Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Iterative Methods for Solving Linear Systems

Laboratory Work
Preconditioned Conjugate Gradient Method implementation

Supported by Intel

K.A. Barkalov,
Software Department

Contents

Purposes of work

- Demonstrate practical implementation of the conjugate gradient method for symmetric sparse matrices using methods of computational error reduction

Objectives of work

- ☐ Study of the conjugate gradient method for sparse matrices
- ☐ Development of the sequential method implementation. Convergence analysis
- ☐ Use of ILU(0)-preconditioner to improve the method convergence. Convergence analysis
- ☐ Use of extended precision numbers to reduce the method computational error (by the example of MPFR library).
- ☐ Analysis of method convergence using extended precision numbers

Objectives of work

- Development of the parallel method version based on OpenMP
- Parallel implementation scalability analysis

Test infrastructure

CPU	Two Intel Xeon E5520 (2.27 GHz) processors
RAM	16 Gb
OS	Microsoft Windows 7
Framework	Microsoft Visual Studio 2008
Compiler, profiler, debugger	Intel Parallel Studio XE 2011
Libraries	Intel® Threading Building Blocks 3.0 for Windows, Update 3 (part of Intel® Parallel Studio XE 2011)

Problem statement (1)

- Let us consider a system of *n* linear equations like

$$Ax=b$$

- $A=(a_{ij})$ is a $n \times n$ real matrix; b and x are vectors consisting of n elements; let the exact system solution be x^*.
- *An iterative method* generates a sequence of vectors $x(s) \in R^m$, $s=0,1,2,\ldots,$ where $x^{(s)}$ is an approximate system solution.
- An iterative method is *convergent* if

$$\forall\ x^{(0)} \in R^m \quad \lim_{s \to \infty} \left\| x^{(s)} - x^* \right\| = 0$$

- *Which iterative method stop criteria do you know?*

Problem Statement (2)

- Iterative method stop criteria: accuracy and number of iterations
  - Stop, if $\|x^{(s)}-x^{(s-1)}\| \le \varepsilon_1$. In this case, $\varepsilon_2 = \|x^{(s)}-x^{(s-1)}\|$ is the attainable method accuracy.
  - Stop, if $\|r^{(s)}\| = \|Ax^{(s)} - b\| \le \varepsilon_1$. In this case, $\varepsilon_2 = \|r^{(s)}\|$ is the attainable method accuracy.
  - Stop, if $s=N$. $x^{(N)}$ is understood as an obtained solution. The maximum number of iterations N is predefined.
- Let us assume that the matrix A is real symmetric positive definite.

CONJUGATE GRADIENT METHOD (CG)

Idea of the method

- The idea of the conjugate gradients method is that solving the linear system $Ax = b$ where *A* is a SPD matrix is equivalent to solving the function minimization problem

$$F(x) = \frac{1}{2}(Ax, x) - (b, x)$$

- This is true if

$$\nabla F(x) = Ax - b = 0$$

Conjugate gradient method algorithm (1)

- Preliminary step: computation of the initial residual vector r_0 and direction vector p_0:

$$r_0 = p_0 = b - Ax_0$$

- Further approximations ($i=0, 1, 2, \ldots, n-1$) are determined using the formulae:

$$\alpha_i = \frac{(r_i, r_i)}{(Ap_i, p_i)}$$

$$x_{i+1} = x_i + \alpha_i p_i$$

$$r_{i+1} = r_i - \alpha_i Ap_i$$

$$\beta_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)}$$

$$p_{i+1} = r_i + \beta_i p_i$$

Conjugate gradient method algorithm (2)

- $r_i = b - Ax_i$ is the residual of the i^{th} approximation,
- β_i is a coefficient corresponding to fulfillment of direction conjugacy condition

$$(Ap_i, p_{i-1}) = 0$$

- α_i is the solution of the function F minimization problem in the direction p_i

$$\alpha_i = \arg\min_{\alpha} F(x_i + \alpha p_i)$$

SEQUENTIAL SOFTWARE IMPLEMENTATION

Project creation (1)

- Create the 01_CG project to implement the conjugate gradient method for a sparse matrix
- Include the following files in the project:
  - **types.h** – description of the involved data structures and constant values
  - **readMatrix.h, readMatrix.c** – declaration and implementation of matrix reading functions from a .mtx file
  - **memoryWork.h, memoryWork.c** – declaration and implementation of functions working with memory
  - **matrixOperations.h, matrixOperations.c** – declaration and implementation of functions for matrix-vector operations with sparse matrices

Project creation (2)

- **CGSolver.h, CGSolver.c** – declaration of functions that solve linear systems using the conjugate gradient method
- **main.c** – call of the conjugate gradient method and running serial experiments.

