Some Computational Results on MPI Parallel Implementation of Dense Simplex Method

El-Said Badr, Mahmoud Moussa, Konstantinos Paparrizos, Nikolaos Samaras, and Angelo Sifaleras

Abstract—There are two major variants of the Simplex Algorithm: the revised method and the standard, or tableau method. Today, all serious implementations are based on the revised method because it is more efficient for sparse linear programming problems. Moreover, there are a number of applications that lead to dense linear problems so our aim in this paper is to present some computational results on parallel implementation of dense Simplex Method. Our implementation is implemented on a SMP cluster using C programming language and the Message Passing Interface MPI. Preliminary computational results on randomly generated dense linear programs support our results.

Keywords—Linear Programming, MPI, Parallel Implementation, Simplex Algorithm.

I. INTRODUCTION

LINEAR programming (LP) is the most important and well studied optimization problem. The simplex algorithm which developed by Dantzig [1] had fascinated researchers for many years because its performance on real world problems is usually better than the theoretical worst case. It is well-known that the simplex algorithm is not polynomial. Despite this, Borgwardt [2] proved that the expected number of iterations of the simplex algorithm is polynomial when it is applied for practical problems solving. The main computational disadvantage of the simplex algorithm is that the total number of iterations can not be predicted. As dimension n increases, the computational time rise up exponentially. The simplex algorithm searches for an optimal solution by moving from one feasible solution to another, along the edges of the feasible set, always in a cost reducing direction. This computational behavior makes parallel solution of linear optimization problems an attractive and promising research area.

Parallel implementations of linear programming algorithms have been studied extensively in the recent years [3]–[5].

Most of the real world linear programming problems are extremely sparse. Nevertheless, dense linear programming problems have important applications. Specifically, some kinds of decomposition (for example Benders, Dantzig–Wolfe) generate full dense linear problems. More applications leading to dense linear programming problems can be found in [6].

In this paper we examine the computational performance of a parallel version of the simplex algorithm on dense linear programming problems. We perform experiments using the communication package Message Passing Interface (MPI) [7]–[8]. Parallel implementations of the simplex algorithm vary in the way that the simplex tableau is distributed among the processors. Specifically, our preliminary results reveal that if the parallel version of the simplex algorithm run in 8 processors then we can achieve a speed-up factor of about 5 times faster comparing with the sequential version of the simplex algorithm.

The paper is organized as follows: A sequential version of the simplex algorithm is presented in Section 2. In Section 3 we give a brief description of a parallel version of the simplex algorithm. To continue with, some preliminary computational results on randomly generated test instances are reported in Section 4. Finally, conclusions and future research directions are discussed in Section 5.

II. SEQUENTIAL VERSION OF THE SIMPLEX ALGORITHM

In linear programming problems, we minimize or maximize a linear function of real variables over a region defined by linear constraints. The mathematical formulation of the linear programming problem, in standard form, is shown in LP.1:

$$
\begin{aligned}
\text{Minimize} \quad & z = c^T x \\
\text{s.t.} \quad & Ax = b \quad\quad\quad (\text{LP.1}) \\
& x \geq 0
\end{aligned}
$$

where $c, x \in \Re^n$, $b \in \Re^m$, $A \in \Re^{m \times n}$ and T denotes transposition. We assume that the set of basis vector (columns of A) is linearly independent. The simplex algorithm consists of two steps: (a) a way of finding out whether a current basic feasible solution is an optimal solution and (b) a procedure of obtaining an adjacent basic feasible solution with the same or better value for the objective function. We shall describe the simplex algorithm using the tableau format. This description has many advantages. It is more efficient for full dense linear problems and it can be easily convert to a distributed version

with a loosely coupled system. A tableau is an $(m+1)\times(m+n+1)$ matrix of the following form:

	x_1	x_2	…	x_n	x_{n+1}	…	x_{n+m}	z	
	$-c_1$	$-c_2$	…	$-c_n$	0	…	0	1	0
x_{n+1}	a_{11}	a_{12}	…	a_{1n}	1	…	0	0	b_1
x_{n+2}	a_{21}	a_{22}	…	a_{2n}	0	…	0	0	b_2
…	…	…	…	…	…	…	…	…	…
x_{n+m}	a_{m1}	a_{m2}	…	a_{mn}	0	…	1	0	b_m

Notice that the entries in the second row are the coefficients of the objective function z and the rightmost column is the value of the objective function for the initial basic feasible solution. Now we can proceed with a formal description of the sequential simplex algorithm:

Sequential Simplex Algorithm

Step 0: **(Initialization)**
Start with a feasible basic solution and construct the corresponding simplex tableau.

