PARALLEL ALGORITHMS FOR LARGE SCALE MACROECONOMETRIC MODELS

Bogdan OANCEA[1]
PhD, Assistant Professor, Artifex University, Bucharest, Romania

E-mail: oanceab@ie.ase.ro

Monica NEDELCU
PhD, Associate Professor, University of Economics, Bucharest, Romania

E-mail: mona.nedelcu@yahoo.com

Abstract: *Macroeconometric models with forward-looking variables give raise to very large systems of equations that requires heavy computations. These models was influenced by the development of new and efficient computational techniques and they are an interesting testing ground for the numerical methods addressed in this research. The most difficult problem in solving such models is to obtain the solution of the linear system that arises during the Newton step. For this purpose we have used both direct methods based on matrix factorization and nonstationary iterative methods, also called Krylov methods that provide an interesting alternative to the direct methods. In this paper we present performance results of both serial and parallel versions of the algorithms involved in solving these models. Although parallel implementation of the most dense linear algebra operations is a well understood process, the availability of general purpose, high performance parallel dense linear algebra libraries is limited by the complexity of implementation. This paper describes PLSS – (Parallel Linear System Solver) - a library which provides routines for linear system solving with an interface easy to use, that mirrors the natural description of sequential linear algebra algorithms.*

Key words: *parallel algorithms; linear algebra; macroeconometric models*

1. Introduction

Macroeconometric models with forward-looking variables are a special class of models which generates large systems of equations. The clasical method used to solve such models is the *extended path algorithm* proposed by Fair and Taylor (Fisher, 1992). They use Gauss-Seidel iterations to solve the model, period after period, for a given time horizon. The convergence of this method depend on the order of the equations.

In this paper we'll take another approach – we'll solve the system by the Newton method together with direct and iterative techniques that are well suited for large models. This approach was avoided in the past because it is computationally intensive. The new techniques described in this paper shows that the Newton method is an interesting and cost-effective alternative even for very large macroeconometric models.

The advantages of the Newton method are its quadratic speed of convergence and some modifications leading to a global convergent behavior. The nonlinear model with rational expectations can be represented like this:

$$h_i(y_t, y_{t-1}, \dots, y_{t-r}, y_{t+1|t-1}, \dots, y_{t+h|t-1}, z_t) = 0 \quad i=1,\dots m \tag{1}$$

where $y_{t+j|t-1}$ is the expectation of y_{t+j} conditional on the information available at the end of the period *t-1* and z_t represents the exogenous and random variables. For consistent expectations, the forward expectations $y_{t+j|t-1}$ have to coincide with the next period's forecast when solving the model conditional on the information available at the end of period *t-1*. These expectations are therefore linked in time and solving the model for each y_t conditional on some start period 0 requires each $y_{t+j|0}$ for $j = 1,2, \dots T\text{-}t$ and a terminal condition $y_{T+j|0}$ $j = 1,2 \dots, h$.

Stacking up the these equations for successive time periods give rise to a large nonlinear system of equations. The Newton method for this model gives the following algorithm:

```
NEWTON Method
Given an initial solution y(0)
For k = 0,1,2, … until convergence
        Evaluate b(k) = - h(y(k),z)
Evaluate J(k) = ∂h(y(k),z)/∂y'
Solve J(k)s(k) = b(k)
y(k+1) = y(k) + s(k)
end
```

The most computational intensive step in the Newton method is the linear system *J(k)s(k) = b(k)* when this system is very large. Direct methods for computing the solution of the system can be very expensive because of the computational cost and high memory requirements. That's why, high performance parallel algorithms are an efficient alternative to the classical serial algorithms.

Another alternative to the serial direct methods are iterative methods that computes only an approximation of the solution. This does not influence the convergence of the Newton method.

An important problem is to decide which level of precission for the solution of the linear system guarantees the rapid convergence of the Newton method at the lowest possible cost. To address this problem we define *r(k) = b(k) – J(k)s(k)* the residual for the approximate solution of the linear system at the *k*th Newton iteration. It can be shown that the Newton method is locally convergent if $||r(k)|| \,/\, ||b(k)||$ is a sequence uniformly less than 1.

