|  |  |
| --- | --- |
|  |  |

The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program   
of the Lobachevsky State University of Nizhni Novgorod   
among the world's research and education centers

Strategic initiative  
“Achieving leading positions in the field of supercomputer technology   
and high-performance computing”

**DIRECT METHODS   
FOR SOLVING SYSTEM OF LINEAR EQUATIONS**

*Practice 3. Algorithmic optimization in sparse algebra problems   
by example of matrix multiplication*

Nizhni Novgorod

2014

# Objectives

The objective of this work is to investigate basic principles of sparse matrices storage and some processing algorithms for them.

# Abstract

The work is organized in the following way: the sparse matrix multiplication problem is stated, the sparse matrices storage questions are discussed. A test problem generator is described; some considerations of choosing such problems are stated. A sequential software implementation for computing CRS(CSR)-format matrices multiplication by definition is described. The transposition algorithms are stated. The experiments results and efficiency analysis are demonstrated. Algorithmic optimization is performed. Another matrix multiplication algorithm with better timing characteristics is implemented. A widespread idea of dividing sparse algebra algorithms into two phases – symbolic and numerical – is worded. This division is demonstrated by example of the fastest algorithm discussed in the work. The parallel implementations of the final algorithm using OpenMP and TBB including load balancing are considered.

# GUIDELINES

In the introduction a sparse matrix notion used in different sources is considered, the fields in which sparse algebra problems arise are discussed. Then the problem of multiplying two square sparse matrices *A* and *B* of size *N*×*N* with elements of type **double** is stated. A resulting matrix *C* is also sparse.

Two variants of sparse matrix storage – the coordinate format and Compressed Row Storage (CRS) format with its modifications – are considered in the work. The CRS format is applied when implementing the algorithms.

Then a question of choosing test problems so that they provide the result reproducibility and representativeness is discussed. The generators for matrices of two classes are used in the work: 1) with a regular pattern – matrices containing at most *K* randomly placed nonzero elements in each row; 2) with an irregular pattern – matrices containing 1 to *K* randomly placed nonzero elements in each row and the number of nonzero elements increases according to a cubic law from one row to another.

The function **mkl\_dcsradd()**of the MKL library is interpreted as a model version for matrix multiplication. In the tests, a product of a matrix *A* with a regular pattern and a matrix *B* with an irregular pattern is computed.

This work considers several sparse matrix multiplication algorithms – by definition, by definition using data orderliness, by definition using optimized scalar multiplication, two-phase optimized version. The description of the Gustavson algorithm is provided. For the implementations efficiency analysis the Intel VTune Amplifier XE tool is used.

When investigating the matrix multiplication algorithm by definition, the implementation problems concerning the matrix sparseness are discussed – laboriousness of extracting a column when storing the matrix in the CRS format, necessity of sequential filling of the matrix *C* to avoid repacking, occurrence of zero elements in the matrix *C* as a result of the computations. As a solution, it is proposed to transpose the matrix *B* preliminarily and fill the matrix *C* sequentially, computing the scalar products of each row of the matrix *A* and each row of the matrix *ВТ*. When multiplying the rows, the columns indices are matched in order to determine the pairs of the nonzero elements. An algorithm from [1] is applied to transpose a sparse matrix. In the base matrix multiplication implementation a value of an element of the matrix *C* is put into the storage structure if it is not equal to zero. Since the number of nonzero elements in the matrix *C* is not known beforehand, vectors of the STL library are used for storing them.

When comparing the results of the base algorithm version with the MKL-implementation, a lag of the order of about 3 was revealed. A program profile analysis shows that the most time-consuming operation in the base version is comparison of the columns indices for the elements of the matrices *A* and *ВТ* when computing scalar product. As an algorithmic optimization it is proposed to use the fact that the elements of each row of the multiplied matrices are stored orderly by the number of column. Thanks to this, it is possible to apply an analog of the ordered arrays merge algorithm in the scalar multiplication. Such optimization enables multiplication time to be reduced in 7 times. A disadvantage of this approach is branch misprediction occurring when computing the scalar multiplication, i.e. the situations when a compiler mispredicts the next instruction to process in conditional jumps. This does not allow to take into account the processor architecture features.