Data types

- ☐ Parametrize the floating point data type (**types.h**)

```
#ifdef F_PRECISION
  #define FLOAT_TYPE float
  #define tsqrt sqrtf
  #define tfabs fabsf
#else
  #ifdef D_PRECISION
    #define FLOAT_TYPE double
    #define tsqrt sqrt
    #define tfabs fabs
  #else
    #ifdef LD_PRECISION
      #define FLOAT_TYPE double
      #define tsqrt sqrtl
      #define tfabs fabsl
    #endif
  #endif
#endif
```

Data structures

- For sparse matrix representation, use the CRS (Column Row Storage) format.
- From **types.h**, declare the structure **CrsMatrix:**

```
typedef struct CrsMatrix
{
  int N;                 // matrix dimension (N x N)
  int NZ;                // number of nonzeroes
  FLOAT_TYPE* Value;     // values array (dimension NZ)
  int* Col;              // column numbers array (NZ)
  int* RowIndex;         // row indices array (dimension N +1)
}
crsMatrix;
```

Elementary function

- Implement the `main()` function as follows:
  1. Enter the system parameters (name of the file with the matrix, required accuracy, maximum iterations number, name of the file for result output).
  2. Reading the system matrix in the coordinate format.
  3. Set the exact value $x_i = 1,\ i = 1, \ldots, n$, compute b.
  4. Call the function for solving linear systems by the CG method, compute the achieved residual r norm.
  5. Display the results or save them in a file.
  6. Release the allocated dynamic memory.

Auxiliary functions (1)

- Functions working with memory (**memoryWork.h, memoryWork.c**):
  - **AllocMatrix()** – memory allocation to store a matrix in the CRS format. Input: dimension of the matrix **n**, number of nonzeroes **nz.** Output: pointers to the dedicated arrays **column, row, val** to store the matrix. The function returns the error code.

```
int AllocMatrix(int n, int nz, int** column, int** row,
FLOAT_TYPE** val);
```

  - **FreeMatrix()** – memory release from the matrix in the CRS format stored in the arrays **column, row, val.**

```
int FreeMatrix(int** column, int** row, FLOAT_TYPE**
val);
```

Auxiliary functions (1)

- ☐ Matrix read function (**readMatrix.h, readMatrix.c**):
  - **ReadMatrixFromFile()** – reading the matrix from the file in the coordinate format, saving it in the CRS format. Input: name **matrixName** of the file containing a matrix. Output: matrix dimension **n**, pointers to initialized arrays **column, row, val**, describing the matrix. The function returns the error code.

```
int ReadMatrixFromFile(char* matrixName, int* n, int**
column, int** row, FLOAT_TYPE** val);
```

Auxiliary functions (2)

- Matrix-vector operations (**matrixOperations.h, matrixOperations.c**):
  - **MultiplicateVV() –** vector scalar product computation Input: pointers to the vectors **a, b**, their **size**. The function returns a scalar product.

```
FLOAT_TYPE MultiplicateVV(FLOAT_TYPE* a,
                          FLOAT_TYPE * b, int size);
```

Auxiliary functions (3)

- **MultiplicateMV() –** matrix-vector product computation. Input: pointers to the matrix **M** in the CRS format, vector **v**. Output: pointer to the product **result**

```
void MultiplicateMV(crsMatrix* M, FLOAT_TYPE* v,
                                FLOAT_TYPE* result);
```

- **InitializeVector()** – filling the vector with equal values. Input: pointers to the **vector**, its **size** and **value.** Output: initialized **vector.**

```
void InitializeVector(FLOAT_TYPE* vector, FLOAT_TYPE
  value, int size);
```

Auxiliary functions (4)

- **ComputeResidualNorm() –** residual norm computation for the system with obtained solution. Input: pointers to the matrix A, right side vector b, solution vector res. Output: obtained residual norm value.

```
FLOAT_TYPE ComputeResidualNorm(crsMatrix* A,
  FLOAT_TYPE* res, FLOAT_TYPE* b);
```

Conjugate gradient method implementation (1)

- Implement the **CGMethod()** function for solving systems using the conjugate gradient method (**CGSolver.h, CGSolver.c**).

```
int CGMethod(crsMatrix* A, FLOAT_TYPE* b, FLOAT_TYPE*
   x0, FLOAT_TYPE eps, FLOAT_TYPE* result, int* count,
   int maxIter);
```

- Function input: pointers to the system matrix **A** in the CRS format, right side vector **b**, initial approximation **x0**, required method accuracy **eps**, maximum allowed number of iterations **maxIter**. Function output: pointer to the computed approximate solution **result**, number of performed iterations **count**.