Parallel versions for the iterative algorithms can also be developed quickly.

2. Iterative and direct methods for linear systems

Iterative methods

An interesting alternative to the stationary iterative methods such as Jacobi or Gauss-Seidel are Krylov techniques. These techniques use information that changes from iteration to iteration. For a linear system *Ax = b* Krylov methods compute the *i*th iterate *x(i)* as :

x(i)=x(i-1)+delta(i)i = 1,2,... *(2)*

The operations involved to compute the *i*th update *delta(i)* are only inner products, saxpy and matrix-vector products, all these beeing level 2 BLAS operations. This is a very good reason to use Krylov methods for very large systems. They are computational attractive comparatively with direct methods that use level 3 BLAS operations.

The best known of the Krylov methods is the *conjugate gradient* (CG) method that solves symmetric positive definite systems. The main idea of the CG method is to update the iterates *x(i)* in a way to ensure the largest decrease of the objective function $\frac{1}{2}x'Ax - x'b$ while keeping the direction vectors *delta(i)* A-orthogonal. The implementation of this method uses only one matrix-vector multiplication per iteration. In exact arithmetic, the CG method yields the solution in at most *n* iterations. The complete description of the CG method can be found in (Golub, 1996). Another Krylov method implemented by the author of this paper is the *BiConjugate Gradient* (BiCG) method. *BiCG* takes a different approach based upon generating two mutually orthogonal sequence of residual vectors and A-orthogonal sequences of direction vectors. The updates for residuals and for the direction vectors are similar to those of the CG method, but are performed using A and its transpose. The disadvantages of the *BiCG* method are an erratic behavior of the norm of the residuals and potential breakdowns. An improved version that solves these disadvantages, called *BiConjugate Gradient Stabilized* (BiCGSTAB) is presented bellow:

```
BiCGSTAB
Given an initial solution x(0) compute r = b – Ax(0)
$\rho_0 = 1, \rho_1 = r(0)'r(0), \alpha = 1, \acute{\omega} = 1, p = 0, v = 0$
for k = 1,2, ... until convergence
        $\beta = (\rho_k / \rho_{k-1})(\alpha/\acute{\omega})$
        $p = r + \beta(p - \acute{\omega}v)$
        $v = Ap$
        $\alpha = \rho_k/(r(0)'v)$
        $s = r - \alpha v$
        $t = As$
        $\acute{\omega} = (t's)(t't)$
        $x(k) = x(k-1) + \alpha p + \acute{\omega}s$
        $r = s - \acute{\omega}t$
        $\rho_{k+1} = -\acute{\omega}r(0)'t$
end
```

The BiCGSTAB method needs to compute 6 saxpy operations, 4 inner products and 2 matrix-vector products per iteration The memory requirements are to store matrix A and 7 vectors of size *n*.

A very widely used Krylov method for general nonsymmetric systems is the *Generalized Minimal Residuals* (GMRES). The pseudo-code for GMRES is:

```
GMRES
Given an initial solution x(0) compute r = b – Ax(0)
ρ = ||r||$_2$, v(1) = r/ ρ, β = ρ
for k = 1,2,... until convergence
        for j = 1,2, ... k,
                h(j,k) = (Av(k))'v(j)
        end
        v(k+1) = Av(k) - $\sum_{j=1}^{k} h(j,k)v(j)$
      ( Gram-Schmidt orthogonalization )
        h(k+1,k) = ||v(k+1)||$_2$
        v(k+1,k) = v(k+1)/h(k+1,k)
end
y(k) = argmin$_y$ ||βe$_1$ – H(k)y||$_2$
x(k) = x(0) + [v(1) ... v(k)] y(k)
```

The main difficulty of the GMRES methods is not to lose the orthogonality of the direction vectors *v(j)*. In order to do this the GMRES method uses a modified Gram-Schmidt orthogonalization process. GMRES requires the storage and computation of an increasing amount of information at each iteration: vectors *v* and matrix *H* . To overcome the increasing memory requirement, the method can be restarted after a chosen number of iterations *m* using the current intermediate results as a new starting point.