Step 1: **(Choice of entering variable)**
If $a_{0j} \geq 0$, j=1, 2, …, *n*, STOP. The current solution is optimal. Otherwise, choose the entering variable using the following pivoting rule:
$a_{0s} = \min\{a_{0j}:\ a_{0j} < 0,\ j = 1,2, …, n\}$

Step 2: **(Choice of leaving variable)**
Let I = {i : a_{is}> 0}. If I=∅, STOP. The problem (LP.1) is unbounded. Otherwise, choose the leaving variable using the minimum ratio test:
$$\frac{b_r}{a_{rs}} = \min\left\{\frac{b_i}{a_{is}}: i \in I\right\}$$

Step 3: **(Pivoting)**
The pivoting element is the variable a_{rs}. Construct the next simplex tableau as follows:

Let $\quad a_{rj} \leftarrow \dfrac{a_{rj}}{a_{rs}}$, j=1, 2, … ,n, n+1

and

$a_{ij} \leftarrow a_{ij} - \dfrac{a_{rj}}{a_{rs}}a_{is}$, i= 0, 1, 2, . . . m (i ≠ r),

$\quad$ j = 1, 2, … , n, n+1
Go to Step1.

The simplex algorithm uses the Gauss-Jordan transformation of the tableau to move from one basic feasible solution to another. Each iteration of the simplex algorithm is relatively expensive. This can be seen by examining the previous formal description of the simplex algorithm. More precisely, the number of multiplications and additions at each iteration is approximately equal to *m(m-n)+n+1* and *m(n-m+1)* respectively, where *m* is the number of constraints and *n* is the number of variables. This happens whenever *n* and *m* are large, in which case nearly 100% of the cpu-time is spent in Step 3 (Pivoting). In this third Step, a multiple of row *r* is added to row I, (this is the only double nested loop executed at each iteration).

III. MPI PARALLEL VERSION OF THE SIMPLEX ALGORITHM

We implemented the simplex method using a straightforward application of the C language tools and the Message Passing Interface (MPI). The storage of the simplex tableau was carried out using a $(0:m)\times(0:m+n)$ dimensioned array. Also, in order to allocate the work across multiple processors, MPI was used. The coefficients of the objective function are represented using the *q* vector, the number of processors is denoted by NPRS, the current tableau is denoted by TABL and finally the rank of processor is denoted by ITSK.

Parallel Simplex Algorithm
Begin
1- for $0 \leq i < m + n$ do
$\quad$ In processor 0 : Set *q[i]:=TABL[0][i]*
$\quad$ for $0 \leq k < NPRS$ pardo
$\quad\quad$ {for $0 \leq i < m / NPRS$ do
$\quad\quad\quad$ for $0 \leq j < m + n$ do
$\quad\quad\quad\quad$ Set *C[i][j]:=TABL[k*m/ NPRS+i]*
$\quad\quad\quad$ for $0 \leq j < m / NPRS$ do
$\quad\quad\quad\quad$ Set *b[j] := TABL[j][M+N]*
$\quad\quad\quad\quad$ }
2- In processor 0
$\quad$ for $0 \leq i < n$ do
$\quad\quad$ search *i* (where *q[i]<0*) Set *column := i*
$\quad$ if failure Goto (10).
3- for $0 \leq k < NPRS$ pardo
$\quad\quad$ *X:=min(b[j]/C[j][column], $0 \leq j < m / NPRS$)*
$\quad\quad$ *Set row_no:=j*
4- for $0 \leq k < NPRS$ pardo
$\quad\quad$ Send (*X,ITSK*) to processor 0
$\quad\quad$ In processor 0
$\quad\quad$ Search *ITSK* which corresponds to *min (X)*
$\quad\quad$ *min_L:=ITSK*
5- From processor *min_L*
$\quad\quad$ Send row *g:=C[row_no][:]* to all processor
$\quad\quad$ Send variable *h:=b[[row_no]* to all processor
6- for $0 \leq k < NPRS$ pardo
$\quad\quad$ for $0 \leq i < m / NPRS$ do{
$\quad\quad\quad$ Set *a:=C[i][column]/g[column]*
$\quad\quad\quad$ for $0 \leq j < m + n$ do
$\quad\quad\quad\quad$ Set *C[i][j]:=C[i][j]-a*g[j]*
$\quad\quad\quad\quad$ Set *b[i]:=b[i]-a*h*
$\quad\quad\quad\quad$ }
7- In processor *min_L*
$\quad\quad$ for $0 \leq j < m + n$ do{
$\quad\quad\quad$ set *C[row_no][j] := g[j]/g[column]*
$\quad\quad\quad$ set *b[row_no] := h / g[column]*
$\quad\quad\quad\quad$ }
8- In processor 0
$\quad\quad$ *a:=q[column]/g[column]*
$\quad\quad$ For $0 \leq i < m + n$ do {

