Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Iterative Methods for Solving Linear Systems

Laboratory Work
Preconditioner Construction by Incomplete LU-factorization

Supported by Intel

K.A. Barkalov,
Software Department

Contents

Introduction (1)

- In case of solving linear systems by iterative methods, the *matrix condition number* must be as small as possible.
  - When the linear system matrix is ill-conditioned, iterative methods have a low convergence rate.
- To reduce the condition number, special approaches and techniques must be used.
  - One of them consists in multiplication of the initial linear system matrix by the *preconditioner* matrix.

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

Introduction (2)

- If a matrix inverse to the initial linear system matrix is used as a preconditioner, the system solution will result from matrix multiplication by a vector.
  - In this case, preconditioner search will reduce to the use of a *direct method* of solving linear systems.
- If the preconditioner matrix is close to the inverse one, the new linear system will, from physical consideration, be well-conditioned, as it will be close to the identity matrix.

Introduction (3)

- To search for the inverse matrix, one can make good use of the *matrix factor*.
  - Factor search is one of the most complex stages of direct methods.
- For iterative methods, to reduce the time required to search the solution, *incomplete LU-factorization* is used.
  - Incomplete factorization makes it possible to obtain the matrices *L* and *U* close to the factor at a significantly shorter time required for incomplete factorization.

Purposes of work

- ***The purpose of the laboratory work*** is to demonstrate practical implementation of the preconditioner search method based on the classical ILU(p) method.

Objectives of work

- Study of the ILU(p) incomplete LU-factorization method.
- Development of serial implementation of the ILU(0) preconditioner searching method.
- Analysis of the method software implementation influence on the matrix condition.
- Development of serial implementation of the ILU(p) preconditioner searching method.
- Analysis of the level modification influence on the matrix condition and ILU(p) search time.

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)

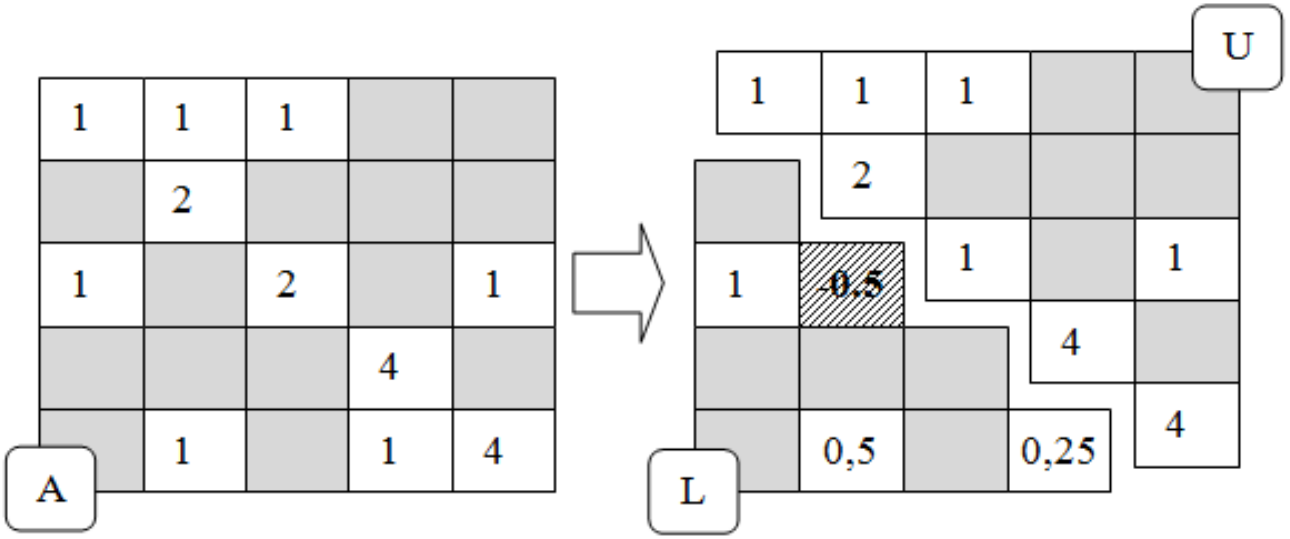
Incomplete LU-factorization (1)

- The basis for the implementation procedure of the incomplete *LU*-factorization is the Gaussian elimination method (or the complete *LU*-factorization).

```
For i = 2 to n do
Begin
  For k = 1 to (i - 1) do
  Begin
    a_{i,k} = a_{i,k} / a_{k,k}
    For j = (k + 1) to n do
    Begin
      a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}
    End
  End
End
```

Incomplete LU-factorization (2)

- Example of complete *LU*-factorization of a matrix



- A complete *LU*-factorization of the initial matrix will result in additional nonzero elements.
  - In the case under consideration, one additional element has appeared. It is represented by a crosshatched square.
- In the picture, matrix elements containing a zero value are shown in grey.

Incomplete LU-factorization (3)

- The idea of ILU(0)-factorization is to eliminate all new non-zero elements that appear in the course of decomposition, from the dct factor.
  - In the classical ILU(0) algorithm, the initial matrix pattern is used as the factor pattern.
- In the numerical part of the algorithm, coefficients of the matrices *L'* and *U'* are computed so that the initial elements of the matrix *A* coincide with *A'=L'*U'* during matrix multiplication.