The operation count per iteration cannot be used to directly compare the performance of BiCGSTAB with GMRES because GMRES converges in much less iterations than BiCGSTAB.

Direct methods

The direct solution for a linear system *Ax = b* takes two steps:

- In the first step the classical decomposition *A=LU* is computed (*L* is a unit lower triangular matrix and *U* is an upper triangular matrix)
- In the second step the two triangular systems *Ux = y* and *Ly = b* are solved by back substitution and forward elimination.

For symmetric positive definite matrices the factorization is achieved by using the Cholesky decomposition $A = LL^T$. A detailed description of the serial algorithms can be found in (Golub, 1996).

3. Parallel algorithms for linear systems

Software packages for solving linear systems have known many generations of evolution in the past 25 years. In '70, LINPACK was the first portable linear system solver package. At the end of '80 the next software package for linear algebra problems was LAPACK (Anderson, 1992) which, few years later, was adapted for parallel computation resulting the ScaLAPACK (Choi, 1992) library. Although parallel algorithms for linear systems are well understood, the availability of general purpose, high performance parallel dense linear algebra libraries is limited by the complexity of implementation. For the purpose of solving very large macroeconometric models we have developed a software package PLSS (Parallel Linear System Solver) that implements parallel algorithms for linear system solving. The PLSS library was designed with an easy to use interface, which is almost identical with the serial algorithms interface. This goal was obtained by means of data encapsulation in opaque objects that hide the complexity of data distribution and communication operations. The PLSS library was developed in C and for the communication between processors we used MPI library (Gropp, 1994) which is a "de facto" standard in message passing environments. The structure of the library is described in (Oancea, 2002),(Oancea, 2003). Here are the most important details about the internal structure of the library.

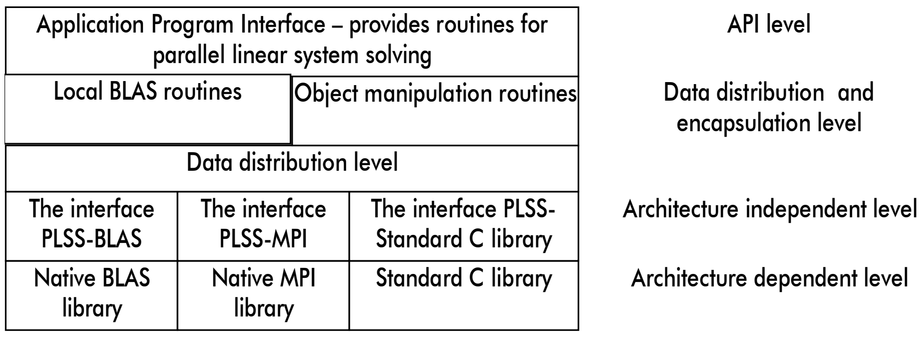

Application Program Interface – provides routines for parallel linear system solving
Local BLAS routines
Object manipulation routines
Data distribution level
The interface PLSS-BLAS
The interface PLSS-MPI
The interface PLSS-Standard C library
Native BLAS library
Native MPI library
Standard C library
API level
Data distribution and encapsulation level
Architecture independent level
Architecture dependent level

Figure 1. The PLSS structure

The first level contains the standard BLAS, MPI and C libraries. This level is architecture dependent. The second level provides the architecture independence. It implements the interface between the base level and the rest of the PLSS package. Using such an approach, the base libraries (BLAS, MPI) can be easily replaced without influencing the rest of the package. This interface has the following components:

- **BLAS-PLSS interface**. Each processor uses the BLAS routines for local computations. Because BLAS library is written in FORTRAN, an interface is needed to call FORTRAN routines from C programs.
- **MPI-PLSS interface**. PLSS uses the following communication operations: MPI_Bcast, MPI_gatherv, MPI_scaterv, MPI_Allgatherv, MPI_Allscatterv, MPI_Reduce, MPI_Allreduce, MPI_Send, MPI_Receive, MPI_Wait. All these MPI operations are encapsulated in PLSS functions in order to decouple the PLSS from MPI.
- **PLSS-Standard C library interface**. This interface encapsulates the standard C library functions (e.g. malloc, calloc, free) in PLSS functions.

