Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Iterative Methods for Solving Linear Systems

Laboratory Work
Solving Sparse Linear Systems Using the Preconditioned Biconjugate Gradient Method

Supported by Intel

K.A. Barkalov,
Software Department

Contents

- The problem of solving linear system using the biconjugate gradient method
- Consecutive implementation of the biconjugate gradient algorithm
- Biconjugate gradient method convergence analysis
- Preconditioned biconjugate gradient method
- The possibility of parallel algorithm implementation

Introduction (1)

- Let us consider a system of n linear equations like

$$\begin{aligned} a_{11}x_1 &+ a_{12}x_2 &+ \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 &+ a_{22}x_2 &+ \ldots + a_{2n}x_n &= b_2 \\ &\ldots & & \\ a_{n1}x_1 &+ a_{n2}x_2 &+ \ldots + a_{nn}x_n &= b_n \end{aligned}$$

- As a matrix, the system may be represented as follows

$$Ax=b$$

- $A=(a_{ij})$ is a $n \times n$ real matrix; A is a sparse matrix; b and x are vectors consisting of n elements.

Introduction (2)

- Methods of solving linear systems may be classified as *direct* and *iterative*.
- Both types have their advantages and disadvantages.
  - The use of direct methods leads to *system matrix filling* in the course of factorization which may cause inefficient memory usage.
  - The use of iterative methods that do not lead to matrix filling may sometimes lead to a *low convergence rate*.
- The purpose of this laboratory works is to study iterative methods of solving linear systems.

Introduction (3)

- *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.
- *Method convergence* is convergence of the sequence $x^{(s)}$ to the exact system solution from any initial approximation.
- *Convergence rate* is determined by the number of approximations performed by the method until the stop criterion is met.
- In practice, convergence is not the only important thing. In the course of computing, computational error is inevitable. A method is numerically stable if the computational error tends to zero when при уменьшении погрешности вычислений.
- *The convergence and numerical stability of iterative methods are the main issues solved as part of iterative method quality study.*

Introduction (4)

- Most methods converge quickly if the matrix is well-conditioned or has few eigenvalues.
- Otherwise, due to computational error accumulation, a method that converges in theory may diverge in reality.
- To overcome poor matrix conditioning, system *preconditioning* is used, i.e. conversion to a linear system with the same solution and a better matrix by multiplying the system by a special matrix.

Purposes of work

- **The purpose of this laboratory work** is to demonstrate practical implementation of the biconjugate gradient method for linear systems with sparse matrices

Objectives of work

- Study of the biconjugate gradient method for dense matrices
- Modification of the biconjugate gradient method for sparse matrices
- Development of the biconjugate gradient method software implementation for sparse matrices
- Developed method convergence analysis
- Development of the consecutive implementation of the biconjugate gradient method
- Developed method convergence analysis

Test infrastructure

CPU	Two Intel Xeon E5520 processors (4 core, 2.27 GHz)
RAM	16 Gb
OS	Microsoft Windows 7
Framework	Microsoft Visual Studio 2008
Compiler, profiler, debugger	Intel® Parallel Studio XE 2011
Libraries	Intel® Math Kernel Library (within Intel® Parallel Studio XE 2011)

Biconjugate gradient method (1)

- The biconjugate gradient method is a generalization of the conjugate gradient method which is intended for linear systems with some *arbitraty nonsingular* matrix.
- $A^t * A$ is known to be a symmetric positive definite matrix.
- Therefore, it is possible to proceed to solution of a new system equivalent to the initial one:

$$A^t * A\, x = A^t * b$$

- This system can be solved by the *conjugate gradient method*, though it is not easy to do it in practice, as the $A^t * A$ product substantially increases the matrix condition.
- Based on the relation, one can obtain an algorithm free from the disadvantages of the $A^t * A\, x$ system solution.
  - For this purpose, the sequence of residuals and directions from the conjugate gradient method and the respective biconjugates are used.

Biconjugate gradient method (2)

- Biconjugate gradient algorithm

1. Вычислить $r_0 = b - Ax_0$; выбрать $\bar{r}_0$ так, чтобы $(r_0, \bar{r}_0) \neq 0$ (например, выбрать $\bar{r}_0 = r_0$).
2. Положить $p_0 = r_0$, $\bar{p}_0 = \bar{r}_0$
3. for j=0,1,… do
4. $\alpha_j = (r_j, \bar{r}_j)/(Ap_j, \bar{p}_j)$
5. $x_{j+1} = x_j + \alpha_j p_j$
6. $r_{j+1} = r_j - \alpha_j A p_j$
7. $\bar{r}_{j+1} = \bar{r}_j - \alpha_j A^T \bar{p}_j$
8. $\beta_j = (r_{j+1}, \bar{r}_{j+1})/(r_j, \bar{r}_j)$. Если $\beta_j = 0$ или $\| r_{j+1} \| < \varepsilon$ то Стоп.
9. $p_{j+1} = r_{j+1} + \beta_j p_j$
10. $\bar{p}_{j+1} = \bar{r}_{j+1} + \beta_j \bar{p}_j$
11. end j

Consecutive implementation of the biconjugate gradient method

Project creation (1)

- For convenience, let us divide the biconjugate gradient method implementation into several projects. A total of three projects will be required:
  - **parser** is the project containing implementation of functionality that enables reading of systems from files and a number of operations required for memory allocation and data initialization.
  - **routine** is the project containing some mathematic operations such as multiplication of matrices and vectors and checking the solution for correctness.
  - **BiCG** is the project containing the biconjugate gradient method implementation and the program main function.