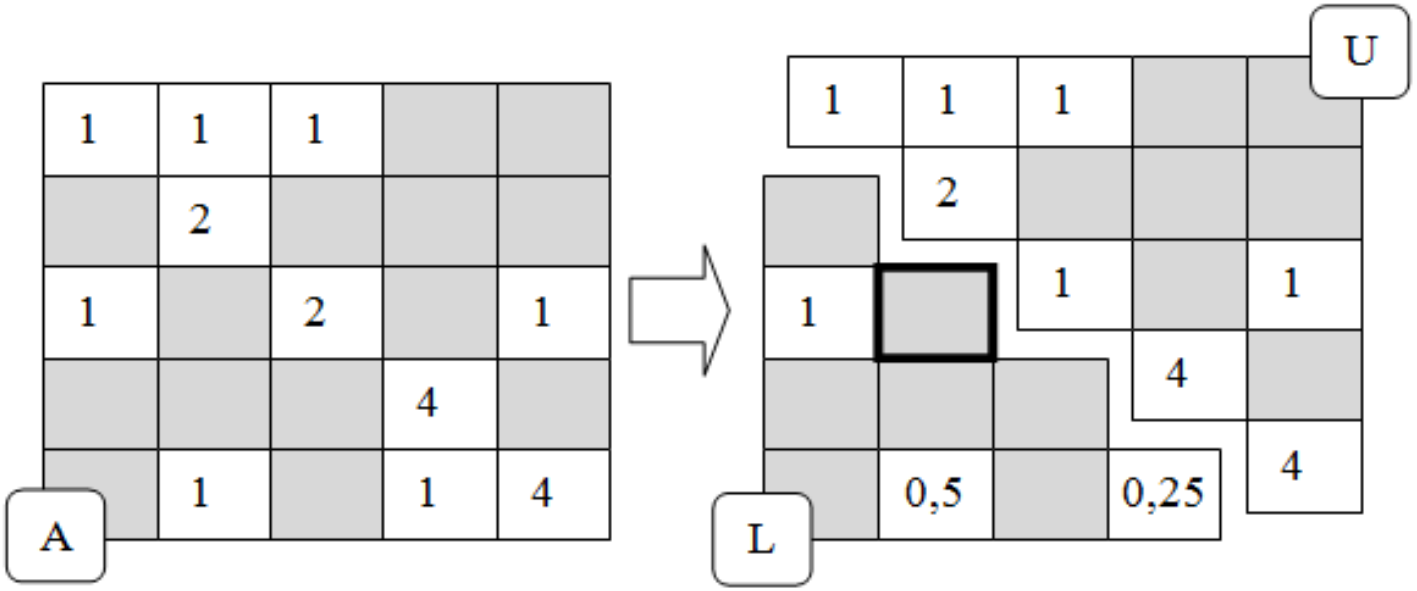
Incomplete LU-factorization (4)

- *ILU(0)* algorithm search pseudocode

```
For i = 2 to n do
Begin
  For k = 1 to (i - 1) and (i, k) ∈ NZ(A) do
  Begin
    a_{i,k} = a_{i,k} / a_{k,k}
    For j = (k + 1) to n and (i, j) ∈ NZ(A) do
    Begin
      a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}
    End
  End
End
```

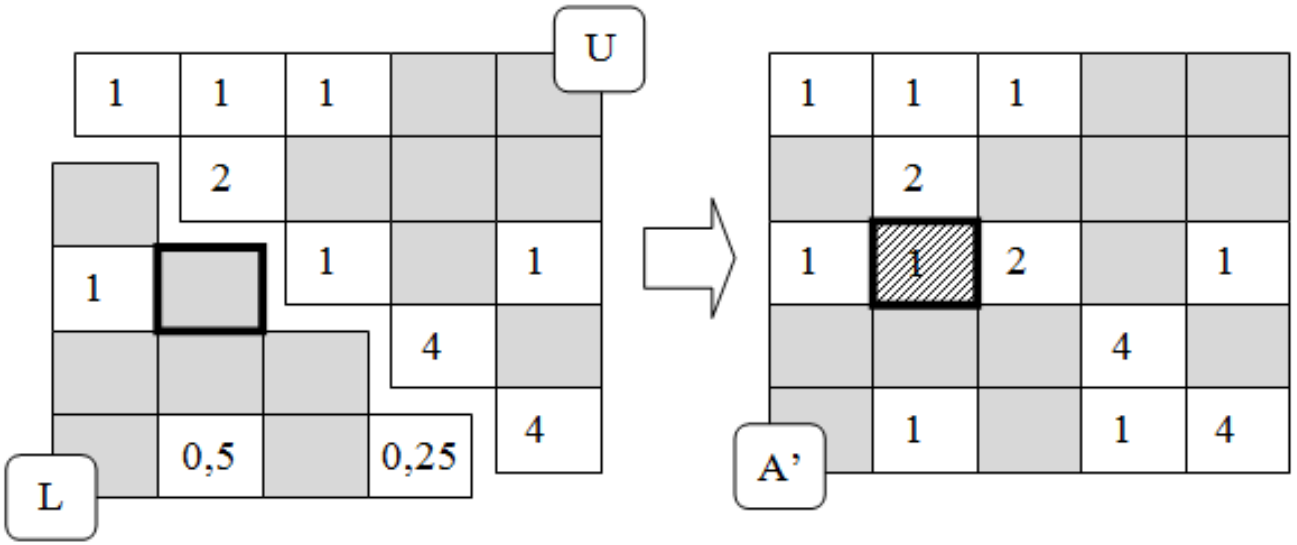
Incomplete LU-factorization (5)

- Example of the found *ILU(0)*-factor of the matrix *A*



Incomplete LU-factorization (6)

- The main advantage of the ILU(0) algorithm is its running speed.
- Multiplication of matrices of the obtained factor may result in new nonzeroes.
- Example:



ILU(0) SOFTWARE IMPLEMENTATION

ILU(0) software implementation

- For convenience, the algorithm software implementation can be divided into three stages.
  - First, develop the ILU(0) software implementation.
  - Second, check the algorithm for consistency.
  - In the end, develop main function of the program to enable experimentation and evaluate the obtained preconditioners.

Software implementation

- Template of the function implementing *ILU(0)*

```
/**
 * API
 *   int ilu0(int n, double* a,int* col,int* row,
 *      double* luval, int* uptr)
 *   ilu0 - matrix factorization
 * INPUT
 *   int    n   - matrix dimension
 *   double * a - nonzero elements
 *   int  * col - column indices
 *   int  * row - row prefixes
 * OUTPUT
 *   double * luval - values of the resolved matrices L and U
 *   int    * uptr - indices of diagonal elements
 *                   in the luval array
 * RETURN
 *   an error code returns
 *   0    - factorization is successful
 *   -(n + 1) - number of row where the diagonal has 0
 **/
int ilu0(int n, double * a, int * col, int * row,
     double * luval, int * uptr);
```

Task 1

- Develop software implementation of the ILU(0) algorithm in accordance with the function template.

Software implementation (1)

- Start algorithm implementation. First, declare the main variables used in the algorithm.

```
int ilu0(int n, double * a, int * col, int * row,
     double * luval, int * uptr)
{
  int j1, j2;      // border of the current row
  int jrow;        // number of the current column
  int k, j, jj;    // cycle counters
  int *iw = NULL; // temporary array
  int jw;
  double t1;
```

Software implementation (2)

- Among the declared variables, the pointer to the temporary array *iw* is very important.
  - This array stores the value index in the CRS array for the current row in the nonzero positions.
  - Instead of searching for each nonzero value, multilevel addressing is used.
- Create a temporary array required for fast computation and clear its memory.

```
iw = new int[n];
memset(iw, 0, n * sizeof(int));
```

Software implementation (3)

- To compute the factor, copy the initial matrix values to the factor value array.

```
memcpy(luval, a, row[n] * sizeof(double));
```

- The factor is computed in place of the initial matrix values.