For further optimization it is offered to implement a scalar multiplication algorithm from the book [1]. This algorithm uses an auxiliary array of size *N* for storing the indices of nonzero elements in a matrix *A* row. This approach makes it possible to reduce calculation time in 3 more times.

Next an experiment is carried out, which demonstrates that knowledge of the additional information about the problem often enables to overtake the implementation optimized for the general case. In order to do this, a product of the matrix *B* with an irregular structure and the matrix *A* with a regular structure is computed. In such experiment the runtime of the optimized version and MKL library function increases in several times in comparison with the previous results since the scalar multiplication algorithm is no longer efficient. For the purpose of further algorithm optimization, a simple heuristics is proposed to apply – to compute either *С*=*ABT* or *СТ*=*BTА* with further matrix *C* transposition, depending on the number of nonzero elements in the matrices *A* and *B*. This approach enabled the authors to reduce the program execution time and overtake the MKL library function.

After that, a case is considered when single-type sparse matrix operations are to be performed many times and a matrix pattern does not change from one iteration to another. In this case calculations are divided into two phases – symbolic and numerical. In the given problem a pattern of the resulting matrix *C* is constructed during the symbolic phase. In the work it is proposed to use the final optimized multiplication version without actual elements computation for the symbolic part.

Then a description and the experiments results for the wide-used-in-practice Gustavson algorithm are given. This algorithm is based on computing a product of every element of the matrix *A* and all the elements of the corresponding row of the matrix *B* in order to accumulate the partial sums step-by-step. The experiments show that the Gustavson algorithm works in times faster than the multiplication algorithm discussed in the given work.

The parallel implementations for the optimized multiplication algorithm using the technologies OpenMP and TBB are considered in the work. Their balancing is discussed as well. An algorithm distributing loop iterations among the threads by rows of the matrix *A* is proposed for this purpose.

In the OpenMP-implementation the directive **pragma omp for** is used for loop parallelization. To avoid threads conflicts of accessing shared data, the arrays **columns** and **values** of the matrix *C* are duplicated according to the number of the rows and the auxiliary array is declared local for each thread. Upon completion of the computations, the structure of the matrix *C* is merged. Such implementation makes it possible to achieve a speedup of 2.5 on 8 threads for the matrices of order 10 000. The profiling demonstrates that the main implementation disadvantage is the great wait time relative to the calculation time. Such waiting occurs due to the small amount of computations assigned to a certain thread. This is confirmed by the experiments on the matrices of great size with rising computational load. Thus, a speedup of 5.3 is achieved on 8 threads for the matrices of order 40 000. For load balancing a static scheme is used in this version. This makes it possible to increase speedup up to 6.9 on 8 threads for the matrices of order 40 000.

In the TBB-implementation the template function **parallel\_for()** on the one-dimensional iteration space blocked\_range is used for loop parallelization. The shared data is accessed as in the OpenMP-version. The base computations are performed in the method operate() of the function class Multuplicator that takes two matrices as input and operates the vectors columns, values and row\_index. The experiments results show a speedup of 7,6 on 8 threads for the matrices of order 40 000. This indicates that load balancing among threads and code optimization is performed better in the TBB-version than in the OpenMP-version. The reader is offered to determine an optimal size for the calculation portion assigned to a certain thread by himself.

# RECOMMENDATIONS for students

Notwithstanding the long-standing edition, the book [1] has not lost its relevance today. The main sparse matrix storage formats allowing the efficient implementation of the standard matrix algorithms are considered in the book. Particularly, the matrix multiplication problem is discussed in detail.

The technological basis of the laboratory work is worded in [2,3]. The description of the technologies OpenMP and TBB is available on the Internet [5, 6]; the Intel MKL library description is also available [6].