Project creation (2)

- Create the following set of files in the **parser** project :
  - **readMTX.h, readMTX.cpp** –files to declare and implement functions required for reading matrices from the file
  - **routines.h, routines.cpp** – files to declare and implement auxiliary functions required for reading matrices from the file and print the read data from the file
  - **type.h** – file to declare the involved structures of data and invariables
  - **util.h, util.cpp** – files to declare and implement the set of functions to distinguish, initialize and delete data for CRS matrices.

Project creation (3)

- Create the following set of files in the **routine** project:
  - **sparseMatrixOperation.h, sparseMatrixOperation.cpp** – files to declare and implement sparse operations with CRS matrices.
  - **timer.hpp, timer.cpp** – files to declare and implement time measurement functions.
  - **validation.h, validation.cpp** – files to declare and implement functions that check the obtained solutions for correctness.

Project creation (4)

- Create the following set of files in the **BiCG** project :
  - **BiCG.h, BiCG.cpp** – file to contain various implementations of the biconjugate gradient method.
  - **main.cpp** – file to contain implementation of the main program function.

Project creation (5)

- Determine relationship between the projects.
- **Routine** depends on **parser** and **BiCG** depends on both **routine** and **parser**.

Involved data structures

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

```
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 dimension)
  int* RowIndex;          // row indices array
                          // (dimension N + 1)
} crsMatrix;
```

Main() function (1)

- Build the **main()** function as follows:
  - Reading the command line arguments
  - Reading the system matrix from the file
  - Initialization of variables
    - Memory allocation
    - Setting the right-hand vector
  - Solving linear systems using the biconjugate gradient method
  - Computation of the system residual over the obtained solution.
  - Method operation data output
  - Memory release

Main() function (2)

- Read command line arguments and announce the variables

```
int main(int argc, char ** argv)
{
  // 1. Reading the command line arguments
  char *matrixName;
  ParseArgv(argc, argv, matrixName);

  // declaring the required variables
  crsMatrix readA;  // read matrix
  crsMatrix *matA;  // pointer to the matrix
                    // used for computation
  int typeOfMatrix; // type of the read matrix
  int error;        //   error code returned by the functions
  double diff;      // computation error
  int iter;         // number of performed iterations
  // timer used to measure
  // algorithm part runtime
  Stopwatch *time = createStopwatch();
  int i;
  double *b;        // right-side vector
  double *x;        // desired linear system solution
```

Main() function (3)

- Read the matrix

```
// 2. Read the matrix from the file
 printf("read matrix (%s) \n", matrixName);
 time->start();
 error = ReadMatrixFromFile(matrixName,
   &(readA.N), &(readA.NZ),
   &(readA.Col), &(readA.RowIndex), &(readA.Value),
   &(typeOfMatrix));

 if(error != BICG_OK)
 {
 printf("error read matrix %d\n", error);
 return error;
 }
```

Main() function (4)

```
// if the matrix is symmetric and
// defined only by the upper triangle,
// make it full
if(typeOfMatrix == UPPER_TRIANGULAR)
{
  matA = UpTriangleMatrixToFullSymmetricMatrix(&readA);
  FreeMatrix(readA);
}
else
{
  matA = &readA;
}
time->stop();

printf("read matrix from file time: %f\n",
  time->getElapsed());
```

Main() function (5)

- Initialize the variables

```
// 3. Initialization of variables
// allocate memory to the right-hand vector and
// solve the linear system
x = new double [matA->N];
b = new double [matA->N];

// initialize the right-hand part
for(i = 0; i < matA->N; i++)
{
  b[i] = 1.0;
}
```

Main() function (6)

- In place of the highlighted “call for the function of system solution using BiCG”comment, call for the function with our method implementation will be written in the future.

```
time->reset();
time->start();
// 4. call the function of solving linear systems by the BiCG method
time->stop();
```

Main() function (7)

- In the end, check the solution for correctness and free the allocated memory.

```
// 5. Checking the BiCG method for correctness
diff = diffSolution(*matA, x, b);

// 6. Method operation data output
printf("BiCG time: %f\n", time->getElapsed());
printf("count of iteration: %d\n", iter);
printf("calc error: %f\n", diff);
// 7.  Dynamic memory release
FreeMatrix(*matA);
delete [] b;
delete [] x;
```