Software implementation (4)

- The factor is computed row-by-row, upside down.

```
for(k = 0; k < n; k++)
{
```

- Then, fill the computation speedup array for each row.

```
j1 = row[k];
j2 = row[k + 1];
for(j = j1; j < j2; j++)
{
  iw[col[j]] = j;
}
```

Software implementation (5)

- Then develop a code that updates the current row value until the diagonal is reached (i. e. for the lower triangle only).

```
for(j = j1; (j < j2) && (col[j] < k); j++)
{
  jrow = col[j];
  t1 = luval[j] / luval[uptr[jrow]];
  luval[j] = t1;
  for(jj = uptr[jrow]+1; jj < row[jrow + 1]; jj++)
  {
  jw = iw[col[jj]];
  if(jw != 0)
  {
    luval[jw] = luval[jw] - t1 * luval[jj];
  }
  }
}
```

Software implementation (6)

- In the end, remember the diagonal element position for the current row.

```
jrow = col[j];
uptr[k] = j;
if((jrow != k) || (fabs(luval[j]) < EPSILON))
{
  break;
}
```

- To record the index, make sure that it exists.
  - If the element does not exist, it is impossible to compute the factor as zero division may be expected.

Software implementation (7)

- To complete row processing, empty the auxiliary array.

```
    for(j = j1; j < j2; j++)
    {
      iw[col[j]] = 0;
    }
  }
```

- In the end, the function must empty the memory and return the error code.

```
  delete [] iw;
  if(k < n)
  return -(k+1);
  return 0;
}
```

Checking the algorithm for consistency

- It is important to check the developed code for consistency.
  - The only verifiable consistency criterion is the situation where, upon multiplication of the obtained factor matrices, the initial matrix values are preserved and new values may appear.
- To check the implemented algorithm for consistency, divide the obtained matrices *L* and *U* by independent matrices and then multiply them using the standard matrix multiplication algorithm. After that, compare the obtained values.

Checking the algorithm for consistency

- Matrix resolution function template:

```
/**
 * API
 *  void LUmatrixSeparation (crsMatrix ilu, int *uptr,
 *                           crsMatrix &L, crsMatrix &U);
 *  matrix resolution into the matrices L and U
 * INPUT
 *   crsMatrix ilu  - ilu matrices in the same structure
 *   int    * uptr - indices of diagonal elements
 *                   in the ilu array
 * OUTPUT
 *   crsMatrix &L   - separated matrix L
 *   crsMatrix &U   - separated matrix U
 * RETURN
 */
void LUmatrixSeparation(crsMatrix ilu, int *uptr, crsMatrix
   &L, crsMatrix &U);
```

Checking the algorithm for consistency

- Matrix pattern consistency check function template

```
/**
 * API
 *  bool structValidation (crsMatrix &A,  crsMatrix &M);
 *  checking the preconditioner structure for consistency
 * INPUT
 *   crsMatrix &A   - initial matrix
 *   crsMatrix &M   - preconditioner
 * OUTPUT
 *
 * RETURN
 * is the structure consistent?
 */
bool structValidation   (crsMatrix &A,  crsMatrix &M)
```

Checking the algorithm for consistency

- Matrix value consistency check function template

```
/**
 * API
 *  double MatrixCompare (crsMatrix &A,  crsMatrix &M);
 *  computation of degree of matrix difference
 * INPUT
 *   crsMatrix &A   - initial matrix
 *   crsMatrix &M   - preconditioner
 * OUTPUT
 *
 * RETURN
 *  degree of difference
 */
double MatrixCompare (crsMatrix &A, crsMatrix &M)
```

Task 2

- Implement preconditioner check for consistency by implementing the declared funtions.

Checking the algorithm for consistency (1)

- Implement the matrix resolution function. The function accepts the matrix and diagonal element index array, returns and

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

- Declare the required variables.

```
  int countL, countU;
  int i, j, s, f, k;
  double *val;
  int    *col;
  countU = 0;
```

Checking the algorithm for consistency (2)

- First of all, count the number of elements in the matrices and

```
for(i = 0; i < ilu.N; i++)
{
  countU += (ilu.RowIndex[i+1] - uptr[i]);
}
countL = ilu.NZ + ilu.N - countU;
```

- Then memory has to be allocated to matrices.

```
InitializeMatrix(ilu.N, countL, L);
InitializeMatrix(ilu.N, countU, U);
```

Checking the algorithm for consistency (3)

- Upon memory allocation, fill the matrix arrays using arrays of the accepted matrix.

```
k = 0;
val = L.Value;  col = L.Col;
L.RowIndex[0] = k;
for(i = 0; i < ilu.N; i++)
{
  s = ilu.RowIndex[i];
  f = uptr[i];
  for(j = s; j < f; j++)
  {
    val[k] = ilu.Value[j];
    col[k] = ilu.Col[j];
    k++;
  }
  val[k] = 1.0; col[k] = i;
  k++;
  L.RowIndex[i + 1] = k;
}
```

Checking the algorithm for consistency (4)

- In the end, values fill the matrix.

```
k = 0;
val = U.Value;
col = U.Col;
U.RowIndex[0] = k;
for(i = 0; i < ilu.N; i++)
{
  s = uptr[i];
  f = ilu.RowIndex[i + 1];
  for(j = s; j < f; j++)
  {
    val[k] = ilu.Value[j];
    col[k] = ilu.Col[j];
    k++;
  }
  U.RowIndex[i + 1] = k;
}
}
```

Checking the algorithm for consistency (5)

- For the purpose of matrix multiplication, one can use the algorithm implemented in the course of the laboratory work on Sparse Matrix Multiplication.

Checking the algorithm for consistency (6)

- Implement the matrix pattern consistency check function

```
bool structValidation   (crsMatrix &A,  crsMatrix &M)
```

- Declare the required variables.

```
int i, j, fA, fM;
i = 0;
j = 0;
int k;
```

- Perform minimum consistency checks

```
if(A.N != M.N)
{
  return false;
}

if(M.NZ <= A.NZ)
{
  return false;
}
```

Checking the algorithm for consistency (7)

- For each row, check that the initial column index array is included into the new one. To speed up the check, remember that index arrays are continuous.

```
for(k = 0; k < A.N; k++)
{
  i  = M.RowIndex[k]; fM = M.RowIndex[k + 1];
  j  = A.RowIndex[k]; fA = A.RowIndex[k + 1];
  while((i < fM) && (j<fA))
  {
    if(M.Col[i] != A.Col[j]) i++;
    else
    {
      j++;
      i++;
    }
  }
  if((i == fM) && (j != fA))
    return false;
}
return true;
}
```

Checking the algorithm for consistency (8)

- After checking matrix patterns, matrix nonzero values also have to be compared.
- Matrix values are compared in the same way as matrix patterns.
- For comparison purposes it is proposed to find the maximum difference modulus for the corresponding matrix elements.