The next level implements the data distribution and encapsulation model. All details regarding distribution of vectors and matrices on local processors are placed at this level. Also at this level we can found data encapsulation in opaque objects, hiding the complexity of communication operations. This level defines:

- Objects that describe vectors and matrices.
- Object manipulation routines – object creation, destroying and addressing routines.
- Local BLAS routines. Because matrices and vectors are encapsulated in objects, we must extract some information from these objects such as vector/matrix dimension, their localization etc, before calling a BLAS routine to perform some computations. Local BLAS routines extract these information and then call the standard BLAS routines.

- Communication functions – these functions implement the communication operations between processors.

The top level of the PLSS library is the application program interface. PLSS API provides a number of routines that implements parallel BLAS operations and parallel linear system solving operations based on LU and Cholesky matrix factorization.

The PLSS library uses a bidimensional mesh of processors. We have chosen this model of processor interconnection based on scalability studies of matrix factorization algorithms (Grama, 2003, Oancea, 2002). For a linear system $Ax = b$, vectors x and b are distributed on processors in a block column cyclic model and the system matrix A is distributed according to the vector distribution – the column $A_{*,j}$ will be assigned to the same processor as x_j.

Here are some examples of parallel implementation of some basic operations in the PLSS package. One of the most used operation in linear system solving is matrix-vector multiplication: $Ax = y$. Figure 2 shows the necessary steps to implement parallel matrix-vector multiplication. The matrix in this example has 8 rows and 8 columns.

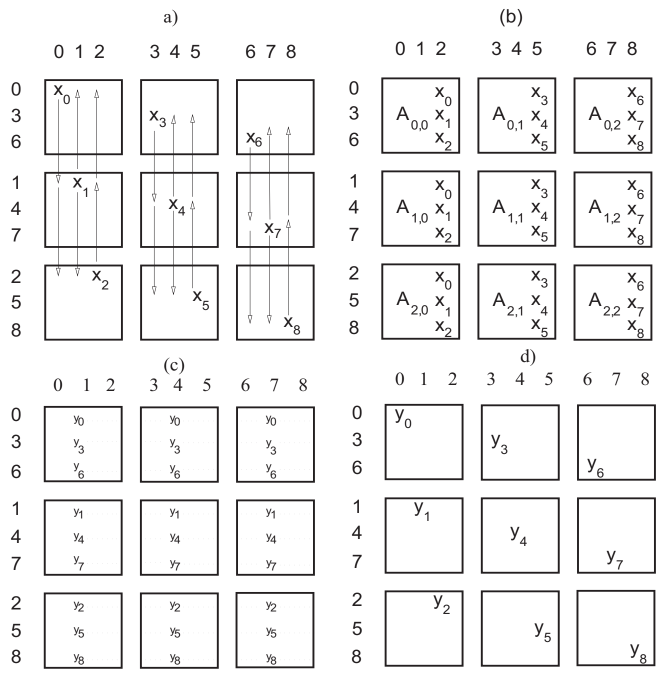


Figure 2. Matrix-Vector multiplication procedure

In the first step (Figure. 2a) the vector components are distributed on the processors columns. After vector distribution it follows a step consisting of local matrix-vector multiplications (Figure. 2b). At this moment each processor owns a part of the final result (Figure. 2c). In the last step, these partial components are summed up along the processor rows (Figure. 2d).

Another frequently used basic operation is rank-1 update. It consists in the following computation: $A = A + yx^t$.

Assuming that x and y have identical distributions on processor columns and rows, each processor has the data needed to perform the local computations.

These two basic operations, matrix-vector multiplication and rank-1 update can be used in order to derive a parallel algorithm for matrix-matrix multiplication. It is easy to observe that the product $C = AB$ can be decomposed in a number of rank-1 updates:

$C = a_0b_0^t + a_1b_1^t + \ldots + a_{n-1}b_{n-1}^t$ (4)

where a_i are the columns of matrix A and b_i^t are the rows of matrix B.