Example for the conjugate gradient method (2)

- Stop criterion: maximum allowed number of iterations **maxIter** or required solution accuracy $||r^{(s)}||/||b||=||Ax^{(s)}-b||/||b||\le\varepsilon$

```
void CGMethod(crsMatrix* A, FLOAT_TYPE* b, FLOAT_TYPE* x0,
   FLOAT_TYPE eps, FLOAT_TYPE* result, int* count, int maxIter)
{
  int size = A->N;
  FLOAT_TYPE* rPrev // rPrev - current approximation residual
  FLOAT_TYPE* rNext // rNext - next approximation residual
  FLOAT_TYPE* p     // p - direction vector
  // Auxiliary variables:
  FLOAT_TYPE* y     // y - vector product A*x0
  FLOAT_TYPE* Ap    // Ap - vector product A*p
  // check - current method accuracy, norm - vector b norm,
  // beta, alpha - computing formula coefficients
  FLOAT_TYPE beta, alpha, check, norm;
  ...
```

Example for the conjugate gradient method (3)

```
...
// Method initialization
*count = 0;
norm = tsqrt(MultiplicateVV(b, b, size));
// residual calculation with the first approximation
MultiplicateMV(A, x0, y);
for(j = 0; j < size; j++)
  rPrev[j] = b[j] - y[j];
// p initialization, result
memcpy(p, rPrev, size * sizeof(FLOAT_TYPE));
memcpy(result, x0, size * sizeof(FLOAT_TYPE));
...
```

Example for the conjugate gradient method (4)

```
// performance of method iterations
do
{
  (*count)++;
  MultiplicateMV(A, p, Ap);
  alpha = MultiplicateVV(rPrev, rPrev, size) /
          MultiplicateVV(p, Ap, size);
  for (j = 0; j < size; j++)
  {
    result[j] += alpha * p[j];
    rNext[j] = rPrev[j] - alpha * Ap[j];
  }
  beta = MultiplicateVV(rNext, rNext, size) /
         MultiplicateVV(rPrev, rPrev, size);
  check = tsqrt(MultiplicateVV(rNext, rNext, size)) / norm;

  for (j = 0; j < size; j++)
    p[j] = beta * p[j] + rNext[j];
  ...
```

Example for the conjugate gradient method (5)

```
    ...
      swap = rNext;
      rNext = rPrev;
      rPrev = swap;
    }
    while ((check > eps) && (*(count) <= maxIter));
    ...
  }
```

Experiments. Test matrices

- The University of Florida Sparse Matrix Collection
  http://www.cise.ufl.edu/research/sparse/matrices/
- Parameters of the matrices involved

Name	*n*	*nz*	*Condition number*
bcsstk01	48	400	882 336
bcsstk05	153	2 423	14 281.1
bcsstk10	1086	22 070	524 225
bcsstk13	2003	83 883	1.06e+10
parabolic_fem	525 825	3 674 625	2.11e+10
tmt_sym	726 713	5 080 961	---

Experimental results (1)

- ☐ Number of method iterations depending on the required accuracy. Single-precision numbers.

Matrix name	Order	Required accuracy	
		10^{-4}	10^{-7}
bcsstk01	48	26	239
bcsstk05	153	185	317
bcsstk10	1086	565	2538
bcsstk13	2003	446	20031
parabolic_fem	525 825	1143	2473
tmt_sym	726 713	22637	54352

Experimental results (2)

- Number of method iterations depending on the required accuracy.Double-precision numbers.

Matrix name	Order	Required accuracy		
		10^{-4}	10^{-8}	10^{-15}
bcsstk01	48	24	125	125
bcsstk05	153	153	272	272
bcsstk10	1086	529	2335	2335
bcsstk13	2003	332	20031	20031
parabolic_fem	525 825	1690	1690	1690
tmt_sym	726 713	5356	5356	5356

Experimental results (3)

- Number of method iterations depending on the floating point number type. Required accuracy 10^{-7}.

Matrix	Order	Accuracy of floating point numbers		
		float	double	long double
bcsstk01	48	239	125	112
bcsstk05	153	317	272	258
bcsstk10	1086	2538	2335	2291
bcsstk13	2003	20031	20031	20031
parabolic_fem	525 825	2473	1690	1690
tmt_sym	726 713	54352	5356	5356

Experimental results. Conclusions

- If the required accuracy increases from 10^{-4} to 10^{-7}, the number of method iterations grows 1.7 to 4.4 times for well-conditioned matrices and up to 10 times for ill-conditioned ones.
- An exception to this rule is the matrix bcsstk13, for which the method is divergent if $\varepsilon > 10^{-4}$, i. e. the maximum number of operations is performed.
- In theory, the number of method iterations does not exceed the matrix order. In practice, it can exceed the matrix order by several times. Why does this happen?
  - This is due to computational error accumulation in the course of arithmetic operation and rounding errors.

