ANALELE ŞTIINŢIFICE ALE UNIVERSITĂŢII „ALEXANDRU IOAN CUZA" DIN IAŞI
Număr special **Ştiinţe Economice** **2010**

NUMERICAL PARALLEL ALGORITHMS FOR LARGE SCALE MACROECONOMETRIC MODELS

Bogdan OANCEA
Artifex University
Bucharest, Romania
oanceab@ie.ase.ro

Tudorel ANDREI
Academy of Economic Studies
Bucharest, Romania
andreitudorel@yahoo.com

Stelian STANCU
Academy of Economic Studies
Bucharest, Romania
stelian_stancu@yahoo.com

Andreea Iluzia IACOB
Academy of Economic Studies
Bucharest, Romania
aiacob@ase.ro

Abstract

In this paper we develop algorithms to solve macro econometric models with forward-looking variables based on Newton method for nonlinear systems of equations. The most difficult step for Newton methods represents the resolution of a large linear system for each iteration. Thus, we compare the performances resulted by solving this linear system using two iterative methods and the direct method.

We also describe an implementation of the parallel versions of such algorithms using a software package. Our experiments confirm that the iterative methods have a low computational complexity and storage requirements, but the parallel versions of direct methods show a superior speedup.

Keywords: macro econometric model, rational expectations model, linear algebra, Newton methods, Krylov techniques, direct methods, software package.
JEL classification: B23, C87

1. INTRODUCTION

Advances in the computational power have a large influence on almost all fields of scientific computing. Although, during the last decade, microprocessors' performance has significantly increased and new architectures like multi-core processors has appeared, there are still problems that cannot be solved on a single desktop computer [9].

One of the fields that need a special attention is macroeconometric modelling. Macroeconometric models with forward-looking variables are a special class of models which involve very large systems of equations. The matrices resulting from these models could be so large that doesn't fit with the internal memory of a single desktop computer. For such models it is necessary to develop high performance parallel algorithms that can be run in parallel execution environments like parallel computers, clusters of workstations or grid environments.

A special kind of macroeconometric models are the rational expectations models [14]. These models contain variables that forecast the economic system state for the future periods $t + 1, t + 2, \ldots, t + T$, where T is the forecast time horizon. Depending on the size of the forecast time horizon, macroeconometric models with rational expectations could give raise to systems with tens or hundreds of thousands of equations.

For example, MULTIMOD model [16], [21] is a dynamic, annual forecast model designed by the International Monetary Fund that describes the economic behaviour of the whole world decomposed in 8 industrial regions and the rest of the countries. The model contains 466 equations. If we want to solve the model for a 30 years time horizon then we will have to solve a nonlinear system containing 13908 equations which is not a simple task nor for the most powerful workstations.

QPM (Quarterly Projection Model) [2] is a quarterly model developed by the Bank of Canada to obtain economic forecasts and as a research tool for the analysis of macroeconomic policies and economic equilibrium on long term. The QPM model has 329 nonlinear equations. The resolution of the model for a 30 years time horizon means to solve a system of 39480 equations.

FRB/US [5], [6] is a quarterly econometric model that describes the U.S. economy and has around 300 equations. An extension of this model is FRB/GLOBAL [that describes the world economy using few thousands equations. The resolution of these models for a 20-30 years time horizon implies nonlinear systems with hundreds of thousands of equations.

Let's consider the general form of the nonlinear model with rational expectations:

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

where $y_{t+j/t-1}$ is the expectation of y_{t+j} conditioned 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 conditioned 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 conditioned on some start period 0 requires each $y_{t+j|0}$ for $j = 1,2, \ldots T\text{-}t$ and a final condition $y_{T+j|0}$, $j = 1, 2 \ldots, h$. Considering these equations for successive time periods a large nonlinear system of equations will result.

One of the first methods used to solve such models was the extended path algorithm proposed by Fair and Taylor [13]. They use Gauss-Seidel iterations to solve the model, period after period, for a given time horizon. The convergence of this method depends on the

order of the equations. The endogenous forecast variables are considered as predetermined and then the model is solved period after period for a time horizon. The solutions thus obtained represent the new values for the forecast variables. The process is repeated until the convergence is obtained. The advantage of this method is it's simplicity in implementation and the low storage requirements but this method has a main disadvantage: if the initial values for the endogenous variables are not "well" chosen, the convergence of the system is very poor or the system is not convergent at all.

