_i^T = \frac{1}{d_i s_i} w_i^T + e_i^T A^{(i-1)}, \quad w_i^T = s_i u_i^T A^{(i-1)}, \quad x_i = A^{(i-1)} v_i,$$

$$d_i = -\text{sign}(a_{i,i}^{(i-1)}) \varphi_i, \quad f_{i+1} = -\text{sign}(g_{i,i+1}) \psi_i,$$

$$c_i = 0{,}5 t_i w_i^T v_i, \quad s_i = \left(d_i a_{i,i}^{(i-1)} - \varphi_i^2\right)^{-1}, \quad t_i = \left(f_{i+1} g_{i,i+1} - \psi_i^2\right)^{-1},$$

$$\varphi_i^2 = \sum_{j=i}^{m} \left(a_{i,j}^{(i-1)}\right)^2, \; \psi_i^2 = \sum_{j=i+1}^{n} \left(g_{i,j}\right)^2, \tag{8.16}$$

and e_i is unity ort.

Thus, at each step (for each i) the sub-matrix of dimension $(m-i+1)\times(n-i+1)$, that is located in the right-hand lower corner is modified, the fulfillment of conditions (8.7), (8.8) is ensured and elements $d_i \equiv a_{i,i}^{(i)}$ and $f_{i+1} \equiv a_{i,i+1}^{(i)}$ are diagonal and off-diagonal elements of two-diagonal matrix $J^{(0)}$, respectively.

Simultaneously with evaluation of matrix $J^{(0)}$ by Formula (8.13) the rectangular (vector) $b^{(n-1)}$ (or $b^{(m-1)}$, if $m < n$) is formed. Formula (8.13) for $i = 1, 2, \ldots, \min(m, n)-1$ can be written in form:

$$b^{(i)} = b^{(i-1)} + u_i h_i^T, \quad h_i^T = s_i u_i^T b^{(i-1)} \quad (m \geq n) \tag{8.17}$$

or

$$b^{(i)} = b^{(i-1)} + v_i h_i^T, \quad h_i^T = t_i v_i^T b^{(i-1)} \quad (m < n). \tag{8.18}$$

Hence, for each i the lower $(m-i+1)\times q$ submatrix is modified.

Matrix of elementary reflection transformations Q of (8.12) can be formed as follows:

$$Q_{(i)} = Q^{(k)} Q_{(i-1)} \equiv Q_{(i-1)} + v_k x_k^T \quad (i = 1, 2, \ldots, n-1), \tag{8.19}$$

where $Q_{(0)} \equiv I$, $Q \equiv Q_{(n-1)}$,

$$x_k = t_k Q_{(i-1)}^T v_k, \quad t_k = \left(f_{k+1} v_{k,k+1}\right)^{-1} \tag{8.20}$$

The reflection vector v_k is determined in (8.16), $f_{k+1} \equiv j_{k,k+1}^{(0)}$ is over-diagonal element of two-diagonal matrix $J^{(0)}$ ($k = n-i$).

Matrix of elementary reflection transformations P of (8.12) is evaluated for $m < n$ that is formed by analogy.

Effective paralleling of algorithm for singular value decomposition is based both on row-cyclic scheme of distributing matrix elements between processes and natural parallelism inherent in matrix-vector operations.

For the arrangement of parallel computations elements of matrices $A^{(i)}$ and $b^{(i)}$, including original matrices $A^{(0)} \equiv A$ и $b^{(0)} \equiv b$, are distributed between processes by row-cyclic scheme. There is no close relation between logical number of the

process and ordinal number of a row: process with logical number r ($r = 0, 1,\ldots, p-1$) receives elements of matrix rows numbered $k+1, p+k+1, 2p+k+1, \ldots$, where p is the number of processes which is being used, while k is equal to the remainder of the division $r+l$ by p for the arbitrary $0 \le l \le p-1$.

Results of reduction are presented by diagonal $a_{j,j}^{(n-1)}$ ($j = 1, 2,\ldots, n$) and over-diagonal elements $a_{j,j+1}^{(n-1)}$ ($j = 1, 2,\ldots, n-1$) of upper two-diagonal matrix $J^{(0)}$; they are evaluated by each process. Besides, non-zero elements of vector v_i ($m \ge n$) or u_i ($m < n$) are stored on the place of elements of the lower triangle of the original matrix according to their distribution between processes. If further computations require parameters of both orthogonal transformations, i.e. vectors u_i and v_i, then non-zero elements of these vectors will be stored in the place of elements of lower and upper triangles of the original matrix, respectively, according to their distribution between processes.

The performance of intermediate computation by Formulas (8.15), (8.16), and (8.17) or (8.18) and data exchanges between processes require three arrays of dimensions $\min(m, n)+q$ each.

Array of over-diagonal elements of the matrix $J^{(0)}$ and reflection vectors v_k, formed during the reduction of rectangular matrix to two-diagonal form which are used as initial data for the forming of matrix Q (8.19) and (8.20). Non-zero elements of these vectors are stored in the place of elements of lower triangle of the original matrix according to their row-cyclic distribution between processes. In doing so of vector v_k are located on the place of elements of the k-th column of the original matrix. The result of computations – square matrix Q turned out to be distributed by row cyclic scheme.

The performance of intermediate computations by Formulas (8.19), (8.20) and data exchanges between processes require three arrays for the storing of vectors: one array for v_k and two arrays for x_k.

Householder's row-cyclic parallel algorithm for the reduction of rectangular $m \times n$ matrix $A^{(0)}$ to the upper two-diagonal form $A^{(n-1)}$ for $i = 1, 2,\ldots, n-1$ consists of the following sequence of operations:

1) for the evaluation of value φ_i and vector w_i of (8.16) as well as vector h_i of (8.17) provided $m \ge n$ each process computes arrays of the last $n-i+1$ and q elements of vectors containing partial sums of products of the transposed i-th vector-column of the matrix $A^{(i-1)}$ by rectangular matrices $A^{(i-1)}$ and $b^{(i-1)}$, respectively;

2) multi-gathering by leading process (in which i-th rows of rectangular matrices are located) of vectors containing products of the transposed i-th vector-column if the matrix $A^{(i-1)}$ by rectangular matrices $A^{(i-1)}$ and $b^{(i-1)}$, these vectors are being involved in vectors of partial sums computed by each process (see introduction for the definition of multi-gathering and broadcasting operations);

3) by means of broadcasting operation the following data are broadcasted to all processes: one-dimensional array of elements $a_{i,i}^{(i-1)}, a_{i,i+1}^{(i-1)}, \ldots, a_{i,n}^{(i-1)}$ of the *i-th* matrix row, the last $n-i+1$ and q elements of vectors containing products of the

transposed i-th vector-column of the matrix $A^{(i-1)}$ by rectangular matrices $A^{(i-1)}$ and $b^{(i-1)}$;

4) all processes simultaneously evaluate values φ_i and $d_i \equiv a_{i,i}^{(i)}$ of (8.16) form the left-hand reflection vector u_i, evaluate the value s_i of (8.16) then form both the last $n-i$ elements of vectors w_i and g_i according to (8.16) and vector h_i of (8.17) provided by $m \geq n$;

5) all processes simultaneously evaluate values ψ_i and $f_{i+1} \equiv a_{i,i+1}^{(i)}$ of (8.16), form right-hand reflection vector v_i, evaluate values t_i and c_i and then form last $n-i$ elements of vector y_i;

6) each process according to both row-cyclic scheme of distribution of rectangular matrices and Formula (8.16) evaluates the last $m-i$ elements of vectors x_i and z_i as well as vector h_i of (8.18) provided by $m < n$;

7) each process according to the row-cyclic scheme for distribution of rectangular matrices performs modification (8.15) of $(m-i)\times(n-i)$ sub-matrix of the matrix $A^{(i-1)}$, located in the right-hand lower corner as well as modification (8.17) or (8.18) of lower $(n-i+1)\times q$ submatrix of the matrix $b^{(i-1)}$.

After performing all of these operations for all $i = 1, 2, \ldots, n-1$ element $a_{n,n}^{(n-1)}$ should be broadcasted to all processes. Thus, as a result of performing of all operations by algorithm under consideration each process will form two n-dimensional arrays of diagonal and over-diagonal elements of two-diagonal matrix $J^{(0)}$.

Row-cyclic algorithm for accumulation of elementary reflection transformations for $i = 1, 2, \ldots, n-1$ and $k = n-i$ consist of the following sequence of operations:

5) each process evaluates array of the last i elements of vector containing partial sums of the product $Q_{(i-1)}^T v_k$ of the square matrix (to be precise its diagonal block of order i) is predetermined by a vector according to row-cyclic scheme;

6) multi-gathering by leading process (in which the $(k+1)$th row of matrix is being formed and located) of the last i elements of vector $Q_{(i-1)}^T v_k$ from vectors of partial sums is evaluated by each process ;

7) evaluation by leading process of the value t_k and last i elements of vector x_k of (8.20);

8) by means of broadcasting operation of one-dimensional array containing last i elements of the vector x_k which is broadcasted to all processes;

9) each process according to row-cyclic scheme for the distribution of elements of the matrix $Q_{(i-1)}$ performs modification (8.19) over its diagonal block of order $i+1$, located in the low right-hand corner.

Efficiency of algorithms. The total number of arithmetic operations is required for the reduction of rectangular $m\times n$ matrix to two-diagonal form by Householder's method and formation of rectangular matrix (vector) $b^{(n-1)}$ (8.13) is

estimated by value $O_1 = \frac{12m-4n}{3}(n+q)n + O(mn) + O(n^2)$ with $m \geq n$, while the number of arithmetic operations performed by each of p processes – by value

$$O_p = \frac{n}{3p}\left((12m-4n)(n+q) + 3m + (15p - 4{,}5 - 2q)n\right) + O(n).$$

At each step of i–cycle the following operations are performed: one operation of multi-gathering of $(n–i+q)$-dimensional vector and one operation of array containing $2n–2i+q$ elements. Hence, the total number of exchanges is estimated by value $O_c \approx 2n\log_2 p$, and at that amount of data by which processes exchange is approximately equal to $O_o \approx n(1{,}5n+2q)\log_2 p$ of double words.

Then coefficients of acceleration and efficiency of the row-cyclic partial algorithm for the reduction of dense symmetric matrix to three-diagonal symmetric matrix are estimated as follows:

$$\tau_1 = \tau_o + \frac{4}{3n+4q}\tau_c$$

$$S_p \approx p\left(1 + \frac{3{,}75(p-1)n}{(3m-n)(n+q)} + \frac{1{,}125\,p\log_2 p}{(3m-n)}\tau_1\right)^{-1}, \qquad (8.21)$$

$$E_p = \frac{S_p}{p};$$

if $\frac{3{,}75(p-1)n}{(3m-n)(n+q)} + \frac{1{,}125\,p\log_2 p}{(3m-n)}\tau_1 << 1$, then

$$E_p \approx 1 - \frac{3{,}75(p-1)n}{(3m-n)(n+q)} - \frac{1{,}125\,p\log_2 p}{(3m-n)}\tau_1. \qquad (8.22)$$

Similar estimate is also derived for $m < n$.

The total number of arithmetic operations required for the formation of matrix of elementary reflection transformations Q (8.12) is estimated by value

$$O_1 \approx 4n^3/3,$$

while the arithmetic operations are performed by each of p processes – by value

$$O_p \approx \frac{4n^3 + 1{,}5n^2 p}{3p}.$$

At each step of i – cycle the following operations are performed: one operation of broadcasting of one-dimensional array containing i elements and one operation of multi-gathering of i-dimensional vector. Hence, the total number of exchanges is estimated by value $O_c \approx 2n\log_2 p$ and at that the amount of data by which processes exchange consists approximately $O_o \approx n^2\log_2 p$ double words.

Then coefficients of acceleration and efficiency of row-cyclic parallel algorithm for forming matrix of elementary reflection Q (8.12) are estimated as follows:

$$S_p \approx p\left(1 + \frac{0{,}375p}{n} + \frac{0{,}75\,p\log_2 p}{n}\tau_2\right)^{-1}, \quad E_p = \frac{S_p}{p}; \qquad (8.23)$$

where $\tau_2 = \tau_o + \frac{2}{n}\tau_c$.

If $\frac{0{,}375p}{n} + \frac{0{,}75p\log_2 p}{n}\tau_2 << 1$, then

$$E_p \approx 1 - \frac{0{,}375p}{n} - \frac{0{,}75p\log_2 p}{n}\tau_2. \quad (8.24)$$

Householder parallel algorithm is realized in function `dsht_prc` in Listing 8.5. Here original matrix *A* is reduced to bidiagonal form by described above transformations. Next section consists of this function for data type `mpf_t`.

Listing 8.5 Function reduction of the original matrix (the size of mxn) to bidiagonal form by Householder transformations

```
  int dsht_prc (int m, int n, int q, int ri, double** A, int wU,
double** U, int wV, double** V, double tol,       double*    D0,
double* buf, MPI_Comm comm)
  {
  int i,j,k,l,r,kp1,kp,mp,np,npq,qU,qV;
  int ir,cr,p,cpr,lpr,err;
  double  dk,ek,fk, c,s,t,z,  absp, maxp;
  double *d, *e, *f, *g, *g0, *gi, *x;
  double ts;

  MPI_Comm_rank( comm, &cpr );
  MPI_Comm_size( comm, &p );

  ir = p-ri;   cr = ir+cpr;
  mp = (m+ir)/p;   np = (n+ir)/p;
  npq = n+q;   qU = q*wV;   qV = q*wU;
  d = D0;   e = buf;   g0 = buf+2;   f = g0+n+qU;
  x = buf+2*(n+qU+1);

  ts = MPI_Wtime();
  for(  e[0]=maxp=0.0, k=0;  k<n;  k++, f--  )
  {
       kp1 = k+1;
       kp = (kp1+ir)/p;
       lpr = (kp1+cr)%p;
  //  Calculation of the parameters of the left transformation
  err = ScalProd_p ( 0, qU, U, kp, mp, A, n, k, f, lpr, cpr, p,
comm );
       if( err )                    return( err-16 );
       g = (cpr==lpr?A[kp-1]:g0);
       if( f[k]<tol )
```

```
        {
                dk = d[k] = 0.0;
                for(  j=kp1, r=0;  j<n;  r++, j++  )
                        x[r] = 0.0;
        }
        else
        {
                d[k] = -msign(sqrt(f[k]),g[k]);
                dk = d[k];    g[k] -= dk;
                s = 1.0/(g[k]*dk);
                for(  j=kp1, r=0;  j<n;  r++, j++  )
                {
                        x[r] = (f[j]-dk*g[j])*s;
                        g[j] = f[j]/dk;
                }
                for(  l=0;  l<qU;  r++, j++, l++  )
                {
                        x[r] = (f[j]-dk*g0[j])*s;
                        if( lpr==cpr )
                                U[kp-1][l] += x[r]*g[k];
                }
        }
  //  Calculation of the parameters of the right transformation
        fk = ScaPro5( n-kp1, g+kp1, g+kp1 );
        if( fk<tol )
        {
                ek = e[kp1] = 0.0;
                c = t = 0.0;
        }
        else
        {
                e[kp1] = -msign(sqrt(fk),g[kp1]);
                ek = e[kp1];    g[kp1] -= ek;
                t = 1.0/(g[kp1]*ek);
                c = 0.5*t*ScaPro5( n-kp1, x, g+kp1);
        }
  //  Bilateral transformation
        for(  j=kp1, r=0;  j<n;  r++, j++  )
                x[r] += c*g[j];
                if( qV )
                {
                err = ScalProd_p ( 0, qV, V, kp, np, &g, 0, 0, f,
lpr, cpr, p, comm );
                if( err )             return( err-24 );
```

```
                for(  l=0;  l<qV;  r++, l++  )
                        x[r] = f[l]*t;
                        gi = g+ri-1;
                }
                for(  i=kp;  i<mp;  i++  )
                {
                s = A[i][k];
                z = t*ScaPro5( n-kp1, A[i]+kp1, g+kp1 )+c*s;
                for(  j=kp1, r=0;  j<n;  r++, j++  )
                        A[i][j] += s*x[r]+z*g[j];
                        for(  l=0;  l<qU;  r++, l++  )
                                U[i][l] += s*x[r];
                                if( qV )   s = gi[p*i];
                                for(l=0;l<qV;r++, l++  )
                                        V[i][l] += s*x[r];
                }
  //  Saving parameters of the right transformations
                if( wV )
                {
                        gi = g+ri-1;
                        for(i=kp, r=i*p;i<np;i++,r+=p)
                        V[i][k] = gi[r];
                }
                if((absp=mabs(d[k])+mabs(e[k]))>maxp)
                        maxp = absp;
        }
  //  Formation the matrix of the right transformations
        ts = MPI_Wtime();
        if( wV )
        for(e[n]=0.0, f=buf+n+1, k=n;  k>0;  f++  )
        {
                kp1 = k;   kp = (kp1+ir)/p;
                lpr = (kp1+1+cr)%p;   k--;
                if( e[kp1]!=0.0 )
                {
                        err = ScalProd_p ( 1, 0, U, kp, np, V, n,
k, f, lpr, cpr, p, comm );
                        if( err )    return( err-32 );
                        s = 1.0/(f[k]*e[kp1]);
                        for(  i=kp;  i<np;  i++  )
                        {
                                t = V[i][k]*s;
                                for(j=kp1;  j<n;  j++  )
                                        V[i][j] += t*f[j];
```

```
                        }
                }
                for(  i=kp;  i<np;  i++  )
                V[i][k] = 0.0;
                if( k%p+1==ri )
                for(V[kp-1][k]=1.0; kp1<n; kp1++)
                        V[kp-1][kp1] = 0.0;
        }
  //  Formation the matrix the left transformations
        ts = MPI_Wtime();
        if( wU )
                for(  f=buf+n+1, k=n;  k>0;  f++  )
                {
                        kp1 = k;   k--;   kp = (k+ir)/p;
                        lpr = (kp1+cr)%p;
                        if( cpr==lpr )
                                for(  j=kp1;  j<n;  j++  )
                                        U[kp][j] = 0.0;
                                        if( d[k]!=0.0 )
                                        {
                                        err = ScalProd_p ( 1, 0, V,
kp, mp, U, n, k, f, lpr, cpr, p, comm );
                                        if( err )
                                        return( err-48 );
                                        s = 1.0/(f[k]*e[kp1]);
                                        for(i=kp;  i<mp;  i++  )
                                        {
                                        t = U[i][k]*s;
                                        U[i][k] = 0.0;
                                        for(  j=k;  j<n;  j++  )
                                                U[i][j] += t*f[j];
                                        }
                                        }
                                        else for(i=kp;i<mp;i++)
                                                U[i][k] = 0.0;
                                        if( cpr==lpr )
                                                U[kp][k] += 1.0;
                }
                                        *buf = maxp;
        return( 0 );
  }
```