PRECONDITIONED GENERALIZED MINIMAL RESIDUAL METHOD

Idea of preconditioning

- $\mu_A=\lambda_{max}/\lambda_{min}$ is the spectral number.
  - $\mu_A \approx 1$ – the method converges quickly (A is well-conditioned)
  - $\mu_A >> 1$ – the method converges slowly (A is ill-conditioned)
- The idea of preconditioning lies in converting the ill-conditioned system $Ax=$□i□b□
  to a well-conditioned one

$$M^{-1}Ax=M^{-1}b.$$

  Here, M is a preconditioner.
- $M^{-1}A$ is not computed explicitly
- Corrective steps allowing for preconditioning are added to the iterative method.

Preconditioned conjugate gradient method algorithm (1)

- Preliminary step: computation of the initial residual vector r_0 and direction vector p_0:

$$r_0 = b - Ax_0 \qquad Mz_0 = r_0 \qquad p_0 = z_0$$

- Further approximations ($i = 0, 1, 2, \dots, n-1$) are determined using the formulae:

$$\alpha_i = \frac{(r_i, r_i)}{(Ap_i, p_i)} \qquad x_{i+1} = x_i + \alpha_i p_i$$

$$r_{i+1} = r_i - \alpha_i Ap_i \qquad z_{i+1} = M^{-1} r_{i+1}$$

$$\beta_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)} \qquad p_{i+1} = r_i + \beta_i p_i$$

Conjugate gradient method algorithm (2)

- Use the ILU(0) type preconditioner.

$$M = LU$$

- Then, instead of computing the matrix M^{-1} solve the system

$$Mz_{i+1} = r_{i+1}$$

- I. e. move to triangular systems

$$Ly = r_{i+1}, Uz_{i+1} = y$$

PRECONDITIONED METHOD SOFTWARE IMPLEMENTATION

Project creation (1)

- Create the **02_PCGMethod** project with the preconditioned conjugate gradient method implementation
- Copy the files to the project from **01_CGMethod**
- Add the project with the ILU(0)-preconditioner from the respective laboratory work.
  - **ilu0.h, ilu0.c** – declaring and implementing functions of the ILU(0)-preconditioner

Project creation (2)

- Set up the project **02_PCGMethod** properties: from **ConfigurationProperties**, select **C\C++** → **General** →**Additional Include Directories** tab, enter **..\ILU** in the field **Additional Input Directories.**
- For the ILU project, indicate 01_CGMethod in the property field.
- Connect Intel MKL to the project.
- Set up **Project Dependencies**: the **ILU** project depends on **01_CGMethod** and **02_PCGMethod** depends on **ILU**.

Auxiliary functions (1)

- Matrix-vector operations (**matrixOperations.h, matrixOperations.c**):
  - **GaussSolve()** – solution of linear systems with a dense upper of lower matrix. Input: pointers to the matrix **A**, system dimension **size**, right side vector **b**, number **isLowTriangle** denoting matrix type, i.e. upper triangle (0) or lower triangle (1). Output: pointer to the system solution **result.**

```
void GaussSolve(crsMatrix* A, int size, int isLowTriangle,
                FLOAT_TYPE* b, FLOAT_TYPE* result);
```

  - This function is required when the preconditioned matrices L and U are used.

Auxiliary functions (2)

- **LUmatrixSeparation()** – forming the multiplier matrices L and U based on the ILU-preconditioner matrix. As an input, the function receives the **ilu** preconditioner matrix and **uptr** array containing diagonal element indices in the Col array of the **ilu** matrix. The function returns structures of the matrices **L** and **U.**

```
void LUmatrixSeparation(crsMatrix ilu, int *uptr, crsMatrix
                        &L, crsMatrix &U);
```

Auxiliary functions (3)

- **UpToFull()** – restoring the lower triangle of the symmetric matrix for which only the upper triangle was stored. Input: pointer to the matrix **A** that contains only the upper triangle elements. Output: pointer to the full matrix **Full (Full = A + A^T).**

```
void UpToFull(crsMatrix* A, crsMatrix* Full);
```

Elementary function

- ❑ Preconditioner call:

```
#include "ilu0.h "
int main( int argc, char* argv[])
{
  ...
  //4. Computing the matrices L and U
  UpToFull(&A, FullA);
  // the matrix LU will be obtained within the FullA matrix
  ilu0(size, FullA->Value, FullA->Col, FullA->RowIndex, FullA->Value,
   uptr);
  LUmatrixSeparation(FullA, uptr, L, U);

  //5. Solving linear systems using the conjugate gradient method
  CGMethod(&A, &L, &U, b, x0, accuracy, res, &iterCount, maxIterNum);
  ...
}
```

Preconditioned method implementation (1)

```
void CGMethod(crsMatrix* A, crsMatrix* L, crsMatrix* U, FLOAT_TYPE* b,
   FLOAT_TYPE* x0, FLOAT_TYPE eps, FLOAT_TYPE* result, int* count, int
   maxIter)
{ ...
  FLOAT_TYPE* zPrev; // zPrev - vector z for the current approximation
  FLOAT_TYPE* zNext; // zNext -vector z for the next approximation
  FLOAT_TYPE* y;     // y - intermediate vector

  // Method initialization
  ...
  for(j = 0; j < size; j++)
    rPrev[j] = b[j] - y[j];
  // Computation of the residual vector r
  // and intermediate vector z from the preconditioner LU
  GaussSolve(L, size, 1, rPrev, y);
  GaussSolve(U, size, 0, y, zPrev);
  ...
```