Auxiliary functions (1)

- Let us study the auxiliary functions necessary to implement the method.
- Place the functions of memory allocation and release for the matrix storage structure in **util.h** and **util.cpp.**
  - **InitializeMatrix()** – memory allocation to store a matrix in the CRS format.The function inputs are dimension of the matrix **n** and number of nonzeroes **NZ.** The outputs are a link to the structure of the matrix in the CRS format with initialized fields and allocated memory. The function returns the error code.

```
int InitializeMatrix(int N, int NZ, crsMatrix
   &mtx);
```

  - **FreeMatrix()** – memory release from the matrix in the CRS format. The output is a link to the structure that contains the matrix. The function returns the error code.

```
int FreeMatrix(crsMatrix &mtx);
```

Auxiliary functions (2)

- From **parser**, announce the **ReadMatrixFromFile()** function, which reads the matrix from the file in the mtx format and stores it in the SRC format, in **readMTX.h** and implement it in **readMatrix.cpp.** As an input, the function will accept **matrixName** - name of the file containing the matrix. The function outputs are the 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);
```

mtx format

- The mtx file is a matrix in coordinate representation.
  - The file contains such matrix parameters as the number of rows, columns and nonzeroes.
  - Then, row-by-row, the parameters of matrix nonzeroes are listed such as the respective row, column and value.
  - Row and column numbering starts from 1.
  - If the matrix is symmetric, the file can contain only its upper or lower triangle.
  - Comment fields start with %.

Task 1

- Implement the abovementioned functions dealing with matrices in the CRS format and read them from file.

Auxiliary functions (3)

- Implement matrix-vector operations of the biconjugate gradient method.
- From **routine**, declare and implement the functions in **sparseMatrixOperation.h** and **sparseMatrixOperation.cpp**, respectively.

Auxiliary functions (4)

- **MatrixVectorMult() –** matrix-vector product computation. As inputs, the function receives pointers to the matrix **A** in the CRS format and vector **b**. Function output is the pointer to their product **x.** As the result, the function returns the error code.

```
int MatrixVectorMult(crsMatrix A, double * b,  double *x)
{
  int i, j;
  int s, f;
  for(i = 0; i < A.N; i++)
  {
    s = A.RowIndex[i];
    f = A.RowIndex[i + 1];
    x[i] = 0.0;
    for(j = s; j < f; j++)
      x[i] += A.Value[j] * b[ A.Col[j] ];
  }
  return BICG_OK;
}
```

Auxiliary functions (5)

- **scalarProduct() –** vector scalar product computation. As inputs, the function receives pointers to the vectors **a**, **b** and their dimension **n**. Output of the function is the scalar product.

```
double scalarProduct(int n, double *a,  double *b)
{
  double sum = 0.0;
  int i;
  for(i = 0; i < n; i++)
  {
    sum += a[i] * b[i];
  }
  return sum;
}
```

Software implementation of the biconjugate gradient method (1)

- Let us start software implementation of the biconjugate gradient method of solving linear systems.
- Declare and implement the **BiCG()** function of solving linear systems using the iterative method in question in **BiCG.h** and **BiCG.cpp**, respectively.
- As inputs, the function will receive the system matrix **A** in the CRS format,righ-hand vector **b** and the maximum allowable number of iterations **CountIteration**. Function outputs are the pointer to the computed approximate solution **x**, number of performed iterations **iter**.

Software implementation of the biconjugate gradient method (2)

- In the function body, compute approximations to the system solution in the **x** array.
- As the method stop criterion, use the maximum allowable number of iterations **CountIteration** or the required solution accuracy.
- The attainable accuracy will be computed in the **check** variable as the relative residual norm $\|r^{(k)}\| / \|b\|$
  - Required accuracy is set by the **EPSILON** invariable in the **type.h** file of the **parser** project.

Software implementation of the biconjugate gradient method (3)

```
int BiCG(crsMatrix A, double * b,  double *x,
  int CountIteration, int &iter)
{
  // To speed up computation, compute
  // the transposed matrix A
  crsMatrix At;

  At.N  = A.N;
  At.NZ = A.NZ;

  Transpose(A.N, A.Col, A.RowIndex, A.Value,
    &(At.Col), &(At.RowIndex), &(At.Value));
```

Software implementation of the biconjugate gradient method (4)

```
// arrays to store the residual
// of the current and next approximations
double * R, * biR;
double * nR, * nbiR;

R    = new double [A.N];
biR  = new double [A.N];
nR   = new double [A.N];
nbiR = new double [A.N];

// arrays to store the current and next
// method step direction vectors
double * P, * biP;
double * nP, * nbiP;

P    = new double [A.N];
biP  = new double [A.N];
nP   = new double [A.N];
nbiP = new double [A.N];
```

Software implementation of the biconjugate gradient method (5)

