Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Networks (PDCN 2011)
February 15 - 17, 2011 Innsbruck, Austria

REEXAMINING THE PARALLELIZATION SCHEMES FOR STANDARD FULL TABLEAU SIMPLEX METHOD ON DISTRIBUTED MEMORY ENVIRONMENTS

Basilis Mamalis[1], Grammati Pantziou[1], Georgios Dimitropoulos[1], Dimitrios Kremmydas[2]
[(1)]Technological Educational Institute of Athens, Ag. Spyridonos, 12210, Egaleo, Greece
[(2)]Agricultural University of Athens, Iera Odos 75, 11855, Athens, Greece
Email:{vmamalis|pantziou|cs000055}@teiath.gr, kremmydas@aua.gr

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 computatios required (especially for the solution of large in size linear problems). In this paper we present a highly scaleable parallel implementation framework of the standard full tableau simplex method on a highly parallel (distributed memory) environment. Specifically, we have designed and implemented a column distribution scheme (similar to the one presented in [24]) as well as a row distribution scheme (similar to the one presented in [3]) and we have entirely tested our implementations over a considerably powerful parallel environment (a linux-cluster of eight powerful Xeon processors connected via a high speed Myrinet network interface). We then compare our approaches (a) among each other for variable number of problem size (number of rows and columns) and (b) to the corresponding ones of [3] and [24] which 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. Moreover, both schemes (even the row distribution scheme over large scale problems) lead to particularly high speed-up and efficiency values, that are considerably better in all cases than the ones achieved by the corresponding implementations of [3] and [24].

KEY WORDS

Linear Programming, Simplex Algorithm, MPI, Parallel Processing, Linux Clusters.

1. Introduction

Linear programming is the most important and well studied optimization problem. The simplex method, which can be found in many textbooks (i.e. see [9,15]), has been succesfully used for solving linear programming problems for many years. Parallel approaches have also extensively been studied (e.g. [1-3,6,10,13-14,16,19-24]) due to the intensive computations required (especially for the solution of large in size linear problems).

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 (most of the real world linear programming problems are extremely sparse). 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 [11]. On the other hand, the standard method is more efficient for dense linear problems (that also appear in real world applications – although much more rarely than the sparse ones) and it can be easily convert to a distributed/parallel version with satisfactory speedup values and good scalability (i.e. see [11,16,24]).

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 DSM environments (e.g. [7]). Hall and McKinnon [10] and Shu and Wu [21] worked on parallel revised methods. Thomadakis and Liu [20] worked on the standard method utilizing the MP-1 and MP-2 MasPar. Eckstein et al. [6] showed in the context of the parallel connection machine CM-2 that the iteration time for parallel revised tended to be significantly higher than for parallel tableau even when the revised method is implemented very carefully. Stunkel [19] 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.

As mentioned above, the standard method can be easily and effectively extended to a coarse-grained, distributed algorithm. Yarmish [23] describes such a coarse grained distributed simplex method, in greater detail. For more details on both the corresponding serial and parallel implementations, see [22]. It should also be noted that although dense problems are uncommon in general, they do occur frequently in some important applications within linear programming [6]. Included among those are wavelet decomposition, image processing [5,17] and digital filter design [8,12,18]. Other applications leading to dense linear programming problems can be found in [4]. All these problem groups

DOI: 10.2316/P.2011.719-028

are well suited to the standard simplex method. Moreover, when the standard simplex method is distributed, aspect ratio becomes less of an issue (it can be easily faced via a well designed column distribution scheme).

Furthermore, existing parallel (distributed memory) implementations of the standard simplex method naturally vary in the way that the simplex tableau is distributed among the processors [3,11,16]. 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 corresponding effort following the column distribution scheme (which is the most popular and widely used for practical problems) have been done by Yarmish et al. [22-24] yielding to high parallel efficiency and corresponding scalability for large-scale problems (as it has already been stated above), as well as (older) by Qin et al [16]. On the other hand, quite recently, Badr et al. [1-3] followed the row distribution scheme and presented a well-designed quite valuable implementation on eight loosely coupled processors (interconnected with Fast Ethernet and SCI interface), targeting and achieving significant speed-up (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 scaleable 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 the one presented in [24]) and a row distribution scheme (similar to the one presented in [3]), 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 eight powerful 3.0GHz XEON processors with 1GB RAM each that are connected via a dedicated Myrinet network interface which provides 10Gbps communication speed.