Preconditioned method implementation (2)

```
   ...
   // performance of method iterations
   do
   {
     ...
     for (j = 0; j < size; j++)
     {
       result[j] += alpha * p[j];
       rNext[j] = rPrev[j] - alpha * Ap[j];
     }

     // Computation of the residual vector r and intermediate vector z
     // from the preconditioner LU
     GaussSolve(L, size, 1, rNext, y);
     GaussSolve(U, size, 0, y, zNext);
     ...
     beta = MultiplicateVV(zNext, rNext, size) /
            MultiplicateVV(zPrev, rPrev, size);
     ...
 }
```

Experimental Results

- Comparison of the number of method iterations with and without preconditioner for double-precision numbers

Matrix name	Order	Accuracy 10^{-7}		Accuracy 10^{-15}	
		Without preconditioner	With preconditioner	Without preconditioner	With preconditioner
bcsstk01	48	125	15	125	23
bcsstk05	153	272	35	272	48
bcsstk10	1086	2335	149	2335	264
bcsstk13	2003	20031	10016	20031	10016
parabolic_fem	525 825	1690	1045	1690	1964
tmt_sym	726 713	5356	1210	5356	2000

Experimental results. Conclusions

- The use of a preconditioner for smaller matrices enabled reduction of the number of iteration by 5 to 9 times when double precision numbers were used or 7 to 15 times in case of single precision numbers.
- For larger matrices, the number of iterations reduced to two times.
- The use of a preconditioner for the matrix bcsstk13 resulted in solving the system.

IMPLEMENTATION BASED ON EXTENDED PRECISION NUMBERS

MPFR library

- MPFR (http://www.mpfr.org/) is a free library for multiple-precision floating-point computations with correct rounding. MPFR is based on the GMP library.
- It has been developed to be used on Linux. There are a number of ways to assemble this library on Windows [6].

MPFR library assembly (1)

❑ Assemble the library based on a Visual Studio 2010 project [8]. For this purpose:

1. Download the latest library version from http://www.mpfr.org/mpfr-current/#download.
2. Download the MPIR auxiliary library (to work with long integer numbers) http://mpir.org/#release
3. Download the MPFR integration project from MPIR for VS 2010 http://gladman.plushost.co.uk/oldsite/computing/mpfr.svn.build.vc10.zip
4. Unpack MPFR and MPIR archives and the project.

MPFR library assembly (2)

5. In the current archive directory, create the mpfr and mpir folders. Create the build.vc10 directory in the mpir folder, and build.vc10 and src. subdirectories in the mpfr folder. This will result in the following system of directories:

   …\ mpir \ build.vc10; …\ mpfr \ build.vc10; …\ mpfr \ src
6. Assemble the MPIR library: assemble the lib_mpir_gc in the Release configuration in mpir.sln. Upon project assembly, the lib folder will appear in the mpir folder.

MPFR library assembly (3)

7. Assemble the MPFR library Correct the lib_mpfr.sln project as required:
   - In **C/C++ -> Preprocessor -> Preprocessor Definitions,** leave one of two macros**, _WIN32** or **_WIN64**, depending on the future application platform.
   - Delete **fpif.c, get_float128.c, rndna.c** and **set_float128.c** from the project.

Project creation

- Create the **03_GMethod_MPFR** project to implement the conjugate gradient method based on the extended precision numbers from MPFR.
- Copy the filed from **01_CGMethod** to the project.
- Connect the MPRF library to the project.
  - From **Configuration Properties,** select **Linker→Input** and enter **mpfr.lib, mpir.lib** as filenames for the assembled libraries; select **Linker→General** and enter in the **Additional Library Directories** the path to the folder that stores **mpfr.lib** and **mpir.lib.**

Data types (1)

- To initialize extended precision numbers, declare from **types.h** the **PRECISION** invariable that stores the size of involved numbers in bits.
- For the floating point number type **FLOAT_TYPE,** indicate the **mpfr_t** extended precision number type from the MPFR library. To use the library in the implementation, connect the **gmp.h, mpfr.h** header files.

```
#include "gmp.h"
#include "mpfr.h"

// bit length of the number
#define PRECISION 128
// type of numbers from MPFR
#define FLOAT_TYPE mpfr_t
```

Basic functions of the MPFR library (1)

- Working with numbers in MPFR has the following peculiarity: variables of the **mpfr_t** type must be initialized before any value is obtained.
- **mpfr_init()** – numerical variable initialization:

```
void mpfr_init(mpfr_t x);
```

  - In this case, the value **x** is equal to **NaN** with a default precision.

