



Highly Scalable Parallelization of Standard Simplex Method on a Myrinet-Connected Cluster Platform

Basilis Mamalis, Grammati Pantziou, Georgios Dimitropoulos & Dimitris Kremmydas


To cite this article: Basilis Mamalis, Grammati Pantziou, Georgios Dimitropoulos & Dimitris Kremmydas (2013) Highly Scalable Parallelization of Standard Simplex Method on a Myrinet-Connected Cluster Platform, International Journal of Computers and Applications, 35:4, 152-161

To link to this article: <https://doi.org/10.2316/Journal.202.2013.4.202-3691>



Published online: 11 Jul 2015.



Submit your article to this journal 



Article views: 6

HIGHLY SCALABLE PARALLELIZATION OF STANDARD SIMPLEX METHOD ON A MYRINET-CONNECTED CLUSTER PLATFORM

Basilis Mamalis,* Grammati Pantziou,* Georgios Dimitropoulos,* and Dimitris Kremmydas**

Abstract

The simplex method has been successfully used in solving linear programming problems for many years. Parallel approaches have also extensively been studied due to the intensive computations required, especially for the solution of large linear problems (LPs). In this paper we present a highly scalable parallel implementation framework of the standard full tableau simplex method on a highly parallel (distributed memory) environment. Specifically, we have designed and implemented a suitable column distribution scheme as well as a row distribution scheme and we have entirely tested our implementations over a considerably powerful distributed platform (linux cluster with myrinet interface). We then compare our approaches (a) among each other for variable number of problem size (number of rows and columns) and (b) to other recent and valuable corresponding efforts in the literature. In most cases, the column distribution scheme performs quite/much better than the row distribution scheme. Moreover, both schemes (even the row distribution scheme over large-scale problems) lead to particularly high speedup and efficiency values, which are considerably better in all cases than the ones achieved in other similar research efforts and implementations. Moreover, we further evaluate our basic parallelization scheme over very large LPs in order to validate more reliably the high efficiency and scalability achieved.

Key Words

Parallel processing, linux clusters, linear programming, simplex algorithm, message passing interface

1. Introduction

Linear programming is the most important and well-studied optimization problem. The simplex method, which

can be found in many textbooks [1], has been successfully used for solving linear programming problems for many years. Parallel approaches have also extensively been studied due to the intensive computations required [2]–[15].

Most research (with regard to sequential simplex method) has been focused on the revised simplex method since it takes advantage of the sparsity that is inherent in most linear programming applications. The revised method is also advantageous for problems with a high aspect ratio; that is, for problems with many more columns than rows. However, there have not been seen many parallel/distributed implementations of the revised method that scale well [2]. On the other hand, the standard method is more efficient for dense linear problems and it can be easily converted to a distributed/parallel version with satisfactory speedup values and good scalability [2]–[5]. Also, lately, some alternative very promising efforts have been made, based on the block angular structure (or decomposition) of the initially transformed problems [6], [7].

Earlier work focused mainly on more complex, and more tightly coupled, networking structures (than a cluster or a network of workstations). Several attempts have also been made over distributed shared memory (DSM) environments [8]. Hall and McKinnon [9] and Shu and Wu [10] worked on parallel revised methods. Thomadakis and Liu [11] worked on the standard method utilizing the MP-1 and MP-2 MasPar. Eckstein *et al.* [12] showed in the context of the parallel connection machine CM-2 that the iteration time for parallel revised method tended to be significantly higher than for parallel full tableau method even when the revised method is implemented very carefully. Stunkel [13] found a way to parallelize both the revised and standard methods so that both obtained a similar advantage in the context of the parallel Intel iPSC hypercube. Two other valuable attempts are presented in [14] and [15] following the primal–dual simplex algorithm and the sparse simplex method, respectively, and they have led to quite satisfactory results for large-scale problems.

As mentioned above, the standard method can be easily and effectively extended to a coarse-grained, distributed

* Department of Informatics, Technological Educational Institute of Athens, Greece; e-mail: {vmamalis, pantziou, cs000055}@teiath.gr

** Department of Agricultural Economics and Rural Development, Agricultural University of Athens, Greece; e-mail: kremmydas@aia.gr

Recommended by Dr. M Hamza

(DOI: 10.2316/Journal.202.2013.4.202-3691)

algorithm [3]. It should also be noted that although dense problems (which are most suited for the standard method) are uncommon in general, they do occur frequently in some important applications within linear programming [12]. Included among those are wavelet decomposition, image processing [16], [17], and digital filter design [18], [19]. Other applications leading to dense linear programming problems can be found in [20]. All these problem groups are well suited to the standard method. Moreover, in distributed implementations, aspect ratio becomes less of an issue.

Furthermore, existing distributed memory implementations of the standard simplex method naturally vary in the way that the simplex tableau is distributed among the processors [2]. Either a column distribution scheme or a row distribution scheme may be applied, depending on several parameters (relative number of rows and columns, total size of the problem, target hardware environment details, *etc.*). The most recent valuable efforts following the column distribution scheme (which is the most popular and widely used one for practical problems) have been done by Yarmish and Slyke [3] yielding to high parallel efficiency and corresponding scalability for large-scale problems, as well as (older) by Qin and Nguyen [5]. On the other hand, quite recently, Badr *et al.* [4] followed the row distribution scheme and presented a well-designed quite valuable implementation on eight loosely coupled processors (interconnected with Fast Ethernet and Scalable Coherent Interface (SCI) interface), targeting and achieving significant speedup (up to five) when solving small random dense linear programming problems.