Specifically, we have tested and compared our approaches among each other for varying number of processors (up to eight processors) and varying problem size (number of rows and columns), as well as to the corresponding implementations of [3] and [24] which (as mentioned above) are two of the most recent and valuable corresponding efforts. In most cases (and almost all the practical cases – except for small problem sizes) the column distribution scheme performs quite/much better than the row distribution scheme. 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 (even the row distribution scheme over large scale problems) lead to particularly high speed-up and efficiency values that are considerably better in all cases than the ones achieved by the corresponding implementations of [3] and [24].

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 (through detailed tables and curves as well as corresponding discussion for each kind of experiments) 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. The mathematical formulation of the linear programming problem, in standard form, is shown below:

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

where A is an mxn 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.

The simplex algorithm consists of two steps; first, a way of finding out whether a current basic feasible solution is an optimal solution, and second, a procedure of obtaining an adjacent basic feasible solution with the same or better value for the objective function. We focus here on the standard full tableau format of the simplex method, which (as it has already been mentioned in the previous section) is more efficient for full dense linear problems and it can be easily convert to a distributed version for cluster computer systems. A tableau is an (m+1)x(m+n+1) matrix of the following form:

Table 1. Simplex full tableau representation

	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

Based on the above 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 simplex tableau.
- **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 – 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 (column-based and row-based) distribution schemes. As it can be easily observed the two schemes mainly differ on the part of the algorithm that remains essentially non-parallelized (step 1 or step 2) 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 (i.e. see [24]) 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 (even the first one of choosing the entering variable – finding the winning column – which is of less computation). Additionally, it is the more 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 the one presented in [24]) that we have implemented with regard to the column-based distribution scheme are given below:

- The simplex table is shared among the processors by columns. Also, the right-hand constraints vector is broadcasted to all processors.
- 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)
- 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 (a suitable reduction-type function of MPI is actually used here in order that computation execute optimally in parallel).
- 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
- The same (winning) processor then broadcasts the winning column as well as the winning row's id to all processors.
- Each processor performs (in parallel) on it's own part (columns) of the table all the calculations required for the global rows pivoting, based on the pivot data received during the previous step.
- 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 partitioning scheme can be followed in order to take advantage of some probable inherent parallelism in the computations of step 2; however (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 communication network among the processors is not quite fast (because the corresponding computational task becomes quite small and naturally the corresponding communication tasks dominate).

Additionally, in the case of the row-based distribution scheme, the parallelization of step 1 becomes more problematic and gets preferable to keep the corresponding task executed centrally on processor 0 (which normally tends to be a disadvantage for large problems with many columns). On the other hand this difference usually tends to be an advantage when the size of the problem (especially the number of columns) is quite small, as well as when the communication network among the processors is not quite fast.

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 (compared to the column distribution scheme) when the number of columns decrease (and generally for small sized problems), whereas the opposite is expected to happen when the number of columns increase (and generally for large sized problems).

We have implemented such a row distribution parallel scheme (similar to the one presented in [3]), based on the following steps:

- The simplex table is shared among the processors by rows. The right-hand constraints vector is also shared in the same way.
- Processor 0 finds the best candidate (winning) column (choice of the entering variable) and broadcasts this column's id to all processors.

- 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).
- 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 (a suitable reduction-type function of MPI is actually used here in order that computation execute optimally in parallel).
- The new pivot row is determined (constructed) and then it's broadcasted to all the other processors by the winning processor.
- Each processor performs (in parallel) a pivot on it's own rows of the table, based on the new pivot row (that was just received from the winning processor during the previous step).
- The above steps are repeated until the best solution is found or the problem gets unbounded.

We also have to note here that in both algorithms, in all steps that require communication among all processors, we have suitably used corresponding MPI collective communication functions (i.e. MPI_Scatter and Scatterv, MPI_Gather and Allgather, MPI_Bcast, MPI_Reduce and Allreduce – with or without the use of MINLOC and MAXLOC operators) in order all these tasks to be executed optimally over the underlying platform.