An alternative method to solve the model is to built a system of equations written for successive periods *t, t + 1, ..., t + T*¸ and to solve this system of *nT* nonlinear equations by one of the existing methods for nonlinear systems. Due to the large scale of the system, this method has been avoided in the past. Due to the recent advances in the parallel algorithms field it is now possible to solve such large scale systems with efficiency.

The Newton method applied to solve this model uses 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)
endfor

If the linear system $J(k)s(k) = b(k)$ is very large, the use of direct methods to determine the solution can be very expensive due to high memory requirements and computational cost. This is a very good reason to develop high performance parallel algorithms as an attractive alternative to the classical serial algorithms. Another alternative to serial direct methods are the iterative methods which determine only an approximation of the solution, but this fact does not influence the convergence of the Newton method. These iterative algorithms can be parallelized too.

We will also analyze high performance iterative and direct methods used to solve large linear systems that result by applying the Newton method, then we will describe an implementation of the parallel versions of such algorithms that we've developed using a software package called PLSS (Parallel Linear System Solver).

2. SERIAL ITERATIVE AND DIRECT METHODS FOR THE SOLVING OF LINEAR SYSTEMS

For very large linear systems, the most appropriate ***iterative methods*** are the so-called Krylov techniques [23]. Contrary to stationary iterative methods such as Jacobi or Gauss-Seidel, Krylov 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 :

$$x(i) = x(i-1) + d(i) \qquad i = 1,2,...$$

Operations involved to find the i^{th} update *d(i)* are only inner products, *saxpy* and matrix-vector products that has the complexity of $\Theta(n^2)$, so that Krylov methods are computational attractive comparing to the direct methods for linear systems.

A perhaps the best known of the Krylov' method is the conjugate gradient method. This method solves symmetric positive definite systems. The 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 *d(i) A*-orthogonal. This method can be implemented using only one matrix-vector multiplication per iteration. In exact arithmetic, the CG method gives the solution for at most *n* iterations. The complete description of the CG method can be found in [15].

Another Krylov method for general non symmetric systems is the Generalized Minimal Residuals (GMRES) introduced by [23]. The pseudo-code for GMRES is:

GMRES
Given an initial solution x(0) compute r = b – Ax(0)
$\rho = ||r||_2$, 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
  $v(k+1) = Av(k) - \sum_{j=1}^{k} h(j,k)v(j)$
  $h(k+1,k) = ||v(k+1)||_2$
  v(k+1,k) = v(k+1)/h(k+1,k)
endfor
$y(k) = \text{argmin}_y ||\beta e_1 - H(k)y||_2$
x(k) = x(0) + [v(1) ... v(k)] y(k)

The most difficult part of this algorithm is not to lose the orthogonality of the direction vectors *v(j).* To achieve this goal the GMRES method uses a Gram-Schmidt orthogonalization process. GMRES requires the storage and computation of an increasing amount of information, vectors *v* and matrix *H*. To overcome these difficulties, the method can be restarted after a chosen number of iterations *m.* The current intermediate results are used as a new starting point.

Another Krylov method implemented by the authors is the BiConjugate Gradient method [6]. BiCG uses a different approach based upon generating two mutually orthogonal sequences 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 disadvantage of the BiCG method is an erratic behaviour of the norm of the residuals and potential breakdowns. An improved version, called BiConjugate Gradient Stabilized BiCGSTAB, is presented below:

BiCGSTAB
Given an initial solution x(0) compute r = b – Ax(0)
$\rho_0 = 1$, ρ_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 + β(p- $\acute{\omega}$v)
v = Ap
$\alpha = \rho_k/$(r(0)'v)
s = r – αv
t = As
$\acute{\omega}$ = (t's)(t't)
x(k) = x(k-1) + αp + $\acute{\omega}$s
r = s – $\acute{\omega}$t
ρ_{k+1} = - $\acute{\omega}$r(0)'t
endfor

For the BiCGSTAB method we need to compute 6 *saxpy* operations, 4 inner products and 2 matrix-vector products per iteration and to store matrix A and 7 vectors of size *n*. The computational complexity of the method is $\Theta(n^2)$ like the other Krylov methods. 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. We have implemented these iterative methods and run experiments to determine the possible advantages of them over the direct methods. The results of our experiments are presented in the next section.

