"HENRI COANDA"
AIR FORCE ACADEMY
ROMANIA

"GENERAL M.R. STEFANIK"
ARMED FORCES ACADEMY
SLOVAK REPUBLIC

INTERNATIONAL CONFERENCE of SCIENTIFIC PAPER
AFASES 2012
Brasov, 24-26 May 2012

OPTIMIZATION METHOD OF SIMPLEX ALGORITHM SOLUTION

Ioan POPOVICIU

Naval Academy "Mircea cel Batran", Constantza

Abstract : *When speaking about linear programming problems of big dimensions with sparse matrix of the system, resolved through simplex method, it is necessary, at each iteration, to calculate the inverse of the base matrix, which leads to the loss of the rarity character of the matrix. The article proposes the replacement of the calculus of the inverse of the base matrix with the solving through iterative parallel methods of a linear system with sparse matrix of the system.*

Keywords: *algorithm, matrix,, parallel, processor, simplex*
MSC 2010: *65Y05*

1. GENERAL PRESENTATION

Linear programs for large real systems are characterized by sparse matrixes, having a low percentage of non-void elements. The sparse character appears at each base matrix, but disappears at the inverse of this matrix. In its classical form, the simplex method uses a square matrix, the inverse of the base matrix, whose value is putting up-to-date at each iteration. The number non-void elements of the inverse matrix increase rapidly and depend on the number of iterations. Because of this, in the place of the calculus of the sparse matrix, one can solve the linear matrixes with the sparse matrix of the system through iterative parallel methods.

Let's take the linear programming problem under the standard form:

$$\begin{aligned} &Ax = b \\ &x \geq 0 \end{aligned} \qquad (1)$$

$$\max(f(x) = c^T x) \qquad (2)$$

where A is a matrix with m lines and n columns, $x \in R^n, b \in R^m, c \in R^n$.

At each iteration, one takes a base, meaning a square matrix of order m, which can be inverted, extracted from matrix A, noted with A^I, where $I \subset N, |I| = m$. A so-called basic solution is associated to the base I defined by:

$$x_I^B = (A^I)^{-1} b$$

$$x_{\bar{I}}^B = 0$$

where $\bar{I}$ is the complement of I in N.

The bases which are being successively generated through simplex method are of the type $x_I^B \geq 0$, meaning the basic solutions considered are all admissible (they fulfill the conditions (1) and (2)).

An iteration consists of a change of the base I into an adjacent base I'; this is a base obtained through the changing of the index $r \in I$ with the index $s \in \bar{I}$:

$$I' = I - r + s. \qquad (3)$$

To determine r and s one has to calculate:

$$u = f^I (A^I)^{-1} \qquad (4)$$

$$d^{\bar{I}} = f^{\bar{I}} - uA^{\bar{I}} \qquad (5)$$

where $u, f^I, d^{\bar{I}}$ are line vectors. This allows that $s \in \bar{I}$ is selected by the condition $d^s > 0$. Then:

$$x_I^B = (A^I)^{-1} b \qquad (6)$$

$$T^s = (A^I)^{-1} a^s \qquad (7)$$

where a^s is column s of matrix A, and x_I^B, b, T^s, a^s are column vectors. One obtains $r \in I$ through condition:

$$\frac{x_r}{T_r^s} = \min\left\{ \frac{x_i}{T_i^s} \mid i \in I, T_i^s > 0 \right\} \qquad (8)$$

Once the values r and s are determined, there follows the updating of the inverse of the base matrix , meaning that $\left(A^{I'}\right)^{-1}$ is determined, which is obtained from the relation:

$$\left(A^{I'}\right)^{-1} = E_r(\eta)(A^I)^{-1} \qquad (9)$$

where $E_r(\eta)$ is the matrix obtained from the unit matrix of order n, by replacing the column e^r with the vector:

$$\eta = \left(-\frac{c_1}{c_r}, \dots, -\frac{c_{r-1}}{c_r}, \frac{1}{c_r}, -\frac{c_{r+1}}{c_r}, \dots, -\frac{c_n}{c_r} \right)$$

where $c = (A^I)^{-1} a^s$.

In this way the mathematical equations are represented by the relations (4-9), and the inverse of the base matrix appears in the relations (4), (6), (7). The last three relations can be replaced by:

$$uA^I = f^I \qquad (4')$$

$$A^I x_I^B = b \qquad (6')$$

$$A^I T^s = a^s \qquad (7')$$

In the first equation, the matrix is the transpose of the base matrix; in the last two equations even the base matrix appears and consequently these two systems benefit from the sparse character of matrix A, an efficient solution being possible through iterative parallel methods.