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 flavor) message passing library and they have been extensively tested (in terms of speed-up 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 [24]. The speed-up for p processors (Sp) 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 (Ep) is computed as the speed-up achieved in p processors divided by p (and represents the fraction of the maximum theoretical speed-up that has been achieved). Our linux-cluster consists of 8 powerful 3.0GHz XEON processors with 1GB RAM each, and a dedicated Myrinet network interface which provides 10Gbps communication speed. This platform is quite more powerful than the corresponding platforms used for experimental testing in [3,24]; it can be regarded as a typical representative of modern distributed memory cluster platforms able to support high performance parallel implementations.

Specifically, we have performed three kinds of experiments: (a) experiments that compare our column distribution scheme to the corresponding one of [24], (b) experiments that compare our row distribution scheme to the corresponding one of [3], and (c) experiments that compare the two distribution schemes (row-based and column-based) among each other for varying number of processors and problem sizes (either taken from the NETLIB library or random ones).

4.1 Comparing to the implementation of [24]

In order to compare our approach to the one presented in [24] (which is the most recent and valuable related attempt for distributed memory platforms, using the column distribution scheme) we have run our relevant implementation (the column-based distribution one) over the large size (1000x5000) linear problem presented there [24], 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/distributed platform used in the experiments of [24] consisted of 7 independent 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. A more clear view can also be found in the graphical representation given in figure 1. Observing the results of table 1, firstly it can easily be noticed that the execution times (in one or more processors) of our algorithm are much better than the ones of [24], which however was expected due to the fact that our platform is quite more powerful than the platform of [24].

The most important, the achieved speed-up values of our implementation are also better than the ones achieved in [24]. To be more precise, they are slightly better for 2 and 4 processors (which are powers of two – 1.99 vs 1.97 and 3.98 vs 3.96 respectively) and quite better for 3, 5, 6 and 7 processors (i.e. 4.89 vs 4.68 for 5 processors and 6.91 vs 6.72 for 7 processors). Furthermore, observing the corresponding efficiency values in the last column someone can easily notice the high scalability (higher and smoother than in [24]) 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 8 processors.

It can also be noticed that in both implementations there is a relatively better performance when the number of processors is a power of two (i.e. for 2, 4 and 8 processors). This happens in our implementation mainly because the tree-based MPI collective communication functions that have been applied (i.e. MPI_Reduce, MPI_Bcast etc) perform better when the number of processes is a power of two; in our approach however these differences are not so significant as in [24] due probably to the high-speed and low-latency (myrinet based) network connections used.

Table 2. Comparing to the implementation of [24] (1000x5000)

P	Yarmish et al [24]			Our Implementation		
#proc	Time/iter	$Sp=T_1/T_P$	Ep=Sp/P	Time/iter	$Sp=T_1/T_P$	Ep=Sp/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%

Table 3. Comparing to the implementation of [3] (200x200)

P	Badr et al [3]			Our Implementation		
#proc	Time/iter	$Sp=T_1/T_P$	Ep=Sp/P	Time/iter	$Sp=T_1/T_P$	Ep=Sp/P
1	0.00512	1.00	100.0%	0.00310	1.00	100.0%
2	0.00264	1.94	97.0%	0.00156	1.99	99.3%
4	0.00253	2.03	50.7%	0.00093	3.32	83.0%
8				0.00063	4.95	61.9%

Table 4. Comparing to the implementation of [3] (300x300)

P	Badr et al [3]			Our Implementation		
#proc	Time/iter	$Sp=T_1/T_P$	Ep=Sp/P	Time/iter	$Sp=T_1/T_P$	Ep=Sp/P
1	0.01403	1.00	100.0%	0.00775	1.00	100.0%
2	0.00763	1.84	92.0%	0.00403	1.92	96.1%
4	0.00456	3.08	77.0%	0.00226	3.43	85.7%
6	0.00382	3.67	61.2%	0.00157	4.95	82.5%
8				0.00132	5.86	73.2%

Table 5. Comparing to the implementation of [3] (400x400)

P	Badr et al [3]			Our Implementation		
#proc	Time/iter	$Sp=T_1/T_P$	Ep=Sp/P	Time/iter	$Sp=T_1/T_P$	Ep=Sp/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%

(*) no results are mentioned in [3] for the execution of (400x400) on 6 processors