In our work, we focus on the parallelization of the standard full tableau simplex method and we present two highly scalable parallel implementations designed for distributed memory environments. We have designed and implemented both the above-mentioned data distribution schemes: a column distribution scheme (similar to [3]) and a row distribution scheme (similar to [4]), in order to compare their performance and scalability under several parameters. Furthermore, we have experimentally evaluated our implementations over a considerably powerful parallel environment; a linux cluster of 16 powerful 3.0 GHz XEON processors with 4 GB RAM each that are connected via a dedicated (low latency) Myrinet network interface of 10 Gbps communication speed.

Specifically, we have tested and compared our approaches among each other for varying number of processors and varying problem size, as well as to the

corresponding implementations of [3] and [4] which (as mentioned above) are two of the most recent and valuable corresponding efforts. In most cases, the column distribution scheme performs quite/much better than the row distribution scheme, and leads to almost linear speedup for large and very large linear problems with high aspect ratio. The row distribution scheme performs better only for small problems as well as for specific problems where the number of rows is significantly greater than the number of columns. Moreover, both schemes lead to particularly high speedup and efficiency values for typical test LPs that are considerably better in all cases than the ones achieved by the corresponding implementations of [3] and [4].

An earlier version of our work (of significantly less extent and results) has been published in [21]. The rest of the paper is organized as follows. In Section 2, a brief description of the standard full tableau simplex method is given. In Section 3, we present the two different parallel approaches (row and column distribution schemes) followed in our implementations with a brief discussion of their basic theoretical advantages and disadvantages. Section 4 outlines the extended experimental results taken over our powerful linux cluster environment, whereas Section 5 concludes the paper.

2. The Simplex Method

In linear programming problems, the goal is to minimize (or maximize) a linear function of real variables over a region defined by linear constraints. In standard form, it can be expressed as shown in Table 1, where A is an $m \times n$ matrix, x is an n -dimensional design variable vector, c is the price vector, b is the right-hand side vector of the constraints (m -dimensional), and T denotes transposition. We assume that the set of basis vector (columns of A) is linearly independent.

We focus here on the standard full tableau format of the simplex method, which is more efficient for dense linear problems and it can be easily converted to a distributed version for cluster computer systems. A tableau is an $(m+1) \times (m+n+1)$ matrix of the form given in Table 1.

Based on the full tableau representation, the basic steps of the standard simplex method can be summarized (without loss of generality) as follows:

- **Step 0:** Initialization: start with a feasible basic solution and construct the corresponding tableau.

Table 1
Standard Simplex Method Formulation and Full Tableau Representation

Standard Full Tableau Simplex Method			x_1	x_2	\dots	x_n	x_{n+1}	\dots	x_{n+m}	z	
			$-c_1$	$-c_2$	\dots	$-c_n$	0	\dots	0	1	0
Minimize	$z = c^T x$	x_{n+1}	a_{11}	a_{12}	\dots	a_{1n}	1	\dots	0	0	b_1
	s.t. $Ax = b$	x_{n+2}	a_{21}	a_{22}	\dots	a_{2n}	0	\dots	0	0	b_2
	$x \geq 0$	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots
		x_{n+m}	a_{m1}	a_{m2}	\dots	a_{mn}	0	\dots	1	0	b_m

- **Step 1:** Choice of entering variable: find the winning column (the one having the larger negative coefficient of the objective function – entering variable).
- **Step 2:** Choice of leaving variable: find the winning row (apply the minimum ratio test to the elements of the winning column and choose the row number with the minimum ratio – leaving variable).
- **Step 3:** Pivoting (involves the most calculations): construct the next simplex tableau by performing pivoting in the previous tableau rows based on the new pivot row found in the previous step.
- **Step 4:** Repeat the above steps until the best solution is found or the problem gets unbounded.

3. The Implemented Parallel Approaches

In the following paragraphs, we briefly present the algorithmic approaches that we followed in our parallel implementations for the two different distribution schemes. As it can be easily observed the two schemes mainly differ on the part of the algorithm that remains essentially non-parallelized as well as on how the distribution scheme affects the parallelization overhead of Step 3.

3.1 The Column Distribution Scheme

First, we focus on the most popular and widely used column-based distribution scheme. This is a relatively straightforward parallelization scheme within the standard simplex method which involves dividing up the columns of the simplex table among all the processors and it is theoretically regarded as the most effective one in the general case. Following this scheme all the computation parts except Step 2 of the basic algorithm are fully parallelized. Additionally, this form of parallelization looks as the most natural choice since in most practical problems the number of columns is larger than the number of rows. The basic steps of the parallel simplex algorithm (similar to [3]) that we have implemented with regard to the column-based distribution scheme are given below:

- **Step 0:** The simplex table is shared among the processors by columns. Also, the right-hand constraints vector is broadcasted to all processors.
- **Step 1:** Each processor searches in its local part and chooses the locally best candidate column – the one with the larger negative coefficient in the objective function part (local contribution for the global determination of the entering variable).
- **Step 2:** The local results are gathered in parallel and the winning processor (the one with the larger negative coefficient among all) is found and globally known. At the end of this step each processor will know which processor is the winner and has the global column choice.
- **Step 3:** The processor with the winning column (entering variable) computes the leaving variable (winning row) using the minimum ratio test over all the winning column's elements.

- **Step 4:** The same (winning) processor then broadcasts the winning column as well as the winning row's id to all processors.
- **Step 5:** Each processor performs (in parallel) on its own part (columns) of the table all the calculations required for the global rows pivoting, based on the pivot data received during Step 4.
- **Step 6:** The above steps are repeated until the best solution is found or the problem gets unbounded.