Parallelization of matrix-matrix multiplication is equivalent with parallelization of a sequence of rank-1 updates. In order to obtain an increase in performance, the rank-1 update can be replaced with rank-k update, but in this case x and y will be rectangular matrices. We conclude this section with the implementation of the block Cholesky factorization. Cholesky factorization consists in finding the factorization of the form $A = LL^T$ where A is a symmetric positive definite matrix. Figure 3 shows the partitioning of matrices A and L.

$$A = \begin{pmatrix} A\_11 & * \\ A\_21 & A\_22 \end{pmatrix}$$

$$L = \begin{pmatrix} L\_11 & 0 \\ L\_21 & L\_22 \end{pmatrix}$$

Figure 3. The partitioning of matrices A and L

From $A = LL^T$ we can derive the following equations :

$A_{11} = L_{11}L_{11}^T$ (5)

$L_{21}L_{11}^T = A_{21}$ (6)

$A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$ (7)

If matrix L will overwrite the inferior triangle of A, then the Cholesky factorization consists in the following three computations:

$A_{11} \leftarrow L_{11} = Cholesky(A_{11})$ (8)

$A_{21} \leftarrow L_{21} = A_{21}L_{21}^{-T}$ (9)

$A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$ (10)

The dimension of matrix block A_{11} is computed such that A_{11} will be stored on only one processor and the factorization from equation (8) will be a local operation. Under these conditions A_{21} is stored on the same column of processors and L_{11} will be distributed to these processors. The parallel Cholesky factorization can be described as follows:

1. Determine the block size such that A_{11} is stored on a single processor.
2. Split matrix A into blocks A_{11}, A_{21}, $A_{\_22}$ according to the block size computed in step 1.
3. Compute the Cholesky factorization of submatrix A_{11} – this is a local operation.
4. Distribute A_{11} on the column of processors.

5. Solve the triangular system given by equation (9) – this is a local operation because A_{11} was distributed in the pervious step to all processors that participate in this computation.
6. Compute the symmetric rank-k update given by equation (10).
7. Recursive apply the same steps to matrix A_{22}.

4. Experimental results

We have conducted performance experiments with both serial and parallel versions of the algorithms for two iterative methods – GMRES(40) and BiCGSTAB and with the direct method that consists in matrix factorization. For our experiments we have considered nonlinear systems with the number of variables between 10000 and 38000. The tolerance for the solution was fixed at 10^{-4} for all methods. The serial algorithms are implemented using the C++ programming language under the Linux operating system. Both iterative methods behave relatively well for our problems but BiCGSTAB is less expensive in number of floating point operations and memory requirements. Table 1 shows the number of floating point operations per iteration for each Newton variant to converge and the amount of memory needed.

Table 1. The number of MFLOP/iteration and memory requirements

Matrix dimension	GMRES(40)		BiCGSTAB	
	MFLOP	Memory (Mb)	MFLOP	Memory (Mb)
10000	2100	3.12	723	1.24
14000	4800	4.92	2880	1.88
18000	14100	6.31	12800	3.12
22000	29500	8.23	27500	3.66
26000	58000	8.99	52000	4.99
30000	125000	12.11	112000	5.82
34000	140000	12.55	120000	8.02
38000	200000	15.02	175000	8.82