Parallel QR-algorithm for upper two-diagonal real matrices. For reasons of efficiency it is advisable to parallel QR-method with implicit shift for the evaluation of singular values of upper two-diagonal matrix. Therefore, it is reasonable to

carry out evaluation of all approximate singular values simultaneously and completely by all process and form elements of matrices of singular vectors according to (8.11) in parallel by starting transformations from matrices Q or P of (8.12). At the same time the matrix (vector) c^T of (8.14) is formed by analogy.

Along with algorithms that we dealt with earlier, this algorithm will be described in [17].

Distribution of data and results. According to the foregoing, all diagonal and over-diagonal elements of upper two-diagonal matrix $J^{(0)}$ are stated in each process. All evaluated approximate singular values are also stored in each process.

Since each specific (elementary) plane rotation which is used in the evaluation of matrix of singular vectors (8.11) consists of pairs of transformations of only two elements (belonging to columns that are being modified), every row and parameters of the planar rotation is evaluated by each process, then elements of matrix of singular vectors are being formed as well as elements of the matrix c^T of (8.14) and are distributed by row-cyclic scheme. In this case there is no need for the data exchanges between processes during the formation of these matrices.

Algorithm. At each iteration $s = 1, 2, \ldots$of parallel *QR*-algorithm the following operation are performed:

- each process simultaneously with other process and without data exchanges between them evaluates the value of the shift k_s, in succession for $j = 1, 2, \ldots, n_s < n$ forms matrices of plane rotations $S_j^{(s)}, T_j^{(s)}$ and modifies two-diagonal matrix $J^{(s-1)}$ according to (8.9);
- each process simultaneously with other processes and without data exchanges between them according to the distribution scheme modifies elements both of matrix of singular values that is being formed and matrix c^T of (8.14);
- each process simultaneously with other processes and without data exchanging between them verifies the condition for attaining the given accuracy during computing of the next approximate singular value.

Modification of both matrix of singular values being formed and matrix c^T of (8.14) can be carried out immediately after formation of each matrix of plane rotations $P_j^{(s)}$ or at the end of each iteration. After evaluation of the next approximate singular value decimal exponent of the part of matrix being processed decreases by 1.

Efficiency of algorithm. Parallel algorithm has been constructed so that all computations take place without data exchanges between processes. The number of arithmetic operations performed by each of p processes is p times less than that required by the computations in mono-processor mode. Hence, coefficient of acceleration of the parallel *QR*-algorithm is close to p, while coefficient of efficiency is close to unity.

Parallel QR-algorithm is realized in function `qris_prc` in Listing 8.6. The next section consists of this function for data type `mpf_t`.

Listing 8.6 Function of reduction of the bidiagonal matrix to diagonal form using QR-algorithm with implicit shift

```
  int qris_prc (int m, int n, double* ep, double* d,
double* e, int p, int ri, int wU, double** U, int wV, double** V,
int q, int cpr)
  {
  int ir, mp,np,qp, qU,qV,conv, i,j,k,l,r, jp1, kp1, km1, lm1,
cs, cs2;
  double   c, s, f, g, h,  w,w0, eps;
  double  *dk, *ek,  *x,*y;
  int   it0, its, rot0,rots;
  double  titpr, tmodV, tmodc, t0, ts;
  ts = MPI_Wtime();
  qU = q*wV;   qV = q*wU;
  x = e+n;    y = e+3*n;    cs = 1;    cs2 = cs+cs;
  ir = p-ri;   mp = (m+ir)/p;
  np = (n+ir)/p;   qp = (q+ir)/p;
  *ep *= e[0];    eps = *ep;
  e[0] = 0.0;    conv = 1;
  its = rots = 0;    titpr = tmodV = tmodc = 0.0;
  // cycle for the annulment of n-1 overdiagonal elements
  for(  km1=n-1, k=n;  k>0;  )
  {
       kp1 = k;   k = km1;   km1--;
       it0 = rot0 =0;
       while(  conv  )
       {
              for(  l=k;  l>=0;  l=lm1  )
              {
                     lm1 = l-1;
                     if( fabs(e[l])<=eps )  break;
                     if( fabs(d[lm1])<=eps )
                     {
  /* annulment overdiagonal element e [l +1],if l> 0*/
                     t0 = MPI_Wtime();
                     for(j=l,r=0,c=0.0,s=1.0;j<=k;r+=2,j=jp1)
                     {
                            rot0++;
                            f = s*e[j]; e[j] *= c;
                            if( mabs(f)<=eps )  break;
                            h = sqrt( d[j]*d[j]+f*f );
                            x[r] = c = d[j]/h; x[r+1]=s=-f/h;
                            d[j] = h;    jp1 =j+1;
                     }
```

```
                        titpr += MPI_Wtime()-t0;
                        t0 = MPI_Wtime();
                        if( wU )
                        for(  i=0;  i<mp;  i++  )
                        {
                                for(j=l, r=0;j<jp1;r+=2, j++)
                                {
                                w = U[i][lm1];
                                U[i][lm1] = x[r]*w+x[r+1]*U[i][j];
                                U[i][j] = x[r]*U[i][j]-x[r+1]*w;
                                }
                        }
                if( qU )
                        for(  i=0;  i<qp;  i++  )
                        {
                        w0 = U[i][lm1];
                        for(j=l, r=0;j<jp1;r+=2, j++)
                        {
                        w = U[i][lm1];
                        U[i][lm1] = x[r]*w+x[r+1]*U[i][j];
                        U[i][j] = x[r]*U[i][j]-x[r+1]*w;
                        }
                        tmodc += MPI_Wtime()-t0;
                        }
                        break;
                }
        }
        dk = d+k;   ek = e+k;
        if( l==k )  break;
  // formation shift f for QR-transformation
        t0 = MPI_Wtime();
        f   =   0.5   *   ((d[km1]-d[k])*(d[km1]+d[k])   +   (e[km1]-
e[k])*(e[km1]+e[k])) / (d[km1]*e[k]);
        g = sqrt(1.0+f*f);    h = f+msign(g,f);
        f = ((d[l]-d[k])*(d[l]+d[k])+e[k]*(d[km1]/h-e[k]))/d[l];
  // next step QR-transformation
        for(j=l, r=0, c=s=1.0, g=d[l];  j<k; r+=2, j=jp1)
        {
                rot0++;    jp1 = j+1;
                h = s*e[jp1]; e[jp1] *= c;
                e[j] = sqrt( f*f+h*h );
                if( e[j]>0.0 )
                {
                        c = f/e[j]; s = h/e[j];
```

```
        }
        x[r] = c; x[r+1] = s; f = c*g+s*e[jp1];
        e[jp1] = c*e[jp1]-s*g; h = s*d[jp1];
        d[jp1] *= c; d[j] = sqrt( f*f+h*h );
        if( d[j]>0.0 )
        {
               c = f/d[j]; s = h/d[j];
        }
        y[r] = c;    y[r+1] = s;
        f = c*e[jp1]+s*d[jp1];
        g = c*d[jp1]-s*e[jp1];
}
titpr += MPI_Wtime()-t0;
t0 = MPI_Wtime();
if( wV )
        for(  i=0;  i<np;  i++  )
        {
               w = V[i][l];
               for(j=l, r=0; j<jp1; r+=2, j++)
               {
               V[i][j] = x[r]*w+x[r+1]*V[i][j+1];
               w = x[r]*V[i][j+1]-x[r+1]*w;   }
               V[i][jp1] = w;
        }
if( qV )
        for(  i=0;  i<qp;  i++  )
        {
               w = V[i][l];
               for(j=l, r=0; j<jp1; r+=2, j++)
               {
               V[i][j] = x[r]*w+x[r+1]*V[i][j+1];
               w = x[r]*V[i][j+1]-x[r+1]*w;
               }
               V[i][jp1] = w;
        }
tmodV += MPI_Wtime()-t0;
t0 = MPI_Wtime();
if( wU )
        for(  i=0;  i<mp;  i++  )
        {
               w = U[i][l];
               for(j=l, r=0; j<jp1; r+=2, j++)
               {
               U[i][j] = y[r]*w+y[r+1]*U[i][j+1];
```

```
                    w = y[r]*U[i][j+1]-y[r+1]*w;
                    }
                    U[i][jp1] = w;
             }
      if( qU )
             for(  i=0;  i<qp;  i++  )
             {
                    w = U[i][l];
                    for(j=l, r=0; j<jp1; r+=2, j++)
                    {
                    U[i][j] = y[r]*w+y[r+1]*U[i][j+1];
                    w = y[r]*U[i][j+1]-y[r+1]*w;
                    }
                    U[i][jp1] = w;
             }
      tmodc += MPI_Wtime()-t0;
      e[l] = 0.0; d[k] = g; e[k] = f; it0++;
      }
      its += it0; rots += rot0;
      if( d[k]<0.0 )
      {
             d[k] = -d[k];
             if( wV )
                    for(  i=0;  i<np;  i++  )
                           V[i][k] = -V[i][k];
             if( qV )
                    for(  i=0;  i<qp;  i++  )
                           V[i][k] = -V[i][k];
      }
}
      return( 0 );
}
```

To find the normal pseudosolution, determining factor is the use of singular value decomposition of a matrix. The main stages of computing: bringing the matrix to upper bidiagonal form, the accumulation of Householder transformations and the formation of the matrix of singular vectors of the singular decomposition of the upper bidiagonal matrix by QR-algorithm. The total number of arithmetic operations that are performed on each of these stages of transformation when $q << n$ is $O\ (mn2)$, and for all other calculations, is not more than $O\ (mn + n2)$, where q - the number of right sides of the system.

Efficiency of parallel algorithm for finding the normal pseudosolution is determined by the efficiency reduction of the original matrix to bidiagonal.

Speedup and efficiency of the cyclic reduction algorithm for parallel rectangular matrix to bidiagonal form are estimated as follows:

$$S_p \approx p\left(1+\frac{3{,}75(p-1)n}{(3m-n)(n+q)}+\frac{1{,}125\,p\log_2 p}{(3m-n)}\tau_1\right)^{-1},\ E_p=\frac{S_p}{p},$$

where $\tau_1=\tau_o+\frac{4}{3n+4q}\tau_c$.

8.2.2 *Parallel Program for Solution SLAEs by SVD with Multiple Precision*

Consider the example of solving linear algebraic system equations

$$Ax=b, \tag{8.25}$$

where matrix A is square matrix of order n, diagonal elements of which are equal n-i, and nondiagonal are equal to n +1- *max {i, j}*, where *n% 3 = 1*

$$A=\left\{a_{ij}\right\}_1^n,\ a_{ii}=n-i,\ a_{ij}=n+1-\max(i,j),\quad i,j=1\div n,$$

matrix b is rectangular matrix of the right sides of size *N*n*, where *N%3=1*

$$b=\left\{b_i\right\}_1^n,$$

$$b_i=n-i,\quad i\le 2,\quad b_i=n+1-i,\quad i>2,\quad n=3w+1=1000\quad (w=333)$$

Exact solution: $x_2=1,\quad x_i=0,\quad (i=1,3,4,...n)$.

Consider the program for solving linear system equations by method of singular value decomposition of matrix. Listing 8.7 presents the program `Slae_svd_p.c`. This is a program for investigating and solving systems of linear system equations with rectangular or square singular matrices of general form, using singular value decomposition of the matrix of system with arbitrary precision using GMP [6] and MPIGMP [14, 18].

The program provides a generalized solution of SLEs with error estimation.

To obtain a generalized solution of linear algebraic equations singular value decomposition of the original matrix $A = U\Sigma V^{\mathrm{T}}$ is used, which runs by a sequence of two-sided orthogonal transformations of Householder and QR-algorithm with implicit shifts. The generalized solution is obtained by the formulas:

$$x^{\#}=V\left(\Sigma^{\#}c\right),\quad c=U^Tb,$$

where $\Sigma^{\#}$ – pseudoinverse matrix of singular values.

At first, the program `Slae_svd_p.c` checks equivalence of value of parameters. Then called function `dsht_prc()` to reduce the original matrix A to upper bidiagonal form by a sequence of two-sided orthogonal transformations, Householder, at that appropriately converted right-hand side b. After this function called `qris_prc()` is used to calculate the singular values of the upper bidiagonal matrix, and to complete the calculation of the matrix right singular vectors V and the matrix product $c = U^{\mathrm{T}}b$. Next, using function `sv_anal()` to analyze the calculated singular values, computed matrix $\Sigma^{\#}$, calculate the rank and the estimate of the

conditioning number of matrix of SLAEs, and also error estimate obtained generalized solution. The work of function `Slae_svd_p()` is completed with calling function `MatrProd_RrRc()` to calculate the generalized solution.

Since the listing of the program takes up much space, we consider only the basic functions of the program. All function from this parallel program you can find it in `http://extras.springer.com/2012/978-3-642-25672-1`.