The other alternative to solve a linear system $Ax = b$ is the ***direct method*** that consists in two steps:

- First, the matrix A is factorized, $A = LU$ where *L* is a lower triangular matrix with 1s on the main diagonal and *U* is an upper triangular matrix; in the case of symmetric positive definite matrices, we have $A = LL^t$.
- Second, we have to solve two linear systems with triangular matrices: $Ly = b$ and $Ux = y$.

The standard LU factorization algorithm with partial pivoting is [15]:

Right-looking LU factorization
for k =1:n-1 **do**
find v with k≤ v≤n such that $|A(v,k)| = \|A(k:n,k)\|_\infty$
A(k,k:n)↔A(v, k:n)
p(k) = v
if A(k,k) ≠ 0 **then**
A(k+1:n, k) = A(k+1:n,k)/A(k,k)
A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n, k) A(k, k+1:n)
endif
endfor

The computational complexity of this algorithm is $\Theta(2n^3/2)$. After we obtain the matrix factors

L and *U* we have to solve two triangular systems: $Ly = b$ and $Ux = y$. These systems are solved using forward and backward substitution that have a computational complexity of $\Theta(n^2)$, so the most important computational step is the matrix factorization. That's why we have to show a special attention to the algorithms for matrix factorization.
In practice, using actual computers with memory hierarchies, the above algorithm is not efficient because it uses only level 1 and level 2 BLAS operations [18], [11]. As it is well-known, level 3 BLAS operations [10] have a better efficiency than level 1 or level 2 operations. The standard way to change a level 2 BLAS operations into a level 3 BLAS operation is delayed updating. In the case of the LU factorization algorithm we will replace *k* rank-1 updates with a single rank-*k* update.

We present a block algorithm for LU factorization that uses level 3 BLAS operations. The $n \times n$ matrix A is partitioned as in Figure 1. The A_{00} block consists of the first *b* columns and rows of the matrix A.

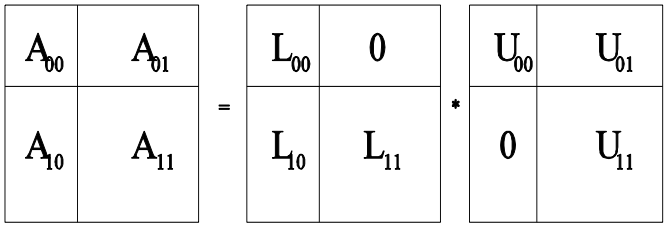

Figure no. 1 Block LU factorization

We can derive the following equations starting from A=LU:

$$L_{00}U_{00} = A_{00} \quad (1)$$

$$L_{10}U_{00} = A_{10} \quad (2)$$

$$L_{00}U_{01} = A_{01} \quad (3)$$

$$L_{10}U_{01} + L_{11}U_{11} = A_{11} \quad (4)$$

Equations (1) and (2) perform the *LU* of the first *b* columns of the matrix *A*. Thus we obtain L_{00}, L_{10} and U_{00} and now we can solve the triangular system from equation (3) that gives U_{01}. The problem of computing L_{11} and U_{11} reduces to compute the factorization of the submatrix $A_{11}' = A_{11} - L_{10}U_{01}$ that can be done using the same algorithm but with A_{11}' instead of A. The block LU factorization algorithm can now be derived easily: suppose we have divided the matrix A in column blocks with *b* columns in each block. The complete block LU factorization algorithm is given below.

Block LU factorization
for k_b =1 to n-1 **step** b **do**
$b_f = \min(k_b + b - 1, n)$
{LU factorization of $A(k_b : n, k_b : b_f)$ with BLAS 2}
for k = k_b **to** b_f **do**
find k such that $|A(k,i)| = \|A(i:n,i)\|_\infty$
if i ≠ k **then**
swap rows i and k
endif
A(i+1:n, i) = A(i+1:n, i)/A(i,i)
A(i+1:n, i+1: b_f) = A(i+1:n, i+1: b_f) - A(i+1:n, i) A(i, i+1: b_f)
endfor
{Let $\tilde{L}$ be unit lower triangular matrix $b \times b$ stored in $A(k_b : b_f, k_b : b_f)$ }
Solve triangular systems $\tilde{L}Z = A(k_b : b_f, b_f + 1 : n)$
Update $A(k_b : b_f, b_f + 1 : n) \leftarrow Z$
{Delayed updating}
$A(b_f + 1 : n, b_f + 1 : n) = A(b_f + 1 : n, b_f + 1 : n) - A(b_f + 1 : n, k_b : b_f)A(k_b : b_f, b_f + 1 : n)$
endfor