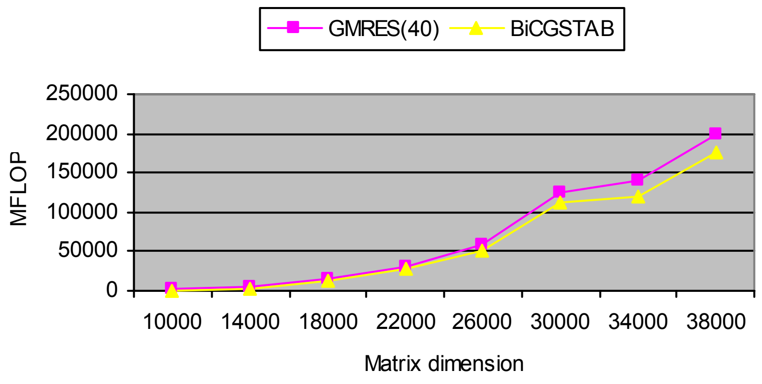


Figure 4. The number of floating point operations per iteration

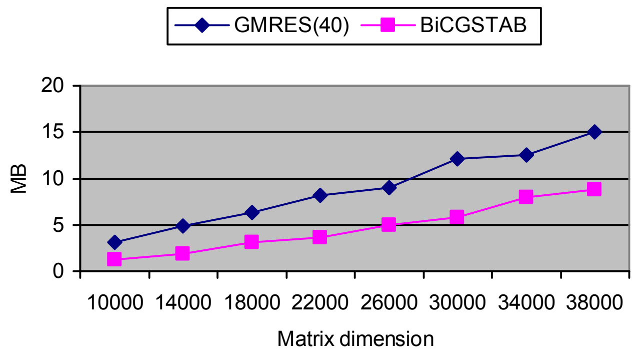
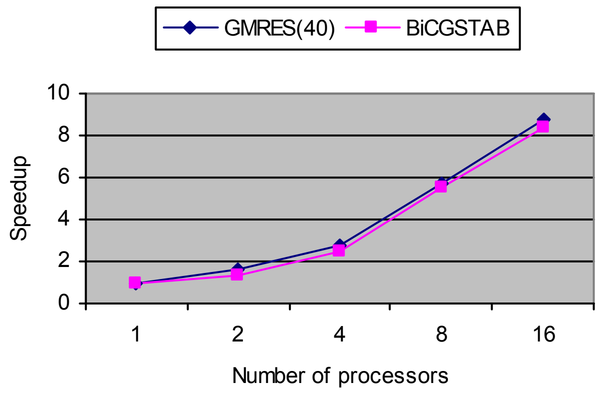


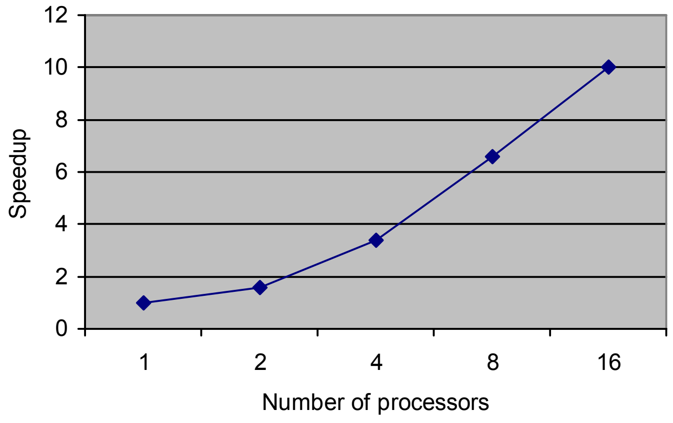
Figure 5. Memory requirement per iteration

These results show that the iterative methods can be a good alternative to direct methods for very large systems of equations.

Parallel versions of the algorithms were executed on a cluster of workstations, connected through a 100Mb Ethernet local network, each station with 1GB of main memory. We have tested the PLSS package for both iterative and direct methods, for 1, 2, 4, 8, and 16 processors. The dimension of the matrix was maintained fixed with 22000 rows and columns. Figure 6(a) shows the speedup of the parallel algorithms in the case when iterative methods are used for solving the model and figure 6(b) shows the speedup in the case of using the direct methods. It can be observed that the direct method has a better speedup the the iterative ones.



a) Parallel iterative methods



b) Direct method (LU factorization)

Figure 6. The speed-up for parallel versions of the algorithms

5. Conclusion

In this paper we have described algorithms for solving macroeconometric models with forward-looking variables based on the Newton method for nonlinear systems of equations. The most computational intensive step in the Newton method consists in solving a large linear system at each iteration. We have compared the performance of solving this linear system for two iterative methods – GMRES(40) and BiCGSTAB and the direct method based on matrix factorization.

For serial algorithms, iterative Krylov methods proved to be an interesting alternative to exact Newton method with LU factorization for large systems. Both the computational cost and memory requirements are inferior in the case of iterative Krylov methods compared with LU factorization.

