2011 11th IEEE International Conference on Computer and Information Technology

Open Multi Processing (OpenMP) of Gauss-Jordan Method for Solving System of Linear Equations

Panagiotis D. Michailidis
Department of Balkan Studies
University of Western Macedonia
Florina, Greece
Email: pmichailidis@uowm.gr

Konstantinos G. Margaritis
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
Email: kmarg@uom.gr

***Abstract*—Solving systems of linear equations is probably one of the most scientific applications of linear algebra and direct-based Gauss-Jordan method as a classical kernel of large system of linear equations has become the focus of research. This paper presents an OpenMP pipeline implementation of Gauss-Jordan method and the corresponding performance model. Then, we conduct an experimental evaluation of the pipeline implementation in comparison with the two other naive parallel versions of Gauss-Jordan method such as row block and row cyclic distribution on a multicore platform. From the experiments are obtained show that the proposed pipelined implementation is a good solution for solving large system of linear equations. Finally, the experimental results of the pipeline implementation confirm the proposed performance model.**

***Keywords*-Linear algebra; Gauss-Jordan; multicore; OpenMP; parallel algorithms;**

I. Introduction

The solving of dense linear systems is an important scientific problem that is used as a kernel in many scientific disciplines ranging from computer science, data analysis to computational economics.

There are many methods for solving a linear system such as direct and iterative methods. In this paper we study the parallelization based on the direct method for solving a linear system. We focus on the Gauss-Jordan method as a kernel can be used to solve the above scientific problem.

In the literature have been studied many parallelization schemes of Gauss-Jordan method on modern high performance computer systems such as multicomputer (i.e. cluster of workstations) and multicore systems (i.e. dual/quad cores) using standard interfaces such as MPI, or parallel extension of language C/C++ such as OpenMP. Serge shows its version adaptive to MIMD [1], N.Melab et al give us its version suiting for MARS which is a kind of network of workstations [2], Aouad et al and Shang et al present its intra-step parallel version on grid platforms [3], [4], [5], [6] using a YML framework [7]. Further, experimental results of Gauss-Jordan on cluster and grid computing environments using MPI library are presented in [8] and [9] respectively. On the other hand, there are implementations on multicore for other similar problems of linear algebra such as factorization algorithms for solving systems of equations (LU, QR and Cholesky) [10], [11], [12], [13], [14]. We recently developed an implementation of the pipeline technique for the LU factorization algorithm using OpenMP [15]. However, there are not research efforts in implementation of Gauss-Jordan method on multicore using OpenMP interface.

The goal of the paper is to implement the Gauss-Jordan method using pipelining technique in OpenMP and compare the performance of the proposed implementation to the two strategies that are based on naive parallelizations of the original algorithm (such as row block and row cyclic data distribution) on a multicore platform. We must note that the implementation of the pipelining technique in OpenMP for different application domains have not been reported in past research literature. We also propose a mathematical model of performance for the pipelined implementation of the Gauss-Jordan method and we verify this model with extensive experimental measurements on multicore platform.

The rest of the paper is organized as follows: In Section II, the Gauss-Jordan method is outlined for solving a linear system. In Section III, the pipelined implementation and the corresponding performance model are discussed. In Section V, experimental results are presented. Finally, some conclusions are drawn in last section.

II. Gauss-Jordan Method

Firstly, we describe the known Gauss-Jordan method for solving a system of linear equations. Consider the following real linear algebraic system $Ax = b$, where $A = (a_{ij})_{n\times n}$ is a known nonsingular $n \times n$ matrix with nonzero diagonal entries, $b = (b_0, b_1, ..., b_{n-1})^T$ is the right-hand side and $x = (x_0, x_1, ..., x_{n-1})^T$ is the vector of the unknowns.