Listing 8.7 Program for solving SLEs by SVD with GMP and MPIGMP

Slae_svd_mpf_p.c

```
#include "svd.h"

int Slae_svd_p (
int  nr,            // number of rows of matrix A
int  nc,            // number of columns of matrix A
int nrs,            // number of right parts
                    // (=0, if needs only SVD )
int  ss,            // index storage scheme:
                    // =0 for row scheme,
                    // =1 for column scheme
int  ri,            // number of the first row|column
                    // from the stored in this processor
mpf_t** A,  // input - elements of the matrix
                    // of the problem;
                    // output - elements of of one of
                    // the matrices SVD:
                    // U (ss=nrs=0) | V (ss|nrs>0)
mpf_t  eA,  //  relative error of setting of
                    // elements of the matrix A
mpf_t** C,  // input - elements of the matrix of
                    // the right sides; output -
                    // elements of matrix of solutions
                    // and estimates or other matrix SVD
mpf_t  eC,  //  relative error of setting of
                    // elements of the right sides
int sar,            // buffer size:
            // sar>=((nrs+p-1)/p)*nr+4*min{nr,nc}+2
mpf_t* ar,  //  working buffer
mpf_t* ep,  //  input - "relative" criterium of
                    // zero for QR algorithm;
                    // output-computer zero for matrix A
int* rank,  // output - calculated and effective
                    // rank of matrix A
MPI_Comm  comm )    //  communicator
```

```
  {

  int  err, p,cpr, m,n, mp,np,qp, wU,wV, srs, i,j,jp, cmpres;
  mpf_t  **V = 0, **B = 0, *SV = 0, *iSV = 0, *buf = 0, *erX =
0, tol, eps;
  void *tmpp1 = 0, *tmpp2 = 0;
  double ts;

  // Monitoring data and presets
  if( (ss==0 && nr<nc) || (ss==1 && nr>=nc) )     return(-100 );

  m = __MMAX(nr,nc);
  n = __MMIN(nr,nc);

  if( nrs==0 )   wU = wV = 1;
  else if( ss )
  {
       wU = 1; wV = 0;
  }
  else
  {
       wU = 0; wV = 1;
  }
  MPI_Comm_rank( comm, &cpr );
  MPI_Comm_size( comm, &p );

  mp = (m-1)/p+1;   np = (n-1)/p+1;   qp = (nrs+p-1)/p;

  if( sar<4*n+3*nrs+2 )             return(-100 );

  // Initialization
  mpf_init_set( eps, *ep );
  mpf_init_set( tol, MEPS );

  SV = ar;  buf = ar+n+1; erX = buf+n+1; B=C+qp; V=A;

  /* Decomposition of the matrix of the problem for singular
values and calculation of the matrix C = U'B, if nrs> 0 */
  MPI_Barrier( comm );
  ts = MPI_Wtime();

  /* Bringing the original matrix to upper bidiagonal by double-
side Householder transformation */
```

```
  err = dsht_prc ( m, n, nrs, ri, A, wU, (wU?V:B), wV, (wU?B:V),
tol, SV, buf, comm );
  MPI_Barrier( comm );

  if( err )          return(-500 );

  // Redistribution of the modified matrix right parts
  if( nrs )
  {
       srs = 0;
       ts = MPI_Wtime();
       err = pc_realloc( nr, nrs, &srs, ri, B, n, ri, C, C[0]-
erX, erX, comm );
       MPI_Barrier( comm );
              if( err<-200 )        return(-500 );
              else if( err!=0 )            return(-100 );
  }
  else C = B;
  /* Computation of singular values by using QR-algorithm with
implicit shift */
  ts = MPI_Wtime();
  err = qris_prc ( m, n, &eps, SV, buf, p, ri, wU, (wU?V:C), wV,
(wU?C:V), nrs, cpr );
  MPI_Barrier( comm );
  if( err )
  {
       printf("  pr.#%2d:  qris_prc returns %d    time=%.4e\n",
cpr,err,MPI_Wtime()-ts);
       fflush(stdout);
       return( err );
  }
  /* Analysis of singular values: the calculation of rank and
cond(A);   Calculate the error of the computed generalized
solutions (if nrs> 0) */
  for(i=0;i<n;++i)
       err = sv_anal( p, n, SV, eps, eA, nrs, eC, ri, C, SV+n,
rank, (iSV=buf), erX );
       if( err )             return( err );
  //Computation of the generalized solution (if nrs> 0)
  if( nrs>0 )
  {
       for(  j=0, np=(nc+p-1)/p;  j<nrs;  j++  )
       {
              if( (j%p+1) == ri )
```

```
                    mpf_set( B[np-1][j], erX[j/p] );
                else
                    mpf_set_d( B[np-1][j], 0.0 );
        }
        tmpp1 = allocbuf_mpf( MPF_PREC, nrs );
        tmpp2 = allocbuf_mpf( MPF_PREC, nrs );

        pack_mpf( B[np][0], nrs, tmpp2 );
        pack_mpf( B[np - 1][0], nrs, tmpp1 );

        err = MPI_Reduce( tmpp1, tmpp2, nrs, MPI_MPF,
MPI_MPF_SUM, 0, comm );

        unpack_mpf( tmpp1, B[np - 1][0], nrs );
        unpack_mpf( tmpp2, B[np][0], nrs );

        freebuf_mpf( tmpp2 );
        freebuf_mpf( tmpp1 );

        if( err!=MPI_SUCCESS )
        {
                printf("Slae_svd_p: pr.#%2d - ERROR in Reduce -
rv=%d\n",cpr,err);
                fflush(stdout);
                return(-500 );
        }
        cmpres = mpf_cmp_d( SV[n], (1.0e+32) );
        if( cmpres >= 0 )
                return( 121 );
        for( jp=0; jp<qp; jp++ )
                for( i=0; i<n; i++ )
                    mpf_mul( C[jp][i],C[jp][i],iSV[i]);
                    err = MatrProd_RrRc( nc, n, ri, V, n, nrs,
ri, C, C[0]-erX, erX, comm, B );
                    if( err<-200 ) return(-500);
                    else if( err!=0 )     return(-100 );
    }

    mpf_clear( tol );
    mpf_clear( eps );

    return( 0 );
    }
```

```
  void mpf_msign( mpf_t dest, mpf_t X, mpf_t Y )
  {
       if( mpf_cmp_ui( Y, 0 ) < 0 )
       {
              mpf_abs( dest, X );
              mpf_neg( dest, dest );
       }
       else
       {
              mpf_abs( dest, X );
       }
  }


  /* Function for calculating scalar product of n-dimensional
vectors */
  void ScaPro5( int n, mpf_t* x, mpf_t* y, mpf_t *out )
  {
  int  i,m, sc=5;
  mpf_t  accm;
  m = n%sc;

  mpf_set_d( (*out), 0.0 );
  mpf_init( accm );

  for(  i=0;  i<m;  i++  )
  {
       //prod += x[i]*y[i];
       mpf_mul( accm, x[i], y[i] );
       mpf_add( (*out), (*out), accm );
  }
  for(  ;  i<n;  i+=sc  )
  {
  //prod+=x[i]*y[i]+x[i+1]*y[i+1]+x[i+2]*y[i+2]+x[i+3]*y[i+3]+x[
i+4]*y[i+4];
       mpf_mul( accm, x[i], y[i] );
       mpf_add( (*out), (*out), accm );
       mpf_mul( accm, x[i+1], y[i+1] );
       mpf_add( (*out), (*out), accm );
       mpf_mul( accm, x[i+2], y[i+2] );
       mpf_add( (*out), (*out), accm );
       mpf_mul( accm, x[i+3], y[i+3] );
       mpf_add( (*out), (*out), accm );
       mpf_mul( accm, x[i+4], y[i+4] );
```

```
        mpf_add( (*out), (*out), accm );
    }
    mpf_clear( accm );
    return;
    }

    /* Function to calculate the n-(k + t) of scalar products of
columns k +1 to the columns k + t +1, k + t +2, ..., n of matrix
W */
    int ScalProd_p (
                int   t,       int   q,       mpf_t** Z,
                int  kp,       int  mp,       mpf_t** W,
                int   n,       int   k,       mpf_t* sp,
                int lpr,       int cpr,       int   p,
        MPI_Comm  comm )
    {
    int   i,j, h=5,hr,kh, kpt, cnt, rv;
    mpf_t  *s = 0, *sb = 0, *rb = 0, *r = 0, Wik, tmp;
    void *tmpp1 = 0, *tmpp2 = 0;

    mpf_init( Wik );
    mpf_init( tmp );

    // Presets
    hr = (mp-kp)%h; kh = kp+hr; kpt = k+t; cnt = q+n-kpt;
    if( p==1 )
    {
        s = sp;
        r = s+n;
                sb = s-cnt;
                rb = sb+n;
        if( t )
                mpf_set( s[k], W[kp][k] );
        else
        {
                for(  j=k;  j<n;  j++  )
                        mpf_set( sb[j], W[kp-1][j] );
                        for(  j=0;  j<q;  j++  )
                                mpf_set( rb[j], Z[kp-1][j] );
        }
    }
    else
    {
        rb = sp+k;  r = rb-q; s = r-n; sb = s+kpt;
```

```
  }
  // Computation of partial sums for the scalar product
  for( j=kpt;  j<n;  j++ )
  {
       if( t||lpr!=cpr )
              mpf_set_d( s[j], 0.0 );
       else
              mpf_mul( s[j], W[kp-1][k], W[kp-1][j] );
  }
  for(  j=0;  j<q;  j++  )
  {
       if( n==0||t||lpr!=cpr )
              mpf_set_d( r[j], 0.0 );
       else
              mpf_mul( r[j], W[kp-1][k], Z[kp-1][j] );
  }
  for(  i=kp;  i<mp;  i++  )
  {
       if( n )
              mpf_set( Wik, W[i][k] );
       else
              mpf_set( Wik, W[0][ (i-kp)*p ] );
              for(  j=kpt;  j<n;  j++  )
              {
                     mpf_mul( tmp, Wik, W[i][j] );
                     mpf_add( s[j], s[j], tmp );
              }
              for(  j=0;  j<q;  j++  )
              {
                     mpf_mul( tmp, Wik, Z[i][j] );
                     mpf_add( r[j], r[j], tmp );
              }
  }
  /*  Completion  of  the  computation  of  scalar  products  and
distribution (if p> 1) */
  if( p>1 )
  {
       if( cnt>0 )
       {
              tmpp1 = allocbuf_mpf( MPF_PREC, cnt );
              tmpp2 = allocbuf_mpf( MPF_PREC, cnt );

              pack_mpf( sb[0], cnt, tmpp1 );
              pack_mpf( (rb + t)[0], cnt, tmpp2 );
```

```
            rv = MPI_Reduce( tmpp1, tmpp2, cnt, MPI_MPF,
MPI_MPF_SUM, lpr, comm );

            unpack_mpf( tmpp1, sb[0], cnt );
            unpack_mpf( tmpp2, (rb + t)[0], cnt );

            freebuf_mpf( tmpp1 );
            freebuf_mpf( tmpp2 );

            if( rv!=MPI_SUCCESS )
            {
                  printf("ScalProd_p: pr.#%2d - ERROR in
Reduce - rv=%d\n",cpr,rv); fflush(stdout);
                  return( -4 );
            }
      }
      if( lpr==cpr && n>0 )
      {
            if( t )
                  mpf_set( rb[0], W[kp][k] );
            else
            {
                  for( j=k; j<n; j++ )
                        mpf_set( s[j], W[kp-1][j] );
                        for( j=0; j<q; j++ )
                        mpf_set( r[j],Z[kp-1][j] );
            }
      }
      if( n && t==0 )
      {
            rb -= cnt; cnt *= 2;
      }
      tmpp1 = allocbuf_mpf( MPF_PREC, cnt+t );
      pack_mpf( rb[0], cnt+t, tmpp1 );
      rv = MPI_Bcast( tmpp1, cnt+t, MPI_MPF,lpr, comm);
      unpack_mpf( tmpp1, rb[0], cnt+t );
      freebuf_mpf( tmpp1 );
      if( rv!=MPI_SUCCESS )
      {
            printf("ScalProd_p: pr.#%2d - ERROR in Bcast -
rv=%d\n",cpr,rv);
            return( -8 );
        }
```

```
}
mpf_clear( Wik );
mpf_clear( tmp );

return( 0 );
}

/// Parallel double-side Householder's algorithm
int dsht_prc (int    m,    int    n,  int    q,
            int   ri,     mpf_t**  A,
            int   wU,     mpf_t**  U,
            int   wV,     mpf_t**  V,
            mpf_t  tol,   mpf_t*  D0, mpf_t* buf,
            MPI_Comm      comm )
{
int i,j,k,l,r, kp1, kp,mp,np,npq,qU,qV,ir,cr, p,cpr,lpr, err;
mpf_t  dk,ek,fk, c,s,t,z,  absp, maxp, tmp, tmp2;
mpf_t *d=0, *e=0, *f=0, *g=0, *g0=0, *gi=0, *x=0;
double ts;

mpf_init( dk );
mpf_init( ek );
mpf_init( fk );
mpf_init( c );
mpf_init( s );
mpf_init( t );
mpf_init( z );
mpf_init( absp );
mpf_init( maxp );
mpf_init( tmp );
mpf_init( tmp2 );

MPI_Comm_rank( comm, &cpr );
MPI_Comm_size( comm, &p );
ir = p-ri;  cr = ir+cpr;
mp = (m+ir)/p;  np = (n+ir)/p;  npq = n+q;
qU = q*wV;  qV = q*wU;  d = D0; e = buf; g0 = buf+2;
f = g0+n+qU;  x = buf+2*(n+qU+1);

ts = MPI_Wtime();

mpf_set_d( e[0], 0.0 );
mpf_set_d( maxp, 0.0 );
for( k=0;  k<n;  k++, f--  )
```

```
  {
       kp1 = k+1; kp = (kp1+ir)/p; lpr = (kp1+cr)%p;

  // Calculation of parameters of left transformation
       err = ScalProd_p ( 0, qU, U, kp, mp, A, n, k, f, lpr,
cpr, p, comm );
       if( err )
              return( err-16 );
              g = (cpr==lpr?A[kp-1]:g0);
              if( mpf_cmp( f[k], tol ) < 0 )
              {
                     mpf_set_d( dk, 0.0 );
                     mpf_set_d( d[k], 0.0 );
                     for(  j=kp1, r=0;  j<n;  r++, j++  )
                            mpf_set_d( x[r], 0.0 );
              }
              else
              {
              //d[k] = -msign(sqrt(f[k]),g[k]);
                     mpf_sqrt( tmp, f[k] );
                     mpf_msign( tmp, tmp, g[k] );
                     mpf_neg( d[k], tmp );
                     mpf_set( dk, d[k] );
                     mpf_sub( g[k], g[k], dk );
                     //s = 1.0/(g[k]*dk);
                     mpf_mul( tmp, g[k], dk );
                     mpf_set_d( s, 1.0 );
                     _CHECK_ZERO_DIVISOR(tmp,"0");
                     mpf_div( s, s, tmp );
                     for(  j=kp1, r=0;  j<n;  r++, j++  )
                     {
                            //x[r] = (f[j]-dk*g[j])*s;
                            mpf_mul( x[r], dk, g[j] );
                            mpf_sub( x[r], f[j], x[r] );
                            mpf_mul( x[r], x[r], s );
                            //g[j] = f[j]/dk;
                            mpf_set( g[j], f[j] );
                            _CHECK_ZERO_DIVISOR(dk,"1");
                            mpf_div( g[j], g[j], dk );
                     }
                     for(  l=0;  l<qU;  r++, j++, l++  )
                     {
                            //x[r] = (f[j]-dk*g0[j])*s;
                            mpf_mul( x[r], dk, g0[j] );
```

```
                        mpf_sub( x[r], f[j], x[r] );
                        mpf_mul( x[r], x[r], s );
                        if( lpr==cpr )
                        {
                        //U[kp-1][l] += x[r]*g[k];
                        mpf_mul( tmp, x[r], g[k] );
                        mpf_add(U[kp-1][l],U[kp-1][l],tmp );
                        }
                }
        }
// Calculation of parameters of the right transformation
        ScaPro5( n-kp1, g+kp1, g+kp1, &fk );
        if( mpf_cmp( fk, tol ) < 0 )
        {
        mpf_set_d( ek, 0.0 );
                mpf_set_d( e[kp1], 0.0 );
                mpf_set_d( c, 0.0 );
                mpf_set_d( t, 0.0 );
        }
        else
        {
        //e[kp1] = -msign(sqrt(fk),g[kp1]);
                mpf_sqrt( e[kp1], fk );
                mpf_msign( e[kp1], e[kp1], g[kp1] );
                mpf_neg( e[kp1], e[kp1] );
                mpf_set( ek, e[kp1] );
                mpf_sub( g[kp1], g[kp1], ek );
                //t = 1.0/(g[kp1]*ek);
                mpf_mul( tmp, g[kp1], ek );
                mpf_set_d( t, 1.0 );
                _CHECK_ZERO_DIVISOR(tmp,"2");
                mpf_div( t, t, tmp );
                //c=0.5*t*ScaPro5(n-kp1, x, g+kp1 );
                ScaPro5( n-kp1, x, g+kp1, &tmp );
                mpf_set_d( c, 0.5 );
                mpf_mul( c, c, t );
                mpf_mul( c, c, tmp );
        }
// Double-side transform
        for(  j=kp1, r=0;  j<n;  r++, j++  )
        {
        //x[r] += c*g[j];
                mpf_mul( tmp, c, g[j] );
                mpf_add( x[r], x[r], tmp );
```

```
                }
                if( qV )
                {
                        err = ScalProd_p ( 0, qV, V, kp, np, &g, 0,
0, f, lpr, cpr, p, comm );
                        if( err )return( err-24 );
                        for(  l=0;  l<qV;  r++, l++  )
                        mpf_mul( x[r], f[l], t );
                        gi = g+ri-1;
                }
                for(  i=kp;  i<mp;  i++  )
                {
                mpf_set( s, A[i][k] );
                //z=t*ScaPro5(n-kp1, A[i]+kp1, g+kp1 )+c*s;
                        ScaPro5(n-kp1,A[i]+kp1,g+kp1,&tmp);
                        mpf_mul( z, t, tmp );
                        mpf_mul( tmp, c, s );
                        mpf_add( z, z, tmp );
                        for(  j=kp1, r=0;  j<n;  r++, j++  )
                        {
                                //A[i][j] += s*x[r]+z*g[j];
                                mpf_mul( tmp, s, x[r] );
                                mpf_mul( tmp2, z, g[j] );
                                mpf_add( tmp, tmp, tmp2 );
                                mpf_add(A[i][j],A[i][j],tmp);