3.2 The Row Distribution Scheme

Alternatively, a row-based partitioning scheme can be followed in order to take advantage of some probable inherent parallelism in the computations of Step 2 (*i.e.*, the computation of the leaving variable/winning row), which is not parallelized at all in the column distribution scheme. We have implemented such a row distribution parallel scheme (similar to [4]), based on the following steps:

- **Step 0:** The simplex table is shared among the processors by rows. The right-hand constraints vector is also shared in the same way.
- **Step 1:** Processor 0 finds the best candidate (winning) column (choice of the entering variable) and broadcasts this column's id to all processors.
- **Step 2:** Each processor then computes locally the minimum ratio for the winning column's elements of its rows (local contribution for the global minimum ratio test).
- **Step 3:** The local results are gathered in parallel and the winning processor (this one with the minimum ratio among all – choice of the leaving variable) is found and globally known.
- **Step 4:** The new pivot row is determined (constructed) and then it's broadcasted to all the other processors by the winning processor.
- **Step 5:** Each processor performs (in parallel) a pivot on its own rows of the table, based on the new pivot row (that was just received from the winning processor during the previous step).
- **Step 6:** The above steps are repeated until the best solution is found or the problem gets unbounded.

In the above manner the computational task of Step 2 is reduced to almost $O(m/p)$ in each processor (where m is the number of rows and p is the number of processors) instead of $O(m)$ in the column distribution scheme, whereas a new communication task of only $O(\log p)$ size is introduced. Consequently, the performance gains are expected to be quite significant for linear problems with large number of rows. In addition, it is expected to perform better than the column distribution scheme for linear problems where the number of rows is larger than the number of columns, as well as to be highly competitive for problems with low aspect ratio.

On the other hand, the above parallelization of Step 2 (although it is certainly an advantage in the general case) it has been experimentally proved not to be a significant benefit in some cases and usually tends to be a relative disadvantage when the number of rows is small and the

Table 2
Comparing to the Implementation of [3] ($1,000 \times 5,000$)

P	Yarmish and Slyke [3]			Our Implementation		
No. of Processors	Time/Iteration	$S_p = T_1/T_P$	$E_p = S_p/P$ (%)	Time/Iteration	$S_p = T_1/T_P$	$E_p = S_p/P$ (%)
1	0.61328	1.00	100.0	0.27344	1.00	100.0
2	0.31150	1.97	98.4	0.13713	1.99	99.7
3	0.21724	2.82	94.1	0.09225	2.96	98.8
4	0.15496	3.96	98.9	0.06877	3.98	99.5
5	0.13114	4.68	93.5	0.05592	4.89	97.8
6	0.10658	5.75	95.9	0.04636	5.90	98.3
7	0.09128	6.72	96.0	0.03958	6.91	98.7
8				0.03453	7.92	99.0

communication network among the processors is not quite fast (because the corresponding computational task becomes quite small and naturally the corresponding communication task dominates).

Another basic difference is that here (row-based distribution) an extra broadcast of a row (the new pivot row) is needed instead of an extra broadcast of a column (the winning column) that was required above in the column distribution scheme. This difference is expected (theoretically) to lead to better results for the row distribution scheme when the number of columns decreases, whereas the opposite is expected to happen when the number of columns increases.

4. Experimental Results

The two outlined parallel simplex algorithms (row and column distribution schemes) have been implemented with use of the MPI-2 (MPICH-MX flavour) message passing library and they have been extensively tested (in terms of speedup and efficiency measures) over a powerful distributed memory environment (linux cluster), yielding to very good results; considerably better in all cases than the ones achieved by the corresponding implementations of [3] and [4]. The speedup for p processors (S_p) is computed as the time required for the execution in one processor divided by the time required for the execution in p processors, whereas the efficiency for p processors (E_p) is computed as the speedup achieved in p processors divided by p (and represents the fraction of the maximum theoretical speedup that has been achieved). Our linux cluster consists of 16 powerful 3.0 GHz XEON processors with 4 GB RAM each, and a dedicated (low latency) Myrinet-based network interface with 10 Gbps communication speed. This platform is quite more powerful than the corresponding platforms used for experimental testing in [3] and [4]; it can be regarded as a typical representative of modern distributed memory cluster platforms able to support high-performance parallel implementations.

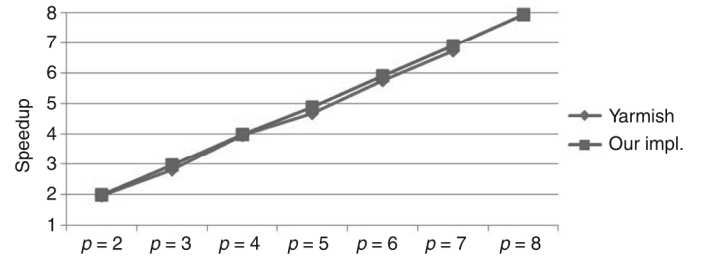


Figure 1. Speedup curves for the results of Table 2.

4.1 Comparing to the Implementation of [3]

In order to compare our approach to the one presented in [3] (which is the most recent and valuable related attempt for distributed memory platforms, using the column distribution scheme) we have run our relevant implementation over the large size ($1,000 \times 5,000$) linear problem presented there [3], with the same characteristics, and we have measured the execution time per iteration for 1, 2, ... up to 8 processors. We have to note that this problem is a large-scale problem with many more columns than rows, so it is expected to have good speedup with use of the column distribution scheme anyway. Note also that the parallel platform used in the experiments of [3] consisted of seven dedicated processors (the exact configuration is not mentioned) connected via Fast Ethernet network interface.