In Gauss-Jordan algorithm, first a working matrix is constructed by augmenting A matrix with b, obtaining $(A|B)$ matrix with n rows and $n + 1$ columns. Then, this matrix is transformed into a diagonal form, using Gaussian elimination. Gauss-Jordan algorithm is executed in two phases. In the first phase of Gauss-Jordan algorithm, the augmented matrix is transformed into a diagonal form in which the

978-0-7695-4388-8/11 $26.00 © 2011 IEEE
DOI 10.1109/CIT.2011.47

elements both above and below the diagonal element of a given column are zero. In the second phase, each x_i $(0 \leq i \leq n)$ solution is computed by dividing the element from row i and column n of the augmented matrix $(a_{i,n})$ with the element from the row i of the principal diagonal $(a_{i,i})$. The serial version of general Gauss-Jordan algorithm for solving a linear system shown in Algorithm 1 consists of three nested loops, which we will adopt for parallel implementation in the remainder of this paper.

Algorithm 1: Sequential Gauss - Jordan algorithm

```
for k ← 0 to n − 1 do
    for i ← 0 to n − 1 do
        if k != i then
            for j ← k + 1 to n do
                a[i][j] ←
                a[i][j] − (a[i][k]/a[k][k]) ∗ a[k][j]
for i ← 0 to n − 1 do
    x[i] ← a[i][n]/a[i][i]
```

The transformation of augmented matrix to diagonal form requires $4n^3/3$ scalar arithmetic operations. Computing solution from diagonal form of the system requires approximately n scalar arithmetic operations, so that the sequential run time of Gauss-Jordan algorithm is $4n^3/3 + n$.

III. Pipelined Implementation and Performance Model

In this section, we present an OpenMP pipelined implementation of Gauss-Jordan method using the queue data structure and we also present a performance model.

A. Implementation

We assume that the rows of matrix A are distributed among the p threads or cores such that each core is assigned $\lceil \frac{n}{p} \rceil$ contiguous rows of the matrix. The general idea of a pipelining algorithm is that each thread executes the n successive steps of Gauss-Jordan algorithm on the rows that it holds. To do so, it must receive the index of the pivot row, send it immediately to the next thread and then proceed with the steps of Gauss-Jordan method. Generally, the parallel algorithm is as follows:
For $k = my\_rank * \frac{n}{p}, ..., my\_rank * \frac{n}{p} + \frac{n}{p}$, do in each thread T_{my_rank}, where my_rank is the rank of the thread and are numbered from 0 to $p - 1$:

1) Receive the index of the kth row from thread T_{my_rank}.
2) Send the index of the kth row just received to the thread T_{my_rank+1}.
3) Perform the elimination step number k for all rows m, that belong to thread T_{my_rank}.

The implementation of the receive and send operations in OpenMP can be realized with the help of the proposed `Get()` and `Put()` procedures, respectively. The `Get()` procedure can take as an argument the rank of the thread which receives the data and it returns a data element, whereas procedure `Put()` can take two arguments: one is for the rank of thread which will send the data and the other is a data element. The procedures `Get()` and `Put()` were implemented using a queue data structure. However, for the further internal details of the `Get()` and `Put()` procedures, the reader is referred in the paper [15]. Algorithm 2, which is called Pipe, shows an OpenMP parallel pipeline algorithm.

Algorithm 2: (Pipe) OpenMP pipelining algorithm of Gauss-Jordan

```
#pragma omp parallel private(k, i, j, row) shared(a)
{
long my_rank = omp_get_thread_num();
bsize = n/p;
if (my_rank != 0) then
    for k ← (my_rank ∗ bsize) to
    (my_rank ∗ bsize) + bsize do
        row = Get(my_rank);
        Put(my_rank + 1, row);
        /* Elimination step */
        for i ← 0 to k + bsize do
            if row != i then
                for j ← k to n + 1 do
                    a[i][j]− =
                    (a[i][row]/a[row][row]) ∗
                    a[row][j]);
else
    for k ← (my_rank ∗ bsize) to
    (my_rank ∗ bsize) + bsize do
        Put(my_rank + 1, k);
        /* Elimination step */
        for i ← 0 to k + bsize do
            if k != i then
                for j ← k + 1 to n + 1 do
                    a[i][j]− = (a[i][k]/a[k][k])∗a[k][j]);
}
```