                        }
                        for(  l=0;  l<qU;  r++, l++  )
                        {
                                //U[i][l] += s*x[r];
                                mpf_mul( tmp, s, x[r] );
                                mpf_add( U[i][l],U[i][l],tmp);
                        }
                        if( qV )
                                mpf_set( s, gi[p*i] );
                                for(  l=0;  l<qV;  r++, l++  )
                                {
                                //V[i][l] += s*x[r];
                                mpf_mul( tmp, s, x[r] );
                                mpf_add(V[i][l],V[i][l],tmp);
                                }
                }
  // Save the settings right transformation
                if( wV )
                {
```

```
                        gi = g+ri-1;
                        for(i=kp, r=i*p; i<np;  i++, r+=p  )
                                mpf_set( V[i][k], gi[r] );
                }
                //if( (absp=mabs(d[k])+mabs(e[k]))>maxp )
                //      maxp = absp;
                mpf_abs( tmp, d[k] );
                mpf_abs( tmp2, e[k] );
                mpf_add( absp, tmp, tmp2 );
                if( mpf_cmp( absp, maxp ) > 0 )
                        mpf_set( maxp, absp );
        }
   // Formation of the matrix ot right transformation
   ts = MPI_Wtime();
   if( wV )
   {
        mpf_set_d( e[n], 0.0 );
        for(  f=buf+n+1, k=n;  k>0;  f++  )
        {
                kp1 = k; kp = (kp1+ir)/p;
                lpr = (kp1+1+cr)%p; k--;
                mpf_set_d( tmp, 0.0 );
                if( !( mpf_cmp( e[kp1], tmp ) == 0) )
                {
                err = ScalProd_p ( 1, 0, U, kp, np, V, n, k, f,
lpr, cpr, p, comm );
                        if( err )               return( err-32 );
                        //s = 1.0/(f[k]*e[kp1]);
                        mpf_set_d( s, 1.0 );
                        mpf_mul( tmp, f[k], e[kp1] );
                        _CHECK_ZERO_DIVISOR(tmp,"3");
                        mpf_div( s, s, tmp );
                        for(  i=kp;  i<np;  i++  )
                        {
                                mpf_mul( t, V[i][k], s );
                                for(  j=kp1;  j<n;  j++  )
                                {
                                //V[i][j] += t*f[j];
                                mpf_mul( tmp, t, f[j] );
                                mpf_add(V[i][j],V[i][j],tmp);
                                }
                                }
                }
                for(  i=kp;  i<np;  i++  )
```

```
                    mpf_set_d( V[i][k], 0.0 );
                    if( k%p+1==ri )
                    {
                            mpf_set_d( V[kp-1][k], 1.0 );
                            for(  ;kp1<n;  kp1++  )
                            mpf_set_d(V[kp-1][kp1], 0.0 );
                    }
        }
  }
  // Formation of the matrix ot left transformation
  ts = MPI_Wtime();
  if( wU )
       for(  f=buf+n+1, k=n;  k>0;  f++  )
       {
              kp1 = k;  k--; kp = (k+ir)/p;
              lpr = (kp1+cr)%p;
              if( cpr == lpr )
                    for(  j=kp1;  j<n;  j++  )
                    mpf_set_d( U[kp][j], 0.0 );
                    mpf_set_d( tmp, 0.0 );
                    if(!( mpf_cmp( d[k], tmp )== 0 ) )
                    {
                    err = ScalProd_p ( 1, 0, V, kp, mp, U, n,
k, f, lpr, cpr, p, comm );
                    if( err )             return( err-48 );
                    //s = 1.0/(f[k]*e[kp1]);
                    mpf_mul( tmp, f[k], e[kp1] );
                    mpf_set_d( s, 1.0 );
                    _CHECK_ZERO_DIVISOR(tmp,"4");
                    mpf_div( s, s, tmp );
                    for(  i=kp;  i<mp;  i++  )
                    {
                    mpf_mul( t, U[i][k], s );
                    mpf_set_d( U[i][k], 0.0 );
                    for(  j=k;  j<n;  j++  )
                    {
                    //U[i][j] += t*f[j];
                    mpf_mul( tmp, t, f[j] );
                    mpf_add( U[i][j], U[i][j], tmp );
                    }
                    }
                    }
                    else
                    for(  i=kp;  i<mp;  i++  )
```

```
                    mpf_set_d(U[i][k],0.0 );
                    if( cpr==lpr )
                    mpf_add_ui( U[kp][k],U[kp][k],1 );
  }
  mpf_set( (*buf), maxp );

  mpf_clear( dk );
  mpf_clear( ek );
  mpf_clear( fk );
  mpf_clear( c );
  mpf_clear( s );
  mpf_clear( t );
  mpf_clear( z );
  mpf_clear( absp );
  mpf_clear( maxp );
  mpf_clear( tmp );
  mpf_clear( tmp2 );

  return( 0 );
  }

  /// Parallel QR-algorithm
  //  Function to bring the bidiagonal matrix to diagonal form
using QR-algorithm with implicit shift
  int qris_prc (     int m, int n, mpf_t* ep, mpf_t* d, mpf_t*
e, int p, int ri, int wU, mpf_t** U, int wV, mpf_t** V, int    q,
int  cpr )
  {
  int ir,mp,np,qp,qU,qV,conv,i,j,k,l,r,jp1,kp1,km1,lm1, cs,cs2;
  mpf_t   c, s, f, g, h,  w,w0, eps, tmp, tmp2;
  mpf_t  *dk = 0, *ek = 0,  *x = 0, *y = 0;
  int   it0, its, rot0,rots;
  double  titpr, tmodV, tmodc, t0, ts;

  mpf_init( tmp );
  mpf_init( tmp2 );
  mpf_init( c );
  mpf_init( s );
  mpf_init( f );
  mpf_init( g );
  mpf_init( h );
  mpf_init( w );
  mpf_init( w0 );
  mpf_init( eps );
```

```
// Presets
ts = MPI_Wtime();
qU = q*wV;  qV = q*wU;  x = e+n;  y = e+3*n; cs = 1;
cs2 = cs+cs;  ir = p-ri;
mp = (m+ir)/p; np = (n+ir)/p; qp = (q+ir)/p;

mpf_mul( (*ep) ,(*ep), e[0] );
mpf_set( eps, (*ep) );
mpf_set_d( e[0], 0.0 );
conv = 1;
its = rots = 0;

titpr = 0.0;
tmodV = 0.0;
tmodc = 0.0;

// Cycle for the cancellation of n-1 upper-diagonal elements
for(  km1=n-1, k=n;  k>0;  )
{
     kp1 = k;  k = km1; km1--;  it0 = rot0 =0;
     while(  conv  )
     {
            for(  l=k;  l>=0;  l=lm1  )
            {
                   lm1 = l-1;
                   mpf_abs( tmp, e[l] );
                   if( mpf_cmp( tmp, eps ) <= 0) break;
                   mpf_abs( tmp, d[lm1] );
                   if( mpf_cmp( tmp, eps ) <= 0 )
                   {
// Cancellation upper-diagonal element e [l +1], if l> 0
                          t0 = MPI_Wtime();
                          mpf_set_d( s, 1.0 );
                          mpf_set_d( c, 0.0 );
                          for(j=l, r=0; j<=k;r+=2,j=jp1)
                          {
                          rot0++;
                          mpf_mul( f, s, e[j] );
                          mpf_mul( e[j],e[j], c );
                          mpf_abs( tmp, f );
                          if(mpf_cmp(tmp,eps)<=0)
                          break;
                          //h = sqrt( d[j]*d[j]+f*f );
```

```
            mpf_mul( tmp,d[j],d[j]);
            mpf_mul( tmp2, f, f );
            mpf_add( tmp,tmp,tmp2 );
            mpf_sqrt( h, tmp );
            _CHECK_ZERO_DIVISOR(h,"5");
            //x[r] = c = d[j]/h;
            mpf_div( tmp, d[j], h );
            mpf_set( x[r], tmp );
            mpf_set( c, tmp );
            _CHECK_ZERO_DIVISOR(h,"6");
            //x[r+1] = s = -f/h;
            mpf_div( tmp, f, h );
            mpf_neg( tmp, tmp );
            mpf_set( x[r+1], tmp );
            mpf_set( s, tmp );
            mpf_set( d[j], h );
            jp1 =j+1;
            }
            titpr += MPI_Wtime()-t0;
            t0 = MPI_Wtime();
            if( wU )
            for(  i=0;  i<mp;  i++  )
            {
            for(j=l,r=0;j<jp1;r+=2,j++)
            {
            mpf_set(w, U[i][lm1]);
    //U[i][lm1]=x[r]*w+x[r+1]*U[i][j];
            mpf_mul( tmp, x[r], w );
            mpf_mul( tmp2,x[r+1],U[i][j]);
            mpf_add(U[i][lm1],tmp, tmp2 );
    //U[i][j] = x[r]*U[i][j]-x[r+1]*w;
            mpf_mul( tmp, x[r], U[i][j] );
            mpf_mul( tmp2, x[r+1], w );
            mpf_sub( U[i][j], tmp, tmp2 );
            }
            }
            if( qU )
            for(  i=0;  i<qp;  i++  )
            {
            mpf_set( w0, U[i][lm1] );
            for( j=l,r=0;j<jp1;r+=2,j++ )
            {
            mpf_set( w, U[i][lm1] );
    //U[i][lm1]=x[r]*w+x[r+1]*U[i][j];
```

```
                                mpf_mul( tmp, x[r], w );
                                mpf_mul( tmp2,x[r+1],U[i][j]);
                                mpf_add( U[i][lm1],tmp,tmp2);
                        //U[i][j] = x[r]*U[i][j]-x[r+1]*w;
                                mpf_mul( tmp, x[r], U[i][j] );
                                mpf_mul( tmp2, x[r+1], w );
                                mpf_sub( U[i][j], tmp, tmp2 );
                                }
                                tmodc += MPI_Wtime()-t0;
                                }
                                break;
                        }
                }
                dk = d+k;
                ek = e+k;
                if( l==k ) break;
  // Formation shift f for QR-transformation
                t0 = MPI_Wtime();
  ////  f  =  0.5  *  ((d[km1]-d[k])*(d[km1]+d[k])  +  (e[km1]-
e[k])*(e[km1]+e[k])) / (d[km1]*e[k]);
                mpf_sub( tmp, d[km1], d[k] );
                mpf_add( tmp2, d[km1], d[k] );
                mpf_mul( f, tmp, tmp2 );
                mpf_sub( tmp, e[km1], e[k] );
                mpf_add( tmp2, e[km1], e[k] );
                mpf_mul( tmp, tmp, tmp2 );
                mpf_add( tmp, f, tmp );
                mpf_set_d( f, 0.5 );
                mpf_mul( tmp, tmp, f );
                mpf_mul( tmp2, d[km1], e[k] );
                _CHECK_ZERO_DIVISOR(tmp2,"7");
                mpf_div( f, tmp, tmp2 );
                //// g = sqrt(1.0+f*f);
                mpf_mul( tmp, f, f );
                mpf_add_ui( tmp, tmp, 1 );
                mpf_sqrt( g, tmp );
                //// h = f+msign(g,f);
                mpf_msign( tmp, g, f );
                mpf_add( h, f, tmp );
  // f=((d[l]-d[k])*(d[l]+d[k])+e[k]*(d[km1]/h-e[k]))/d[l];
                mpf_sub( tmp, d[l], d[k] );
                mpf_add( tmp2, d[l], d[k] );
                mpf_mul( f, tmp, tmp2 );
                mpf_sub( tmp, h, e[k] );
```

```
            _CHECK_ZERO_DIVISOR(tmp,"8");
            mpf_div( tmp, d[km1], tmp );
            mpf_mul( tmp, e[k], tmp );
            mpf_add( f, f, tmp );
            _CHECK_ZERO_DIVISOR(d[l],"9");
            mpf_div( f, f, d[l] );
// Next step QR-transformation
            mpf_set_d( c, 1.0 );
            mpf_set_d( s, 1.0 );
            mpf_set( g, d[l] );
            for(  j=l, r=0;  j<k;  r+=2, j=jp1  )
            {
                  rot0++;
                  jp1 = j+1;
                  mpf_mul( h, s, e[jp1] );
                  mpf_mul( e[jp1], e[jp1], c );
                  mpf_mul( tmp, f, f );
                  mpf_mul( tmp2, h, h );
                  mpf_add( tmp, tmp, tmp2 );
                  mpf_sqrt( e[j], tmp );
                  mpf_set_d( tmp, 0.0 );
                  if( mpf_cmp( e[j], tmp ) > 0 )
                  {
                  _CHECK_ZERO_DIVISOR(e[j],"10");
                  mpf_div( c, f, e[j] );
                  _CHECK_ZERO_DIVISOR(e[j],"11");
                  mpf_div( s, h, e[j] );
                  }
                  mpf_set( x[r], c );
                  mpf_set( x[r+1], s );
                  //// f = c*g+s*e[jp1];
                  mpf_mul( f, c, g );
                  mpf_mul( tmp, s, e[jp1] );
                  mpf_add( f, f, tmp );
                  //// e[jp1] = c*e[jp1]-s*g;
                  mpf_mul( e[jp1], c, e[jp1] );
                  mpf_mul( tmp, s, g );
                  mpf_sub( e[jp1], e[jp1], tmp );
                  mpf_mul( h, s, d[jp1] );
                  mpf_mul( d[jp1], d[jp1], c );
               //// d[j] = sqrt( f*f+h*h );
                  mpf_mul( tmp, f, f );
                  mpf_mul( tmp2, h, h );
                  mpf_add( tmp, tmp, tmp2 );
```

```
            mpf_sqrt( d[j], tmp );
            mpf_set_d( tmp, 0.0 );
            if( mpf_cmp( d[j], tmp ) > 0 )
            {
            _CHECK_ZERO_DIVISOR(d[j],"12");
            mpf_div( c, f, d[j] );
            _CHECK_ZERO_DIVISOR(d[j],"13");
            mpf_div( s, h, d[j] );
            }
            mpf_set( y[r], c );
            mpf_set( y[r+1], s );
            //// f = c*e[jp1]+s*d[jp1];
            mpf_mul( tmp, c, e[jp1] );
            mpf_mul( tmp2, s, d[jp1] );
            mpf_add( f, tmp, tmp2 );
            //// g = c*d1]-s*e[jp1];
            mpf_mul( tmp, c, d[jp1] );
            mpf_mul( tmp2, s, e[jp1] );
            mpf_sub( g, tmp, tmp2 );
    }
    titpr += MPI_Wtime()-t0;
    t0 = MPI_Wtime();
    if( wV )
    for(  i=0;  i<np;  i++  )
    {
            mpf_set( w, V[i][l] );
            for(  j=l, r=0;  j<jp1;  r+=2, j++)
            {
            //V[i][j] = x[r]*w+x[r+1]*V[i][j+1];
            mpf_mul( tmp, x[r], w );
            mpf_mul( tmp2, x[r+1], V[i][j+1] );
            mpf_add( V[i][j], tmp, tmp2 );
            //w = x[r]*V[i][j+1]-x[r+1]*w;
            mpf_mul( tmp, x[r], V[i][j+1] );
            mpf_mul( tmp2, x[r+1], w );
            mpf_sub( w, tmp, tmp2 );
            }
            mpf_set( V[i][jp1], w );
    }
    if( qV )
   for(  i=0;  i<qp;  i++  )
    {
            mpf_set( w, V[i][l] );
            for(  j=l, r=0;  j<jp1;  r+=2, j++)
```

```
        {
        //V[i][j] = x[r]*w+x[r+1]*V[i][j+1];
        mpf_mul( tmp, x[r], w );
        mpf_mul( tmp2, x[r+1], V[i][j+1] );
        mpf_add( V[i][j], tmp, tmp2 );
        //w = x[r]*V[i][j+1]-x[r+1]*w;
        mpf_mul( tmp, x[r], V[i][j+1] );
        mpf_mul( tmp2, x[r+1], w );
        mpf_sub( w, tmp, tmp2 );
        }
        mpf_set( V[i][jp1], w );
}
tmodV += MPI_Wtime()-t0;
t0 = MPI_Wtime();
if( wU )
for(  i=0;  i<mp;  i++  )
{
        mpf_set( w, U[i][l] );
        for(  j=l, r=0;  j<jp1;  r+=2, j++)
        {
        //U[i][j] = y[r]*w+y[r+1]*U[i][j+1];
        mpf_mul( tmp, y[r], w );
        mpf_mul( tmp2, y[r+1], U[i][j+1] );
        mpf_add( U[i][j], tmp, tmp2 );
        //w = y[r]*U[i][j+1]-y[r+1]*w;
        mpf_mul( tmp, y[r], U[i][j+1] );
        mpf_mul( tmp2, y[r+1], w );
        mpf_sub( w, tmp, tmp2 );
        }
        mpf_set( U[i][jp1], w );
}
if( qU )
        for(  i=0;  i<qp;  i++  )
        {
        mpf_set( w, U[i][l] );
        for(  j=l, r=0;  j<jp1;  r+=2, j++)
        {
        //U[i][j] = y[r]*w+y[r+1]*U[i][j+1];
        mpf_mul( tmp, y[r], w );
        mpf_mul(tmp2, y[r+1], U[i][j+1] );
        mpf_add( U[i][j], tmp, tmp2 );
        //w = y[r]*U[i][j+1]-y[r+1]*w;
        mpf_mul( tmp, y[r], U[i][j+1] );
        mpf_mul( tmp2, y[r+1], w );
```

```
                    mpf_sub( w, tmp, tmp2 );
                    }
                    mpf_set( U[i][jp1], w );
                    }
                    tmodc += MPI_Wtime()-t0;
                    mpf_set_d( e[l], 0.0 );
                    mpf_set( d[k], g );
                    mpf_set( e[k], f );
                    it0++;
        }
        its += it0;
        rots += rot0;
        mpf_set_d( tmp, 0.0 );
        if( mpf_cmp( d[k], tmp ) < 0 )
        {
                mpf_neg( d[k], d[k] );
                if( wV )for(  i=0;  i<np;  i++  )
                        mpf_neg( V[i][k], V[i][k] );
                if( qV )for(  i=0;  i<qp;  i++  )
                        mpf_neg( V[i][k], V[i][k] );
        }
    }

        mpf_clear( tmp );
        mpf_clear( tmp2 );
        mpf_clear( c );
        mpf_clear( s );
        mpf_clear( f );
        mpf_clear( g );
        mpf_clear( h );
        mpf_clear( w );
        mpf_clear( w0 );
        mpf_clear( eps );

        return( 0 );
    }
```

