Simplex Parallelization in a Fully Hybrid Hardware Platform

Basilis Mamalis
Technological Educational Institute of Athens
Agiou Spyridonos, 12210, Egaleo
Athens, GREECE

Marios Perlitis
Democritus University of Thrace
University Campus, 69100
Komotini, GREECE

***Abstract*—The simplex method has been successfully used in solving linear programming (LP) problems for many years. Parallel approaches have also extensively been studied due to the intensive computations required, especially for the solution of large LP problems. Furthermore, the rapid proliferation of multicore CPU architectures as well as the computational power provided by the massive parallelism of modern GPUs have turned CPU / GPU collaboration models increasingly into focus over the last years for better performance. In this paper, a highly scalable implementation framework of the standard full tableau simplex method is first presented, over a hybrid parallel platform which consists of multiple multicore nodes interconnected via a high-speed communication network. The proposed approach is based on the combined use of MPI and OpenMP, adopting a suitable column-based distribution scheme for the simplex tableau. The parallelization framework is then extended in such a way that it can exploit concurrently the full power of the provided resources on a multicore single-node environment with a CUDA-enabled GPU (i.e. using the CPU cores and the GPU concurrently), based on a suitable hybrid multithreading/GPU offloading scheme with OpenMP and CUDA. The corresponding experimental results show that the hybrid MPI+OpenMP based parallelization scheme leads to particularly high speed-up and efficiency values, considerably better than in other competitive approaches, and scaling well even for very large / huge linear problems. Furthermore, the performance of the hybrid multithreading/GPU offloading scheme is clearly superior to both the OpenMP-only and the GPU-only based implementations in almost all cases, which validates the worth of using both resources concurrently. The most important, when it is used in combination with MPI in a multi-node (fully hybrid) environment, it leads to substantial improvements in the speedup achieved for large and very large LP problems.**

Keywords*—*Parallel Processing; Linear Programming; Simplex Algorithm; MPI; OpenMP; CUDA

I. INTRODUCTION

Linear programming is the most important and well studied optimization problem. The simplex method, which can be found in many textbooks, has been successfully used for solving linear programming problems for many years. Parallel approaches have also extensively been studied due to the very intensive computations required and the substantial need for faster implementations that make effective use of modern computer architectures.

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 [1]. 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 [1-4]. A detailed overview is given in Section II. Also, lately, some alternative very promising efforts have been made, based on the block angular structure (or decomposition) of the initially transformed problems [5-6], and they have led to very good results for large scale problems over distributed memory multicore environments.

Furthermore, with regard to parallelism, until recently, the relevant models, languages, and libraries for shared-memory and distributed-memory architectures have evolved separately, with MPI [7] becoming the dominant approach for the distributed-memory (message-passing) model, and OpenMP [8] emerging as the dominant high-level approach for shared memory with threads. Recently, the hybrid model has begun to attract more attention, for at least two reasons. The first is that it is relatively easy to pick a language/library instantiation of the hybrid model (OpenMP, MPI, MPI 3.0 Shared Memory etc.). The second reason is that scalable parallel computers now appear to encourage this model. The fastest machines now virtually all consist of multi-core nodes connected by a high speed network. The idea of using OpenMP threads to exploit the multiple cores per node (with one multithreaded process per node) while using MPI to communicate among the nodes is the most known. The last 3-4 years however, another strong alternative has evolved; the MPI 3.0 Shared Memory support mechanism, which improves significantly the previous existed Remote Memory Access utilities of MPI, towards the direction of optimized operation inside a multicore node. As analyzed in [9-11] both the above referred hybrid models (MPI+OpenMP, MPI+MPI 3.0 Shared Memory) have their pros and cons, and it's not straightforward that they outperform pure MPI implementations in all cases. Among all the alternatives the MPI+OpenMP hybrid approach is still regarded as the most efficient one, however the MPI+MPI 3.0 Shared Memory approach is highly competitive.

Moreover, nowadays the computational power provided by the massive parallelism of modern graphics processing units (GPUs), has brought increasingly into focus several kinds of

GPU-accelerated solutions. Although in simplex parallelization there have not been noticed as many relevant attempts as one would expect and no parallel GPU-based implementation of the simplex algorithm has yet offered significantly better performance relative to an efficient sequential simplex solver (at least not in all types of problems), quite significant progress has been achieved at least for dense LP problems [12]. There also exist various approaches in the field of parallel scientific computing that adopt extended CPU/GPU collaboration [13-17]. However, the efficient CPU/GPU collaboration through the combined use of relevant programming models (such as OpenMP and CUDA) still remains a major research challenge. Also, to the best of our knowledge there is no relevant approach in the literature adopting extended CPU/GPU collaboration for parallelizing the simplex method.