B. Performance Analysis

The performance model of the pipeline algorithm depends on two main aspects: the computational cost and the communication cost. In the case of multicore, communications are performed through direct `Put()/Get()` operations by two or more threads. An analytical model based on the number of operations and their cost in CPU times can be used to determine the computational cost. Similarly, to predict

the communication cost we calculate the number of get/put operations between threads and measure the number of cycles for put and get operations.

The execution time of the pipeline algorithm can be broken up into two terms:

- T_{elim}: It is the total time to perform the elimination step in parallel. The elimination step involves divisions, multiplications and subtractions. The number of arithmetic operations on a maximally-loaded thread in the kth iteration involves $n/p(n-k-1)$ divisions, multiplications and subtractions. The total operations spent in elimination step in the kth iteration is $2(n/p)(n-k-1)$. Therefore, the total time T_{elim} is given by:

$$T_{elim} = 2(\frac{n}{p})\Sigma_{k=0}^{n-1}(n-k-1)t_{elim} = 2(\frac{n}{p})\frac{n(n-1)}{2}t_{elim} \quad (1)$$

  where t_{elim} is the elimination time for a floating point operation. The elimination time is included the time for division, multiplication and substraction operations.
- T_{comm}: It is the total communication time to exchange elements between threads of the pipeline. The communication step involves the `Put()` and `Get()` operations. The number of communication operations on a maximally-loaded thread in the kth iteration involves one `Get()` and one `Put()` operation. The total operations spent in communication step on a thread is $2(n/p)$. Then, the total time T_{comm} is given by:

$$T_{comm} = 2(\frac{n}{p})t_{comm} \quad (2)$$

  where t_{comm} is the communication (put and get) time for a floating point operation.

The total parallel execution time of the pipeline algorithm, T_{pipe}, using p threads is the summation of the two terms and is given by:

$$T_{pipe}(p) = T_{elim} + T_{comm} \quad (3)$$

IV. Experimental Results

A. System Platform and Experimental Process

For our experimental evaluation we used an Intel Core 2 Quad CPU with four processor cores, a 2.40 GHz clock speed and 8 Gb of memory. The system ran GNU/Linux, kernel version 2.6, for the x84 64 ISA. All programs were implemented in C using OpenMP interface and were compiled using gcc, version 4.4.3, with the "-O2" optimization flag.

Several sets of test matrices were used to evaluate the performance of the parallel algorithms, a set of randomly generated input matrices with sizes ranging from 32×32 to 4096×4096. To compare the parallel algorithms, the practical execution time was used as a measure. Practical execution time is the total time in seconds an algorithm needs to complete the computation, and it was measured using the `omp_get_wtime()` function of OpenMP. To decrease random variation, the execution time was measured as an average of 50 runs.

B. Analysis based on Experimental Results

We must note that we compare the performance of the pipeline implementation to the two naive parallel algorithms are based on the two different data distribution schemes among the available threads such as row block and row cyclic. In row block data distribution, the $n \times n$ coefficient matrix A is block-striped among p threads or cores such that each core is assigned $\lceil \frac{n}{p} \rceil$ contiguous rows of the matrix. This algorithm we call RowBlock. In this row block data distribution we used the static schedule without a specified chunk size that implies the OpenMP divides the iterations into p blocks of equal size of $\lceil \frac{n}{p} \rceil$ and is statically assigned to the threads in a blockwise distribution. The allocation of iterations is done at the beginning of the loop, and each thread will only execute those iterations assigned to it. In row cyclic data distribution, rows of matrix A are distributed among the p threads or cores in a round-robin fashion. This algorithm we call RowCyclic. In this row cyclic data distribution we used the static schedule with a variable specified chunk size that specifies a static distribution of iterations to threads which assign blocks of size bs in a round-robin fashion to the threads available, where bs takes the values 1, 2, 4, 8, 16, 32 and 64 rows.