The result of the program with normal completion is listed data below.

- A generalized solution $x^{\#}$ (in place of the right sides). Elements of the matrix (vector) of the generalized solutions are distributed to processors according to horizontal cyclic scheme, which coincide with the scheme of distribution of the elements of the matrix.
- Error estimates for generalized solutions.
- Calculated and effective rank of the matrix.

- Estimating the conditioning number of the matrix.
- The diagonal elements of diagonal matrices of singular values Σ and $\Sigma^{\#}$ (located fully in each processor in the working buffer).
- The matrix of right singular vectors V; its elements are distributed to processors according to horizontal cyclic scheme, which coincides with the scheme of distribution of the elements of the original matrix, and are located on their site.
- The matrix of left singular vectors of U (if set to q = 0); its elements are distributed to processors according to horizontal cyclic scheme, which coincide with the scheme of distribution of the elements of the original matrix.

The protocols of executed programs with `double` and increased digit were presented in Listings 8.8 and 8.9.

Listing 8.8 The protocol of the program Slae_svd_p.c with double precision

```
PROBLEM:
     solving of system of linear algebraic equations
     with rectangular general matrix

    Input Parameters:
        - the number of rows                =  1000
        - the number of columns             =  1000
        - the number of right hand sides    =     1
        - maximal relative errors
           of matrix elements    = 0.000e+00
           of right hand sides elements    = 0.000e+00

   Process of research and solution of the problem

   Method:
       - singular value decomposition of a general matrix

   Results:         SOLUTION WAS CALCULATED
   the first 12 components of solution (vector 1) are:
 5.6333892e-10 1.0000000e+00  9.4955344e-10 -3.9295606e-10
 3.4584592e-11 -3.1325660e-11 -5.2191194e-11 1.0333934e-10
 4.0820428e-11 1.2171034e-10 -1.3408585e-10 -1.3067941e-10
 The vecror(s) of solution are successfully stored in the file
result.out

       Error estimations:   4.99145e-08

       Properties:
        - estimation of conditional number: 7.49316e+07
        - matrix rank: 999 (999)
```

```
     Number of processors: 16
     Time of the problem solving: 2.43078e+00

***  SINGULAR VALUES:
Maximal
4.0568920395844785e+05  4.5075763402881799e+04
1.6226688158594263e+04
8.2784735506754641e+03
5.0076033418825928e+03  3.3518942488331431e+03
2.3996165932057975e+03
1.8021505384225013e+03  1.4028553799870913e+03
1.1228786850331160e+03
minimal
2.6589222401030971e-02  2.2155492320442959e-02
2.1366524181246346e-02
1.6540565028568651e-02  1.6096816909122732e-02
1.0976747353899348e-02
1.0779540714116454e-02  5.4634267893085391e-03
5.4141273411846850e-03
1.7284405921760462e-14
#result=0
```

Listing 8.9 The protocol of the program Slae_svd_mpf_p.c

```
MEPS=    2.9387358770557188e-39

PROBLEM:
     solving of system of linear algebraic equations
     with rectangular general matrix

   Input Parameters:
         - the number of rows               =  1000
         - the number of columns            =  1000
         - the number of right hand sides   =     1
         - maximal relative errors
                   of matrix elements    = 0.000e+00
         of right hand sides elements    = 0.000e+00

Process of research and solution of the problem

Method:
  - singular value decomposition of a general matrix

Results:         SOLUTION WAS CALCULATED
```

```
  the first 12 components of solution (vector 1) are:
  4.54202365317217102961341237270854073423 7e-10
  9.99999988133053712363884862610545339883e-1
  8.59633785524852192094904251130341068299 9e-10
  -5.77874554552283761371930465705715516256 0e-10
  2.47884244100045702896906265507688417889 2e-10
  -4.61473200574616371177285928040703957299 4e-11
  -1.33205881774091489405517782293700810636 1e-10
  2.47514735329583591123106309160447666673 1e-10
  -1.41351130547047135142908327081825499533 4e-10
  4.64629488131611662502568200121516672827 1e-11
  1.57934479073569176162088912917338771491 7e-10
  -1.32193356536517252300590911524967191966 1e-10

  The vecror(s) of solution are successfully stored in the file
result.out

       Error estimations:   4.99145e-08

       Properties:
       - estimation of conditional number: 7.49316e+07
       - matrix rank: 999 (999)

       Number of processors: 256
       Time of the problem solving: 1.07412e+01

  ***  SINGULAR VALUES:
  maximal
  4.05689203958447683098188137130537612501 1e5
  4.50757634028817772976716184953658531212 3e4
  1.62266881585942612488155837733065656492 6e4
  8.27847355067545217907040824146830381138 7e3
  5.00760334188258998336870932463919418384 1e3
  3.35189424883314488685251841304853594720 4e3
  2.39961659320579800307402550753466679831 9e3
  1.80215053842250162162445627094580674852 6e3
  1.40285537998709170769255999845227550395 9e3
  1.12287868503311620599399753832529514974 5e3

  minimal
  2.65892224010156592957076209577722327884 7e-2
  2.21554923204427722079488629680503218705 3e-2
  2.13665241811318761275480185431817213490 0e-2
```

```
1.65405650285687673465127665648883105382 8e-2
1.60968169090590137690033624296025566352 4e-2
1.09767473539086282341381841216693475969 1e-2
1.07795407140743012597401271640857593206 1e-2
5.46342678935915851583799867119247464868 9e-3
5.41412734116999940645828573166797025073 1e-3
2.72554407447103686180265188962863468333 9e-36

#result=0
```

Testing program performed on Inparcom-256 [10, 11], with precision `double` (53), 64 and 128 bit. Results of testing program are presented in Tables 8.3-8.6.

Table 8.3 Time solution (sec) of problem (2) with GMP and MPI for N=1000

Prec	1	4	8	32	128	192	256
double	11,0791	1,52111	0,803173	1,04195	1,57758	1,51271	1,90184
64	450,829	117,160	62,8410	18,5836	11,0297	10,9702	10,7412
128	594,989	154,374	80,8474	26,0746	24,0095	12,5435	13,0108

Table 8.4 Time solution (sec)of problem (2) with GMP and MPI for N=2500

Prec	4	8	16	32	128	192	256
double	48,0993	35,8675	11,7230	4,88885	6,41928	6,02911	8,42860
64	1659,39	846,960	443,934	243,781	103,786	77,5082	72,9634
128	2307.96	1190.85	631,778	339,839	125,317	105,785	94,0263

Table 8.5 Time solution (sec) of problem (2) with GMP and MPI for N=4000

Prec	32	64	128	192	256
double	32,5347	13,9563	14,0896	13,0781	19,1600
64	927,812	523,079	346,555	251,406	230,815
128	1302,40	743,607	444,035	344,978	303,514

Table 8.6 Time solution (sec) of problem (2) with GMP and MPI for N=7000

Prec	32	64	128	192	256
double	241.233	112.685	57.3805	39.4785	42.1095
64	4763,24	2522,31	1601,05	1093,58	933,728
128	6794,95	3641,33	2044,11	1557,08	1288,25

Dependence of time of solution of system of linear equations (8.25) by singular value decomposition of matrix with different order of matrices from required accuracy `MPF_PREC` using 256 processes is presented in Fig. 8.3. Dependence of

speedup of program of order of matrix N=1000 with different precision from number of processes is presented on Fig. 8.4.

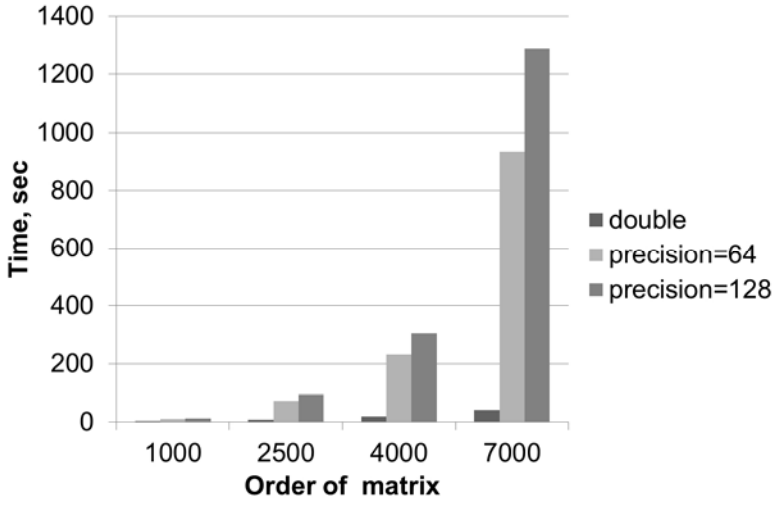

Fig. 8.3 Dependence of time of solution of problem (8.25) with different order of matrices from required accuracy MPF_PREC using 256 processes

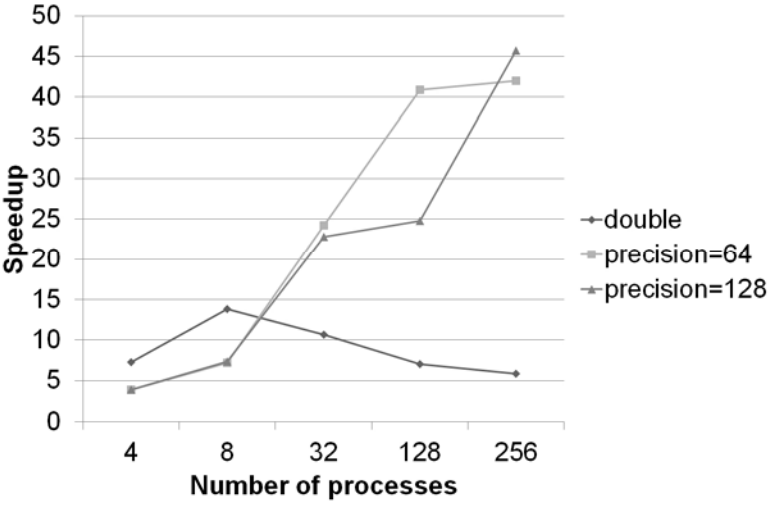

Fig. 8.4 Dependence of speedup of program for solution of SLAEs by SVD (order of matrix N=1000) with different precision from number of processes

As it is seen from the tables and graphs, we can talk about the preferred use of parallel computers to calculate with high accuracy.