```
// pointer to change pointers for the vectors of the current
// and next method steps
double * tmp;

// arrays to store the product of matrix multiplication by
// the direction vector and the biconjugate vector
double * multAP, * multAtbiP;
multAP    = new double [A.N];
multAtbiP = new double [A.N];

// beta and alfa - computing formula coefficients
double alfa, beta;
// numerator and denominator of beta and alfa
double numerator, denominator;

// variables for computation
// of the current approximation accuracy
double check, norm;
norm = sqrt(scalarProduct(A.N, b, b));
```

Software implementation of the biconjugate gradient method (6)

- As the initial approximation, take a unit vector

```
// setting the initial approximation
int i;
int n = A.N;

for(i = 0; i < n; i++)
{
  x[i] = 1.0;
}

// Method initialization
MatrixVectorMult(A, x, multAP);
for(i = 0; i < n; i++)
{
  R[i] = biR[i] = P[i] = biP[i] = b[i] - multAP[i];
}
```

Software implementation of the biconjugate gradient method (7)

```
// Method initialization
for(iter = 0; iter < CountIteration; iter++)
{
  MatrixVectorMult(A, P, multAP);
  MatrixVectorMult(At, biP, multAtbiP);

  numerator   = scalarProduct(A.N, biR, R);
  denominator = scalarProduct(A.N, biP, multAP);
  alfa = numerator / denominator;

  for(i = 0; i < n; i++)
  {
    nR[i] = R[i] - alfa * multAP[i];
  }

  for(i = 0; i < n; i++)
  {
    nbiR[i] = biR[i] - alfa * multAtbiP[i];
  }

  denominator = numerator;
  numerator   = scalarProduct(A.N, nbiR, nR);
  beta = numerator / denominator;
```

Software implementation of the biconjugate gradient method (8)

```
  for(i = 0; i < n; i++)
  {
    nP[i] = nR[i] + beta * P[i];
  }
  for(i = 0; i < n; i++)
  {
    nbiP[i] = nbiR[i] + beta * biP[i];
  }

  // control compliance with accuracy requirements
  check = sqrt(scalarProduct(n, R, R)) / norm;
  if (check < EPSILON)
    break;
  for(i = 0; i < n; i++)
  {
    x[i] += alfa * P[i];
  }

  // swap positions of the current and next step arrays
  tmp = R; R = nR; nR = tmp;
  tmp = P; P = nP; nP = tmp;
  tmp = biR; biR = nbiR; nbiR = tmp;
  tmp = biP; biP = nbiP; nbiP = tmp;
}
```

Software implementation of the biconjugate gradient method (9)

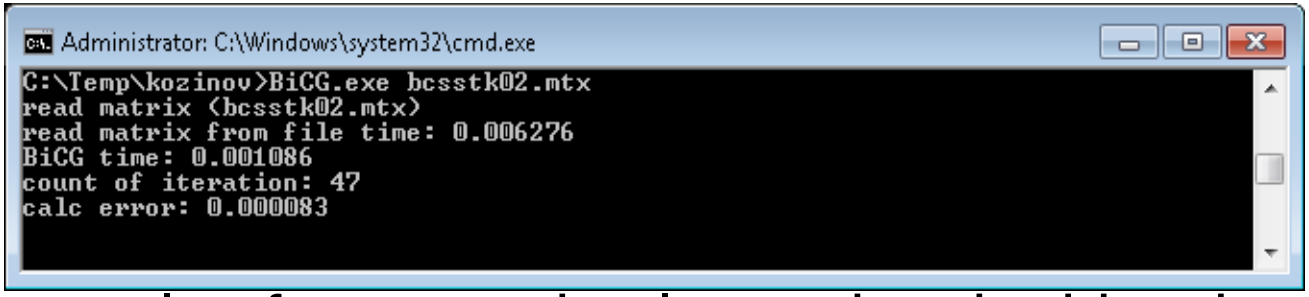
```
    // memory release
    FreeMatrix(At);
    delete [] R;
    delete [] biR;
    delete [] nR;
    delete [] nbiR;

    delete [] P;
    delete [] biP;
    delete [] nP;
    delete [] nbiP;

    delete [] multAP;
    delete [] multAtbiP;