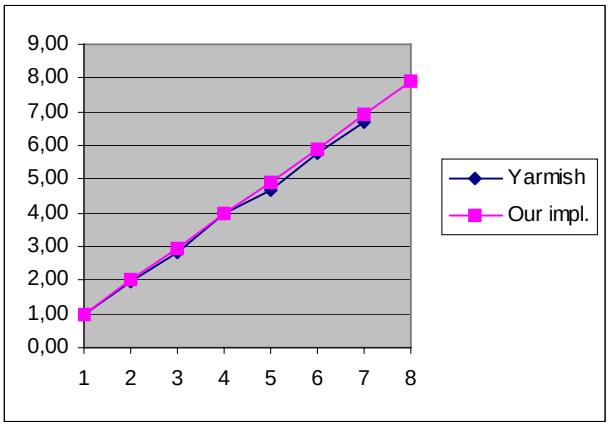

Figure 1. Speed-up curve for table 2 (1000x5000)

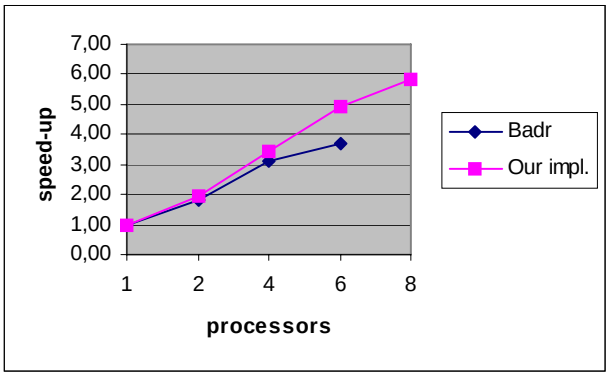

Figure 2. Speed-up curve for table 4 (300x300)

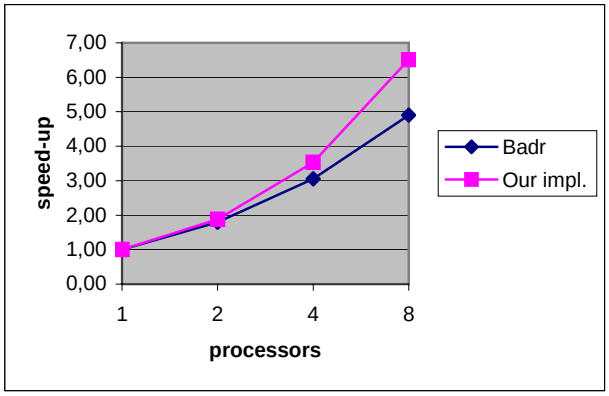

Figure 3. Speed-up curve for table 5 (400x400)

Moreover, consider that the above implementation of [24] (compared to which our implementation scales even better) has also been compared to MINOS, a well-known and commonly used implementation of the revised simplex method, and it has been shown to be highly competitive; actually faster even for problems with density close or little less than 10% provided that the required number of processors (i.e. up to seventeen) is available (see [24] for more details). We also have to note here that the performance of our simplex implementations (as well as the one of [24]) remains unchanged with changes of density; as it was expected since they refer to the standard full tableau method (not the revised).

4.2 Comparing to the implementation of [3]

In order to compare our approach to the one presented in [3] (which is the most recent found attempt for distributed memory platforms using the row distribution scheme and mainly targeting to small size linear problems) we have run our corresponding implementation (the row-based distribution one) over the three classes of small and medium sized (200x200, 300x300, 400x400) linear problems presented there [3], with same characteristics, and we have measured the execution time per iteration for # of processors up to 8 as in [3]. 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 (since they are square problems – i.e. with not much more rows than columns). Additionally, for small-sized problems (where the computation tasks decrease a lot) it is natural that the speed of the interconnection network plays an even more important role in the parallel implementation. Note also that the parallel/distributed platform used in the experiments of [3] consisted of 8 Intel Pentium processors connected via Fast Ethernet and Scalable Coherent Interface (SCI).

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 tables 3, 4 and 5. A more clear view for the results of tables 4 and 5 can also be found in the graphical representations given in figures 2 and 3. Observing the results of these tables, firstly, 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 here too due to the fact that our platform is quite more powerful than the platform used in [3]. Secondly, it's also clear that in all cases the achieved speed-up and efficiency values of our implementation are quite better or much better than the ones of [3], except the special case of 2 processors where the results of [3] are quite close (slightly worse) to our results.