Firstly, we evaluate the RowCyclic algorithm in order to examine the relation between the execution time of the RowCyclic algorithm and the block size. For this reason, we ran the RowCyclic algorithm for different block sizes. Tables I, II and III present the average execution time of the RowCyclic algorithm for different block sizes on one, two and four cores, respectively. From these Tables we conclude that the execution time of the RowCyclic algorithm is not affected by changes in the block size. This is due to the fact that the RowCyclic method has poor locality of reference. For this reason we have used the RowCyclic algorithm with a block of size 1 for the comparison to the other two parallel algorithms.

Figure 1 shows the average execution time of all parallel algorithms for variable matrix size on one, two and four cores. As can be seen from Figure 1, the execution time of all algorithms is increased as the matrix size is increased. We observe that the parallel implementations execute fast on matrix of small sizes (from 32 to 2048) because these matrices fit entirely in the L1 and L2 cache. For large matrix sizes there is a slowdown in the execution time because the matrices are larger than the cache (which is the more likely case) and this leads to being served primarily by the slow main memory.

Figure 2 presents the way the performance of all parallel algorithms was affected when parallel processed using OpenMP for 1 to 4 threads, for small, medium and large

Table I
EXECUTION TIME (IN SECONDS) OF THE ROW CYCLIC ALGORITHM FOR DIFFERENT BLOCK SIZES AND MATRIX SIZES ON 1 CORE

	1	2	4	8	16	32	64
32	0,00019	0,00019	0,00019	0,00018	0,00019	0,00018	0,00018
64	0,00124	0,00126	0,00126	0,00124	0,00123	0,00123	0,00123
128	0,00978	0,00972	0,0097	0,00968	0,00968	0,00971	0,00959
256	0,07685	0,07637	0,0758	0,07599	0,0754	0,07578	0,07589
512	0,60477	0,60442	0,60342	0,60115	0,60117	0,60118	0,60118
1024	4,89064	4,88948	4,88231	4,87806	4,87231	4,87124	4,88124
2048	38,79325	38,75918	38,79122	38,74208	38,7341	38,7431	38,75068
4096	383,48853	383,5622	383,7488	385,3828	384,642	385,542	385,7211

Table II
EXECUTION TIME (IN SECONDS) OF THE ROW CYCLIC ALGORITHM FOR DIFFERENT BLOCK SIZES AND MATRIX SIZES ON 2 CORES

	1	2	4	8	16	32	64
32	0,0002	0,00018	0,00017	0,00017	0,00017	0,00019	0,00021
64	0,00088	0,00082	0,0008	0,0008	0,0008	0,0009	0,0013
128	0,00556	0,0053	0,00547	0,00527	0,00521	0,00527	0,00522
256	0,04051	0,03928	0,03815	0,03915	0,03821	0,03915	0,03889
512	0,38483	0,31027	0,31002	0,31005	0,30722	0,30123	0,30889
1024	3,06108	2,98264	2,89892	2,46892	2,48892	2,50212	3,01635
2048	24,40194	24,23195	24,42808	24,39808	23,4833	24,39808	21,92683
4096	193,60249	193,41822	193,40811	193,36971	188,44212	189,3831	189,67035

Table III
EXECUTION TIME (IN SECONDS) OF THE ROW CYCLIC ALGORITHM FOR DIFFERENT BLOCK SIZES AND MATRIX SIZES ON 4 CORES

	1	2	4	8	16	32	64
32	0,00018	0,00015	0,00014	0,00014	0,00014	0,00014	0,00021
64	0,00061	0,00055	0,00054	0,00053	0,00055	0,0006	0,00128
128	0,00312	0,00294	0,00279	0,00289	0,00381	0,00339	0,00504
256	0,0211	0,02018	0,02017	0,02002	0,02102	0,02112	0,01959
512	0,16003	0,15382	0,14981	0,15304	0,1521	0,15981	0,15161
1024	1,27937	1,25728	1,2635	1,25835	1,26275	1,25835	1,25478
2048	11,22809	11,13837	11,17514	11,17514	11,1911	11,1489	11,13339
4096	88,02839	87,4224	87,722566	87,85566	88,14321	89,35655	89,30532