- **mpfr_set_default_prec()** – setting the variable precision using any integer value prec from **MPFR_PREC_MIN** through **MPFR_PREC_MAX** bit.

```
void mpfr_set_default_prec(mp_prec_t prec);
```

Basic functions of the MPFR library (2)

- **mpfr_set_d()** – setting the extended precision variable **x**, equal to the doublr-precision variable **val**:

```
int mpfr_set_d (mpfr_t x, double val, mp_rnd_t rnd);
```

- **mpfr_init_set_d()** – variable initialization and assignment of a double-precision value at the same time:

```
int mpfr_init_set_d(mpfr_t x, double val, mp_rnd_t rnd);
```

- **mpfr_set_str()** – setting the variable **x** from the row **s** containing number with a **base:**

```
int mpfr_set_str(mpfr_t x, const char *s, int base,
                 mp_rnd_t rnd);
```

Basic functions of the MPFR library (3)

- **rnd**, the last parameter of the mentioned function, means the type of rounding. The MPFR library offers five rounding modules. In our case, use **MPFR_RNDD**, which ensures rounding towards negative infinity.
- **mpfr_clear(), mpfr_clears()** – clearing extended precision variables from the memory:

```
void mpfr_clear(mpfr_t x);
void mpfr_clears(mpfr_t x, ...);
```

  - **mpfr_clears()** – emptying memory for lists of variables such as mpfr_t ending in a nil pointer

Basic functions of the MPFR library (4)

- **mpfr_get_str()** – convert the **mpfr_t** type variable x with a **base** into a string with **n** significant characters and **rnd** rounding module:

```
char* mpfr_get_str(char* str, mp_exp_t* expptr, int base,
                   size_t n, mpfr_t x, mp_rnd_t rnd)
```

  - The function returns the number in an exponent representation: **str** contains all its characters, **expptr** shows the number exponent.

- Modify the **main()** function using the MPFR library functions.

Basic functions of the MPFR library (5)

- Mathematic functions of the MPFR library:

```
// addition: rop = po1 + po2
int mpfr_add(mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd);
// subtraction: rop = po1 - po2
int mpfr_sub(mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd);
// multiplication: rop = po1 * po2
int mpfr_mul(mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd);
// division: rop = po1 / po2
int mpfr_div(mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd);
// square root computation
int mpfr_sqrt(mpfr_t rop, mpfr_t op, mp_rnd_t rnd);
// comparison of numbers. Returns a positive number if op1 > op2,
// a negative number if op1 < op2, zero, if op1 = op2.
int mpfr_cmp(mpfr_t op1, mpfr_t op2);
```

Auxiliary functions (1)

- Modify implementation of the auxiliary functions (**matrixOperations.h, matrixOperations.c**)
- **InitializeVector()** – initialization of vector components by **value**

```
void InitializeVector(FLOAT_TYPE* vector, int size, double value)
{
  int i;
  for (i = 0; i < size; i++)
    mpfr_init_set_d(vector[i], value, MPFR_RNDD);
}
```

Auxiliary functions (2)

- **InitializeEmptyVector()** – vector initialization without initial values

```
void InitializeEmptyVector(FLOAT_TYPE* vector, int size)
{
  int i;
  for (i = 0; i < size; i++)
    mpfr_init(vector[i]);
}
```

- **ReleaseVector() -** vector release

```
void ReleaseVector(FLOAT_TYPE* vector, int size)
{
  int i;
  for (i = 0; i < size; i++)
    mpfr_clear(vector[i]);
  free(vector);
}
```

Auxiliary functions (3)

- **MultiplicateVV() –** scalar product computation

```
void MultiplicateVV(FLOAT_TYPE* a, FLOAT_TYPE* b, int size,
                    FLOAT_TYPE * res)
{
  int i;
  // product is the auxiliary variable to store
  // the product of components
  FLOAT_TYPE product;
  mpfr_init(product);
  mpfr_set_d(*res, 0, MPFR_RNDD);

  for (i = 0; i < size; i++)
  {
    mpfr_mul(product, a[i], b[i], MPFR_RNDD);
    mpfr_add(*res, *res, product, MPFR_RNDD);
  }
  mpfr_clear(product);
}
```

Auxiliary functions (4)