2. THE PARALLEL ALGORITHM OF THE CONJUGATED GRADIENT

In order to solve linear systems of large dimensions of the type (4'-7'), we are going to present a parallel implementation of the algorithm of the conjugated gradient, method where, in the first place, one has to make the operations of multiplication between a sparse matrix and a vector parallel.

Let's take the product $y = Ax$ where A is a sparse matrix $n \times n$, and x and y are vectors of n dimension. In order to accomplish a parallel execution of the product $y = Ax$ one has to perform a partitioning of the matrix A into a matrix distributed over many processors. In this view, a subset of the components of vector x and consequently a subset of the lines of matrix A are being allocated to a processor so that the components of vectors x and y can be divided into three groups:

- **-internal** are those components which belong (and consequently are calculated) to the processor and do not take part into the communication between the processors. We say in consequence that y_j is an **internal component** if it is calculated by the processor which it belongs to and if the index j of the column corresponding to the element a_{ij} unlike zero from line i correspond to a component x_j which also belongs to the same processor;
- **-border set** are those components which belong (and by consequence are calculated) to the processor, but they require a communication with other processors in order to calculate them. Thus, we may say that y_j is a **border set component** if it is calculated by the processor which it belongs to and if at least one column index j associated to the non-void elements a_{ij} from line i, corresponds to a component x_j which does not belong to the processor;
- **-external** are those components which do not belong and by consequence are calculated) to the processor, but which correspond to column indexes associated to non-void elements from the lines belonging to the processor.

In conformity with this organisation, there corresponds to each processor a vector whose components are ordered as follows:

- the first components are numbered from 0 to N_i-1, where N_i is the number of internal components ;
- the next components are border set components and occupy the positions from N_i to N_i+N_f-1 , where

N_f is the number of border set components;

- the last components are external components and occupy the positions comprised between N_i+N_f and $N_i+N_f+N_e-1$, where N_e is the number of external components.

Within this vector associated with a processor, the external components are being ordered so that those which are used by the processor occupy successive positions.

For example let's take A(6, 6) and presuppose that x_0, x_1, x_2 and by consequence the lines 0, 1, 2 of matrix A are allocated to the processor 0; x_3 and x_4 and by consequence the lines 3 and 4 are allocated to the processor 2; x_5 and by consequence line 5 are allocated to the processor 1. The matrix A has the non-void elements marked by a * in the following description.

$$A=\begin{array}{ll} & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{l} \text{proc.0} \left\{\begin{array}{c} 0 \\ 1 \\ 2 \end{array}\right. \\ \text{proc.2} \left\{\begin{array}{c} 3 \\ 4 \end{array}\right. \\ \text{proc.1} \;\; 5 \end{array} & \left(\begin{array}{cccccc} * & * & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 & * \\ * & * & * & * & * & * \\ & & & & & \\ & & & & & \\ & & & & & \end{array}\right) \end{array}$$

For processor 0 which has the lines 0, 1, 2 attached to the matrix A and respectively the components x_0, x_1, x_2, we have:

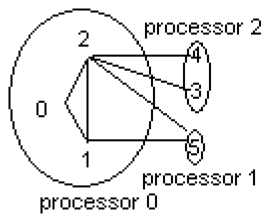
$N_i=1$: a sole internal component y_0 because in calculating the y_0 only there appears only those x_0, x_1, x_2 that belong to the processor 0.

$N_f=2$: two border set components y_1 and y_2 in whose calculus the elements belonging to other processors also appear:

- in the calculus of y_1 there also appears x_5 which belongs to the processor 1
- in the calculus of y_2 there also appears x_5 which belongs to the processor 1 and x_3, x_4 belonging to the processor 2

$N_e=3$: three external components because in the calculus of y_0, y_1, y_2 there appear three components x_5, x_3, x_4 which belong to other processors.

The communication graph corresponding to the processor 0 is defined in the following picture:



To the lines 0, 1, 2, the following vectors correspond, vectors in which the indexes of the columns corresponding to the external components are grouped and sorted into processors:

Line the indexes of columns with non-void elements

0 $\longrightarrow$ 0 1 2
1 $\longrightarrow$ 1 0 2 5
2 $\longrightarrow$ 2 0 1 5 3 4

Each processor has to acknowledge on which of the processors the external components are being calculated: in the above example, processor 1 calculates the component y_5 and processor 2 calculates the components y_3 and y_4. At the same time, each processor has to acknowledge which of its internal components are being used by other processors.