The process of factorization is shown in Figure 2. The factorization of the current column block is done with the usual BLAS 2 operations and the active part of the matrix A will be updated with *b* rank-one updates simultaneously which in fact is a matrix-matrix multiplication (level 3 BLAS). If $n >> b$ almost all floating point operations are done in the matrix-matrix multiplication operation.

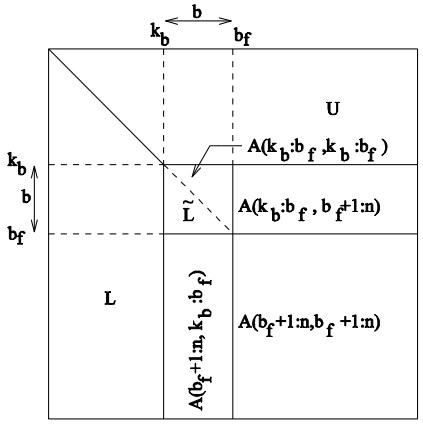

Figure no. 2 Block LU factorization with BLAS 3 operations

3. THE IMPLEMENTATION OF PARALLEL ALGORITHMS FOR LINEAR SYSTEMS

For very large matrices that result for the econometric models presented above, the serial algorithms may not be appropriate to solve the models. Thus, parallel versions of the above presented algorithms have to be developed and implemented.

Software packages for solving linear systems have known a powerful evolution during the last 35 years. LINPACK was the first portable linear system solver package followed at the end of '80 by a new software package for linear algebra problems LAPACK [1] which was adapted for parallel computation resulting ScaLAPACK [7] library. Other software packages for parallel computation have been developed over the years: PETSc [3] is a parallel library that implements iterative methods, PARPACK [20] package can handle matrices in sparse format; SuperLU [26] package implements a supernodal parallel algorithm for sparse matrix factorization.

Although parallel algorithms for linear systems are studied and very well understood nowadays, the availability for general purpose, high performance parallel linear algebra libraries is limited by the complexity of implementation. Almost all parallel libraries have a complicated interface, very difficult to use due to the complexity of the parallel algorithms.

We have developed a library that implements parallel algorithms for linear systems solving - PLSS (Parallel Linear System Solver). The library was designed with an easy to use interface, which is almost identical with the serial algorithms' interface. This makes the software developing process very easy because the parallelism is hidden from the user and the algorithms are almost identical with their serial versions. 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 [24], which is a "de facto" standard for message passing environments.

The PLSS library is structured on four levels, as we can see in Figure no. 3.

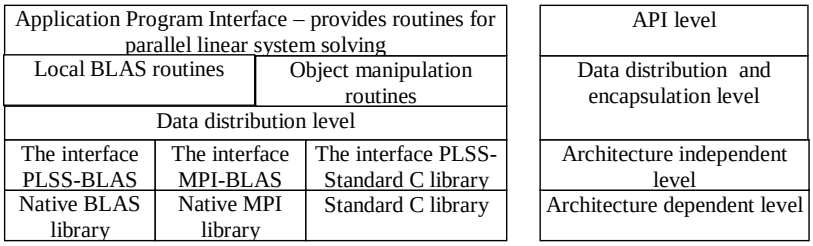

Figure no. 3 PLSS structure

The first level contains the standard BLAS, MPI and C libraries. This level is architecture dependent. The second level provides the architecture independence, which implements the interface between the first level and the rest of the PLSS package. The next level implements the data distribution model – all details regarding distribution of vectors and matrices on processors are localized at this level. At this level, the data are encapsulated in objects that are opaque to users, hiding thus the complexity of communication operations.

This level defines:

•Objects that describe vectors and matrices.

•Object manipulation routines – object creation, destroying routines and object 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, in fact, the application programming interface. PLSS API provides a number of routines that implements parallel BLAS operations and parallel linear system solving operations: direct methods based on LU and Cholesky matrix factorization and nonstationary iterative methods GMRES, BiCG, BiCGSTAB.