# References

1. Sergio Pissanetzky. Sparse matrix technology. Academic Press, 1984.
2. Intel® ThreadingBuilding Blocks. Reference Manual. Version 1.6. Intel® Corporation, 2007.
3. Intel® Threading Building Blocks. Tutorial. Version 1.6. Intel® Corporation, 2007.
4. Official Website OpenMP [www.openmp.org].
5. TBB library page on the Intel Corporation Website [http://software.intel.com/en-us/articles/intel-tbb/].
6. MKL library page on the Intel® Corporation Website [http://software.intel.com/en-us/articles/intel-mkl/].

# EXERCISES

1. Carry out the experiments on multiplication of the matrices in the CCS format. Reveal and explain the effects concerning correlation of the execution times for the different sequential algorithms. Compare with the base version presented in the work (matrices in the CRS format). Develop and set up a parallel implementation.
2. Carry out the experiments on multiplication of the matrices in the coordinate format. Reveal and explain the effects concerning correlation of the execution times for the different sequential algorithms. Compare with the base version presented in the work (matrices in the CRS format). Develop and set up a parallel implementation.
3. Carry out the experiments on multiplication of the matrices of another structure. Consider matrices with the same number of elements per row. Reveal and explain the effects concerning correlation of the execution times for the different sequential algorithms. Develop and set up a parallel implementation.
4. Adapt the algorithms used in the work to rectangular matrices. Develop a software implementation. Carry out the computational experiments.

# TEST QUESTIONS

1. What data does the element **RowIndeх[i]** store when using the CRS format?
   1. The number of the row of the **i**-th nonzero matrix element
   2. + The starting index of the elements of the **i**-th row in thearray **Values**
   3. The starting index of the elements of the **i**-th column in thearray **Values**
2. What function of the MKL library performs addition of two sparse matrices with the elements of **double** type?
   1. + **mkl\_dcsradd()**
   2. **mkl\_scsradd()**
   3. **mkl\_dcrsblasadd(**)
3. What mode of the IntelVTuneAmplifier XE tool should be used to profile the work of the program call stack?
   1. BasicHotspotsAnalysis
   2. IntermediateHotspotsAnalysis
   3. + Advanced HotspotsAnalysis
4. What is the name of the metric of the IntelVTuneAmplifier XE tool, which counts the number of cycles per instruction?
   1. + CPI
   2. Instructions retired
   3. Instructions frequency
5. What operation was the most time-consuming in the naive algorithm version?
   1. + Comparison of the indices of the matrices А and В elements
   2. Addition of the elements of the matrices А and В
   3. Matrix zeroing
6. What is the core idea of the first matrix multiplication algorithmic optimization?
   1. Using of matrix transposition
   2. Branching prediction
   3. + Using of arrays orderliness
7. What is the function of the matrix multiplication symbolic part?
   1. + Construction of the matrix product pattern, memory allocation for the matrix product
   2. Calculation of the elements of the matrix product
   3. Calculation of the necessary memory volume for the matrix product without constructing its pattern
8. What is the main idea of the Gustavson algorithm?
   1. Using of double transposition to order the pattern of the matrix C
   2. Calculation of each element of the matrix C as a product of a matrix A row and a matrix BT row
   3. + Calculation of product of each matrix A element and all the elements of the corresponding matrix B row
9. What mode of the IntelVTuneAmplifier XE tool should be used to obtain information about program parallelism quality?
   1. + Concurrency
   2. Advanced Hotspots
   3. Locks and Waits
10. What parallelization scheme is used for matrix multiplication?
    1. + Loop parallelization by rows of the matrix А
    2. Loop parallelization by rows of the matrix В
    3. Loop parallelization by columns of the matrix ВТ
11. What parameters does the OpenMP-directive **for** need to divide loop iterations among the threads into the blocks of size **chunk** before the start of the loop?
    1. **schedule(chunk, static)**
    2. **schedule(dynamic, chunk)**
    3. + **schedule(static, chunk)**
12. What is passed as a mandatory argument to the TBB-function **parallel\_for**?
    1. + A loop iteration space and object of a function class
    2. An object of a function class
    3. A loop iteration space and pointer to the method **operator()** of a function class