We consider two parallel programs with high accuracy using the GMP library. And we can make a conclusion that for ill-conditioned systems with exactly given initial data which use the GMP library functions, for increasing the bit of calculation can obtain a solution with the required accuracy. The time of solving problems using programs of the GMP library decreases with increasing volume of tasks. Also functions of GMP can be used as in a traditional computer program and as in parallel programs.

References

Several books contain listings of algorithms, or even come with a disk containing software, for example, the introductory books [4, 5]. The monograph [12] is devoted to the development and investigation of parallel algorithms for the solving of basic classes of problems in the computational mathematics on parallel computers: linear algebraic systems, algebraic eigenvalue problems, nonlinear equations and systems as well as initial-value problem for systems of 1st order ordinary differential equations.

1. Bjorck, A.: Numerical Methods for Least Squares Problems. SIAM, Philadelphia (1996)
2. Choi, J., Dongarra, J.J., Ostrouchov, L.S.: The design and implementation of the scalapack LU, QR and Cholesky factorization routines. Scientific Programming 5(3), 173–184 (1996)
3. Dongarra, J.J., Duff, I.S., Sorensen, D.C., Van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia (1998)
4. Forsythe, G.E., Malcolm, M.A., Moler, C.B.: Computer Methods for Mathematical Computations. Prentice-Hall, Englewood Cliffs (1977)
5. Forsythe, G.E., Moler, C.B.: Computer Solution of Linear Algebraic Systems. Prentice-Hall, Englewood Cliffs (1967)
6. GMP Library, http://www.gmplib.org/
7. Golub, G.H.: Least squares, singular values and matrix approximations. Aplikace Matematiky 13, 44–51 (1968)
8. Golub, G.H., Kahan, W.: Calculating the singular values and pseudo-inverse of a matrix. SIAM J. Numer. Anal. Ser. B. 2, 205–224 (1965)
9. Golub, G.H., Reinsch, C.: Singular value decomposition and least squares solution. Numer. Math. 14, 403–420 (1970)
10. Inparcom (family of intelligent parallel computers), http://www.geopoisk.com/inparcom/eng/index_eng.htm
11. Khimich, A.N., Molchanov, I.N., Mova, V.I., et al.: Numerical program software of intelligent computer Inparcom. Naukova dumka, Kiev (2007)
12. Khimich, A.N., Polyanko, V.V.: Block-cyclic Gauss parallel algorithms for solving systems of linear algebraic equations with sparse matrices. Herald of Lviv University. Series Appl. Math. and Inform. 15, 109–116 (2009)
13. Khimich, A.N., Polyanko, V.V.: The effectiveness of two-dimensional block-cyclic parallel algorithms. Problems of Programming 3, 145–149 (2008)

14. Kouya, T.: A Brief Introduction to MPIGMP & MPIBNCpack (2008), http://na-inet.jp/na/bnc/brief_intro_mpibncpack.pdf (accessed August 21, 2011)
15. Lawson, C.L., Hanson, R.J.: Solving Least Squares Problems. SIAM, Philadelphia (1995)
16. Molchanov, I.N.: Machine methods for solving applied problems. Algebra, approximation of functions, ordinary differential equations. Naukova dumka, Kiev (2007)
17. Molchanov, I.N., Popov, A.V., Khimich, A.N.: Parallel algorithm for the singular value decomposition of matrices. Theory of optimal solutions, 80–83 (2001) (in Russian)
18. MPIGMP Library, http://na-inet.jp/na/bnc
19. Stewart, G.W.: On the early history of the singular value decomposition. SIAM Review 35, 551–566 (1993)
20. Wilkinson, J.H.: The algebraic eigenvalue problem. Clarendon Press, Oxford (1965)
21. Wilkinson, J.H., Reinsch, C.: Handbook for automatic computation. Linear Algebra. Springer, Berlin (1971)

Appendix A
Main `mpf_t` and `mpfr_t` Functions

In this book, we were introduced you to the basic functions of libraries. We could not cover all the functions of the different types of variables. Our task was to help the reader to understand how to write programs with arbitrary precision with using the appropriate functions.

For specific tasks, you must use the appropriate set of functions. For your convenience and understanding, we propose in this application, a set of tables with the basic GMP and MPFR functions with different types of variables, including those that have not been described previously. We have given a comparative table for types `mpf_t` and `mpfr_t`, respectively.

These tables are below:

Table A1. Initialization functions

Table A2. Assignment functions

Table A3. Init&Assign functions

Table A4. Conversion functions

Table A5. Arithmetic functions

Table A6. Comparison functions

Table A7. Input and Output functions

Table A8. Rounding Related MPFR functions

Table A1 Initialization functions

mpf_t	**mpfr_t**
`void mpf_set_default_prec(mp_bitcnt_t prec)`	`void mpfr_set_default_prec (mpfr_prec_t prec)`
Set the default precision to be at least prec bits.	Set the default precision to be exactly prec bits, where prec can be any integer between MPFR_PREC_MIN and MPFR_PREC_MAX.
`mp_bitcnt_t mpf_get_default_prec (void)`	`mpfr_prec_t mpfr_get_default_prec (void)`
Return the default precision actually used.	Return the current default MPFR precision in bits.
`void mpf_init (mpf_t x)`	`void mpfr_init (mpfr_t x)`
Initialize `x` to 0.	Initialize `x`, set its precision to the default precision, and set its value to `NaN`.
	`void mpfr_inits (mpfr_t x, ...)`
	Initialize all the `mpfr_t` variables of the given va_list, set their precision to the default precision and their value to `NaN`
`void mpf_init2 (mpf_t x, mp_bitcnt_t prec)`	`void mpfr_init2 (mpfr_t x, mpfr_prec_t prec)`
Initialize `x` to 0 and set its precision to be at least `prec` bits.	Initialize `x`, set its precision to be exactly `prec` bits and its value to `NaN`
`void mpf_inits (mpf_t x, ...)`	`void mpfr_inits2 (mpfr_prec_t prec, mpfr_t x, ...)`
Initialize a NULL-terminated list of `mpf_t` variables, and set their values to 0.	Initialize all the `mpfr_t` variables of the given variable argument va_list, set their precision to be exactly `prec` bits and their value to `NaN`
`void mpf_clear (mpf_t x)`	`void mpfr_clear (mpfr_t x)`
Free the space occupied by `x`.	Free the space occupied by the significand of `x`.
`void mpf_clears (mpf_t x, ...)`	`void mpfr_clears (mpfr_t x, ...)`
Free the space occupied by a NULL-terminated list of `mpf_t` variables.	Free the space occupied by all the `mpfr_t` variables of the given va_list.
`mp_bitcnt_t mpf_get_prec (mpf_t x)`	`mpfr_prec_t mpfr_get_prec (mpfr_t x)`
Return the current precision of `x`, in bits.	Return the precision of `x`, i.e., the number of bits used to store its significand.
`void mpf_set_prec (mpf_t x, mp_bitcnt_t prec)`	`void mpfr_set_prec (mpfr_t x, mpfr_prec_t prec)`
Set the precision of `x` to be at least `prec` bits.	Reset the precision of `x` to be exactly `prec` bits, and set its value to `NaN`.
`void mpf_set_prec_raw (mpf_t x, mp_bitcnt_t prec)`	
Set the precision of `x` to be at least `prec` bits, without changing the memory allocated.	

Table A2 Assignment functions

mpf_t	**mpfr_t**
`void mpf_set (mpf_t x, mpf_t y)`	`int mpfr_set (mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
`void mpf_set_ui (mpf_t x, unsigned long int y)`	`int mpfr_set_ui (mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)`
`void mpf_set_si (mpf_t x, signed long int y)`	`int mpfr_set_si (mpfr_t x, long int y, mpfr_rnd_t rnd)`
`void mpf_set_d (mpf_t x, double y)`	`int mpfr_set_uj (mpfr_t x, uintmax_t y, mpfr_rnd_t rnd)`
`void mpf_set_z (mpf_t x, mpz_t y)`	`int mpfr_set_sj (mpfr_t x, intmax_t y, mpfr_rnd_t rnd)`
`void mpf_set_q (mpf_t x, mpq_t y)`	`int mpfr_set_flt (mpfr_t x, float y, mpfr_rnd_t rnd)`
	`int mpfr_set_d (mpfr_t x, double y, mpfr_rnd_t rnd)`
	`int mpfr_set_ld (mpfr_t x, long double y, mpfr_rnd_t rnd)`
	`int mpfr_set_decimal64 (mpfr_t x, _Decimal64 y, mpfr_rnd_t rnd)`
	`int mpfr_set_z (mpfr_t x, mpz_t y, mpfr_rnd_t rnd)`
	`int mpfr_set_q (mpfr_t x, mpq_t y, mpfr_rnd_t rnd)`
	`int mpfr_set_f (mpfr_t x, mpf_t y, mpfr_rnd_t rnd)`
Set the value of `x` from `y`.	Set the value of `x` from `y`, rounded toward the given direction `rnd`.
	`int mpfr_set_ui_2exp (mpfr_t x, unsigned long int y, mpfr_exp_t e, mpfr_rnd_t rnd)`
	`int mpfr_set_si_2exp (mpfr_t x, long int y, mpfr_exp_t e, mpfr_rnd_t rnd)`
	`int mpfr_set_uj_2exp (mpfr_t x, uintmax_t y, intmax_t e, mpfr_rnd_t rnd)`
	`int mpfr_set_sj_2exp (mpfr_t x, intmax_t y, intmax_t e, mpfr_rnd_t rnd)`
	`int mpfr_set_z_2exp (mpfr_t x, mpz_t y, mpfr_exp_t e, mpfr_rnd_t rnd)`
	Set the value of `x` from `y` multiplied by two to the power `e`, rounded toward the given direction rnd.
`int mpf_set_str (mpf_t x, char *str, int base)`	`int mpfr_set_str (mpfr_t x, const char *s, int base, mpfr_rnd_t rnd)`
Set the value of `x` from the string in `str`.	Set `x` to the value of the string `s` in base `base`, rounded in the direction `rnd`.

Table A2 *(continued)*

mpf_t	mpfr_t
	`int mpfr_strtofr (mpfr_t x, const char *nptr, char **endptr, int base, mpfr_rnd_t rnd)`
	Read a floating-point number from a string `nptr` in base `base`, rounded in the direction `rnd`; `base` must be either 0 or a number from 2 to 62.

	`void mpfr_set_nan (mpfr_t x)`
	`void mpfr_set_inf (mpfr_t x, int sign)`
	`void mpfr_set_zero (mpfr_t x, int sign)`

	Set the variable `x` to `NaN` (Not-a-Number), infinity or zero respectively.
`void mpf_swap (mpf_t x, mpf_t y)`	`void mpfr_swap (mpfr_t x, mpfr_t y)`
Swap `x` and `y` efficiently.	Swap the values `x` and `y` efficiently.

Table A3 Init&Assign functions

mpf_t	mpfr_t
`void mpf_init_set (mpf_t x, mpf_t y)`	`int mpfr_init_set (mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
`void mpf_init_set_ui (mpf_t x, unsigned long int y)`	`int mpfr_init_set_ui (mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)`
`void mpf_init_set_si (mpf_t x, signed long int y)`	`int mpfr_init_set_si (mpfr_t x, long int y, mpfr_rnd_t rnd)`
`void mpf_init_set_d (mpf_t x, double y)`	`int mpfr_init_set_d (mpfr_t x, double y, mpfr_rnd_t rnd)`
	`int mpfr_init_set_ld (mpfr_t x, long double y, mpfr_rnd_t rnd)`
	`int mpfr_init_set_z (mpfr_t x, mpz_t y, mpfr_rnd_t rnd)`
	`int mpfr_init_set_q (mpfr_t x, mpq_t y, mpfr_rnd_t rnd)`
	`int mpfr_init_set_f (mpfr_t x, mpf_t y, mpfr_rnd_t rnd)`
Initialize `x` and set its value from `y`.	Initialize `x` and set its value from `y`, rounded in the direction `rnd`.
`int mpf_init_set_str (mpf_t x, char *str, int base)`	`int mpfr_init_set_str (mpfr_t x, const char *s, int base, mpfr_rnd_t rnd)`
Initialize `x` and set its value from the string in `str`.	Initialize `x` and set its value from the string `s` in base `base`, rounded in the direction `rnd`.

Table A4 Conversion functions

mpf_t	**mpfr_t**

`double mpf_get_d (mpf_t x)`	`float mpfr_get_flt (mpfr_t x, mpfr_rnd_t rnd)`
	`double mpfr_get_d (mpfr_t x, mpfr_rnd_t rnd)`
	`long double mpfr_get_ld (mpfr_t x, mpfr_rnd_t rnd)`
	`_Decimal64 mpfr_get_decimal64 (mpfr_t x, mpfr_rnd_t rnd)`

Convert `x` to a `double`, truncating if necessary (ie. rounding to zero).	Convert `x` to a `float` (respectively `double`, `long double` or `_Decimal64`), using the rounding mode `rnd`.

`long mpf_get_si (mpf_t x)`
`unsigned long mpf_get_ui(mpf_t x)`
	`long mpfr_get_si (mpfr_t x, mpfr_rnd_t rnd)`
	`unsigned long mpfr_get_ui (mpfr_t x, mpfr_rnd_t rnd)`
	`intmax_t mpfr_get_sj (mpfr_t x, mpfr_rnd_t rnd)`
	`uintmax_t mpfr_get_uj (mpfr_t x, mpfr_rnd_t rnd)`

Convert `x` to a `long` or `unsigned long`, truncating any fraction part. If op is too big for the return type, the result is undefined.	Convert `x` to a `long`, an `unsigned long`, an `intmax_t` or an `uintmax_t` (respectively) after rounding it with respect to `rnd`.
`double mpf_get_d_2exp (signed long int *exp, mpf_t x)`	`double mpfr_get_d_2exp (long *exp, mpfr_t x, mpfr_rnd_t rnd)` `long double mpfr_get_ld_2exp (long *exp, mpfr_t x, mpfr_rnd_t rnd)`
Convert `x` to a `double`, truncating if necessary (ie. rounding to zero), and with an exponent returned separately.	Returned and set `exp` (formally, the value pointed to by `exp`) such that 0.5•|d|<1 and d times 2 raised to `exp` equals `x` rounded to `double` (resp. `long double`) precision, using the given rounding mode.
	`mpfr_exp_t mpfr_get_z_2exp (mpz_t x, mpfr_t y)`
	Put the scaled significand of `y` (regarded as an integer, with the precision of `y`) into `x`, and return the exponent exp (which may be outside the current exponent range) such that y exactly equals `x` times 2 raised to the power exp.
	`int mpfr_get_z (mpz_t x, mpfr_t y, mpfr_rnd_t rnd)`
	Convert `y` to a `mpz_t`, after rounding it with respect to `rnd`.
	`int mpfr_get_f (mpf_t x, mpfr_t y, mpfr_rnd_t rnd)`
	Convert `y` to a `mpf_t`, after rounding it with respect to `rnd`.