Set *q[i] := q[i]-a*g[i]*
Set *q[n+m] := q[m+n] - a*h*
}

9- Goto (2)
10- For $0 \leq i < m+n$ do
In processor 0
Set *TABL[0][i] := q[i]*
for $0 \leq k < NPRS$ pardo{
for $0 \leq i < m/NPRS$ do
For $0 \leq j < m+n$ do
Set *TABL[k*m/NPRS+i][j] := C[i][j]*
for $0 \leq j < n/NPRS$ do
Set *TABL[j][m+n] := b[j]*
}

End.

IV. COMPUTATIONAL RESULTS

The algorithm described in Section 3 has been experimentally implemented. In this Section, the numerical experiments are presented. It must be mentioned that the computational results demonstrate a speedup for traditional simplex algorithm on dense linear programs.

All test runs were carried out on 16 uniprocessors Intel Pentium III 500MHz with 512 KB L2 Cache. The processors were interconnected using Fast Ethernet and Scalable Coherent Interface (SCI). Furthermore, the machine precision was 32 decimal digits. The reported CPU times were measured in seconds. MPI implementation MPICH v.1.2.6 was used and appropriately configured for our cluster. Usage of this machine was provided by the National Technical University of Athens, School of Electrical and Computer Engineering.

We run a total of 30 random dense linear problems. In our computational study we use only square problems (*nxn*). This dimension case includes three different classes of problems corresponding to the values *n* = 200, 300, 400; each of these classes contains ten random dense linear programs. The dense linear optimization problems that have been solved are of the general form

Minimize $c^{T}x$
s.t.   Ax $\leq$ b
     x $\geq$ 0

The planes of the constraints are tangent on a sphere, so that its center is feasible. Also, these problems have a feasible region that is a closed polyhedron. The ranges of values, being used for randomly generated linear programs for all problems, are $c \in [1\ 500]$, $A \in [-700\ 1800]$, *r* = 7 and center = 35. The feasibility tolerance used is 10^{-8} and the tolerance on a pivot row and column is 10^{-10}.

For each problem size we bring together some statistics on the test instances used in our computational study. All the results are summarized using a table and a figure. In Tables I-III we present our computational results. The first column shows the number of processors, the second the average number of iteration, the third the average cpu-time, the fourth the cpu-time per iteration and the last one shows the speed-up factor among different number of processors.

TABLE I
PROBLEM SIZE (200x200)

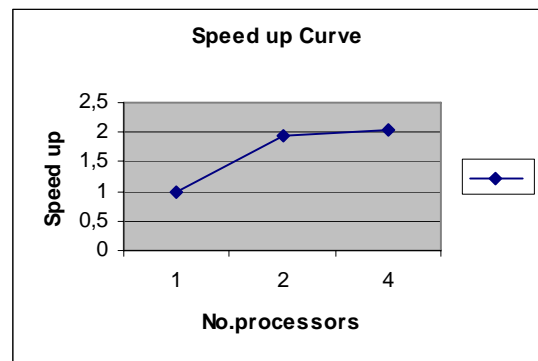| No. processors	niter	Time(secs.)	Secs./niter	Speed up
1	382.1	1.9571	0.0051	1
2	382.1	1.0086	0.0026	1.9404
4	382.1	0.9658	0.0025	2.0263

Fig. 1 Speed up Curve for Problem Size (200x200)

In Fig. 1 one can see that as the number of processors increases (and less than 4 processors) the speed up also increases. Our experiments show that if we use more than 4 processors, then we have not speeded up. This fact can be justified because the communication time is more than the computational time for problem size (200x200).