- **MultiplicateMV() –** matrix-vector product computation

```
void MultiplicateMV(crsMatrix* M, FLOAT_TYPE* v, FLOAT_TYPE* result)
{ int j, i; FLOAT_TYPE product;
  mpfr_init(product); InitializeVector(result, M->N, 0);
  for (i = 0; i < M->N; i++)
  {
    for (j = M->RowIndex[i] + 1; j < M->RowIndex[i + 1]; j++)
    {
      mpfr_mul(product, M->Value[j], v[M->Col[j]], MPFR_RNDD);
      mpfr_add(result[i], result[i], product, MPFR_RNDD);
      mpfr_mul(product, M->Value[j], v[i], MPFR_RNDD);
      mpfr_add(result[M->Col[j]], result[M->Col[j]], product,
                 MPFR_RNDD);
    }
    mpfr_mul(product, M->Value[M->RowIndex[i]],
                 v[M->Col[M->RowIndex[i]]], MPFR_RNDD);
    mpfr_add(result[i], result[i], product, MPFR_RNDD);
  }
  mpfr_clear(product);
}
```

Auxiliary functions (5)

- **ComputeResidualNorm() –** system residual norm computation

```
void ComputeResidualNorm(crsMatrix* A, FLOAT_TYPE* res, FLOAT_TYPE* b,
   int size, FLOAT_TYPE* resNorm)
{
  int i;
  FLOAT_TYPE* err; //auxiliary product array
   //A*res
  FLOAT_TYPE scalarProduct; // auxiliary variable to store
            // the scalar product
  err = (FLOAT_TYPE *) malloc (sizeof(FLOAT_TYPE) * size);
  InitializeEmptyVector(err, size);
  MultiplicateMV(A, res, err);
  for (i = 0; i < size; i++)
    mpfr_sub(err[i], err[i], b[i], MPFR_RNDD);

  MultiplicateVV(err, err, size, &scalarProduct);
  mpfr_sqrt(*resNorm, scalarProduct, MPFR_RNDD);

  ReleaseVector(err, size);
}
```

Conjugate gradient method implementation (1)

- ☐ Modify the function **CGMethod():**

```
void CGMethod(crsMatrix* A, FLOAT_TYPE* b, FLOAT_TYPE* x0,
   FLOAT_TYPE eps, FLOAT_TYPE* result, int* count, int maxIter)
{
  ...
  // auxiliary variables
  FLOAT_TYPE numerator, denominator, product;

  // initialization of variables: alpha, check, beta, numerator,
  //denominator, product by the function mpfr_init()

  // initialization of vectors rPrev, rNext, y, p, Ap by the function
  //InitializeEmptyVector()

  // compute the norm b
  MultiplicateVV(b, b, size, &product);
  mpfr_sqrt(norm, product, MPFR_RNDD);
  ...
```

Example for the conjugate gradient method (2)

```
// compute the first residual rPrev
MultiplicateMV(A, x0, y);

for(j = 0; j < size; j++)
{
  mpfr_sub(rPrev[j], b[j], y[j], MPFR_RNDD);
  mpfr_set(p[j], rPrev[j], MPFR_RNDD);
  mpfr_set(result[j], x0[j], MPFR_RNDD);
}

// performance of method iterations
do
{
  (*count)++;
  MultiplicateMV(A, p, Ap);
  ...
```

Example for the conjugate gradient method (3)

```
// compute alpha
MultiplicateVV(rPrev, rPrev, size, &numerator);
MultiplicateVV(p, Ap, size, &denominator);
mpfr_div(alpha, numerator, denominator, MPFR_RNDD);

for (j = 0; j < size; j++)
{
  // compute the next approximation of the result
  mpfr_mul(product, alpha, p[j], MPFR_RNDD);
  mpfr_add(result[j], result[j], product, MPFR_RNDD);

  // compute rNext
  mpfr_mul(product, alpha,  Ap[j], MPFR_RNDD);
  mpfr_sub(rNext[j], rPrev[j], product, MPFR_RNDD);
}

// compute the residual norm
MultiplicateVV(rNext, rNext, size, &product);
mpfr_sqrt(product, product, MPFR_RNDD);
mpfr_div(check, product, norm, MPFR_RNDD);
```

Example for the conjugate gradient method (4)

```
    // compute beta
    MultiplicateVV(rNext, rNext, size, &numerator);
    MultiplicateVV(rPrev, rPrev, size, &denominator);
    mpfr_div(beta, numerator, denominator, MPFR_RNDD);

    // compute p
    for (j = 0; j < size; j++)
    {
      mpfr_mul(p[j], beta, p[j], MPFR_RNDD);
      mpfr_add(p[j], rNext[j], p[j], MPFR_RNDD);
    }
    swap = rNext;
    rNext = rPrev;
    rPrev = swap;
  }
  while ((mpfr_cmp(check, eps) > 0) && (*(count) <= maxIter));

  // empty the memory allocated to variables and vectors
  // by the mpfr_clear(), ReleaseVector() functions
}
```

Experimental Results

- Number of operations for the conjugate gradient method depending on the floating point numbers type. Required accuracy is 10^{-7}.