Let's remind the schematic structure of the algorithm CG:

x=initial value
r=b-Ax …
p=r …initial direction

```
repeat
        v=A*p
                ...multiplication matrix-vector
        a=(r^T*r)/(p^T*v)
                ...product "dot"
        x=x+a*p
                ...update solution vector
                ...operation "saxpy"
        new_r=new_r-a*v
                ...update rest vector
                ...operation "saxpy"
        g=(new_r^T*new_r)/(r^T*r)
                ...product "dot"
        p=new_r+g*p
                ...update new direction
                ...operaţtion "saxpy"
        r=new_r
until (new_r^T*new_r suficient de mic)
```

It is noticed that the following operations are necessary in the algorithm CG:

1. A product sparse matrix-vector
2. Three vector updatings (operations "SAXPY")
3. Two scalar products (operations "DOT")
4. Two scalar dividings
5. A scalar comparison for the testing of the convergence

For the parallel implementation of the algorithm CG, the following distinct parts appear:

a) Distribution of the date on processors

The date are being distributed on processors on lines so that each processor has a consecutive number of lines from the sparse matrix assignated:

```
typedef struct tag_dsp_matrix_t
   {
    int N;              /* dimension matrix N×N */
    int row_i, row_f;   /* rank of beginninbg and ending line which belongs to the processor*/
    int nnz;            /* number of non-void elements from the local matrix */
    double* val;        /* elements of the matrix */
    int* row_ptr;       /* beginning of a matrix */
    int* col_ind;       /* column index*/
   } dsp_matrix_t;
```

Each processor will store the rank of the lines belonging to it, the elements of the matrix and two pointers row_ptr and col_ind used in the storing of the compressed on lines of a matrix.

b) In/out operation

In/out operations comprise the reading of the matrix and its stiring in a compressed lines format.

c) Operations on vectors

The operations on vectors are of two types:

- operations "saxpy" for updating of the vectors, which do not require communication between processors;
- operations "dot" (scalar product) which do not require communication between processors with the help of function MPI_Allreduce.

d) Multiplication matrix-vector

Each processor uses at the calculus the lines from the matrix which belong to it, but needs elements of the vector *x* which belong to other processors. This is why a processor receives these elements from the other processors and sends at the same time its part to all the other processors. In this way, we can write schematically the following sequence:

```
new_element_x=my_element_x
for i=0, num_proces
        - send my_element_x to the processor my_proc+i
        - calculate locally with new_element_x
        - receive new_element_x from the processor my_proc-i
Repeat
```

3. THE OPTIMISATION OF THE COMMUNICATION

A given processor does not need the complete part of *x* which belongs to other processors, but only the elements corresponding to the columns which contain non-void elements. At the same time it sends to the other processors only the non-void elements of *x*. This is the reason why the structure presented above comprises the field *col_ind* which indicates the rank of the column that contains a non-void element. In this way, we can schematically write the following sequence:

- each processor creates a mask which indicates the rank of the columns of non-void elements from *A*
- communication between processors:

new_element_x=my_element_x
for i=0, num_proces
- if communication necessary between my_proc and my_proc+i
- transmite my_element_x la procesorul my_proc+i
endif
- calculate locally with new_element_x
- receive new_element_x from processorl my_proc-i
repeat

The algorithm implemented for a matrix of a dimension N×N=960×960, with 8402 non-void elements has given the following results:

* time is expressed in minutes.

Number of processors	Number of iterations	Calculus duration	Duration of communication between processors	Time for the memory allocation	Time for operations with vectors	Total duration
1	205	3.074	0.027	0.002	0.281	3.384
2	205	2.090	0.341	0.002	0.136	2.568
4	204	1.539	0.500	0.002	0.070	2.110

4. THE ANALYSIS OF THE PERFORMANCE OF ALGORITHM CG

The analysis of the performance of algorithm CG is done from the point of view of the time necessary for the execution of the algorithm. In this model the initiation times for the matrix *A* and of the other vectors used is neglectable. At the same time, the time necessary for the verification of the convergence is disregarded and it is presupposed that the initializations necessary for an iteration have been done.