The PLSS library uses a logical bidimensional mesh of processors. We have chosen this model of processor interconnection based on scalability studies of the matrix factorization algorithms [22]. In this case, the isoefficency function [17] for LU matrix factorization is $O(p\sqrt{p})$, that means a highly scalable algorithm.

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.

We also present some examples of parallel implementation of some basic operations for the PLSS package. Matrix-vector multiplication *Ax = y* is a frequent operation for linear system solving algorithms. Figure 4 shows the necessary steps to implement parallel matrix-vector multiplication.

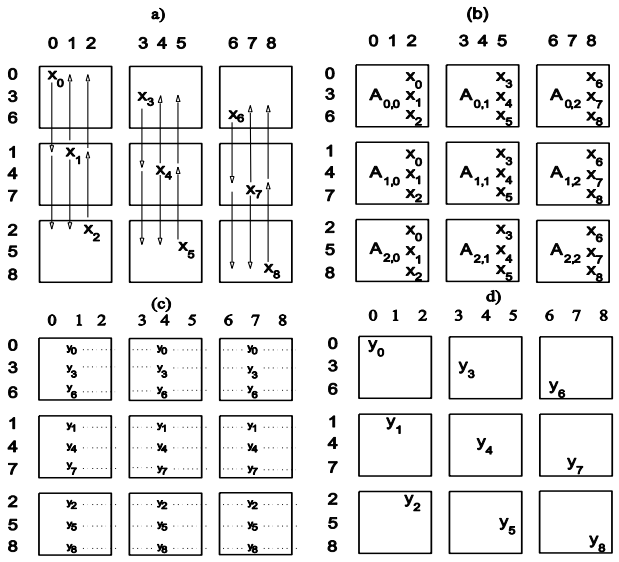

Figure no. 4 Matrix-Vector multiplication

At the first step (Figure 4a) the vector components are distributed on the processors columns. After vector distribution it follows a step consisting in local matrix-vector multiplications (Figure 4b). At this moment each processor owns a part of the final result (Figure 4c). At the last step, these partial components are summed up along the processor rows (Figure 4d).

Rank-1 update is another basic operation which 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:

$$C = a_0 b_0^t + a_1 b_1^t + \ldots + a_{n-1} b_{n-1}^t \quad (5)$$

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

The parallelization of matrix-matrix multiplication is equivalent with the 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 5 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 no. 5 The partitioning of matrices *A* and *L*

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

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

$$L_{21} L_{11}^T = A_{21} \quad (7)$$

$$A_{22} - L_{21} L_{21}^T = L_{22} L_{22}^T \quad (8)$$

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

$$A_{11} \leftarrow L_{11} = Cholesky(A_{11}) \quad (9)$$

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

$$A_{22} \leftarrow A_{22} - L_{21} L_{21}^T \quad (11)$$

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 (9) 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:

Step 1: Determine the block size such that A_{11} is stored on a single processor.

Step 2: Split matrix A into blocks A_{11}, A_{21}, $A_{\_22}$ according to the block size computed at the previous step.

Step 3: Compute the Cholesky factorization of submatrix A_{11} – this is a local operation.

Step 4: Distribute A_{11} on the column of processors.

Step 5: Solve the triangular system given by equation (8) – this is a local operation because A_{11} was distributed at the previous step on all processors that participate at this computation.

Step 6: Compute the symmetric rank-k update given by equation (9).

Step 7: Recursive, apply the same steps to matrix A_{22}.

The current version of the PLSS library implements the following parallel BLAS routines:

Table no. 1 Level 1 BLAS routines

int Axpy(Object alpha, Object x, Object y)	computes $y = \alpha x x + y$
int Dot(Object x, Object y, Object alpha)	computes the dot product $\alpha = x^T y$
int Nrm2(Object x, Object alpha)	computes the euclidian norm of vector *x.*
int Scal(Object x, Object alpha)	scales vector *x* : $x = \alpha x x$
int Iamax(Object x, Object k, Object xmax)	computes the maximum value (*xmax*) and the global offset (*k*) for object *x*.