Main function implementation (1)

- The necessary functionality has been implemented, so it can be incorporated in the main program function.

Main function implementation (2)

- First, accept the resolved matrix name using command line arguments.

```
int main(int argc, char **argv)
{
  // review of parameters
  char *matrixName;
  ParseArgv(argc, argv, matrixName);
```

Main function implementation (3)

- Then, declare the necessary variables to store matrices and concomitant structures.

```
crsMatrix readA;
crsMatrix *matA;
crsMatrix lu;
crsMatrix L;
crsMatrix U;
crsMatrix M;
int *uptr;
int typeOfMatrix;
int error;
double diff;
```

- Create a respective timer.

```
Stopwatch *time = createStopwatch();
```

Main function implementation (4)

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

Main function implementation (5)

- If the matrix is symmetric and specified only by the upper triangle, the algorithm will not work.
  - In this case, to ensure algorithm operability, the lower triangle must be added to the matrix.

```
if(typeOfMatrix == UPPER_TRIANGULAR)
{
  matA = UpTriangleMatrixToFullSymmetricMatrix(&readA);
}
else
{
  matA = &readA;
}
time->stop();
printf("read matrix from file time: %f\n", time->
                                          getElapsed());

uptr = new int [matA->N];
```

Main function implementation (6)

- Having read the initial matrix, create a matrix to store the computed factor.

```
time->reset();
time->start();
InitializeMatrix(matA->N, matA->NZ, lu);
memcpy(lu.RowIndex, matA->RowIndex,
       (matA->N + 1) * sizeof(int));
```

- Everything is ready to call the factor computation function.

```
error = ilu0(matA->N, matA->Value, matA->Col,
             matA->RowIndex, lu.Value, uptr);
time->stop();
```

- To analyze runtime-related algorithm efficiency, derive the factor computing time.

```
printf("ILU factorization time: %f\n", time->
                                    getElapsed());
```

Main function implementation (7)

- In the end, the developed algorithm must be checked for consistency

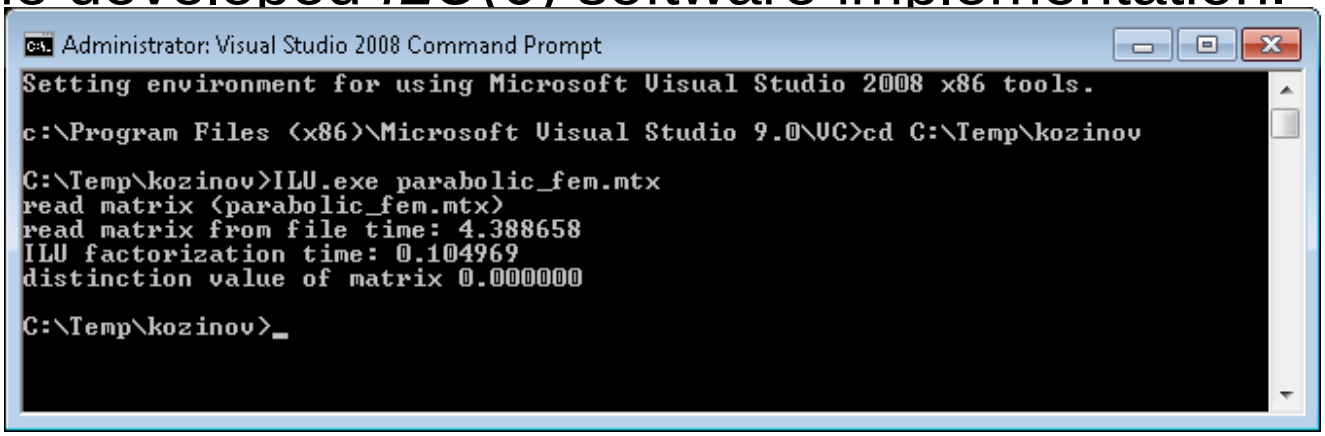
```
// checking ILU for consistency
// matrix resolution into L and U
LUmatrixSeparation(lu, uptr, L, U);

// matrix multiplication
ProductSparseMatrix(L, U, M);
```

Main function implementation (8)

```
  // checking the obtained matrix structure for
  consistency
  if(!structValidation(*matA, M))
  {
    printf("invalid struct of matrix M \n");
    return -2;
  }

  // computing difference of values
  diff = MatrixCompare(*matA, M);

  printf("distinction value of matrix %f \n", diff);

  return 0;
}
```

ILU(0) algorithm implementation run analysis

- Run the developed *ILU*(0) software implementation.



- The run results prove that the time required to search for the factor is short indeed.
- The algorithm runs in a consistent manner.

Time and consistency of ILU(0) implementation running for symmetric matrix sample

Matrix	matrix dimension	factor search time	error
bcsstk01	48	0,000018	0,00000
bcsstk05	153	0,000117	0,00000
bcsstk10	1 086	0,000642	0,00000
bcsstk13	2 003	0,005276	0,00156
parabolic_fem	525 825	0,090491	0,00000
tmt_sym	726 713	0,090719	0,00000

Time and consistency of ILU(0) implementation running for non-symmetric matrix sample

Matrix	matrix dimension	factor search time	Error
fs_541_1	541	0,000066	0,00000
ex22	839	0,000929	0,00000
sherman2	1 080	0,000812	0,00000
cage10	11 397	0,003938	0,00000

ILU(0) algorithm implementation run analysis

- As you can see from these tables, the algorithm runtime is very short even for large matrices.
- However, short runtimes do not always mean good algorithm performance.
- Evaluate quality of the obtained preconditioner.
  - For this purpose, compute condition numbers for the matrix without preconditioner and the preconditioned one.
- To compute condition numbers, you can use the respective MKL functionality.

ILU(0) algorithm implementation run analysis

- To start with, implement the auxiliary functionality.
- The first function to be developed is conversion of a sparse matrix into a dense one.