Table A4 *(continued)*

mpf_t	mpfr_t
`char * mpf_get_str (char *str, mp_exp_t *expptr, int base, size_t n_digits, mpf_t x)`	`char * mpfr_get_str (char *str, mpfr_exp_t *expptr, int b, size_t n, mpfr_t x, mpfr_rnd_t rnd)`
Convert `x` to a string of digits in base `base`.	Convert `x` to a string of digits in base `b`, with rounding in the direction `rnd`, where `n` is either zero (see below) or the number of significant digits output in the string; in the latter case, `n` must be greater or equal to 2.
	`void mpfr_free_str (char *str)`
	Free a string allocated by `mpfr_get_str` using the current unallocation function.
	`int mpfr_fits_ulong_p (mpfr_t x, mpfr_rnd_t rnd)`
	`int mpfr_fits_slong_p (mpfr_t x, mpfr_rnd_t rnd)`
	`int mpfr_fits_uint_p (mpfr_t x, mpfr_rnd_t rnd)`
	`int mpfr_fits_sint_p (mpfr_t x, mpfr_rnd_t rnd)`
	`int mpfr_fits_ushort_p (mpfr_t x, mpfr_rnd_t rnd)`
	`int mpfr_fits_sshort_p (mpfr_t x, mpfr_rnd_t rnd)`
	`int mpfr_fits_uintmax_p (mpfr_t x, mpfr_rnd_t rnd)`
	`int mpfr_fits_intmax_p (mpfr_t x, mpfr_rnd_t rnd)`
	Return non-zero if `x` would fit in the respective C data type, respectively `unsigned long`, `long`, `unsigned int`, `int`, `unsigned short`, `short`, `uintmax_t`, `intmax_t`, when rounded to an integer in the direction `rnd`.

Table A5 Arithmetic functions

mpf_t	mpfr_t
`void mpf_add (mpf_t z, mpf_t x, mpf_t y)`	`int mpfr_add (mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
`void mpf_add_ui (mpf_t z, mpf_t x, unsigned long int y)`	`int mpfr_add_ui (mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)`
	`int mpfr_add_si (mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)`
	`int mpfr_add_d (mpfr_t z, mpfr_t x, double y, mpfr_rnd_t rnd)`
	`int mpfr_add_z (mpfr_t z, mpfr_t x, mpz_t y, mpfr_rnd_t rnd)`

Table A5 *(continued)*

mpf_t	mpfr_t
	`int mpfr_add_q (mpfr_t z, mpfr_t x, mpq_t y, mpfr_rnd_t rnd)`
Set `z` to `x+y`.	Set `z` to `x+y` rounded in the direction `rnd`.
`void mpf_sub (mpf_t z, mpf_t x, mpf_t y)`	`mpfr_sub (mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
`void mpf_ui_sub (mpf_t z, unsigned long int x, mpf_t y)`	`int mpfr_ui_sub (mpfr_t z, unsigned long int x, mpfr_t y, mpfr_rnd_t rnd)`
`void mpf_sub_ui (mpf_t z, mpf_t x, unsigned long int y)`	`int mpfr_sub_ui (mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)`
	`int mpfr_si_sub (mpfr_t z, long int x, mpfr_t y, mpfr_rnd_t rnd)`
	`int mpfr_sub_si (mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)`
	`int mpfr_d_sub (mpfr_t z, double x, mpfr_t y, mpfr_rnd_t rnd)`
	`int mpfr_sub_d (mpfr_t z, mpfr_t x, double y, mpfr_rnd_t rnd)`
	`int mpfr_sub_z (mpfr_t z, mpfr_t x, mpz_t y, mpfr_rnd_t rnd)`
	`int mpfr_sub_q (mpfr_t z, mpfr_t x, mpq_t y, mpfr_rnd_t rnd)`
Set `z` to `x-y`.	Set `z` to `x-y` rounded in the direction `rnd`.
`void mpf_mul (mpf_t z, mpf_t x, mpf_t y)`	`int mpfr_mul (mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
`void mpf_mul_ui (mpf_t z, mpf_t x, unsigned long int y)`	`int mpfr_mul_ui (mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)`
	`int mpfr_mul_si (mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)`
	`int mpfr_mul_d (mpfr_t z, mpfr_t x, double y, mpfr_rnd_t rnd)`
	`int mpfr_mul_z (mpfr_t z, mpfr_t x, mpz_t y, mpfr_rnd_t rnd)`
	`int mpfr_mul_q (mpfr_t z, mpfr_t x, mpq_t y, mpfr_rnd_t rnd)`
Set `z` to `x` times `y`.	Set `z` to `x` times `y` rounded in the direction `rnd`.
`void mpf_div (mpf_t z, mpf_t x, mpf_t y)`	`int mpfr_div (mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
`void mpf_ui_div (mpf_t z, unsigned long int x, mpf_t y)`	`int mpfr_ui_div (mpfr_t z, unsigned long int x, mpfr_t y, mpfr_rnd_t rnd)`
`void mpf_div_ui (mpf_t z, mpf_t x, unsigned long int y)`	`int mpfr_div_ui (mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)`
	`int mpfr_si_div (mpfr_t z, long int x, mpfr_t y, mpfr_rnd_t rnd)`

Table A5 *(continued)*

mpf_t	**mpfr_t**

	`int mpfr_div_si (mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)`
	`int mpfr_d_div (mpfr_t z, double x, mpfr_t y, mpfr_rnd_t rnd)`
	`int mpfr_div_d (mpfr_t z, mpfr_t x, double y, mpfr_rnd_t rnd)`
	`int mpfr_div_z (mpfr_t z, mpfr_t x, mpz_t y, mpfr_rnd_t rnd)`
	`int mpfr_div_q (mpfr_t z, mpfr_t x, mpq_t y, mpfr_rnd_t rnd)`

Set `z` to `x/y`.	Set `z` to `x/y` rounded in the direction `rnd`.
	`int mpfr_sqr (mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
	Set `x` to the square of `y` rounded in the direction `rnd`.
`void mpf_sqrt (mpf_t x, mpf_t y)` `void mpf_sqrt_ui (mpf_t x, unsigned long int y)`	`int mpfr_sqrt (mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)` `int mpfr_sqrt_ui (mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)`
Set `x` to the square root of `y`.	Set `x` to the square root of `y` rounded in the direction `rnd` (set `x` to −0 if `y` is −0, to be consistent with the IEEE 754 standard).
	`int mpfr_rec_sqrt (mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
	Set `x` to the reciprocal square root of `y` rounded in the direction `rnd`.
	`int mpfr_cbrt (mpfr_t x, mpfr_t y mpfr_rnd_t rnd)` `int mpfr_root (mpfr_t x, mpfr_t y, unsigned long int k, mpfr_rnd_t rnd)`
	Set `x` to the cubic root (resp. the `k`-th root) of `y` rounded in the direction `rnd`.

`void mpf_pow_ui (mpf_t z, mpf_t x, unsigned long int y)`	`int mpfr_pow (mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
	`int mpfr_pow_ui (mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)`
	`int mpfr_pow_si (mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)`
	`int mpfr_pow_z (mpfr_t z, mpfr_t x, mpz_t y, mpfr_rnd_t rnd)`
	`int mpfr_ui_pow_ui (mpfr_t z, unsigned long int x, unsigned long int y, mpfr_rnd_t rnd)`
	`int mpfr_ui_pow (mpfr_t z, unsigned long int x, mpfr_t y, mpfr_rnd_t rnd)`

Set `z` to `x` raised to the power `y`.	Set `z` to `x` raised to the `y`, rounded in the direction `rnd`.

Table A5 *(continued)*

mpf_t	**mpfr_t**
`void mpf_neg (mpf_t x, mpf_t y)`	`int mpfr_neg (mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
Set x to −y.	Set x to −y, rounded in the direction rnd.
`void mpf_abs (mpf_t x, mpf_t y)`	`int mpfr_abs (mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
Set x to the absolute value of y.	Set x absolute value of y, rounded in the direction rnd.
	`int mpfr_dim (mpfr_t z, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)`
	Set z to the positive difference of x and y, i.e., x−y rounded in the direction rnd if x>y, +0 if x≤y, and NaN if x or y is NaN.
`void mpf_mul_2exp (mpf_t z, mpf_t x, mp_bitcnt_t y)`	`int mpfr_mul_2ui (mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)` `int mpfr_mul_2si (mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)`
Set z to x times 2 raised to y.	Set z to x times 2 raised to y rounded in the direction rnd.
`void mpf_div_2exp (mpf_t z, mpf_t x, mp_bitcnt_t y)`	`int mpfr_div_2ui (mpfr_t z, mpfr_t x, unsigned long int y, mpfr_rnd_t rnd)` `int mpfr_div_2si (mpfr_t z, mpfr_t x, long int y, mpfr_rnd_t rnd)`
Set z to x divided by 2 raised to y.	Set z to x divided by 2 raised to y rounded in the direction rnd.

Table A6 Comparison functions

mpf_t	**mpfr_t**
`int mpf_cmp (mpf_t x, mpf_t y)`	`int mpfr_cmp (mpfr_t x, mpfr_t y)`
`int mpf_cmp_d (mpf_t x, double y)`	`int mpfr_cmp_ui (mpfr_t x, unsigned long int y)`
`int mpf_cmp_ui (mpf_t x, unsigned long int y)`	`int mpfr_cmp_si (mpfr_t x, long int y)`
`int mpf_cmp_si (mpf_t x, signed long int y)`	`int mpfr_cmp_d (mpfr_t x, double y)`
	`int mpfr_cmp_ld (mpfr_t x, long double y)`
	`int mpfr_cmp_z (mpfr_t x, mpz_t y)`
	`int mpfr_cmp_q (mpfr_t x, mpq_t y)`
	`int mpfr_cmp_f (mpfr_t x, mpf_t y)`

Table A6 *(continued)*

mpf_t	**mpfr_t**
Compare x and y. Return a positive value if x>y, zero if x=y, and a negative value if x<y.	Compare x and y. Return a positive value if x>y, zero if x=y, and a negative value if x<y. Both x and y are considered to their full own precision, which may differ. If one of the operands is NaN, set the erange flag and return zero.
	`int mpfr_cmp_ui_2exp (mpfr_t x, unsigned long int y, mpfr_exp_t e)` `int mpfr_cmp_si_2exp (mpfr_t x, long int y, mpfr_exp_t e)`
	Compare x and y multiplied by two to the power e.
	`int mpfr_cmpabs (mpfr_t x, mpfr_t y)`
	Compare |x| and |y|. Return a positive value if |x|>|y|, zero if |x|=|y|, and a negative value if |x|<|y|. If one of the operands is NaN, set the erange flag and return zero.

	`int mpfr_nan_p (mpfr_t x)`
	`int mpfr_inf_p (mpfr_t x)`
	`int mpfr_number_p (mpfr_t x)`
	`int mpfr_zero_p (mpfr_t x)`
	`int mpfr_regular_p (mpfr_t x)`

	Return non-zero if x is respectively NaN, an infinity, an ordinary number (i.e., neither NaN nor an infinity), zero, or a regular number (i.e., neither NaN, nor an infinity nor zero). Return zero otherwise.

	`int mpfr_greater_p (mpfr_t x, mpfr_t y)`
	`int mpfr_greaterequal_p (mpfr_t x, mpfr_t y)`
	`int mpfr_less_p (mpfr_t x, mpfr_t y)`
	`int mpfr_lessequal_p (mpfr_t x, mpfr_t y)`
	`int mpfr_equal_p (mpfr_t x, mpfr_t y)`

	Return non-zero if x>y, x≥y, x<y, x≤y, x=y respectively, and zero otherwise. Those functions return zero whenever x and/or y is NaN.
	`int mpfr_lessgreater_p (mpfr_t x, mpfr_t y)`

Table A6 *(continued)*

mpf_t	mpfr_t
	Return non-zero if x<y or x>y (i.e., neither x, nor y is NaN, and x<>y), zero otherwise (i.e., x and/or y is NaN, or x=y).
	`int mpfr_unordered_p (mpfr_t x, mpfr_t y)`
	Return non-zero if x or y is a NaN (i.e., they cannot be compared), zero otherwise.
`int mpf_eq (mpf_t x, mpf_t y, mp_bitcnt_t z)`	
Return non-zero if the first z bits of x and y are equal, zero otherwise. I.e., test if x and y are approximately equal.	
`void mpf_reldiff (mpf_t z, mpf_t x, mpf_t y)`	
Compute the relative difference between x and y and store the result in z. This is |x-y|/x.	
`int mpf_sgn (mpf_t x)`	`int mpfr_sgn (mpfr_t x)`
Return +1 if x>0, 0 if x=0, and -1 if x<0.	Return a +1 if x>0, 0 if x=0, and -1 if x<0. If the operand is NaN, set the erange flag and return zero. This is equivalent to `mpfr_cmp_ui(x,0)`, but more efficient.

Table A7 Input and Output functions

mpf_t	mpfr_t
`size_t mpf_out_str (FILE *stream, int base, size_t n_digits, mpf_t x)`	`size_t mpfr_out_str (FILE *stream, int base, size_t n, mpfr_t x, mpfr_rnd_t rnd)`
Print x to stream, as a string of digits. Return the number of bytes written, or if an error occurred, return 0.	Output x on stream `stream`, as a string of digits in base `base`, rounded in the direction `rnd`. The `base` may vary from 2 to 62. Print n significant digits exactly, or if n is 0, enough digits so that x can be read back exactly (see `mpfr_get_str`).
`size_t mpf_inp_str (mpf_t y, FILE *stream, int base)`	`size_t mpfr_inp_str (mpfr_t y, FILE *stream, int base, mpfr_rnd_t rnd)`
Read a string in base `base` from `stream`, and put the read `float` in y.	Input a string in base `base` from stream `stream`, rounded in the direction `rnd`, and put the read float in y. This function reads a word (defined as a sequence of characters between whitespace) and parses it using `mpfr_set_str`.

Table A8 Rounding Related MPFR functions

Functions	
`void mpfr_set_default_rounding_mode (mpfr_rnd_t rnd)`	Set the default rounding mode to `rnd`. The default rounding mode is to nearest initially.
`mpfr_rnd_t mpfr_get_default_rounding_mode (void)`	Get the default rounding mode.
`int mpfr_prec_round (mpfr_t x, mpfr_prec_t prec, mpfr_rnd_t rnd)`	Round `x` according to `rnd` with precision `prec`, which must be an integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX` (otherwise the behavior is undefined). If `prec` is greater or equal to the precision of `x`, then new space is allocated for the significand, and it is filled with zeros. Otherwise, the significand is rounded to precision `prec` with the given direction. In both cases, the precision of `x` is changed to `prec`.
`int mpfr_can_round (mpfr_t b, mpfr_exp_t err, mpfr_rnd_t rnd1, mpfr_rnd_t rnd2, mpfr_prec_t prec)`	Assuming `b` is an approximation of an unknown number `x` in the direction `rnd1` with error at most two to the power `E(b)-err` where `E(b)` is the exponent of `b`, return a non-zero value if one is able to round correctly `x` to precision `prec` with the direction `rnd2`, and 0 otherwise (including for `NaN` and `Inf`). This function does not modify its arguments.
`mpfr_prec_t mpfr_min_prec(mpfr_t x)`	Return the minimal number of bits required to store the significand of `x`, and 0 for special values, including 0. (Warning: the returned value can be less than `MPFR_PREC_MIN`.)
`const char * mpfr_print_rnd_mode (mpfr_rnd_t rnd)`	Return a string (`"MPFR_RNDD"`, `"MPFR_RNDU"`, `"MPFR_RNDN"`, `"MPFR_RNDZ"`, `"MPFR_RNDA"`) corresponding to the rounding mode `rnd`, or a null pointer if `rnd` is an invalid rounding mode.

Appendix B
The List of the Examples of Programs

Book consist of many examples of programs. Some examples is not full, because it have listings very large. So all full programs you can find on `http://extras.springer.com/2012/978-3-642-25672-1`, that attached to this book.

Here we describe list of programs, that you can find in this book, and also on `http://extras.springer.com/2012/978-3-642-25672-1`. If program consist of many functions, we include figures (Fig. B1-B4), that show their dependence.