In this work we focus on the parallelization of the standard full tableau simplex method and we firstly present and evaluate a relevant highly scalable implementation (on the basis of a carefully designed column-based distribution scheme) for the most efficient of the hybrid parallelization alternatives referred above (MPI+OpenMP) assuming there are not GPU-accelerators in our hybrid hardware platform. We then demostrate the high efficiency and scalability of the proposed hybrid MPI+OpenMP parallelization scheme, over a suitable subset of the well known and widely used NETLIB test LP problems. The corresponding experiments have been perfromed over a hybrid, newly developed, parallel platform which consists of up to 8 quad-core processors (making a total of 32 cores) connected via Gigabit ethernet interface. In all cases the hybrid MPI+OpenMP based parallelization scheme leads to considerably high speedup and efficiency values and performs better than other alternatives [18]. Note also that it has been shown (over the less powerful platform of [18]) to perform quite better than the relevant, higly competitive, approach presented in the work of [4].

Secondly, we extend the proposed hybrid parallelization scheme (MPI+OpenMP) over multicore platforms with CUDA-enabled GPUs, and we present a highly efficient framework which can exploit the full power of both the provided multiple CPU-cores and the GPU, concurrently. In the above context, we've designed and implemented a hybrid multithreading/GPU offloading scheme (based on the combined use of OpenMP and CUDA) that efficiently adopts full CPU/GPU collaboration, as well as two complementary schemes for comparison purposes, i.e. a (multithreading) OpenMP-based only scheme, and a GPU-based only (CUDA). The corresponding experimental results show that the performance of our GPU-based only implementation is comparable to other relevant approaches in the literature [18], and superior to our OpenMP-based only implementation (leading to speedup values up to 14.06 for a GTX 760 GPU and up to 23.22 for a GTX TitanX GPU - compared to the sequential implementation). Moreover, the performance of our hybrid multithreading/GPU offloading scheme is clearly superior to the GPU-based only implementation in almost all cases (leading to an additional speedup of up to 1.28 for the GTX 760 GPU and up to 1.23 for the GTX TitanX GPU), which validates the worth of using both resources concurrently. The most important, when the proposed CPU/GPU collaboration scheme is used in combination with MPI in a multi-node (fully hybrid) environment, it leads to substantial performance gains that rise up to 15,9%.

A very early version of this work (in a much less powerful platform, without adopting full CPU/GPU collaboration, and without addressing the fully hybrid platform integration) has been presented in [18,20]. The rest of the paper is organized as follows. In Section II the related work is summarized. In Section III the necessary background is stated with regard to simplex method. In Section IV the detailed description of our basic hybrid parallelization scheme (using MPI and OpenMP) is given. In Section V the relevant extension with CPU/GPU collaboration is presented. In Section VI the experimental results of our basic schemes (MPI+OpenMP and OpenMP+ CUDA) are given, whereas in Section VII we present the performane gains of our fully hybrid parallel approach (MPI+OpenMP+CUDA). Section VIII concludes the paper.

II. Related Work

Earlier work on simplex parallelization focused mainly on tightly coupled or shared memory hardware structures as well as on clusters and networks of workstations. Hall & McKinnon [21] and Shu & Wu [22] worked on the parallel revised method over powerful shared memory and hypercube platforms respectively. Thomadakis & Liu [23] worked on the standard method utilizing the MP-1 and MP-2 MasPar. Eckstein et al. [24] showed in the context of the parallel connection machine CM-2 that the iteration time for parallel revised method tended to be higher than for parallel full tableau method even when the revised method is implemented very carefully. Stunkel [25] 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 [26-27] following the primal-dual simplex method and the sparse simplex method, and they've led to satisfactory results for large scale problems. Till recently, no other valuable attempts have been made to parallelize the classical revised simplex method, thus making the one presented by Huangfu and Hall [28-29] a distinguished one. The authors in [28-29] have designed and implemented a very efficient parallelization scheme of the dual revised method with use of the suboptimization technique, and they have obtained speedup values comparable to those of the best commercial simplex solvers. A relevant survey which covers adequately all the recent advances in simplex parallelization can be found in [12].