ILU(0) algorithm implementation run analysis

```
/**
 * API
 *    double * CRStoGeneral(crsMatrix A)
 *    matrix conversion from CRS to dense format
 * INPUT
 *    crsMatrix A - matrix in a CRS format
 * OUTPUT
 *
 * RETURN
 *    matrix in a dense form
 **/
double * CRStoGeneral(crsMatrix A)
```

ILU(0) algorithm implementation run analysis

```
double * CRStoGeneral(crsMatrix A)
{
  int i, j, s, f;
  double * mat;
  mat = new double[A.N * A.N];
  memset(mat, 0, A.N * A.N * sizeof(double));
  for(i = 0; i < A.N; i++)
  {
    s = A.RowIndex[i];
    f = A.RowIndex[i + 1];
    for(j = s; j < f; j++)
    {
      mat[i * A.N + A.Col[j]] = A.Value[j];
    }
  }
  return mat;
}
```

ILU(0) algorithm implementation run analysis

- In MKL, the condition number is approximated.
- For computation purposes, the matrix norm is used.
  - One can use either 1-norm or infinite norm.
  - For the purposes of this laboratory work, the infinite matrix norm will be used.

ILU(0) algorithm implementation run analysis

```
/**
 * API
 *   double MatrixNormI(double* Matrix, int size)
 *   infinite matrix norm
 * INPUT
 *   double* Matrix - matrix in a dense form
 *   int     size   - matrix dimension
 * OUTPUT
 * RETURN
 *   infinite matrix norm
 **/
double MatrixNormI(double* Matrix, int size)
{
  double norm = 0.0;
  double sum;
  for (int i = 0; i < size; i++)
  {
    sum = 0.0;
    for (int j = 0; j < size; j++)
      sum += fabs(Matrix[i*size + j]);
    if (sum > norm) norm = sum;
  }
  return norm;
}
```

ILU(0) algorithm implementation run analysis

- Using the functional for matrix conversion and norm calculation, one can compute the initial matrix condition number.

```
/**
 * API
 *   double getConditionNumber(crsMatrix A)
 *   condition number evaluation
 * INPUT
 *   crsMatrix A - matrix in a CRS format
 * OUTPUT
 *
 * RETURN
 *   condition number
 **/
double getConditionNumber(crsMatrix A)
```

ILU(0) algorithm implementation run analysis

- First, declare the necessary set of variables.

```
// auxiliary variables
double *Matrix;
int* Size = &(A.N);
int* ipiv = new int [A.N];
double* ANorm;
double* work = new double [A.N*4];
int* iwork = new int [A.N];
int lda = A.N;
int* Lda = &lda;
int info = 0;
int *Info = &info;
double rcond;
double* Rcond = &rcond;
// norms and condition numbers
double norm, cond = -1.0;
```

ILU(0) algorithm implementation run analysis

- Second, convert the matrix from sparse into dense.

```
// make the matrix dense
Matrix = CRStoGeneral(A);
```

- In the end, call the condition number evaluation algorithm implemented in MLK.

```
// compute the norm
norm = MatrixNormI(Matrix, A.N);
//PLU factorization
dgetrf(Size,Size,Matrix,Lda,ipiv,Info);
// condition number 1 norm
ANorm = &norm;
dgecon("O",Size,Matrix,Lda,ANorm,Rcond,work,iwork,Info);
cond = 1.0/(rcond);

delete [] Matrix;
delete [] ipiv;
delete [] work;
delete [] iwork;
return cond;
```

ILU(0) algorithm implementation run analysis

- The condition number for a preconditioned matrix is computed in a similar way.
- The only difference is that the initial matrix has to be multiplied by the matrix inverse to A’ = L’ U’.
- One can solve two triangular systems instead of finding the inverse matrix A’. This functionality is implemented in **mkl_dcsrsm.**

Task 3

- Implement the preconditioned matrix condition number search.

Comparison of condition numbers for symmetric and non-symmetric matrices

Matrix	matrix dimension	Matrix condition number A	Matrix condition number	condition numbers ratio
bcsstk01	48	1597600	231583	6,90
bcsstk05	153	35319	5869	6,02
bcsstk10	1 086	1318823	1499780	0,88

Matrix	matrix dimension	Matrix condition number, A	Matrix condition number, M^-1 * A	condition numbers ratio
fs_541_1	541	1060	1	1060,00
ex22	839	61837	18969	3,26
sherman2	1 080	1,68E+12	3869118	434334,24

ILU(P) SOFTWARE IMPLEMENTATION

An approach to ILU(p) matrix pattern search

- When the preconditioner quality is poor, it may be improved.
- One of the ways to improve the *ILU* algorithm quality is to construct a factor which is closed to the matrix factor (i. e. a certain factor filling is allowable compared to *ILU*(0)).
- Let us study one of these algorithms, the *ILU(p)* factorization method based on the idea of *p filling level*.

An approach to ILU(p) matrix pattern search

- The classical ILU(p) searching algorithm pseudocode

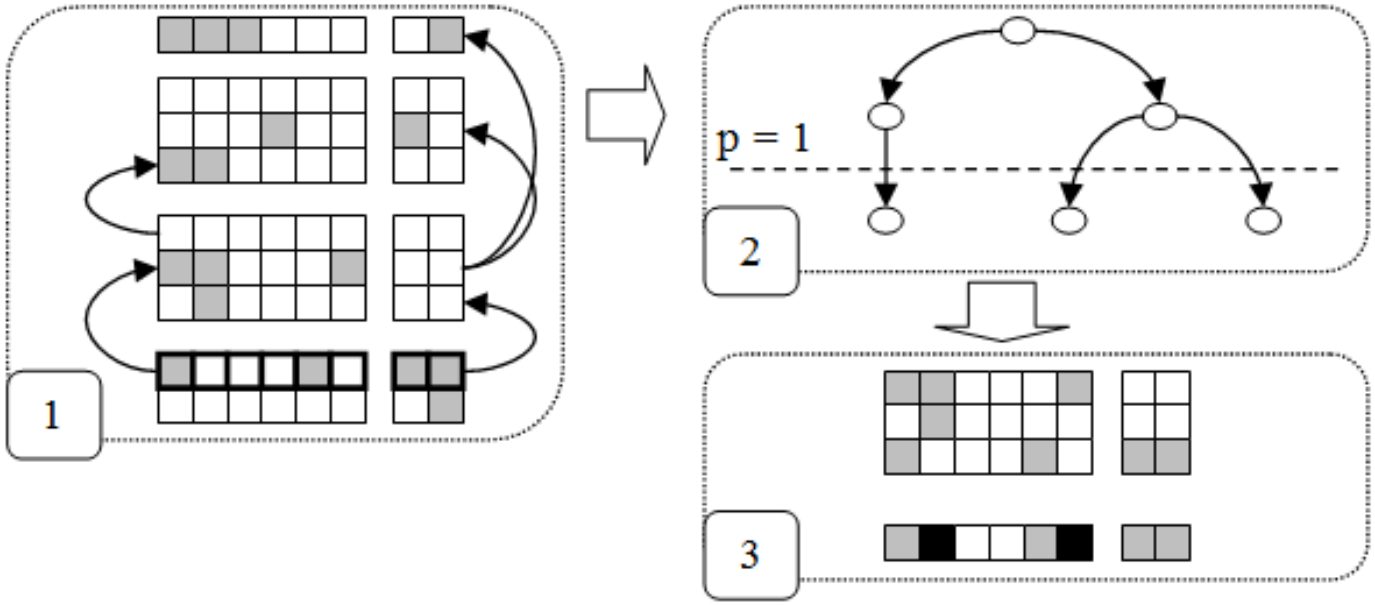
For all non zero element $a_{i,j}$ $Level[a_{i,j}] = 0$
For i = 2 **to** n **do**
Begin
  For k = 1 to (i - 1) and **For** $Level[a_{i,j}] < p$ do
  Begin
    $a_{i,k} = \frac{a_{i,k}}{a_{k,k}}$
    For j = (k + 1) to n do
    Begin
      $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
      $Level[a_{i,j}] = \min\{Level[a_{i,j}], Level[a_{i,k}] + Level[a_{k,j}] + 1\}$
    End
    Заменить на ноль,
    все элементы в строке где $Level[a_{i,j}] > p$
  End
End