The corresponding results (in terms of speedup and efficiency measures – based on the execution time per iteration) for varying number of processors are presented in Table 2 and Fig. 1. Observing the results of Table 2, first it can easily be noticed that the execution times (in one or more processors) of our algorithm are much better than the ones of [3], which however was expected due to the fact that our platform is quite more powerful than the platform of [3].

The most important, the achieved speedup values of our implementation are also better than the ones achieved in [3]. To be more precise, they are slightly better for two

Table 3
Comparing to the Implementation of [4] (400×400)

P	Badr <i>et al.</i> [4]			Our Implementation		
No. of Processors	Time/Iteration	$S_p = T_1/T_P$	$E_p = S_p/P$ (%)	Time/Iteration	$S_p = T_1/T_P$	$E_p = S_p/P$ (%)
1	0.02491	1.00	100.0	0.01360	1.00	100.0
2	0.01384	1.80	90.0	0.00723	1.88	94.1
4	0.00817	3.05	76.2	0.00385	3.53	88.3
8	0.00508	4.91	61.3	0.00209	6.51	81.4

and four processors (which are powers of two – 1.99 versus 1.97 and 3.98 versus 3.96, respectively) and quite better for 3, 5, 6, and 7 processors (*i.e.*, 4.89 versus 4.68 for five processors and 6.91 versus 6.72 for seven processors). Furthermore, observing the corresponding efficiency values in the last column someone can easily notice the high scalability (higher and smoother than in [3]) achieved by our implementation. Note also that the achieved speedup remains very high (close to the maximum/speedup = 7.92, efficiency = 99.0%) even for eight processors.

Moreover, consider that the implementation of [3] has also been compared to MINOS, a well-known implementation of the revised simplex method, and it has been shown to be highly competitive, even for very low density problems (the reader may refer to [3] for more details). We have to note here that the performance of our simplex implementations remains unchanged with changes of density.

4.2 Comparing to the Implementation of [4]

In order to compare our approach to the one presented in [4] (which is the most recent found attempt for distributed memory platforms using the row distribution scheme) we have run our corresponding implementation over the three classes of small- and medium-sized (200×200 , 300×300 , 400×400) linear problems presented there [4], with the same characteristics, and we have measured the execution time per iteration for number of processors up to eight as in [4]. We have to note that these problems are relatively small-sized linear problems, so it is normally expected not to have quite good speedup performance, especially with use of the row distribution scheme. Additionally, for small-sized problems it is natural that the speed of the network plays an even more important role in the parallel implementation. Note also that the parallel/distributed platform used in the experiments of [4] consisted of eight Intel Pentium processors connected via Fast Ethernet and SCI.

The corresponding results for problem size 400×400 and varying number of processors are presented in Table 3 and Fig. 2. The corresponding results for the other two problem sizes (200×200 and 300×300) can be found in [21]. Observing the obtained results, first, it can easily be noticed that the execution times (in one or more processors) of our algorithm are much better than the ones of [4], which

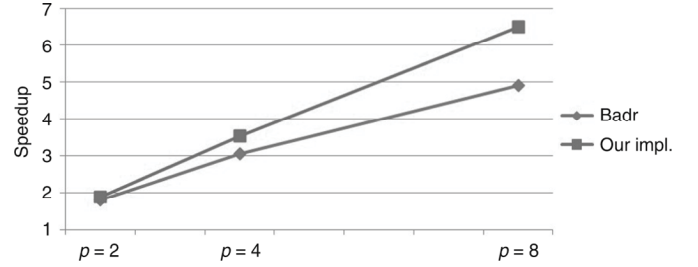


Figure 2. Speedup curves for the results of Table 3.

however was expected here too due to the fact that our platform is quite more powerful than the platform used in [4]. Second, it's also clear that in all cases the achieved speedup and efficiency values of our implementation are quite better or much better than the ones of [4], except the special case of two processors where the results of [4] are quite close (slightly worse) to our results.

Considering in more details the case of problem size 400×400 (Table 3), in [4] a very good speedup (1.80) is achieved for two processors, it remains quite good for four processors (3.05) and continues to increase even for eight processors (and becomes equal to 4.91 which is the maximum speedup value achieved in the experiments presented in [4]). In our implementation, we achieve slightly better speedup for two processors (1.88), quite better speedup for four processors (3.53) and much better speedup for eight processors (6.51) which means that our algorithm scales quite well (with the increase of the number of processors – see also Fig. 2) even for this relatively small-sized problem. Concluding, it's clear that our row distribution scheme implementation (taking advantage, among else, from the particularly high-speed and low-latency interconnection network used) achieves quite good parallel performance even for small- and medium-sized problems; much better in most cases than the implementation of [4].