Table no. 2 Level 2 BLAS routines

int Ger (Object alpha, Object x, Object A)	computes $A = \alpha x y^T + A$
int Gemv (int transa, Object alpha, Object A, Object x, Object beta, Object y)	computes the matrix-vector multiplication: $y = \alpha Ax + \beta y$
int Symv (int uplo, Object alpha, Object A, Object x, Object beta, Object y)	computes the matrix-vector multiplication for symmetric matrices : $y = \alpha Ax + \beta y$
int Trmv (int uplo, int trans, int diag, Object A, Object x)	computes the matrix-vector multiplication for triangular matrices.
int Trsv (int uplo, int trans, int diag, Object A, object x)	solves the linear system $Ax=b$ where A is a triangular matrix.

Table no. 3 Level 3 BLAS routines

int Syrk (int uplo, int trans, Object alpha, Object A, Object beta, Object C)	symmetric rank-k update: $C = \alpha AA^T + \beta C$
int Trsm (int side, int uplo, int trans, int diag, Object alpha, Object A, Object C)	solves the multiple right hand linear system: $AX = B$
int Gemm(int trans, int transb, Object alpha, Object A, Object B, Object beta, Object C)	Matrix multiplication: $C = \alpha AB + \beta C$

The name of routines and the significance of parameters *trans, side, uplo, diag,* are the same as for the original BLAS library.

For matrix factorization, PLSS library has two routines:

- *Cholesky*(Object A) – computes the Cholesky factorization of a SPD matrix *A*.
- *LU*(Object A, Object pivots) – computes the LU factorization with partial pivoting.

Finally, the PLSS package contains routines that implement the GMRES, BiCG and BICGSTAT iterative methods.

4. EXPERIMENTAL RESULTS

We have conducted performance experiments for both serial and parallel versions of the algorithms for two iterative methods – GMRES(35) and BiCGSTAB and for the direct method that consists in matrix factorization. For our experiments we have considered nonlinear systems containing between 2000 and 20000 variables. The tolerance for the solution was fixed at 10^{-4} for all methods. The serial versions of the 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 slightly less expensive in number of floating point operations and memory requirements. Table 4 shows the number of floating point operations per iteration for each Newton variant to converge and the amount of memory needed.

Table no. 4 The number of MFLOP/iteration and memory requirements

GMRES(35)			BiCGSTAB		
Size	MFLOP	Memory (Mb)	Size	MFLOP	Memory (Mb)
2000	149	0.66	2000	135	0.38
4000	261	1.51	4000	260	0.66
8000	962	2.82	8000	744	1.32
12000	3800	4.05	12000	2670	1.88
16000	8310	5.65	16000	6790	2.68
20000	21540	7.24	20000	20830	3.22

These results show that the iterative methods can be a good alternative to direct methods for systems containing a higher number of equations mainly due to the low memory requirements. This allow that large problems to be solved on a single workstation. Although the data used for such problems fits on a single workstation internal memory, the time needed to solve them can be very high. In practice, a compromise between memory requirements and the solving time is always the best solution. For the cases when the time is too high, parallel solvers are necessary.

Parallel versions of the algorithms implemented using the PLSS package were executed on a cluster of workstations, connected through a 100Mb Ethernet local network, each station having 512 MB of main memory. The PLSS package uses the MPICH implementation of the MPI library and, for the local BLAS operations, uses the ATLAS library [25] that provides a high performance for local operations. The ATLAS library can be replaced with a vendor specific library, but this would introduce architecture dependence. 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 20000 rows and columns. Figure 6(a) shows the speedup of the parallel algorithms for the case when iterative methods are used to solve the model and figure 6(b) shows the speedup in the case of using direct methods.