matrix sizes. As can be seen, the performance of the algorithms improved with each additional thread. We observe that the performance of the RowBlock and RowCyclic algorithms increased at a decreasing rate. This is due to the fact that the implicit synchronization cost of parallel 'for' loops (i.e. start and stop parallel execution of the threads) dominates the execution time. In this case, the cost of synchronization is about n^2. On the other hand, it is clear the performance of the Pipe algorithm on two and four cores resulted in the approximate doubling and quadrupling of its performance. With the Pipe algorithm, the decrease in performance is much slower than it is with the others. Therefore, we conclude that the parallel execution time of the Pipe algorithm depends on the number of threads, since the total communication and overhead time is much lower than the processing time on each thread. Finally, we expect that the performance of the Pipe algorithm will be better than that of the other two algorithms for large numbers of cores, such as 8 and 16 cores.

As we can see in the results, the ranking of the parallel algorithms is obvious. The parallel algorithm with the best resulting performance is the Pipe, the second best is RowBlock and the third best is RowCyclic.

C. Evaluation of the Performance Model

The final experiment was conducted to verify the correctness of the proposed performance model for the pipeline algorithm. The performance model of equation 3 is plotted in Figure 3 using time parameters. In order to obtain these predicted results, we have determined the time parameters for elimination and communications operations (i.e. t_{elim}, t_{comm}) for different matrix sizes. These parameters of the target machine are determined experimentally with the results shown in Figure 4. More specifically, the t_{elim} and t_{comm} parameters are measured, executing the serial Gauss-Jordan algorithm 50 times for each different matrix size. Then we take the average these 50 runs for each different matrix size.

As can be seen from Figure 3, the predicted execution times are quite close to the real ones. This verifies that the proposed performance model for the pipeline algorithm is fairly accurate and hence it provides a means to test the viability of the pipeline implementation on any multicore (i.e. dual core, quad core) without taking the burden of real

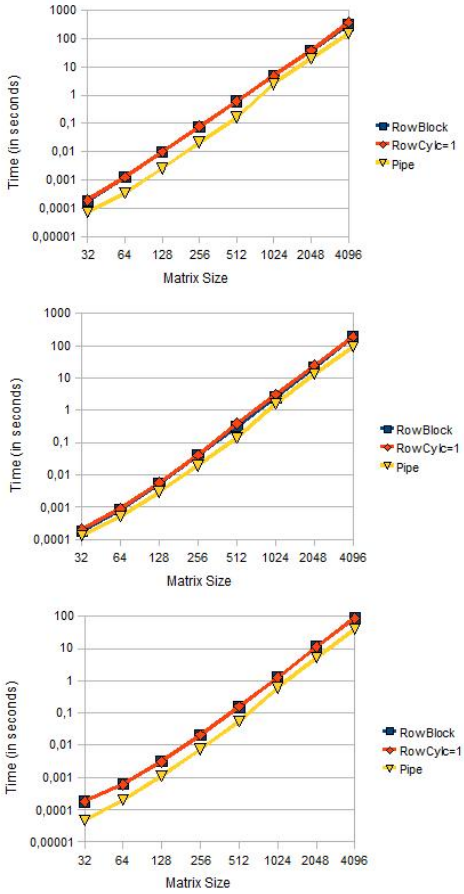

Figure 1. Execution times (in seconds) on log scale of all parallel algorithms for variable matrix sizes on 1 core (top), 2 cores (center) and 4 cores (bottom)