4.3 Comparing the Two Distribution Schemes

In order to compare the performance of the two different data distribution schemes among each other, we have run on our platform a suitable subset of the well-known and widely used NETLIB test linear problems, as well as an additional subset of random linear problems of our own of varying sizes. The corresponding results (execution times,

Table 4
Comparing the Two Distribution Schemes (Execution Times)

Linear Problems		Column Distribution Scheme				Row Distribution Scheme			
		4 Processors		8 Processors		4 Processors		8 Processors	
	No. of Iterations	Time (s)		Time (s)		Time (s)		Time (s)	
		1 Iteration	Total	1 Iteration	Total	1 Iteration	Total	1 Iteration	Total
SC50A (50×48)	77	0.00051	0.040	0.00039	0.030	0.00050	0.039	0.00036	0.028
ADLITTLE (56×97)	208	0.00096	0.199	0.00064	0.133	0.00106	0.220	0.00077	0.160
SHARE2B (96×79)	199	0.00102	0.202	0.00069	0.137	0.00098	0.194	0.00064	0.128
SC105 (105×103)	191	0.00115	0.219	0.00076	0.145	0.00113	0.216	0.00075	0.143
BRANDY (220×249)	2,952	0.00286	8.434	0.00164	4.853	0.00303	8.943	0.00188	5.554
AGG (488×163)	2,151	0.00523	11.26	0.00292	6.288	0.00506	10.87	0.00276	5.945
AGG2 (516×302)	2,414	0.00854	20.62	0.00464	11.20	0.00842	20.33	0.00450	10.87
BANDM (305×472)	2,875	0.00494	14.21	0.00270	7.752	0.00538	15.46	0.00305	8.767
SCFXM3 ($990 \times 1,371$)	7,147	0.02239	160.0	0.01183	84.55	0.02376	169.8	0.01304	93.22
RANDOM1 (200×200)	592	0.00179	1.061	0.00121	0.718	0.00178	1.051	0.00119	0.705
RANDOM2 (400×400)	3,146	0.00381	11.98	0.00205	6.449	0.00385	12.11	0.00209	6.575
RANDOM3 ($5,000 \times 1,000$)	9,212	0.07603	700.4	0.03964	365.2	0.07227	665.7	0.03632	334.6
RANDOM4 ($1,000 \times 5,000$)	5,567	0.06877	382.8	0.03453	192.2	0.07603	423.3	0.03955	220.2
RANDOM5 ($2,250 \times 2,250$)	6,871	0.06991	480.4	0.03537	243.0	0.07216	495.8	0.03672	252.3
RANDOM6 ($2,500 \times 2,000$)	8,123	0.07034	571.4	0.03574	290.3	0.07109	577.5	0.03603	292.6

speedup and efficiency values) for all these test problems (nine from NETLIB and six random ones) are presented in Tables 4 and 5. We have chosen to base our evaluation on this specific total set of test LPs as they form a sufficient range of different sizes and aspect ratios. For simplicity we present the results for four and eight processors, omitting the cases of one and two processors due to lack of space. By observing Table 5 we can note the following:

For large-sized problems (SCFXM3 and Random3-6) the speedup values given by the column distribution scheme are clearly better (slightly, quite or much better depending on the relative number of rows and columns) in most cases than the ones of the row distribution scheme. The only cases in which the row distribution scheme achieves better speedup values than the column distribution scheme are the cases where the number of rows is much bigger than the number of columns.

For medium-sized problems (BRANDY, AGG/AGG2, BANDM, Random2), the behaviour of the two distribution schemes is similar as for the large-sized problems; however, the differences in speedup and efficiency values are quite less (depending again on the relative number of rows and columns). Also, the cases where the row distribution scheme behaves better are more clear now; however, it is still necessary to have much more rows than columns in order to observe significant difference.

On the contrary, for small-sized problems (SC50A, ADLITTLE, SHARE2B, SC105, Random1) the speedup values given by the row distribution scheme are rather a little better than the ones of the column distribution in the general case. For example, for equal or almost equal number of rows and columns (SC50A, SC105 and Random1), the speedup of the row distribution scheme on four processors is slightly better than for the column scheme. Moreover, when the number of rows is clearly larger than the number of columns the speedup of the row scheme is quite better than the one of the column scheme, whereas when the opposite happens (the number of columns is clearly larger than the number of rows) the speedup of the column scheme is much better than the one of the row scheme.

All the above differences are similar when we run our tests on eight processors; however, in most cases they become slightly or quite sharper (depending again on the relative number of rows and columns – see [21] for a detailed discussion). Also, by staring the differences in speedup values for the above two opposite large-sized problems (Random3 and Random4) as well as the corresponding differences for the random square problems, one can easily notice as a general conclusion that the influence of having more columns than rows in favour of the column distribution scheme is greater than the influence

Table 5
Comparing the Two Distribution Schemes (Speedup and Efficiency)

Linear Problems	Column Distribution Scheme				Row Distribution Scheme			
	4 Processors		8 Processors		4 Processors		8 Processors	
	S_p	E_p (%)	S_p	E_p (%)	S_p	E_p (%)	S_p	E_p (%)
SC50A (50×48)	2.45	61.3	3.26	40.7	2.51	62.8	3.48	43.5
ADLITTLE (56×97)	2.82	70.5	4.22	52.8	2.55	63.8	3.51	43.9
SHARE2B (96×79)	2.80	70.0	4.13	51.6	2.91	72.8	4.43	55.4
SC105 (105×103)	2.91	72.8	4.40	55.0	2.95	73.8	4.46	55.8
BRANDY (220×249)	3.51	87.8	6.10	76.3	3.31	82.8	5.33	66.6
AGG (488×163)	3.39	84.8	6.07	75.9	3.51	87.8	6.42	80.3
AGG2 (516×302)	3.51	87.8	6.46	80.7	3.56	89.0	6.66	83.2
BANDM (305×472)	3.72	93.0	6.82	85.2	3.42	85.5	6.03	75.4
SCFXM3 ($990 \times 1,371$)	3.81	95.3	7.21	90.1	3.59	89.8	6.54	81.7
RANDOM1 (200×200)	3.29	82.3	4.86	60.8	3.32	83.0	4.95	61.9
RANDOM2 (400×400)	3.57	89.3	6.63	82.9	3.53	88.3	6.51	81.4
RANDOM3 ($5,000 \times 1,000$)	3.65	91.3	7.00	87.5	3.84	96.0	7.64	95.5
RANDOM4 ($1,000 \times 5,000$)	3.98	99.5	7.92	99.0	3.60	90.0	6.92	86.5
RANDOM5 ($2,250 \times 2,250$)	3.85	96.3	7.61	95.1	3.73	93.3	7.33	91.6
RANDOM6 ($2,500 \times 2,000$)	3.79	94.8	7.46	93.2	3.75	93.8	7.40	92.5