Matrix name	Order	Accuracy of floating point numbers				
		float	double	long double	mpfr_t 128 bit	mpfr_t 512 bit
bcsstk01	48	239	125	112	77	48
bcsstk05	153	317	272	258	213	152
bcsstk10	1086	2538	2335	2291	2009	1325
bcsstk13	2003	20031	20031	20031	17151	5651
parabolic_fem	525 825	2473	1690	1690	1690	1690
tmt_sym	726 713	54352	5356	5356	4994	4710

Experimental results. Conclusions

- The use of extended precision numbers makes it possible to attain the required accuracy within the number of iterations close to the matrix order. This is in line with theoretical convergence estimate.
- The use of single-precision and double-precision numbers for the ill-conditioned matrix bcsstk13 led to the method divergence; an increased precision resulted in a solution.

PARALLEL SOFTWARE IMPLEMENTATION

Project creation

- Create the **04_GMethod_OMP** project for parallel implementation of the conjugate gradient method
- Copy the files to the project from **01_CGMethod**
- Set up the project to support OpenMP.

- Add the number of OpenMP flows to the parameter list. Set this value in the main() function.

- *How can one parallelize the iterative algorithm?*

Parallel implementation (1)

- ❑ Parallelize matrix-vector operations, i.e. scalar product and matrix-vector product computation.

```
FLOAT_TYPE MultiplicateVV(FLOAT_TYPE* a, FLOAT_TYPE* b, int size)
{
  FLOAT_TYPE result = 0;
  int i;

  #pragma omp parallel for reduction(+:result)
  for (i = 0; i < size; i++)
  {
    result += a[i] * b[i];
  }

  return result;
}
```

Parallel implementation (2)

```
void MultiplicateMV(crsMatrix* M, FLOAT_TYPE* v, FLOAT_TYPE* result)
{
...
#pragma omp parallel for private(i, j)
  for (i = 0; i < M->N; i++)
  {
    for (j = M->RowIndex[i] + 1; j < M->RowIndex[i + 1]; j++)
    {
      #pragma omp atomic
      result[i] += M->Value[j] * v[M->Col[j]];
      #pragma omp atomic
      result[M->Col[j]] += M->Value[j] * v[i];
    }
    #pragma omp atomic
    result[i] += M->Value[M->RowIndex[i]] * v[M->Col[M->RowIndex[i]]];
  }
}
```

Experimental results (1)

- Parallel version run results with an accuracy of 10^{-7}. T – runtime in seconds, S – acceleration compared with the sequential version.

			2 flows		4 flows		8 flows	
Matrix	Order, n	Sequential version	T	S	T	S	T	S
bcsstk10	1086	0,19	0,39	0,48	0,27	0,70	0,17	1,09
bcsstk13	2003	5,98	13,40	0,45	8,17	0,73	5,75	1,04
parabolic_fem	525 825	42,2	84,32	0,50	63,22	0,67	38,33	1,10
tmt_sym	726 713	203,85	284,39	0,72	171,11	1,19	111,25	1,83

Experimental results (2)

- Parallel version run results with an accuracy of 10^{-15}. T – runtime in seconds, S – acceleration compared with the sequential version.

			2 flows		4 flows		8 flows	
Matrix	Order, n	Sequential version	T	S	T	S	T	S
bcsstk10	1086	0,22	0,90	0,24	0,59	0,37	0,39	0,56
bcsstk13	2003	5,93	13,48	0,44	8,29	0,72	5,85	1,01
parabolic_fem	525 825	39,27	186,88	0,21	124,80	0,31	78,19	0,50
tmt_sym	726 713	191,28	473,57	0,40	282,52	0,68	172,82	1,11

- *Why this parallelization approach does not result in runtime reduction?*

Experimental results Conclusions

- Parallelization of easy operations (e. g. scalar product computation) is inefficient.
- Contingencies for flow operation support is comparable to computational load on these flows in terms of time.

References

1. Gene H. Golub, Charles F. Van Loan. Matrix Computations. The John Hopkins University Press, 1996.
2. J. Dongarra et al. Templates for the solution of linear systems: building blocks for iterative methods. SIAM, 1994.
3. Y. Saad. Iterative Methods for Sparse Linear Systems. SIAM, 2003.
4. Fousse L., Hanrot G., Lefèvre V., Pélissier P., Zimmermann P. MPFR: A multiple-precision binary floating-point library with correct rounding // ACM Trans. Math. Softw. – 2007. – Vol. 33, No. 2. – P. 1–14.

Internet resources

5. GNU MPFR Manual [http://www.mpfr.org/mpfr-current/mpfr.pdf]
6. The GNU MPFR library (discussion, bug reports...). MPFR under Windows [http://permalink.gmane.org/gmane.comp.lib.mpfr.general/819]
7. MPIR – Multiple Precision Integers and Rationals [http://mpir.org/]
8. A Native GMP Port Using Microsoft Visual Studio [http://gladman.plushost.co.uk/oldsite/computing/gmp4win.php]