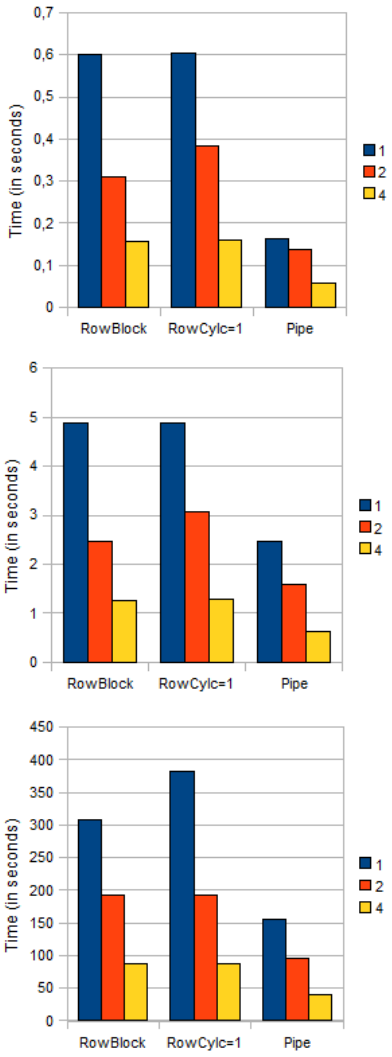

Figure 2. Execution times (in seconds) of all parallel algorithms for 1 to 4 threads for matrix of size 512 (top), matrix of size 1024 (middle) and matrix of size 4096 (bottom)

testing. Further, the proposed performance model is able to predict the parallel performance and the general behavior of the implementation. However, there are minor differences between the measured and predicted results in the pipeline implementation.

V. Conclusions

In this paper, we presented a pipeline implementation of Gauss-Jordan method for solving a linear system of equations using OpenMP interface. Contrary to previous works, no work has been done to implement the Pipeline technique for the Gauss-Jordan method in an OpenMP programming environment. We concluded from the experiments that the pipeline implementation achieves good overall performance in comparison with the other two naive algorithms such as RowBlock and RowCyclic.

Moreover, in this paper we presented a mathematical model of performance in order to predict the performance of the pipeline implementation. We confirmed this model with experimental measurements and showed that this model is good and accurately.

Though many advantages existing, there is still something to improve. One is to improve the experimental study of the pipeline implementation and two naive parallel implementations for large numbers of cores, such as 8 and 16 cores. Another work is to verify the proposed performance model for large number of cores.

References

[1] S. Petiton, "Parallelization on an MIMD computer with real-time Scheduler", *Aspects of Computation on Asynchronous Parallel Processors*, North Holland, 1989.

[2] N. Melab, E. Talbi, and S. Petiton, "A parallel adaptive version

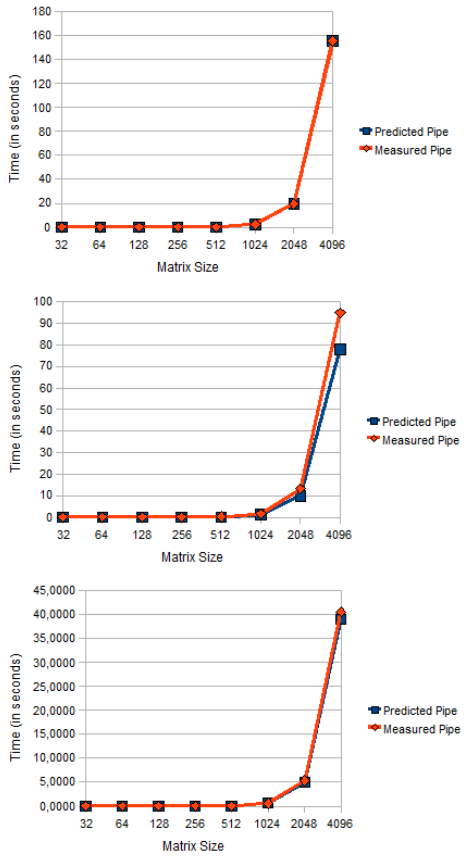

Figure 3. The measured and predicted execution times of the pipeline algorithm on 1 core (top,) 2 cores (center) and 4 cores (bottom)