Table 6
Comparing the Two Distribution Schemes (Pros and Cons)

	Column Distribution Scheme	Row Distribution Scheme
Pros:	It offers better parallelization of Step 3. Since Step 3 involves the most calculations, the larger the problem size (and especially the number of columns) the larger the influence in favour of the column distribution scheme.	Step 2 is adequately parallelized. It turns to be a drawback only in the case of very small number of rows.
Cons:	Step 2 is not parallelized at all. It's the main drawback of the column distribution scheme, especially in the case of very large number of rows.	It leads to worse parallelization of Step 3. It's the main drawback of the row distribution scheme, especially in cases that the number of columns is large and clearly larger than the number of rows.
Experimental Results	The column distribution scheme performs better than the row distribution scheme in cases that the number of columns is greater than the number of rows and <i>vice versa</i> . Moreover, the influence of having more columns than rows in favour of the column-based scheme is greater than the influence of having more rows than columns in favour of the row-based scheme, due to the fact that Step 3 involves much more computations than Step 2 (and naturally its efficient parallelization tends to be the most crucial factor).	

of having more rows than columns in favour of the row distribution scheme; which means that the communication overhead caused (see Section 3) by the parallelization of the row distribution scheme is more significant in

the general case than the one caused by the parallelization of the column distribution scheme. The pros and cons of the two distribution schemes are summarized in Table 6.

Table 7
Execution Times for Very Large Problems

Linear Problems		Speedup and Efficiency/Column Distribution Scheme							
		2 Processors		4 Processors		8 Processors		16 Processors	
	No. of Iterations	Time (s)		Time (s)		Time (s)		Time (s)	
		1 Iteration	Total	1 Iteration	Total	1 Iteration	Total	1 Iteration	Total
FIT2P ($3,000 \times 13,525$)	16,081	0.69	11,051	0.34	5,545	0.17	2,801	0.089	1,433
80BAU3B ($2,263 \times 9,799$)	13,119	0.37	4,914	0.19	2,474	0.10	1,253	0.049	648
QAP15 ($6,330 \times 22,275$)	25,675	1.97	50,456	0.99	25,461	0.51	12,997	0.267	6,843
MAROS-R7 ($3,136 \times 9,408$)	14,549	0.64	9,273	0.32	4,689	0.17	2,407	0.088	1,285
QAP12 ($3,192 \times 8,856$)	7,912	0.49	3,853	0.25	1,949	0.13	1,003	0.068	539
DFL001 ($6,071 \times 12,230$)	23,496	0.72	16,863	0.36	8,519	0.19	4,373	0.099	2,335
GREENBEA ($2,392 \times 5,405$)	10,506	0.19	1,986	0.10	1,008	0.05	523	0.027	285
STOCFOR3 ($16,675 \times 15,695$)	29,811	1.70	50,694	0.86	25,747	0.45	13,495	0.256	7,626
RANDOM7 ($15,000 \times 45,000$)	30,547	5.50	168,002	2.80	85,676	1.48	45,344	0.902	27,544
RANDOM8 ($35,000 \times 105,000$)	33,803	12.75	430,851	6.63	223,968	3.53	119,358	2.294	77,553

Table 8
Speedup and Efficiency for Very Large Problems

Linear Problems	Speedup and Efficiency/Column Distribution Scheme							
	2 Processors		4 Processors		8 Processors		16 Processors	
	S_p	E_p (%)	S_p	E_p (%)	S_p	E_p (%)	S_p	E_p (%)
FIT2P ($3,000 \times 13,525$)	1.982	99.1	3.95	98.7	7.82	97.8	15.28	95.5
80BAU3B ($2,263 \times 9,799$)	1.974	98.7	3.92	98.1	7.74	96.8	14.96	93.5
QAP15 ($6,330 \times 22,275$)	1.968	98.4	3.90	97.5	7.64	95.5	14.51	90.7
MAROS-R7 ($3,136 \times 9,408$)	1.962	98.1	3.88	97.0	7.56	94.5	14.16	88.5
QAP12 ($3,192 \times 8,856$)	1.958	97.9	3.87	96.7	7.52	94.0	14.00	87.5
DFL001 ($6,071 \times 12,230$)	1.950	97.5	3.86	96.5	7.52	94.0	14.08	88.0
GREENBEA ($2,392 \times 5,405$)	1.954	97.7	3.85	96.2	7.42	92.7	13.62	85.1
STOCFOR3 ($16,675 \times 15,695$)	1.930	96.5	3.80	95.1	7.25	90.6	12.83	80.2
RANDOM7 ($15,000 \times 45,000$)	1.892	94.6	3.71	92.8	7.01	87.6	11.54	72.1
RANDOM8 ($35,000 \times 105,000$)	1.809	90.5	3.48	87.0	6.53	81.6	10.05	62.8

4.4 Scalability of the Column Distribution Scheme

Additionally, in order to further validate the high efficiency and scalability of our column distribution scheme, in more representative (closer to the real word) cases, we've also performed corresponding experiments (a) for very large NETLIB problems and (b) for up to all the 16 processors of our cluster platform. Most of the additional problems

chosen for these experiments have medium/high aspect ratio except the last one (STOCFOR3) which has almost equal number of rows and columns.