    return BICG_OK;
}
```

Biconjugate gradient method convergence analysis (1)

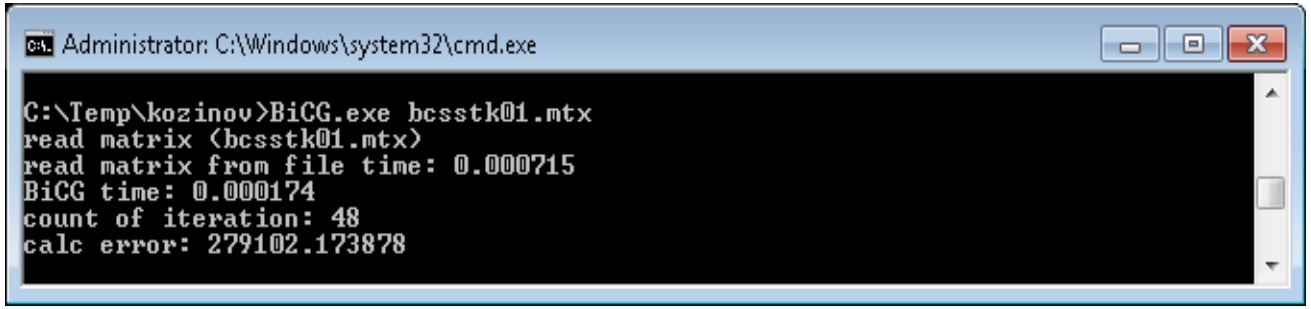
- Incorporate the call for **BiCG()** in the **main()** function body.
- Now the project can be compiled from **Build→Rebuild** and the respective correctness check can be performed.

Biconjugate gradient method convergence analysis (2)

- Example of program implementing the biconjugate gradient method for a well-conditioned matrix



- Example of program implementing the biconjugate gradient method for an ill-conditioned matrix



Biconjugate gradient method convergence analysis (3)

- No solution has been found for the second matrix due to two factors.
  - First, the matrix **bcsstk01.mtx** is ill-conditioned.
  - Second, the parameter restricting the number of algorithm iterations during experiments was the matrix size (theoretical estimate).

Biconjugate gradient method convergence analysis (4)

- Results of running the software implementation of the biconjugate gradient method for symmetric matrices (required computational accuracy is 0.0001).

Matrix	matrix dimension	attainable method accuracy	number of iterations	algorithm runtime
bcsstk01	48	116213.2199	48	0.000
bcsstk05	153	27.2275	153	0.002
bcsstk10	1 086	162.2504	1 086	0.092
bcsstk12	1 473	8576.2735	1 473	0.187
parabolic_fem	525 825	0.0012	717	25.089
tmt_sym	726 713	0.0062	2 487	122.543

Biconjugate gradient method convergence analysis (5)

- Results of running the software implementation of the biconjugate gradient method for non-symmetric matrices (required computational accuracy is 0.0001).

Matrix	matrix dimension	attainable method accuracy	number of iterations	algorithm runtime
fs_541_1	541	0.00030	6	0.000
ex22	839	1.63999	839	0.068
sherman2	1080	5075084918.6	1080	0.103
cage10	11397	0.00213	10	0.011

Biconjugate gradient method convergence analysis (6)

- To improve the method convergence rate, preconditioning is used.
- Let us implement the preconditioned biconjugate gradient method.

Software implementation of the preconditioned conjugate gradient method

Preconditioned biconjugate gradient method implementation

- *A*lgorithm pseudocode:

1. Вычислить $r_0 = b - Ax_0$; выбрать $\bar{r}_0$ так, чтобы $(r_0, \bar{r}_0) \neq 0$
(например, выбрать $\bar{r}_0 = r_0$).
2. Вычислить $z_0 = M^{-1} r_0$, при известном факторе
($sol = L^{-1} r_0$, $z_0 = U^{-1} sol$)
3. Вычислить $\bar{z}_0 = M^{-1} \bar{r}_0$, при известном факторе
($sol = L^{-1} \bar{r}_0$, $\bar{z}_0 = U^{-T} sol$)
4. Положить $p_0 = z_0$, $\bar{p}_0 = \bar{z}_0$
5. for j=0,1,… do
6. $\alpha_j = \frac{(z_j, \bar{r}_j)}{(Ap_j, \bar{p}_j)}$
7. $x_{j+1} = x_j + \alpha_j p_j$
8. $r_{j+1} = r_j - \alpha_j A p_j$
9. $\bar{r}_{j+1} = \bar{r}_j - \alpha_j A^T \bar{p}_j$
10. Вычислить $z_{j+1} = M^{-1} r_{j+1}$, при известном факторе
($sol = L^{-1} r_{j+1}$, $z_{j+1} = U^{-1} sol$)
11. Вычислить $\bar{z}_{j+1} = M^{-T} \bar{r}_{j+1}$, при известном факторе
($sol = L^{-1} \bar{r}_{j+1}$, $\bar{z}_{j+1} = U^{-1} sol$)
10.

8. $\beta_j = \frac{(z_{j+1}, \bar{r}_{j+1})}{(z_j, \bar{r}_j)}$. Если $\beta_j = 0$ или $\| r_{j+1} \| < \varepsilon$ то Стоп.
9. $p_{j+1} = z_{j+1} + \beta_j p_j$
10. $\bar{p}_{j+1} = \bar{z}_{j+1} + \beta_j \bar{p}_j$
11. end j

Project creation

- Preconditioner implementation is outside the scope of this laboratory work. Use the *ILU*-preconditioner from the respective laboratory work.
- Add to the solution a project containing implementation of the *ILU*(p)-preconditioner. The project contains the following files:
  - **ilup.h** and **ilup.cpp** – files containing declaration and software implementation of the symbolic and numerical pars of the *ILU*(p) algorithm
  - **validation.h** and **validation.cpp** – files containing declaration and software implementation of factorization check for correctness and the function that divides the matrix containing both *L* and *U* into two separate matrices.

Auxiliary functions (1)

- To use the ILU-preconditioner, implement the additional function of solving triangular systems **GaussSolve()** in the **sparseMatrixOperation.cpp** and **sparseMatrixOperation.h** files.
- The **GaussSolve()** function solves linear systems with triangular matrices. As inputs, the function receives the structure of the system matrix **A**, right-hand vector **b** and symbol denoting system type **uplo** - *L* means the lower triangular system and *U* means the upper triangular one. The output is the system solution **x**.
- To implement the function, use the existing functionality represented in the MKL.
  - To use the **mkl_dcsrtrsv()** function of triangular system solution, represent the matrix so that its indices start from 1.
  - Having solved the system, go back to numbering accepted in C/C++ starting from zero.

Auxiliary functions (2)