Considering in more details the case of problem size 200x200 (table 3), in [3] a very good speedup (1.94) is achieved for 2 processors; with very small increase however when we go to 4 processors (2.03). Moreover, for more than 4 processors there is not any increase in speedup. In our implementation (although it's a quite small problem) the achieved speedup is slightly better for two processors (1.99) and much better for 4 processors (3.32) and continue to increase even for 8 processors, giving a quite good absolute speedup value of 4.95.

Considering the case of problem size 300x300 (table 4), in [3] a very good speedup (1.84) is achieved again for

2 processors; whereas now the speedup is quite good for 4 processors (3.08) and continues to increase even for 6 processors (3.67 – for more than 6 processors there is not any increase in speedup). In our implementation we achieve slightly better speedup for 2 processors (1.92), quite better speedup for 4 processors (3.43) and much better speedup for 6 processors (4.95) which means that our algorithm starts to scale quite well (with the increase of the number of processors – see also fig. 2) even for this relatively small size problem (the speedup continues to increase even for 8 processors, giving again a quite good absolute speedup value of 5.86).

Finally, with regard to the case of problem size 400x400 (table 5), in [3] we can observe a rather similar behaviour as for the problem size of 300x300 (table 4), with the basic difference that here the speedup continues to increase even for 8 processors (and becomes equal to 4.91 which is the maximum speedup value achieved in the experiments presented in [3]). On the contrary, in our implementation we achieve even better scalability (than for the 300x300 problem size) as the number of processors increase (see also fig. 3), leading to a very good absolute speedup value of 6.51 for 8 processors, which means that our algorithm scales very well with the increase of the problem size too.

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-sized problems; much better in most cases than the implementation of [3].

4.3 Comparing the two distribution schemes

In order to compare the performance of the two different data distribution schemes (row-based and column-based respectively) among each other, we have run on our platform the extended set of the well known and widely used NETLIB test linear problems, as well as an additional set of random problems of our own of different sizes. The corresponding results (in terms of speedup and efficiency measures) for 16 of these test problems (10 from NETLIB and 6 random ones) are presented in table 6. We have chosen to present the above subset as they form a sufficient range of different sizes (both in total tableau size and in relative numbers of rows and columns). For simplicity (and without loss of generality) we present the results for 4 and 8 processors (omitting the execution times per iteration due to lack of space). By observing table 6 we can note the following:

For large sized problems, 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. E.g. on 4 processors for Random4 (1000x5000) the speedup of the column scheme is too much better (3.98 vs 3.60) than the one of the row scheme, for SCFXM3 (991x1371) the speedup of the column scheme is quite much better (3.81 vs 3.59), for Random5 (2250x2250) it is quite better (3.85 vs 3.73) and for Random6 (2500x2000) slightly better (3.79 vs 3.75). 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 – i.e. on 4 processors for Random3 (5000x1000) the speedup of the row distribution scheme is quite better (3.84 vs 3.65) than the one of the column scheme.

For medium sized problems, the behaviour of the two distribution schemes is similar as for the large sized problems (the performance of the column scheme remains better in most cases), however the differences in speedup and efficiency values are quite less than for the large sized problems (depending again on the relative number of rows and columns). E.g. on 8 processors for BANDM (306x472) the speedup of the column scheme is much better (3.72 vs 3.42) than the one of the row scheme and for BRANDY (221x249) the speedup of the column scheme is quite better (3.51 vs 3.31), whereas for the corresponding square problem (Random2 – 400x400) the speedup difference gets only slightly better (3.57 vs 3.53). Moreover, 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 – i.e. on 4 processors for AGG2 (517x302) the speedup of the row distribution scheme is slightly better (3.56 vs 3.51) than the one of the column scheme, whereas for AGG (489x163) it becomes quite better (3.51 vs 3.39).

On the contrary, for small sized problems 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. E.g. for equal or almost equal number of rows and columns (problems SC50A/51x48, SC105/106x103 and Random1/200x200), the speedup of the row distribution scheme on 4 processors is slightly better (2.51 vs 2.45, 2.95 vs 2.91 and 3.32 vs 3.29 respectively) than the one of 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 (e.g. on 4 processors for SHARE2B/97x79 the speedup difference is 2.91 vs 2.80 in favor of the row 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 (e.g. on 4 processors for ADLITTLE/57x97 the speedup difference is 2.82 vs 2.55 in favor of the column scheme).