4.1 Analysis of the sequential algorithm

Notations:
m=vectors dimension
N=total number of non-void elements of matrix *A*
k=number of iterations for which the algorithm is executed
T_{comp1s}=total calculus time for the vectors updating (3 operations SAXPY)
T_{comp2s}=total calculus time for the product *Ap* and for the scalar product (*r*, *r*)
T_{comp3s}=total calculus time for the scalar products (*A*, *Ap*) and (*p*, *Ap*)
T_{comp4s}=total calculus time for the scalars α and β
T_{seq}=total calculus time for the sequential algorithm
Then $T_{seq} = T_{comp1s} + T_{comp2s} + T_{comp3s} + T_{comp4s}$

Within the algorithm there are three operations SAXPY, each vectoir being of dimension *m*. If we suppose that t_{comp} is the total calculus time for the multiplication of two real numbers with double precision and for the adding of the results, then

$$T_{comp1s} = 3 * m * k * t_{comp}$$

T_{comp2s} is the total calculus time for the product sparse matrix-vector and for the two scalar products. The product matrix-vector implies N elements and the scalar product implies m elements. Then

$$T_{comp2s} = (N + m) * k * t_{comp}$$

T_{comp3s} is the calculus time of two scalar products and can be written as

$$T_{comp3s} = 2 * m * k * t_{comp} .$$

The calculus for the scalars α and β implies two operations of division and a subtraction of real numbers. Let's take $t_{comp\alpha}$ calculus time for all these operations. Then $T_{comp4s} = 2 * k * t_{comp\alpha}$.

The total calculus time for the sequential algorithm CG is:

$$T_{seq} = (6 * m + N) * t_{comp} + 2 * k * t_{comp\alpha} .$$

4.2 Analysis of the parallel algorithm

Within the parallel algorithm each processor executes k iterations of the algorithm in parallel. We define:

b=dimension of the block from matrix A and from vectors x, r, p belonging to each processor

p=number of processors

T_{comp1p}=total calculus time for the vectors updating on each processor

T_{comp2p}=total calculus and communication time for the Ap and (r, r)

T_{comp3p}=total calculus time for the calculus of the scalar products and of the global communication

T_{comp4p}=total calculus time for the scalars α and β

Here T_{comp1p} is the total time for the calculus of 3b vectors updating. If matrix A is very sparse (the density is smaller than 5 percentages) the communication time exceeds the calculus time. This way T_{comp2p} is taken equal with t_{comm}, the communication time of a block of dimension b to all the p processors. T_{comp3p} implies the global calculus and communication time, noted with t_{glb}. Then:

$$T_{par} = T_{comp1p} + T_{comp2p} + T_{comp3p} + T_{comp4p}$$

where:

$$T_{comp1p} = 3 * b * k * t_{comp}$$

$$T_{comp2p} = t_{comm}$$

$$T_{comp3p} = 2 * b * k * t_{comp} + t_{glb}$$

$$T_{comp4p} = 2 * k * t_{comp\alpha}$$

Therefore, to estimate T_{seq} and T_{par} it is necessary to estimate the values of t_{comp}, $t_{comp\alpha}$, t_{comp} and t_{glb}.

REFERENCES

1. Axelsson O. *Solution of liniar systems of equations: iterative methods*. In: Barker [1] 1-51.
2. Bank R, Chan T. *An analysis of the composite step biconjugate gradient method*. 1992.
3. Bank R, Douglas C. Bank R, Dupont T. *Analysis of atwa-level scheme for solving finite element equations*. Univ. Texas, Austin, 1980.
4. Braess D. *Finite element*. Springer-Verlag 1992.
5. George J. *Solution of liniar systems of equations: Direct methods for finite element problems*. In Barker [1] 52-101.
6. Golub G, Van Loan. *Matrix computations*. Notrh Oxford Academic, Oxford 1983.
7. Ichim I, Marinescu G. *Methods of numeric approximation*. Ed. Acad. Bucureşti 1986.
8. Mandel J. *A multilevel iterative method for symmetric, positive definite problems*. Appl. Math. Optim. 1984.
9. Oden J, Reddy J. *An introduction to the mathematical theory of finite elements*. New Tork, 1976.
10. Ortega J.M.,Voigt R.G. *Solution of partial differential equation on vector and parallel computers*, SIAM Rev 27(1985),149-270
11. Ostrovski,A.M. *On the linear iteration procedures for symmetric matrices*, Rend. Math. Appl.. 14(1964)140-163
12. Samarski A, Nicolaev E. *Numerical methods for grid equations. Iterative methods*. Basel 1989.
13. Ştefanescu A, Zidãroiu C, *Operational research*, Ed. Didacticã şi pedagogicã, Bucureşti, 1981
14. Varga R., *Matrix Iterative Analysis*, Prentice Hall, Englewood Cliff, New Jersey, 1962.

15. Yong D.M. Iterative solution of large linear systems, Academic Press, New York, 1971.
16. ---- *An Introduction to MPI for C Programmers*, The University of Texax at Austin, 1999.
17. ---- *A User's Guide to MPI*, Departament of Mathematics, University of San Francisco, 1998.
18. ---- *MPI: A Message-Passing Interface Standard*,The University of Tennessee,2000