```
void GaussSolve(crsMatrix* A, char uplo,
  double* b, double* x)
{
  char transa = 'N';
  char diag   = 'N';
  int i;
  for(i = 0; i < A->N + 1; i++)
    A->RowIndex[i] ++;
  for(i = 0; i < A->NZ; i++)
    A->Col[i] ++;
  mkl_dcsrtrsv(&uplo, &transa, &diag, &(A->N), A->Value,
    A->RowIndex, A->Col, b, x);
  for(i = 0; i < A->N + 1; i++)
    A->RowIndex[i] --;
  for(i = 0; i < A->NZ; i++)
    A->Col[i] --;
}
```

Software implementation of the preconditioned biconjugate gradient method (1)

- Develop **BiCG_M()**, a new function of solving linear systems using the preconditioned biconjugate gradient method.
- Main features of the function
  1. The function accepts the preconditioner in the form of two matrices, *L* and *U*.
  2. Для вычислений необходимо хранить помимо транспонированной матрицы транспонированные матрицу фактора предобуславливателя
  3. The biconjugate gradient method requires additional steps to apply the preconditioner to the linear system.

Software implementation of the preconditioned biconjugate gradient method (2)

```
int BiCG_M(crsMatrix A, double * b,  double *x, crsMatrix L, crsMatrix U,
           int CountIteration, int &iter)
{
  // To speed up computation, compute the transposed matrix A,
  crsMatrix At;
  ...
  // To compute the matrix inverse to the transposed
  // preconditioner matrix, compute the transposed
  // matrices L and U
  crsMatrix Lt;

  Lt.N  = L.N;
  Lt.NZ = L.NZ;

  Transpose(L.N, L.Col, L.RowIndex, L.Value,
    &(Lt.Col), &(Lt.RowIndex), &(Lt.Value));

  crsMatrix Ut;

  Ut.N  = U.N;
  Ut.NZ = U.NZ;

  Transpose(U.N, U.Col, U.RowIndex, U.Value,
    &(Ut.Col), &(Ut.RowIndex), &(Ut.Value));
```

Software implementation of the preconditioned biconjugate gradient method (3)

```
// arrays to store the residue of the current
// and next approximations
double * R, * biR;
double * nR, * nbiR;
...
// auxiliary vector and the biconjugate vector
// to use the preconditioner
double * Z, * biZ;
double * nZ, * nbiZ;
double * sol;

Z    = new double [A.N];
biZ  = new double [A.N];
nZ   = new double [A.N];
nbiZ = new double [A.N];
sol  = new double [A.N];

// arrays to store the current and next
// method step direction vectors
double * P, * biP;
double * nP, * nbiP;
...
```

Software implementation of the preconditioned biconjugate gradient method (4)

```
// Method initialization
MatrixVectorMult(A, x, multAP);
for(i = 0; i < n; i++)
{
  R[i] = biR[i] = b[i] - multAP[i];
}

GaussSolve(&L, 'L', R  , sol);
GaussSolve(&U, 'U', sol, Z);
GaussSolve(&Ut, 'L', biR, sol);
GaussSolve(&Lt, 'U', sol, biZ);

for(i = 0; i < n; i++)
{
  P[i] = Z[i];
  biP[i] = biZ[i];
}
```

Software implementation of the preconditioned biconjugate gradient method (5)