TABLE II
PROBLEM SIZE (300x300)

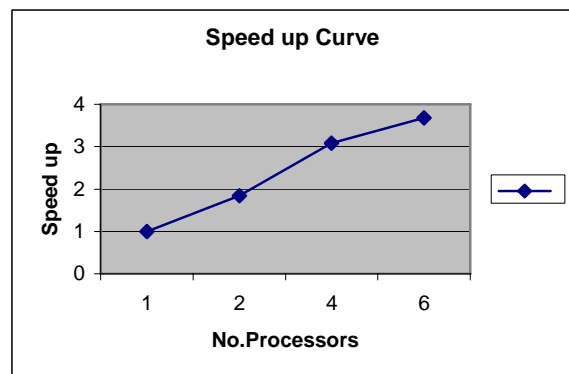| No. processors	niter	Time(secs.)	Secs./niter	Speed up
1	636.4	8.9318	0.0140	1
2	636.4	4.8551	0.0076	1.8397
4	636.4	2.9004	0.0045	3.0795
6	636.4	2.4307	0.0038	3.6746

Fig. 2 Speed up Curve for Problem Size (300x300)

Moreover, in Fig. 2 one can see that as the number of processor increases (and less than 6 processors) the speed up also increases. Our experiments show that if we use more than 6 processors, then we have not speeded up. This happens, because the communication time is more than the computational time for problem size (300x300). It can be seen that, when we increase the problem size from (200x200) to (300x300), we have speed up as long as we use less than 6 processors.

TABLE III
PROBLEM SIZE (400x400)

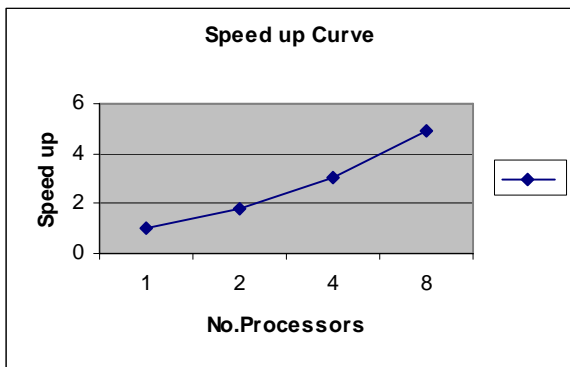| No. processors	niter	Time(secs.)	Secs./niter	Speed up
1	897	22.3404	0.0249	1
2	897	12.4163	0.0138	1.7992
4	897	7.3259	0.0082	3.0495
8	897	4.5529	0.0051	4.9067

Fig. 3 Speed up Curve for Problem Size (400x400)

In Fig. 3 one can observe that, as the number of processor increases (and less than 8 processors), the speed up also increases. Our experiments show that if we use more than 8 processors, then we have not speeded up. This happens again, because the communication time is more than the computational time for problem size (400x400). We can also see that when we increase the problem size from (300x300) to (400x400) we have speed up, as long as we use less than 8 processors.

V. CONCLUSION

We have presented a parallel implementation of the simplex algorithm using the Message Passing Interface. The proposed implementation has an advantage. It leads to important reduction in the total solution time of a linear programming problem. The performance analysis also, shows that the speed-up obtained is highly sensitive to communication among the processors.

ACKNOWLEDGMENT

REFERENCES

[1] G. B. Dantzig, *Linear Programming and Extensions*. NJ: Princeton, Princeton University Press, 1963.
[2] K. H. Borgwardt, "Some distribution independent results about the asymptotic order of the average number of pivot steps in the simplex method", *Mathematics of Operations Research*, vol. 7, no. 3, pp. 441-462, 1982.
[3] I. Maros, and G. Mitra, "Investigating the sparse simplex method on a distributed memory multiprocessor", *Parallel Computing*, vol. 26, pp. 151-170, 2000.
[4] D. Klabjan, L. E. Johnson, and L. G. Nemhauser, "A parallel primal-dual simplex algorithm", *Operations Research Letters*, vol. 27, no. 2, pp. 47-55, 2000.
[5] J. Eckstein, I. Bodurglu, L. Polymenakos, and D. Goldfarb, "Data-Parallel Implementations of Dense Simplex Methods on the Connection Machine CM-2", *ORSA Journal on Computing*, vol. 7, no. 4, pp. 402-416, 1995.
[6] S. P. Bradley, U. M. Fayyad, and O. L. Mangasarian, "Mathematical Programming for Data Mining: Formulations and Challenges", *INFORMS Journal on Computing*, vol. 11, no. 3, pp. 217-238, 1999.
[7] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.
[8] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing-Interface*. MIT Press, 1994.