We have developed a library (PLSS - Parallel Linear System Solver) that implements parallel algorithms for linear system solving. Because of the complexity of parallel algorithms it is difficult to design an easy to use parallel linear system solver. The PLSS infrastructure was designed to provide users a simple interface, close to the description of the serial algorithms. This goal was achieved through data encapsulation, hiding the complexity of data distribution and communication operations from users. PLSS was developed in C using MPI and can be run on many different kinds of parallel computers – it can be run on real parallel computers as well as on simple cluster of workstations

Comparing the performance of the parallel algorithms, LU factorization showed a better scalability then the iterative methods because the iterative algorithms involve a global communication step at the end of each iteration. This communication step slows down the overall execution of the program.

References

1. Anderson, E., *et all.* **LAPACK Users's Guide,** SIAM, Philadelphia, 1992
2. Barrett, R. *et. al.* **Templates for the solution of linear system building blocks for iterative methods**, SIAM, 1994

3. Choi, J., Dongarra, J., Pozo, R., and Walker, D. W. **ScaLAPACK : a scalable linear algebra library for distributed memory concurrent computers**, Proceedings of the fourth Symposium on the Frontiers of Massively Parallel Computers, IEEE Comput. Soc. Press, 1992
4. Dongarra, J., Du Croz, J., Hammarling, S., and Duff, I. **A set of level 3 basic linear algebra subprograms,** ACM Trans. Math. Soft., 16(1), 1990, pp.1-17
5. Dongarra, J., Du Croz, J., Hammarling, S., and Hanson, R. **An extended set of FORTRAN basic linear algebra subprograms**. ACM Trans. Math. Soft., 14(1), March, 1988, pp. 1-17
6. Fisher, P. **Rational Expectations in Macroeconometric Models**, Kluwer Academic Publisher, Dordrecht, 1992
7. Golub G. H., and Van Loan, C. F., **Matrix Computations,** Johns Hopkins Series in Mathematical Sciences, The Johns Hopkins University Press, 1996
8. Grama, A., Gupta, A., Karypis, G., and Kumar V. **Introduction to Parallel Computing, 2nd edition,** Addison-Wesley, 2003
9. Grigori, L., Demmel, J., and Li, X. **Parallel Symbolic Factorization for Sparse LU Factorization with Static Pivoting,** in "Second International Workshop on Combinatorial Scientific Computing", Tolouse, France, June, 2005
10. Gropp, W., Lusk, E., and Skjellum, A., **Using MPI: Portable Parallel Programming with the Message-Passing Interface,** The MIT Press, Cambridge, Massachusets, 1994
11. Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh F.T. **Basic linear algebra subprograms for FORTRAN usage,** ACM Trans. Math. Soft., 5(3), 1979, pp. 308-323
12. Oancea, B. **Parallel algorithms for mathematical models in economics**, Ph.D. Thesis, ASE Bucharest, December 2002.
13. Oancea, B., and Zota, R. **The design and implementation of a dense parallel linear system solver,** in "Proceedings of the First Balkan Conference in Informatics, BCI'2003", Thessaloniki, Greece, 21-23 Nov. 2003
14. Oancea, B., and Zota, R. **The Design and Implementation of a Parallel Linear System Solver,** in "RoEduNet International Conference, Second Edition", Iassy, 5-6 June, 2003
15. Oancea, B. **Implementation of Parallel Algorithms for Dense Matrix Computations**, in "Proceedings of the 7th International Conference on Informatics in Economy", Bucharest, May 2005
16. Saad, Y. **Iterative methods for sparse linear systems**, PWS Publishing Company, 1996

[1] Bogdan Oancea is assistant professor at Artifex University from Bucharest. Since 2002 he has a PhD degree in Cybernetics and Economic Statistics. His main research fields are parallel algorithms applied in mathematical models in economy, geographical information systems and numerical computation. He wrote 6 textbooks in operating systems, parallel computations and he is author of more than 60 papers published in scientific journals or in different conference proceedings. He teaches "Electronic payment systems", "E-marketing" and "Introduction in computer science" courses.