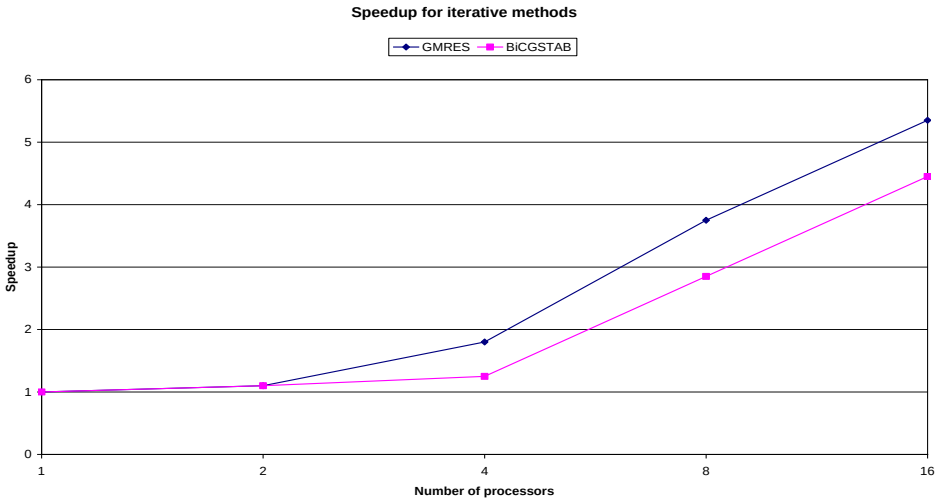

Figure no. 6.1 The speedup for parallel versions of the algorithms - Parallel iterative methods

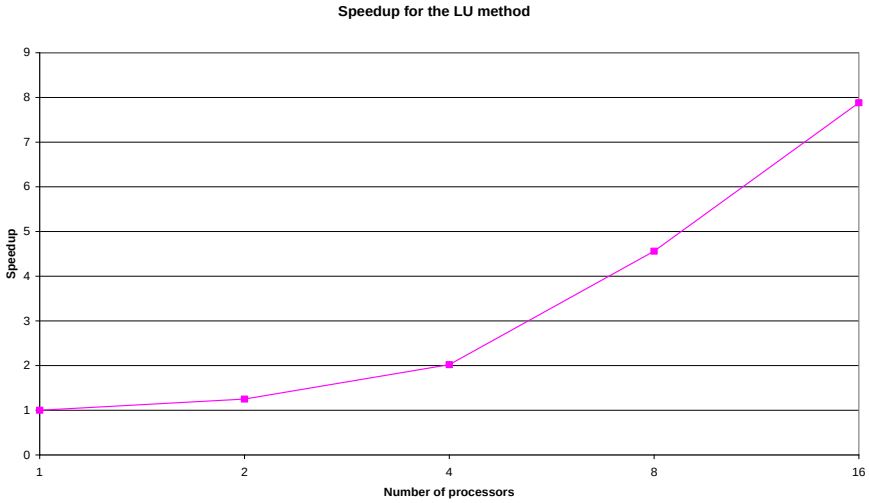

Figure no. 6.2 The speedup for parallel versions of the algorithms- Direct method (LU factorization)

As we can observe, the speedup for the GMRES is slightly better than for the BiCGSTAB method. Compared to the iterative methods, the direct method based on matrix factorization shows a better speedup, meaning that these algorithms are more scalable than the iterative ones. They are better suited for very large problems which run on parallel environments with a large number of processors.

5. CONCLUSIONS

In this paper we have developed some algorithms to solve macroeconometric models with forward-looking variables based on the Newton method for nonlinear systems of equations. The most difficult step for Newton methods represents the resolution of a large linear system for each iteration. We also compared the performances resulted by solving this linear system using two iterative methods and the direct method.

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

Regarding the parallel algorithms, we have developed a parallel library PLSS with an interface easy to use. All the complexity of the parallel algorithms is hidden from the users by encapsulating the matrices and vectors in opaque objects. The experiments using our library for the direct methods using LU factorization showed a better scalability compared to iterative methods because the iterative algorithms involve a global communication step at the end of each iteration. These results recommend the use of parallel algorithms for very large systems in parallel environments with a large number of processors.

References

[1] Anderson, E., Z. Bai, J. Demmel, J., Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. Mckenney, S. Ostrouchov, And D. Sorensen (1992): *LAPACK Users's Guide*. SIAM, Philadelphia.

[2] Armstrong, J., R. Black, D. Laxton, and D. Rose (1995): "The Bank of Canada's New Quarterly Projection Model QPM. Part 2: A Robust Method for Simulating Forward-Looking Models", Technical Report No. 73, Ottawa: The Bank of Canada.

[3] Balay, S., K. Buschelman, V. Eijkhout, V. D. Gropp, D. Kaushik, M.G. Knepley, L. C. Mcinnes, B. F. Smith, and H. Zhang (2004): *PETSc Users Manual*, Technical Report, ANL-95/11 - Revision 2.1.5, Argonne National Laboratory.

[4] Black, R., D. Laxton, and R. Tetlow (1994): "The Bank of Canada's New Quarterly Projection Model QPM. Part 1: The Steady-State Model", Technical Report No. 72, Ottawa: The Bank of Canada, November.