An approach to ILU(p) matrix pattern search

- In practice, implementation of this algorithm is very inefficient.
  - The algorithm must compute all elements of the complete factor and only after that reject elements whose filling level exceeds *p*.
  - The algorithm does not let asses the required memory before its completion.
  - ***The array with levels must be stored either for all elements or in the dynamic data structures.***

An approach to ILU(p) matrix pattern search

- Hysom D. and Pothen A. proposed an ILU(p) algorithm modification free from the mentioned disadvantages.
  - The algorithm provides for accurate calculation of nonzeroes in the row and row structure without storing all the information about levels.

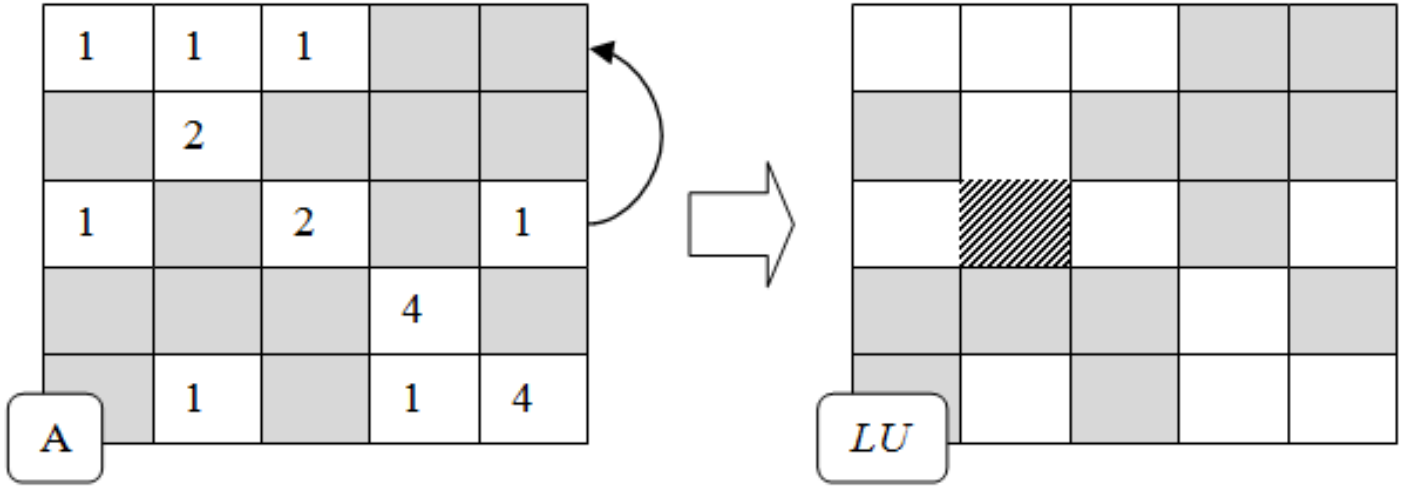


An approach to ILU(p) matrix pattern search

- For incomplete LU-factorization, use the matrix graph.
- Based on the matrix graph, an exclusive tree is formed.
  - The easiest way to form an exclusive tree is graph traversal without reentering the nodes.
- To obtain *ILU*(p) all the exclusive tree nodes whose height exceed *p* are rejected.
- The reduced exclusive tree lets construct the matrix factor row pattern. The pattern results from merging the patterns of rows that correspond to the exclusive tree nodes.

An approach to ILU(p) matrix pattern search

- Let us illustrate the algorithm by an example.



- The element added as a result of ILU(1) factor pattern search is cross-hatched.

An approach to ILU(p) matrix pattern search

- Exclusive tree-based row structure search algorithm

```
Создать массив Level из n элементов
Создать массив Visited из n элементов
Создать пустую очередь Q
Q.push(i) // Поместить в очередь номер текущей строки
For j = 1 to n do
Begin
  Level  [j] = 0
  Visited[j] = -1
End
While !(Q.isEmpty()) do
Begin
  h <- Q.shift()
  For t ∈ adj[h] and Visited[j] <> i do
  Begin
    Visited[t] = i
    if t > i and Level[h] < i then
    Begin
      Q.push(t)
      Level[t] = Level[h] + 1
    End
    if t < i then
      добавить в структуру строки t
  End
End
```

Software implementation of the ILU(p) symbolic phase

```
/**
 * API
 *   int symbolicILUp(int p, int n, int * col, int * row,
 *                    int * lucol, int * lurow,
                      double * &luval,
 *                    int * uptr, int &countL, int &countU);
 *   ILU(p) symbolic phase
 * INPUT
 *   int    n     - matrix dimension
 *   matrix A
 *   int  * col   - column indices of the matrix a
 *   int  * row   - row prefixes of the matrix a
 *   matrix LU pattern
 *   int  * &lucol - column indices of the matrix lu
 *   int  * &lurow - row prefixes of the matrix lu
 *   double * &luval - matrix values
 *   int  * uptr  - indices of diagonal elements
 *                  in the luval array
 * OUTPUT
 *   double * luval - values of the resolved matrices L and U
 *   int &countL    - L matrix dimension
 *   int &countU    - U matrix dimension
 * RETURN
 *   an error code returns
 **/
int symbolicILUp(int p, int n, int * col, int * row,
                 int * &lucol, int * &lurow,
                 double * &luval,
                 int * uptr, int &countL, int &countU);
```

Task 4

- Implement the symbolic *ILU*(p) factor search.