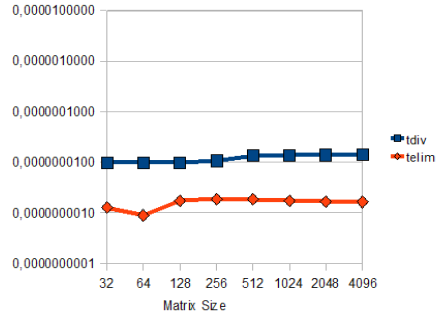

Figure 4. Multicore execution performance as a function of the matrix size for operations elimination and communication on one core

of the block-based gauss-jordan algorithm," in *IPPS/SPDP*, pp. 350-354, 1999.

[3] L. M. Aouad and S. G. Petiton, "Parallel basic matrix algebra on the grid'5000 large scale distributed platform," in *CLUSTER*, 2006.

[4] L. M. Aouad, S. Petiton, and M. Sato, "Grid and Cluster Matrix Computation with Persistent Storage and Out-of-Core Programming," in *The 2005 IEEE International Conference on Cluster Computing, September 2005. Boston, Massachusetts*, 2005.

[5] L. M. Aouad and S. G. Petiton, "Experimentation and Programming Paradigms for Matrix Computing on Peer to Peer Grid", *GRID 2004, The fifth IEEE/ACM International Workshop on Grid Computing*, 2004.

[6] L. Shang, M. Hugues, and S. G. Petiton, "A Fine-grained Task Based Parallel Programming Paradigm of Gauss-Jordan Algorithm", *Journal of Computers*, vol. 5, no. 10, pp. 1510-1519, 2010.

[7] S. Noel, O. Delannoy, N. Emad, P. Manneback, and S. Petiton, "A multilevel scheduler for the Grid computing YML framework," in *Euro-Par 2006: Parallel Processing*, pp. 87-100, 2007.

[8] G. Dimitriu and F. Ionescu, "Solving System of Linear Equations in a Network of Workstations", in *5th International Symposium on Parallel and Distributed Computing*, pp. 323-328, 2006.

[9] F. Ionescu, M. Chiru, V. Sandulescu and M. Ionescu, "Efficient execution of parallel applications in grid with MPI library", in *11th WSEAS International Conference on Mathematical Methods and Computational Techniques in Electrical Engineering*, pp. 384-389, 2009.

[10] A. Buttari, J. J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick, "Multithreading for synchronization tolerance in matrix factorization", *in Proc. of Scientific Discovery through Advanced Computing, Journal of Physics: Conference Series 78:012028*, IOP Publishing, 2007.

[11] E. Elmroth and F. G. Gustavson, "High performance library software for QR factorization", *in Proc. of 5th International Workshop Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, Lecture Notes in Computer Science 1947, pp. 53-63, 2000.

[12] B. C. Gunter and R. A. van de Geijn, "Parallel out-of-core computation and updating the QR factorization", *ACM Transactions on Mathematical Software*, vol. 31, no. 1, pp. 60-78, 2005.

[13] M. Marques, G. Quintana-Orti, E.S. Quintana-Orti and R. van de Geijn, "Out-of-Core Computation of the QR Factorization on Multi-core Processors", *in Proc. of the 2009 Euro-Par Parallel Processing*, Lecture Notes in Computer Science 5704, pp. 809-820, 2009.

[14] S.F. McGinn and R.E. Shaw, "Parallel Gaussian Elimination Using OpenMP and MPI", *in Proc. of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, pp. 169-173, 2002.

[15] P.D. Michailidis and K.G. Margaritis, "Implementing Parallel LU factorization with Pipelining on multicore using OpenMP", *in Proc. of the 13th IEEE International Conference on Computational Science and Engineering*, pp. 253-260, 2010.