[5] Brayton, F., and P. Tinsley (1996): "A guide to FRB/US : A Macroeconomic Model of the United States", Technical Report, Finance and Economics Discussion Series, Federal Reserve Board.

[6] Brayton, F., E. Mauskopf, D. Reifschneider, P. Tinsley, and J. Williams (1997): "The Role of Expectations in the FRB/US Macroeconomic Model", Federal Reserve Bulletin.

[7] Choi, J., J. Dongarra, R. Pozo, and D.W. Walker (1992): *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, 120-127.

[8] Coletti, D., B. Hunt, D. Roseand, And R. Tetlow(1996): "The Bank of Canada's New Quarterly Projection Model QPM. Part 3: The Dynamic Model", Technical Report No. 75, Ottawa: The Bank of Canada.

[9] Creel, M., and W. L. Goffe (2008): "Multi-core CPUs, Clusters, and Grid Computing: a Tutorial", *Computational Economics*, 32 (4), 353-382.

[10] Dongarra, J., J. Du Croz, S. Hammarling, and I. Duff (1990): "A set of level 3 basic linear Algebra subprograms", *ACM Transactions on Mathematical Software*, 16 (1), 1-17.

[11] Dongarra, J., J. Du Croz, S. Hammarling, and R. Hanson (1988): "An extended set of FORTRAN basic linear algebra subprograms", *ACM Transactions on Mathematical Software*, 14, (1), 1-17.
[12] Doornik, J. A., D. F. Hendry, and N. Shephard (2007): "Parallel Computation in Econometrics: A Simplified Approach" Chapter 15 in *Handbook of Parallel Computing and Statistics*, Chapman & Hall/CRC, 449-476.
[13] Fair, R.C., and J. B.Taylor (1983): "Solution and Maximum Likelihood Estimation of Dynamic Nonlinear Rational Expectations Models", *Econometrica*, 51(4), 1169-1185.
[14] Fisher, P., (1992): *Rational Expectations in Macroeconomic Models*. Kluwer Academic Publishers, Dordrecht.
[15] Golub, G. H., and C. F. Van Loan (1996): *Matrix Computations***,** Johns Hopkins Series in Mathematical Sciences, The Johns Hopkins University Press.
[16] Isard, P. (2000): "The Role of MULTIMOD in IMF's Policy Analysis", Technical Report IMF Policy Discussion Paper, International Monetary Fund, Washington DC.
[17] Kumar, V., Grama, A., Gupta, A., and Karypis, G., (1994): *Introduction to Parallel Computing*, The Benjamin/Cummings Publishing Company.
[18] Lawson, C. L., R. J. Hanson, D. R. Kincaid, and F. T. Krogh (1979): "Basic linear algebra subprograms for Fortran usage", *ACM Transactions on Mathematical Software*, 5 (3), 308-323.
[19] Levin, J., and R. Tryon (1997): "Evaluating International Economic Policy with the Federal Reserves Global Model", Federal Reserve Bulletin.
[20] Maschhoff, K. J., and D. C. Sorensen (1996): "A portable implementation of ARPACK for Distributed Memory Parallel Architectures", Proceedings of the Copper Mountain Conference on Iterative Methods.
[21] Masson, P., S. Symanski, and G. Meredith (1990): "MULTIMOD Mark II: A Revised and Extended Model", Technical Report, Occasional paper 71, International Monetary Fund, Washington DC.
[22] Oancea, B., and R. Zota (2003): "The design and implementation of a dense parallel linear system solver", Proceedings of the 1st Balkan Conference in Informatics, Thessaloniki, Greece.
[23] Saad, Y. (1996): *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company.
[24] Snir, M., S. W. Otto, S. Huss-Lederman, D. W. Walker, And J. Dongarra (1996): *MPI: the Complete Reference*, MIT Press.
[25] Whaley, R. C., A. Petitet, and J. Dongarra (2001): "Automated Empirical Optimization of Software and the ATLAS project", *Parallel Computing*, 27(1-2), 3-35.
[26] Xiaoye, S. Li, and J. W. Demmel (2003): "A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems", *ACM Transactions on Mathematical Software*, 29 (2), 110-140.

Acknowledgements

This work was supported by CNCSIS – UEFISCSU, project number PNII – IDEI code 1793/2008, financing contract no. 862/2009.