Software implementation of the ILU(p) symbolic phase (1)

- Proceed to implementation of the symbolic part of the algorithm.

```
int symbolicILUp(int p, int n, int * col, int * row,
                 int * &lucol, int * &lurow, double *
   &luval,
                 int * uptr, int &countL, int &countU)
{
```

- Start implementation by declaring the necessary variables.

```
  int i, j, h, s, f;    // cycle counters
  int jcol;             // and temporary variables
  int * len;
  int * visited;
  len = new int[n];
  adj = new int[n];
  visited = new int[n];
  countL = 0;
  countU = 0;
```

Software implementation of the ILU(p) symbolic phase (2)

- For computational convenience, calculate the number of added nonzeroes in the *adj* array for each row.

```
int * adj;
```

- The algorithm requires a queue. We will use the queue implementation from STL.

```
queue<int> Q;
```

- To ensure algorithm consistency, initialize the values of the revisited nodes array and the number of added nodes.
  - As it can be seen from the algorithm, the complete array initialization is required only once.

```
for(j = 0; j < n; j++)
{
  visited[j] = -1;
  adj[j] = 0;
}
```

Software implementation of the ILU(p) symbolic phase (3)

- Then, count the number of nonzeroes for each row according to the algorithm described above.

```
for(i = 0; i < n; i++)
{
  Q.push(i); len[i] = 0; visited[i] = i;
  while(!(Q.empty()))
  {
    h = Q.front(); Q.pop();
    s = row[h];  f = row[h + 1];
    for(j = s; j < f; j++) {
      jcol = col[j];
      if(visited[jcol] != i) {
        visited[jcol] = i;
        if((jcol > i) && (len[h]<p)) {
          Q.push(jcol);
          len[jcol] = len[h] + 1;
        }
        if(jcol < i) {
          countL++;
          adj[i]++;
        }
      }
    }
  }
}
```

Software implementation of the ILU(p) symbolic phase (4)

- Then one can calculate nonzeroes for *L* and *U*. At the same time, one can calculate the values of the array resolving the factor into the matrices *L* and *U*.

```
for(i = 0; i < n; i++)
{
  s = row[i];
  f = row[i + 1];
  for(j = s; (j < f) && (col[j] < i); j++);
  uptr[i] = j;
  if(col[uptr[i]] != i)
  {
    return -(i + 1);
  }
  countU += (f - j);
  adj[i] += (f - j);
}
```

Software implementation of the ILU(p) symbolic phase (5)

- Then allocate enough memory to store the factor.

```
if(lucol != NULL)
{
  delete []lucol;
}
if(lurow != NULL)
{
  delete []lurow;
}
if(luval != NULL)
{
  delete []luval;
}

lucol = new int[countL + countU];
lurow = new int[n + 1];
luval = new double[countL + countU];
```

Software implementation of the ILU(p) symbolic phase (6)

- Initialize the dedicated arrays.

```
memset(luval, 0, (countL + countU) * sizeof
(double));
lurow[0] = 0;
for(i = 0; i < n; i++)
{
  lurow[i + 1] = lurow[i] + adj[i];
  adj[i] = 0;
}
```

Software implementation of the ILU(p) symbolic phase (7)

- To search for the factor structure, use the factor row pattern search algorithm for each row.

```
for(i = 0; i < n; i++)
{
  Q.push(i); len[i] = 0; visited[i] = i;
  while(!(Q.empty()))
  {
    h = Q.front(); Q.pop();
    s = row[h]; f = row[h + 1];
    for(j = s; j < f; j++)
    {
      jcol = col[j];
      if(visited[jcol] != i) {
        visited[jcol] = i;
        if((jcol > i) && (len[h]<p)) {
          Q.push(jcol);
          len[jcol] = len[h] + 1;
        }
        if(jcol < i) {
```

Software implementation of the ILU(p) symbolic phase (8)

- It is important to note that column indices in this code area are added to the end, so they may not be sorted.

```
            lucol[lurow[i] + adj[i]] = jcol;
            adj[i]++;
          }
        }
      }
    }

    s = uptr[i];
    f = row[i + 1];
    uptr[i] = lurow[i] + adj[i];
    for(j = s; j < f; j++)
    {
      lucol[lurow[i] + adj[i]] = col[j];
      adj[i]++;
    }
  }
```

Software implementation of the ILU(p) symbolic phase (9)

- To sort the factor pattern, perform double transposition.

```
  int *tCol;
  int *tRow;
  StructTranspose(n, lucol, lurow, tCol, tRow);
  delete []lucol;
  delete []lurow;
  StructTranspose(n, tCol, tRow, lucol, lurow);
  delete []tCol;
  delete []tRow;

  delete[] len;
  delete[] adj;

  return ILU_OK;
}
```