Moreover, we've performed tests over two other random problems with much more rows and columns than any of the problems available in NETLIB. These problems (Random7 and Random8) are also of high aspect ratio ($15,000 \times 45,000$ and $35,000 \times 105,000$, respectively) and

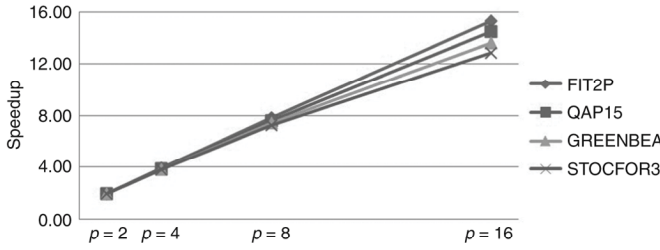


Figure 3. Speedup curves for very large NETLIB problems.

they have been built according to the same characteristics (density, *etc.*) as MAROS-R7 problem of NETLIB.

The corresponding results for varying number of processors (from 2 up to 16) are given in Tables 7 and 8, whereas a more clear view for the performance of some of the test problems is given in Fig. 3. One can easily observe that the efficiency values decrease with the increase of the number of processors. However, this decrease is quite slow, and both the speedup and efficiency values remain high even for 16 processors, in all cases. Moreover, particularly high-efficiency values (almost linear speedup – see Fig. 3) are achieved for all the high aspect ratio NETLIB problems (*e.g.*, see the values for problems FIT2P, 80BAU3B and QAP15 where the speedup even for 16 processors is over 90%).

The speedup and efficiency values remain also quite high in the case of the much larger random problems (Random7 and Random8 – the efficiency falls under 80% only for 16 processors). In the case of 16 processors the communication overhead tends to be very large compared to the computational load (which has been distributed over the 16 processors), and finally leads to relatively lower speedup and efficiency values independently to the high aspect ratio.

Furthermore, with regard to the real-world NETLIB problems, one can easily observe that for constant number of processors, as the aspect ratio or the total size of the problem increases the speedup and efficiency values increase too. Also it must be noted that for larger problems the decrease in the efficiency values as the number of processors increase is less (smoother). As an example, the efficiency value of problem DFL001 is slightly worse in the beginning (for parallel execution on two processors) than the one of GREENBEA (which is a problem with slightly greater – almost the same – aspect ratio than DFL001), however, it becomes much better at the end (for 16 processors).

Concluding, as it comes out from the above discussion, our column distributed scheme is highly scalable for large and very large LP problems. Moreover, it must be noted that the achieved efficiency values (as well as the related scalability) are considerably higher than the values achieved either by corresponding efforts for the parallelization of the revised simplex method [2] or by some more recent, modern and quite more promising efforts based on the block angular structure (or decomposition) of the initially transformed problems [6], [7].

5. Conclusion

A highly scalable parallel implementation framework of the standard full tableau simplex method on a highly parallel – distributed memory – environment has been presented and evaluated throughout the paper, in terms of typical performance measures. Both the two different possible data distribution schemes (column- and row-based) were carefully designed and implemented and were entirely tested over a considerably powerful parallel environment. The experimental tests platform was a linux cluster that consists of 16 powerful 3.0 GHz Xeon processors connected via a dedicated Myrinet network interface with 10 Gbps communication speed.

Our two different data distribution schemes were experimentally compared (with use of standard NETLIB test problems as well as of suitable random problems of our own) both to each other and to the corresponding implementations of [3] and [4], which are two of the most recent and valuable related efforts. In most cases (and almost all the practical cases – except for small problem sizes) the column-based scheme performs quite/much better than the row distribution scheme. The row distribution scheme performs better only for small problems as well as for problems where the number of rows is significantly greater than the number of columns.

Moreover, both the above schemes (even the row-based scheme for large problems) lead to particularly good speedup and efficiency values, that are considerably better in all cases than the ones achieved by the corresponding implementations of [3] and [4]. Finally, the column distribution scheme is further evaluated and experimentally proved to be highly scalable, leading to almost linear speedup for large and very large NETLIB problems with high aspect ratio.

References

- [1] K. Murty, *Linear programming* (New York, NY: John Wiley & Sons, 1983).
- [2] J.A. Hall, Towards a practical parallelization of the simplex method, *Computational Management Science*, 7(2), 2010, 139–170.
- [3] G. Yarmish and R.V. Slyke, A distributed scaleable simplex method, *Journal of Supercomputing*, 49(3), 2009, 373–381.
- [4] E.S. Badr, M. Moussa, K. Paparrizos, N. Samaras, and A. Sifaleras, Some computational results on MPI parallel implementation of dense simplex method, *World Academy of Science, Engineering and Technology (WASET)*, 23, 2008, 778–781.
- [5] J. Qin and D.T. Nguyen, A parallel-vector simplex algorithm on distributed-memory computers, *Structural Optimizations*, 11(3), 1996, 260–262.
- [6] M. Lubin, J.A. Hall, C.G. Petra, and M. Anitescu, Parallel distributed-memory simplex for large-scale stochastic LP problems, *Computational Optimization and Applications*, 55(3), 2013, 571–596.
- [7] K.K. Sivaramakrishnan, A parallel interior point decomposition algorithm for block angular semidefinite programs, *Computational Optimization and Applications*, 46(1), 2010, 1–29.
- [8] M. Geva and Y. Wiseman, Distributed shared memory integration, *Proc. IEEE Conf. on Information Reuse and Integration*, Las Vegas, NV, 2007, 146–151.
- [9] J.A. Hall and K. McKinnon, ASYNPLEX an asynchronous parallel revised simplex algorithm, *Annals of Operations Research*, 81, 1998, 27–49.