```
// Method implementation
for(iter = 0; iter < CountIteration; iter++)
{
  MatrixVectorMult(A, P, multAP);
  MatrixVectorMult(At, biP, multAtbiP);
  numerator   = scalarProduct(A.N, biR, Z);
  denominator = scalarProduct(A.N, biP, multAP);
  alfa = numerator / denominator;
  for(i = 0; i < n; i++)
  {
    nR[i] = R[i] - alfa * multAP[i];
  }
  for(i = 0; i < n; i++)
  {
    nbiR[i] = biR[i] - alfa * multAtbiP[i];
  }
  GaussSolve(&L, 'L', nR , sol);
  GaussSolve(&U, 'U', sol, nZ);
  GaussSolve(&Ut, 'L', nbiR, sol);
  GaussSolve(&Lt, 'U', sol , nbiZ);

  denominator = numerator;
  numerator   = scalarProduct(A.N, nbiR, nZ);
  beta = numerator / denominator;
```

Software implementation of the preconditioned biconjugate gradient method (6)

```
  for(i = 0; i < n; i++)
  {
    nP[i] = nZ[i] + beta * P[i];
  }
  for(i = 0; i < n; i++)
  {
    nbiP[i] = nbiZ[i] + beta * biP[i];
  }
  check = sqrt(scalarProduct(n, R, R)) / norm;
 if (check < EPSILON)
     break;
  for(i = 0; i < n; i++)
  {
    x[i] += alfa * P[i];
  }
  // swap array positions
  tmp = R; R = nR; nR = tmp;
  tmp = P; P = nP; nP = tmp;
  tmp = biR; biR = nbiR; nbiR = tmp;
  tmp = biP; biP = nbiP; nbiP = tmp;

  tmp = Z; Z = nZ; nZ = tmp;
  tmp = biZ; biZ = nbiZ; nbiZ = tmp;
}
```

Software implementation of the preconditioned biconjugate gradient method (7)

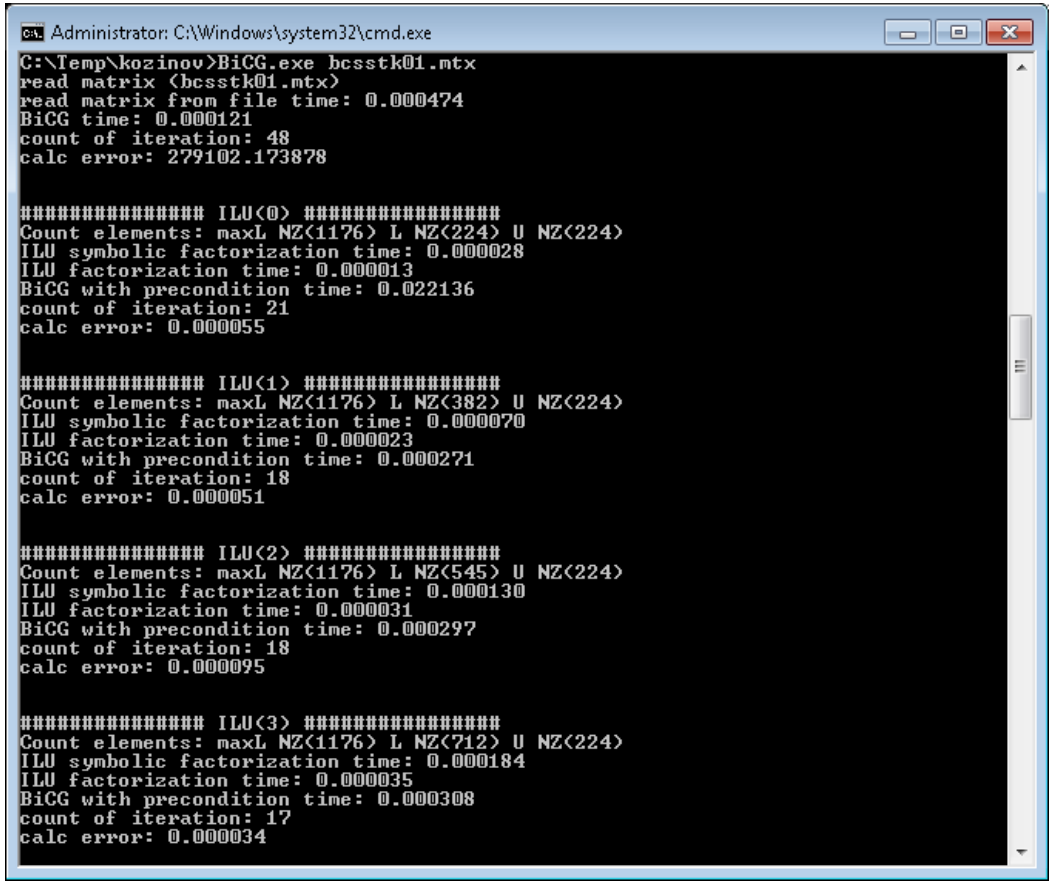
```
  // memory release
   ...
   FreeMatrix(Lt);
   FreeMatrix(Ut);

   delete [] Z;
   delete [] biZ;
   delete [] nZ;
   delete [] nbiZ;
   delete [] sol;