All the above differences are similar when we run our tests on 8 processors, however in most cases they become slightly or quite sharper (depending again on the relative number of rows and columns). E.g. for the large sized problem Random4 (1000x5000) the speedup difference in favor of the column scheme becomes 7.92 vs 6.92 instead of 3.98 vs 3.60 that was for 4 processors;

Table 6. Comparing the two data distribution schemes

Linear Problems	Column distribution scheme				Row distribution scheme			
	4 processors		8 processors		4 processors		8 processors	
	Sp	Ep	Sp	Ep	Sp	Ep	Sp	Ep
SC50A (51x48)	2.45	61.3%	3.26	40.7%	2.51	62.8%	3.48	43.5%
ADLITTLE (57x97)	2.82	70.5%	4.22	52.8%	2.55	63.8%	3.51	43.9%
SHARE2B (97x79)	2.80	70.0%	4.13	51.6%	2.91	72.8%	4.43	55.4%
SC105 (106x103)	2.91	72.8%	4.40	55.0%	2.95	73.8%	4.46	55.8%
LOTFI (154x308)	3.35	83.8%	5.78	72.2%	3.02	75.5%	4.56	57.0%
BRANDY (221x249)	3.51	87.8%	6.10	76.3%	3.31	82.8%	5.33	66.6%
AGG (489x163)	3.39	84.8%	6.07	75.9%	3.51	87.8%	6.42	80.3%
AGG2 (517x302)	3.51	87.8%	6.46	80.7%	3.56	89.0%	6.66	83.2%
BANDM (306x472)	3.72	93.0%	6.82	85.2%	3.42	85.5%	6.03	75.4%
SCFXM3 (991x1371)	3.81	95.3%	7.21	90.1%	3.59	89.8%	6.54	81.7%
RANDOM1 (200x200)	3.29	82.3%	4.86	60.8%	3.32	83.0%	4.95	61.9%
RANDOM2 (400x400)	3.57	89.3%	6.63	82.9%	3.53	88.3%	6.51	81.4%
RANDOM3 (5000x1000)	3.65	91.3%	7.00	87.5%	3.84	96.0%	7.64	95.5%
RANDOM4 (1000x5000)	3.98	99.5%	7.92	99.0%	3.60	90.0%	6.92	86.5%
RANDOM5 (2250x2250)	3.85	96.3%	7.61	95.1%	3.73	93.3%	7.33	91.6%
RANDOM6 (2500x2000)	3.79	94.8%	7.46	93.2%	3.75	93.8%	7.40	92.5%

The corresponding differences in the efficiency values are 99% vs 86.5% on 8 processors instead of 99.5% vs 90% on 4 processors, which means an increase of 3 units (12.5% vs 9.5%) on the efficiency differences when we go from 4 to 8 processors.

This phenomenon mainly happens because (a) the efficiency values become worse with the increase of the number of processors (i.e. from 4 to 8 processors) as it was expected since the communication overhead becomes generally more important than the decrease of the computational work for each processor, and (b) they become even more worse for the distribution scheme that has the most disadvantages with respect to the communication overhead caused.

E.g. for a problem with many more columns than rows (as Random4 1000x5000) it's natural to have larger decrease in the efficiency value for the row distribution scheme when we go from 4 to 8 processors because the relative communication overhead becomes bigger than the one of the column distribution scheme; due to the non-parallelization of step 1 as well as due to the greater overhead in the parallelization of step 3 where the length of the extra row that has to be broadcasted is much bigger than the length of the extra column that has to be broadcasted in the column distribution scheme.

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 favor of the column distribution scheme is greater than the influence of having more rows than columns in favor 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.

5. Conclusion

A highly scaleable 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 were carefully designed and implemented (a column distribution scheme similar to the one of [24], as well as a row distribution scheme similar to the one of

[3]) and were entirely tested over a considerably powerful parallel environment. The experimental tests platform was a linux-cluster that consists of eight powerful 3.0GHz Xeon processors connected via a dedicated Myrinet network interface with 10Gbps 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 [24], 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 high speed-up and efficiency values, that are considerably better in all cases than the ones achieved by the corresponding implementations of [3] and [24].

References

[1] E.S. Badr, K. Paparrizos, N. Samaras & A. Sifaleras, On the basis inverse of the exterior point simplex algorithm. *In Proc. of 17th Conf. of Hellenic Operational Research Society*, Rio, Greece, 2005, 677-687.