- [10] W. Shu and M.Y. Wu, Sparse implementation of revised simplex algorithms on parallel computers, *Proc. 6th SIAM Conf. in Parallel Processing for Scientific Computing*, Norfolk, VA, 1993, 501–509.
- [11] M.E. Thomadakis and J.C. Liu, An efficient steepest-edge simplex algorithm for SIMD computers, *Proc. Int. Conf. on Supercomputing*, Philadelphia, PA, 1996, 286–293.
- [12] J. Eckstein, I. Boduroglu, L. Polymenakos, and D. Goldfarb, Data-parallel implementations of dense simplex methods on the connection machine CM-2, *ORSA Journal on Computing*, 7(4), 1995, 402–416.
- [13] C.B. Stunkel, Linear optimization via message-based parallel processing, *Proc. Int. Conf. on Parallel Processing*, Pennsylvania, PA, 1988, 264–271.
- [14] D. Klabjan, L.E. Johnson, and L.G. Nemhauser, A parallel primal–dual simplex algorithm, *Operations Research Letters*, 27(2), 2000, 47–55.
- [15] I. Maros and G. Mitra, Investigating the sparse simplex method on a distributed memory multiprocessor, *Parallel Computing*, 26(1), 2000, 151–170.
- [16] S.S. Chen, D.L. Donoho, and M.A. Saunders, Atomic decomposition by basis pursuit, *SIAM Journal on Scientific Computing*, 20(1), 1998, 33–61.
- [17] I.W. Selesnick, R.V. Slyke, and O.G. Guleryuz, Pixel recovery via l_1 minimization in the wavelet domain, *Proc. Int. Conf. on Image Processing*, Singapore, 2004, 1819–1822.
- [18] E. Gislason, M. Johansen, K. Conradsen, and B. Ersboll, Three different criteria for the design of two-dimensional zero phase FIR digital filters, *IEEE Transactions on Signal Processing*, 41(10), 1993, 3070–3074.
- [19] K. Steiglitz, T.W. Parks, and J.F. Kaiser, METEOR: a constraint-based FIR filter design program, *IEEE Transactions on Signal Processing*, 40(8), 1992, 1901–1909.
- [20] S.P. Bradley, U.M. Fayyad, and O.L. Mangasarian, Mathematical programming for data mining: formulations and challenges, *INFORMS Journal on Computing*, 11(3), 1999, 217–238.
- [21] B. Mamalis, G. Pantziou, D. Kremmydas, and G. Dimitropoulos, Reexamining the parallelization schemes for standard full tableau simplex method on distributed memory environments, *Proc. 10th IASTED PDCN Conf.*, Innsbruck, Austria, 2011, 115–123.

Biographies



Basilis Mamalis obtained his diploma in computer engineering and informatics as well as his Ph.D. from the Polytechnic School of the University of Patras (Greece), in 1993 and 1998, respectively. He has worked for almost 10 years (from 1994 to 2004 – in several projects and research programs) in Research Academic Computer Technology Institute of Patras. He currently

teaches at the Technological Educational Institute of Athens, Department of Informatics, as an assistant professor from 2004 to 2009, and as an associate professor since 2009 till today. He also teaches in the Hellenic Open University, Department of Informatics, as an external partner. His research interests include information retrieval, document and knowledge management, parallel algorithms and architectures, distributed systems and databases, operating systems and mobile computing. He has over 40 publications in international journals and conferences in the above-mentioned research areas and he was also a referee in several international conferences.



Grammati Pantziou received her Ph.D. in computer science from the University of Patras, Greece, in 1991. She is currently a professor in the Department of Informatics of the Technological Educational Institution of Athens, Greece. She was a researcher in the Computer Technology Institute, Patras (1985–1992 and 1995–1998), a research assistant professor at Dartmouth College, NH, USA (1992–1994), and an assistant professor at the University of Central Florida, USA (1994–1995). Her main research interests include parallel and distributed computing, optimization algorithms, mobile *ad hoc* and wireless sensor networks, pervasive and mobile computing, and security issues in computing environments. She has published over 100 articles in major international computer science journals and conferences. She has served as a program and organizing committee member of several international conferences and schools. Her research was funded by the European Community, the Greek State and the National Science Foundation (NSF) of the United States of America.



Georgios Dimitropoulos obtained his B.Sc. in informatics from the Technological Applications School of the Technological Educational Institute of Athens (Greece), in 2010. He is currently a postgraduate student in the Department of Informatics of the Technological Educational Institute of Athens. He works in the private sector, as a teacher and director of a higher education institute. His main research interests include scientific computing, applied mathematics, advanced information systems and parallel and distributed computing.



Dimitris Kremmydas is currently a laboratory assistant and researcher at the agribusiness unit at the Department of Agricultural Economics and Rural Development of the Agricultural University of Athens. He holds a B.Sc. and an M.Sc. in agricultural economics (from the Agricultural University of Athens), and a B.Sc. in informatics (from the Hellenic Open University). He has participated in many research projects as an IT expert. His main research interests include mathematical programming for farm modelling, agent-based modelling in agriculture and parallel and distributed computing.