   ...

   return BICG_OK;
}
```

Task 2

- Modify the **main()** function to call for the biconjugate gradient method.
  - For this purpose, connect the **ilup.h** header file.
  - Call the *ILU*(p) computation function.
  - Divide matrices into *L* and *U*.
  - Call the implemented *BiCG_M*() function.
- Study how the preconditioner quality influences the method operation quality.
  - For this, add the call for system solution using the biconjugate gradient method with a preconditioner computed at various *p* levels.

Preconditioned biconjugate gradient method convergence analysis (1)

- Run the biconjugate gradient algorithm for the **bcsstk01.mtx** matrix.



Preconditioned biconjugate gradient method convergence analysis (2)

- The matrix **bcsstk01.mtx** is ill-conditioned.
  - As a result, this matrix showed a greater iterative method error in the context of restricted number of iterations.
- The use of preconditioner enabled highly accurate system solution.
- As ***p*** (algorithm level parameter) grows, the number of iterations required to find the solution reduces.
  - For this matrix, the higher is the level, the better is the preconditioner.

Preconditioned biconjugate gradient method convergence analysis (3)

- Results of running the preconditioned biconjugate method software implementation for the bcsstk10 matrix.

matrix:	bcsstk10		**dimension:**	1 086		
p	number of iterations	attainable method accuracy	ILU(p) symbolic part time	ILU(p) numerical part time	BiCG time	Total time
without ILU	*1086*	*162.250*			*0.0922*	*0.0922*
0	211	0.00030	0,0010	0.0007	0.0910	0.0927
1	83	0.00059	0,0030	0.0009	0.0335	0.0374
2	70	0.00032	0,0059	0.0010	0.0303	0.0373
3	92	0.00012	0,0093	0.0011	0.0407	0.0511

Preconditioned biconjugate gradient method convergence analysis (4)

- Results of running the preconditioned biconjugate method software implementation for the tmt_sym matrix.

matrix:	tmt_sym		**dimension:**	726 713		
P	number of iterations	attainable method accuracy	ILU(p) symbolic part time	ILU(p) numerical part time	BiCG time	Total time
without ILU	*2487*	*0.006*			*122.5435*	*122.5435*
0	894	0.00933	0.2886	0.0906	140.6882	141.0673
1	896	0.00813	0.6497	0.1218	151.9161	152.6875
2	894	0.00870	1.3676	0.1608	163.7241	165.2525
3	895	0.01176	2.4250	0.2079	183.1238	185.7567

Preconditioned biconjugate gradient method convergence analysis (5)

- From the tables above one can see that the use of preconditioner lets considerably reduce the number of iterations required by the method to solve a linear system.
- Reduction of method iterations may not always improve the solution time in general (the most important thing is the preconditioner quantity, not the level).
  - For the matrix **tmt_sym**, the number of iterations reduced more than twice but the solution time has increased.
  - At the same time, computation speedup is observed for the **bcsstk10** matrix**.**

Preconditioned biconjugate gradient method convergence analysis (5)

- The preconditioner quality in the algorithm may not always depend on the level.
  - For the **bcsstk10** matrix, the best level value is 1 or 2.
  - For the **tmt_sym** matrix, the 0 level will be the best.
- To obtain better preconditioners that are less dependent on parameters, other preconditioner search algorithms should be employed.

Software implementation of the algorithm

- The main computation operation of the biconjugate gradient algorithm is scalar product computation.
  - The scalar product has a low computational complexity but is used more than once for the algorithm purposes. Therefore, scalar product parallelization will definitely be inefficient.
  - Low efficiency is due to considerable contingencies for parallel computation.
- It will be more efficient to replace the developed software implementation of the scalar product by function call from the library.
  - Such libraries as Intel MKL can be used.
- To parallelize the biconjugate gradient algorithm in a more efficient manner, one can compute independent scalar products in parallel.

Added tasks

1. Analyze the method convergence rate depending on precision of the floating point arithmetics.
2. Implement the biconjugate gradient method with a preconditioner resulting from the ILU^T algorithm.
3. Analyze the efficiency and scalability of the BiCG algorithm parallel modification with parallelization on the scalar product level.
4. Evaluate the efficiency of library implementations of the mathematic operations offered by Intel MLK for the biconjugate gradient method.
5. Analyze the efficiency and scalability of the BiCG algorithm parallel modification with parallelization by parallel computation of independent scalar products.

References

1. Saad Y. Iterative methods for sparse linear systems. – SIAM, 2003.
2. Laboratory Work on Solving sparse linear systems using the preconditioned conjugate gradient method
3. Laboratory Work on Multiplication of Matrices
4. ILU(p) Laboratory Work

Questions

- ???

Authors

- Evgeny Kozinov,
  Assistant, Software Department, CMC Faculty,
  Nizhny Novgorod State University
  Evgeniy.Kozinov@gmail.com