Software implementation of the ILU(p) numerical phase

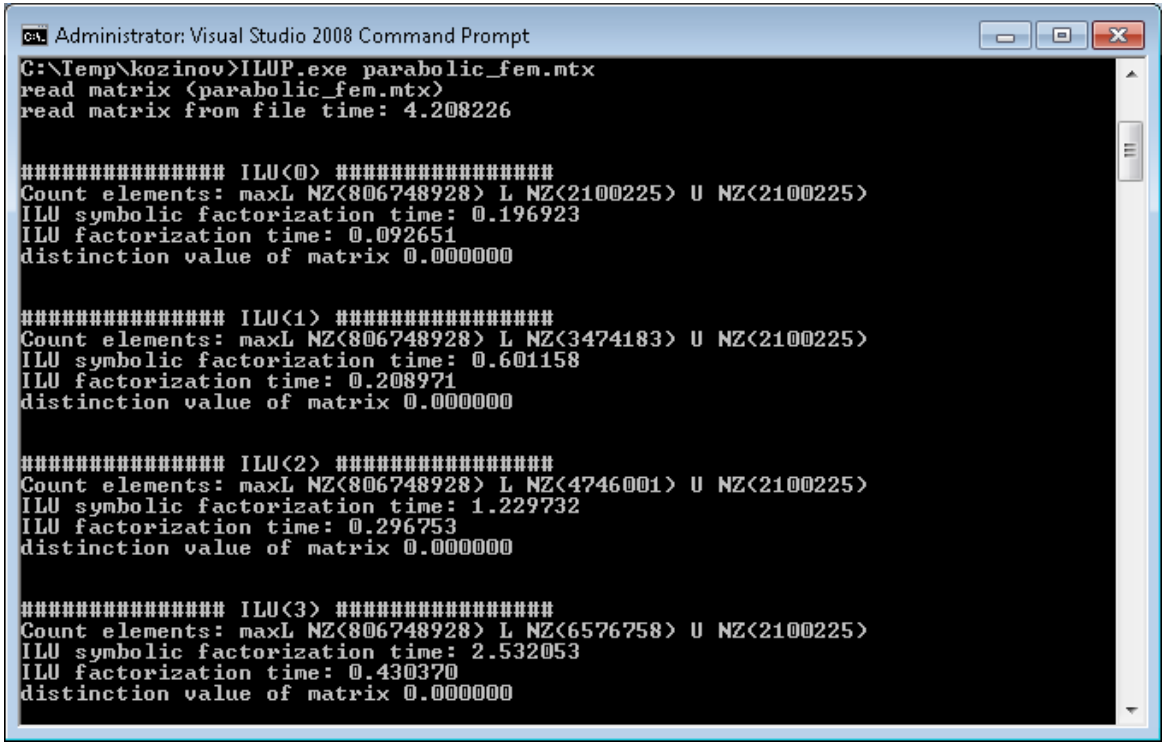
```
/**
* API
*   int numericalILUp(int n, double * a, int * col, int * row,
*                     int * lucol, int * lurow, int * uptr, double * luval);
*   ILU(p) numerical phase
* INPUT
*   int    n     - matrix dimension
*   double * a   - nonzero elements
*   int  * col   - column indices of the matrix a
*   int  * row   - row prefixes of the matrix a
*   int  * lucol - column indices of the matrix lu
*   int  * lurow - row prefixes of the matrix lu
*   int  * uptr  - indices of diagonal elements
*                  in the luval array
* OUTPUT
*   double * luval - values of the resolved matrices L and U
* RETURN
*   an error code returns
*   0        - factorization is successful
*   -(n + 1) - number of row where the diagonal has a 0
**/
int numericalILUp(int n, double * a, int * col, int * row,
                  int * lucol, int * lurow, int * uptr, double * luval)
```

Task 5

- Implementation of the *ILU*(p) numerical part is almost similar to that of *ILU*(0). Implement this part of algorithm on your own.

ILU(p) algorithm implementation run analysis

- Run the developed *ILU*(p) software implementation for several *p* values.



Runtime depending on the level for symmetric matrices

Matrix	matrix dimension	p	Symbolic phase	Numerical phase	total time
bcsstk01	48	0	0,000027	0,000016	0,000043
		1	0,000077	0,000029	0,000106
		2	0,000159	0,000040	0,000199
		3	0,000229	0,000045	0,000274
bcsstk05	153	0	0,0001	0,0001	0,0002
		1	0,0005	0,0002	0,0006
		2	0,0006	0,0001	0,0007
		3	0,0009	0,0001	0,0010
bcsstk10	1086	0	0,0006	0,0005	0,0011
		1	0,0022	0,0008	0,0030
		2	0,0046	0,0009	0,0055
		3	0,0071	0,0010	0,0081
bcsstk13	2003	0	0,00	0,01	0,01
		1	0,02	0,01	0,03
		2	0,05	0,02	0,07
		3	0,11	0,02	0,14
parabolic_fem	525825	0	0,17	0,08	0,26
		1	0,51	0,18	0,68
		2	1,28	0,26	1,55
		3	2,35	0,37	2,72
tmt_sym	726713	0	0,20	0,08	0,28
		1	0,50	0,10	0,60
		2	1,18	0,15	1,33
		3	2,12	0,18	2,30

Runtime depending on the level for non-symmetric matrices

Matrix	matrix dimension	p	Symbolic phase	Numerical phase	total time
fs_541_1	541	0	0,0001	0,0001	0,0002
		1	0,0003	0,0001	0,0004
		2	0,0008	0,0002	0,0010
		3	0,0018	0,0003	0,0021
ex22	839	0	0,001	0,001	0,002
		1	0,003	0,001	0,004
		2	0,008	0,002	0,009
		3	0,013	0,002	0,015
sherman2	1080	0	0,001	0,001	0,001
		1	0,003	0,002	0,004
		2	0,008	0,002	0,010
		3	0,016	0,004	0,020
cage10	11397	0	0,005	0,004	0,009
		1	0,026	0,016	0,042
		2	0,142	0,059	0,201
		3	0,509	0,182	0,692

ILU(p) algorithm implementation run analysis

- The tables show that the higher is the level, the longer is the algorithm runtime.
  - Remaining time will be used for matrix pattern search
- This disadvantage of the classical algorithm may be remedied, for example, by means of more complex algorithms that analyze factor using complete blocks instead of a single row.

ILU(p) performance quality for symmetric matrices depending on the level

Matrix	matrix dimension	p	Condition number A	Condition number $M^{-1}*A$	Condition number reduction
bcsstk01	48	0	1597600	231583	6,8
		1		69385	23,0
		2		66197	24,1
		3		59113	27,0
bcsstk05	153	0	35319	5869	6,0
		1		3977	8,9
		2		4287	8,2
		3		4278	8,3
bcsstk10	1086	0	1318823	1499780	0,9
		1		123576	10,6
		2		258998	5,1
		3		758553	1,7

ILU(p) performance quality for non-symmetric matrices depending on the level

Matrix	matrix dimension	p	Condition number A	Condition number $M^{-1}*A$	Condition number reduction
ex22	839	0	61837	18969	3,3
		1		17808	3,5
		2		23713	2,6
		3		25986	2,4
sherman2	1080	0	1,68E+12	3869118	434334,2
		1		3975617	422699,3
		2		3951175	425314,1
		3		3932957	427284,2

ILU(p) algorithm implementation run analysis

- Experimental results listed in the tables show that the level increase can influence the preconditioner quality in various ways.
  - The preconditioner quality can either gradually improve (see bcsstk01) or change in an unpredictable way (see bcsstk10).
- The main reason is that the algorithm accounts only for the pattern of the obtained factor. The algorithm takes no account of the values rejected in the course of factorization.
  - For due account of the values added to the matrix factors, other euristic approached are used, such as *ILUT*.

Added tasks

1. Implement a parallel ILU(p) algorithm version and analyze its scalability.
2. Implement the ILU – ILUT modification.
3. Implement a block modification of the ILU(p) algorithm to increase the algorithm efficiency for large matrices.

References

1. Saad Y. Iterative methods for sparse linear systems. – SIAM, 2003.
2. Hysom D., Pothen A. Level-based Incomplete LU Factorization: Graph Model and Algorithms // SIAM Journal On Matrix Analysis and Applications, nov. 2002.
3. Laboratory Work on Multiplication of Matrices

Questions

- ???