[2] E.S. Badr, Parallel Programming Algorithms for Linear Programming Problems. *Ph.D dissertation,* University of Macedonia, Thessaloniki, Greece, 2006.

[3] E.S. Badr, M. Moussa, K. Paparrizos, N. Samaras & A. Sifaleras, Some computational results on MPI Parallel Implementation of Dense Simplex Method. *In Proc. of World Academy of Science, Engineering and Technology (WASET), 23*, 2006, 39-42.

[4] S.P. Bradley, U.M. Fayyad, & O.L. Mangasarian, Mathematical Programming for Data Mining: Formulations and Challenges. *INFORMS Journal on Computing, 11*(3), 1999, 217-238.

[5] S.S. Chen, D.L. Donoho & M.A. Saunders, Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing 20*(1), 1998, 33-61.

[6] J. Eckstein, I. Boduroglu, L. Polymenakos, & D. Goldfarb, Data-parallel implementations of dense simplex methods on the connection machine CM-2. *ORSA Journal on Computing 7*(4), 1995, 402-416.

[7] M. Geva & Y. Wiseman, Distributed Shared Memory Integration. *IEEE Conference on Information Reuse and Integration (IEEE IRI-2007)*, Las Vegas, Nevada, 2007, 146-151.

[8] E. Gislason, M. Johansen, K. Conradsen, B. Ersboll, & S. Jacobsen, Three different criteria for the design of two-dimensional zero phase FIR digital filters. *IEEE Signal Processing 41*(10), 1993, 3070-3074.

[9] G. Hadley, *Linear programming* (Massachusetts: Addison-Wesley, 1963).

[10] J.A. Hall & K. McKinnon, ASYNPLEX an asynchronous parallel revised simplex algorithm. *Annals of Operations Research, 81,* 1998, 27-49.

[11] J.A. Hall, Towards a practical parallelisation of the simplex method. *Optimization Online*, 2007, http://www.optimization-online.org/ DB_FILE/2005/02/1061.ps

[12] J.V. Hu & L.R. Rabiner, Design techniques for two-dimensional digital filters. *IEEE Transactions on Audio Electroacoust AU-20*(4), 1972, 249-257.

[13] D. Klabjan, L.E. Johnson & L.G. Nemhauser, A parallel primal-dual simplex algorithm. *Operations Research Letters, 27*(2), 2000, 47-55.

[14] I. Maros & G. Mitra, Investigating the sparse simplex method on a distributed memory multiprocessor. *Parallel Computing, 26*(1), 2000, 151-170.

[15] K. Murty, *Linear and combinational programming* (New York: John Wiley & Sons, 1976).

[16] J. Qin & D.T. Nguyen, A parallel-vector simplex algorithm on distributed-memory computers. *Structural Optimizations, 11*(3), 1996, 260-262.

[17] I.W. Selesnick, R.V. Slyke & O.G. Guleryuz, Pixel recovery via *l*1 minimization in the wavelet domain. *In: IEEE Proceedings of International Conference on Image Processing (ICIP)*, Singapore, 2004, 1819-1822.

[18] K. Steiglitz, T.W. Parks & J.F. Kaiser, METEOR: a constraint-based FIR filter design program. *IEEE Trans Signal Proc 40*(8), 1992, 1901-1909.

[19] C.B. Stunkel, Linear optimization via message-based parallel processing. *In: ICPP: international conference on parallel processing,* 1988, 264-271.

[20] M.E. Thomadakis & J.C. Liu, An efficient steepest-edge simplex algorithm for SIMD computers. *In: Proc. of the International Conference on Supercomputing (ICS '96)*, 1996, 286-293.

[21] W. Shu, & M.Y. Wu, Sparse implementation of revised simplex algorithms on parallel computers. *In: 6th SIAM conference in parallel processing for scientific computing*, 1993, 501-509.

[22] G. Yarmish, A distributed implementation of the simplex method. *Ph.D dissertation*, Polytechnic University, Brooklyn, NY, 2001.

[23] G. Yarmish, Wavelet decomposition via the standard tableau simplex method of linear programming. *WSEAS Transactions on Mathematics 6*(1), 2007, 170-177.

[24] G. Yarmish & R.V. Slyke, A Distributed Scaleable Simplex Method. *Journal of Supercomputing, 49*(3), 2009, 373-381.