1. Programs for calculation of macheps

1.1. P. for calculation of macheps……..…….....Chap.1(p.7)

- `macheps_double.c`

1.2. P. for calculation of macheps with GMP……..…….Sec.2.4(p.29)

- `macheps_mpf.c`

2. Programs for adding of two variables

2.1. P. for adding two variables……..…….....Sec.2.2(p.17)

- `add.c`

2.2. P. for adding two variables with GMP…….....Sec.2.2(p.18)

- `add_mpf.c`

3. Programs for solution system of linear algebraic equations by Gauss method

3.1. P. for solution SLAE by Gauss method…….....Sec.3.2(p.36)

- `gauss.c`

3.2. P. for solution SLAE by Gauss method using GMP……..Sec.3.2(p.38)

- `gauss_mpf.c`

3.3. P. for solution SLAE by Gauss method using MPFR……Sec.4.3(p.50)

- `gauss_mpfr.c`

4. Program C++ for solution system of linear algebraic equations by LU-decomposition using GMP……..…….Sec.5.3(p.89)

- `LU_mpf.cpp`
- `Lu.h`
- `matrices.h`

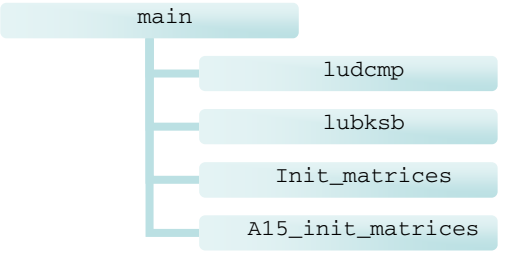

Fig. B1 Functions of program for solution SLAE by LU-decomposition

5. Programs for matrix multiplications

5.1. P. for multiplication matrix…….Sec.5.2(p.80)

- `m_mult_m_d.cpp`
- `matrix_d.h`

5.2. P. for multiplication matrix with GMP …….Sec.5.2(p.83)

- `m_mult_m_mpf.cpp`
- `matrix_mpf.h`

5.3. Parallel p. for matrix multiplication……...…..Subsec.6.2.2(p.110)

- `mpi_mult_m_d.cpp`
- `mpi_matrix.h`

5.4. Parallel p. for matrix multiplication with GMP using MPI_BYTE in communications……..Sec.6.3(p.116)

- `mpi_mult_m_mpf.cpp`
- `mpi_matrixg.h`

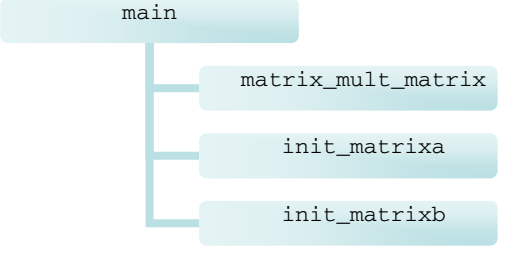

Fig. B2 Functions of programs for matrix multiplication

6. Using MPIGMP library in parallel programming

6.1. Parallel p. for calculating $f(a)=4/(1+a^2)$ and number π using GMP……Sec.7.2(p.122)

- `cpi_gmp.c`

6.2. Parallel MPI-program for solution system of linear algebraic equations by Gauss method with `double` accuracy…….....Subsec.8.1.1(p.137)

- `DenseSolver.c`
- `DenseData.c`

- DenseExchange.c
- DenseFactorization.c
- DenseMath.c
- DensePrinter.c
- DenseSolver.h

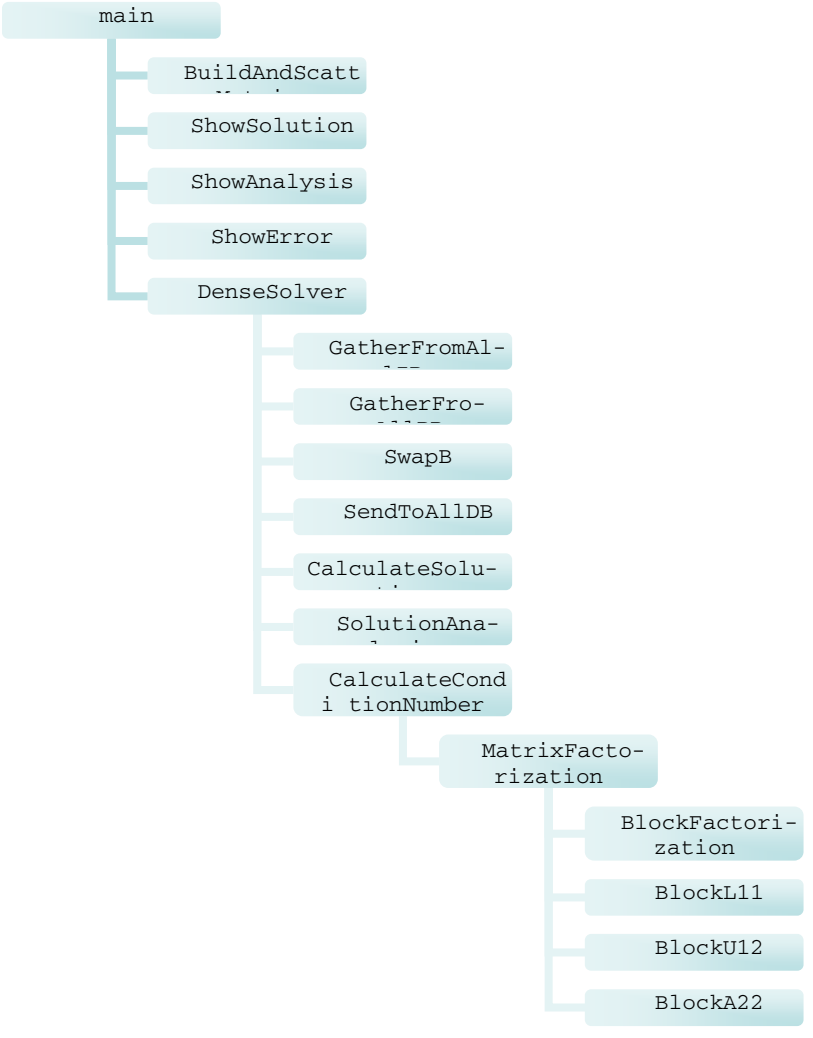

Fig. B3 Functions for parallel programs for solution SLAE by LU-decomposition

6.3. Parallel MPI-program for solution system of linear algebraic equations by Gauss method with GMP……Subsec.8.1.2(p.146)

- GausSolver.c

- GausData.c
- GausExchange.c
- GausFactorization.c
- GausMath.c
- GausPrinter.c
- GausSolver.h

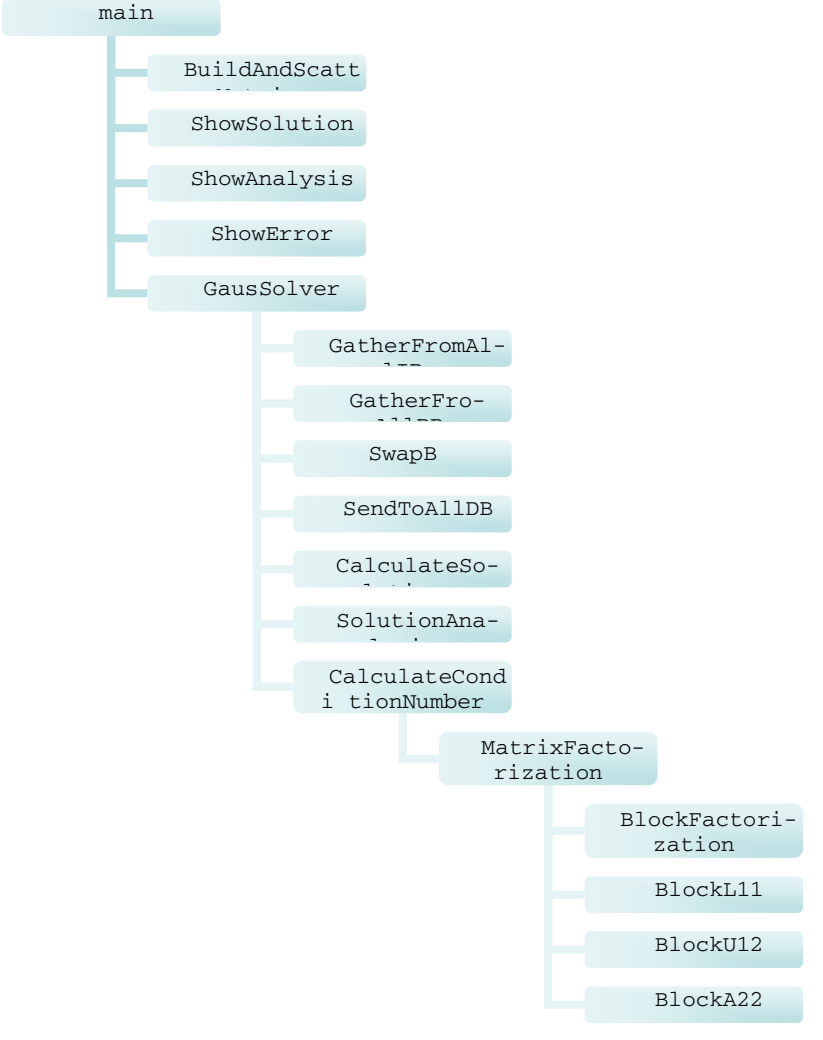

Fig. B4. Functions for parallel programs for solution SLAE by LU-decomposition with GMP

6.4. Parallel MPI-program for solution SLAE by singular value decomposition with `double` accuracy.....Subsec.8.2.1(p.168, 173)

- `Slae_svd_p.c`

- Slae_svd_p_test.c
- A41.c
- B41.c
- svd.h

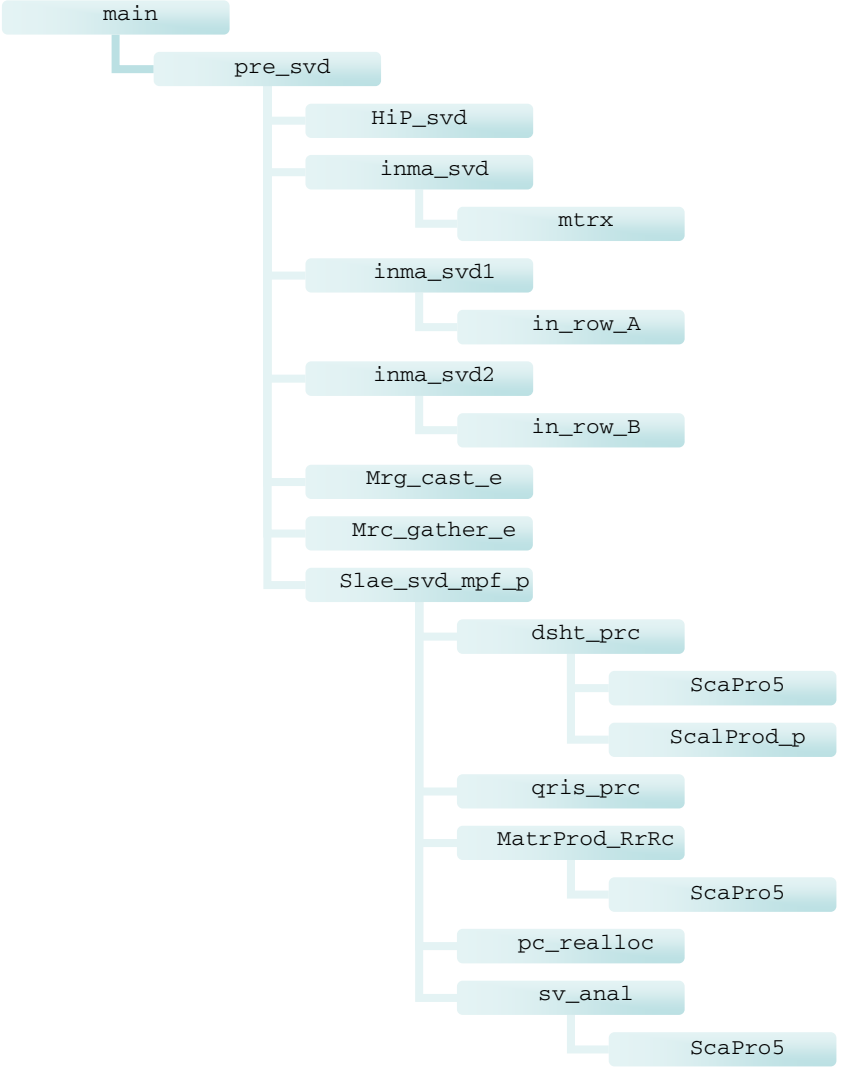

Fig. B5 Functions of programs for solution SLAE by SVD method

6.5. Parallel MPI-program for solution SLAE by singular value decomposition using functions of GMP…….....Subsec.8.2.2(p.178)

- Slae_svd_mpf_p.c

- funk_GMP.c
- Slae_svd_p_test.c
- A41.c
- B41.c
- mpi_gmp.h
- svd.h

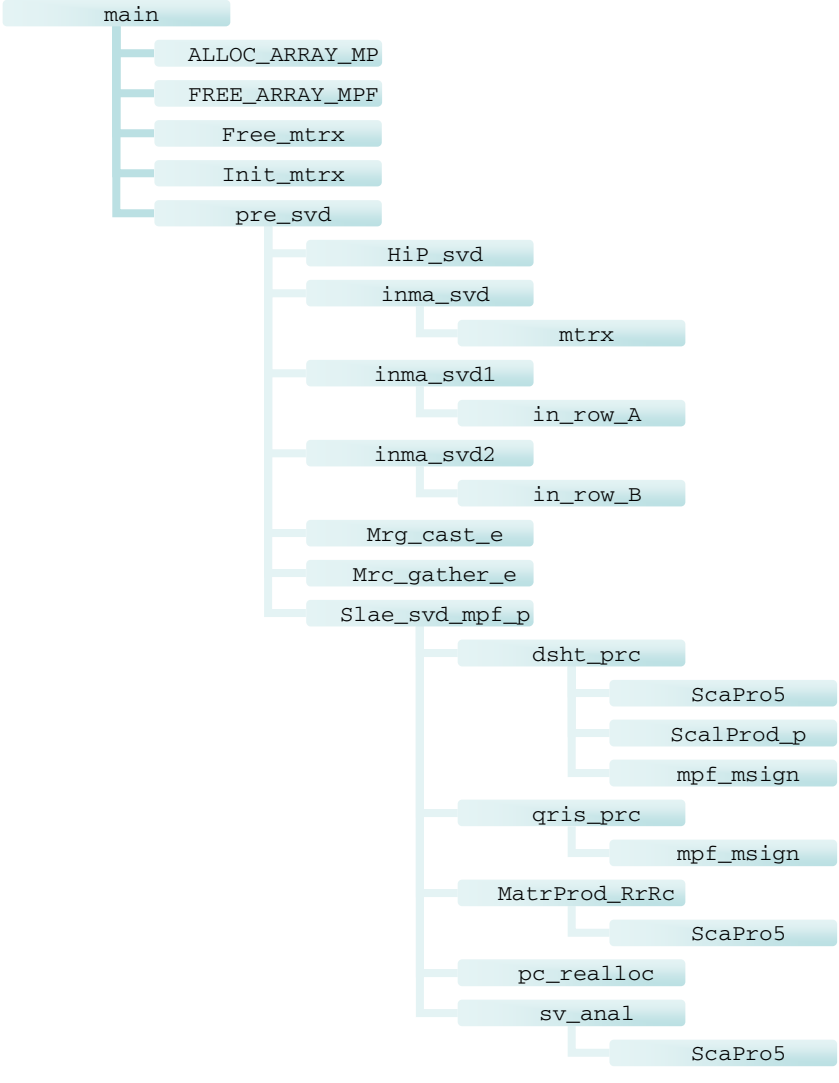

Fig. B6 Functions of programs for solution SLAE by SVD method with GMP

Author Index

Subject Index