As already mentioned, the standard method can be easily and effectively extended to a coarse-grained, distributed algorithm [4]. It should also be noted that although dense problems (which suit better the standard method) are uncommon in general, they do occur frequently in some important applications within linear programming [24]. Furthermore, existing distributed memory implementations of the standard simplex method naturally vary in the way that the simplex tableau is distributed among the processors [1,30]. 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 works following the column distribution scheme (mostly used for practical problems) were by Yarmish et al. [4] as well as (an older one) by Qin et al [3]. On the other hand, the work of Badr et al. [2] referred above followed the row distribution scheme and presented a quite efficient implementa-tion for loosely coupled processors. A comprehensive study and comparison of the above distribution schemes as well as corresponding implementations achieving particularly high speedup values are given in the recent works of Mamalis et al. [18,30]. As shown in [30], the column distribution scheme is the most efficient one for the most types of LP problems.

A number of valuable simplex parallelization attempts have also been seen in the literature with use of GPU accelerators. Spampinato et al [31] have proposed a parallel implementation of the revised simplex method based on NVIDIA CUBLAS and LAPACK libraries, with a maximum speedup of 2.5, using a GTX 280 GPU vs. the sequential implementation on CPU with Intel Core2 Quad 2.83 GHz for randomly generated LP problems of size 2000x2000. In [32] another implementation of the revised simplex method on GPU was proposed, which permits one to speed up solution with a maximum factor of 18 in single precision on a GeForce 9600 GT GPU card as compared with GLPK solver run on Intel Core 2 Duo 3GHz CPU. Lalami et al. [19] have presented a GPU mainly based parallel implementation via CUDA of the standard simplex algorithm for dense LP problems. Experiments carried out on an Intel Xeon 3GHz and a GTX 260 GPU have shown substantial speedup of 12.5 in double precision, for randomly generated LP problems of size up to 8000x8000. The authors have also extended their work on a multi-GPU implementation [33] and their computational results showed a maximum speedup of 24.5, using two Tesla C2050 boards. Meyer et al. [34] proposed a mono- and a multi-GPU implementation of the tableau simplex algorithm, and they compared its performance to the serial Clp solver, using a Tesla S1070 board with T10 GPUs. Their implementation outperformed Clp solver on large sparse LP problems. Finally, Ploskas and Samaras [35] proposed two efficient GPU-based implementations of the revised simplex and a primal–dual exterior point simplex algorithm, using Matlab. The experimental results showed great speedups for the exterior point algorithm and quite worse for the revised simplex method. Other valuable attempts can also be found in [36-38] achieving very satisfactory speedups with C1060, S1070 and GTX 670 boards.

Considering also the general field of parallel scientific computing, several attempts have been made adopting extended CPU/GPU collaboration. Harmony [15] is an integrated programming model which allows the coding and executing of programs for CPU/GPU systems. It also includes an automated distribution of the computational load on the CPU and the GPU, and achieves very good performance mainly in audio-processing systems. In [14,16-17] more recent, relevant approaches are presented in the field of linear algebra and systems. In [17] an additional speedup of up to 1.25 is achieved (compared to the GPU-only implementation) for the parallel execution of the conjugate gradient method, whereas in [14,16] the CPU/GPU collaboration schemes in the field of linear algebra achieved a speedup ranging from 1.15 up to 1.35 for different sizes and types of problems. The works of [14,17] have the major advantage of supporting dynamic distribution of the workload between CPU and GPU. Finally in [39] the authors present a novel generic framework that transparently orchestrates collaborative execution of a single data-parallel kernel across multiple asymmetric CPUs and GPUs. To our knowledge there is no relevant approach in the literature adopting extended CPU/GPU collaboration (i.e. not only for the reduction operations as in [19] or semi-hybrid as in [20]) for parallelizing the simplex method.

III. The Simplex Method

In linear programming problems [34], 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 I (full tableau representation), 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 is more efficient for full dense linear problems and it can be easily converted to a distributed version for cluster platforms or hybrid environments.

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

Initialization Step: Start with a feasible basic solution and construct the corresponding 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 min ratio test to the elements of the winning column and choose the row number with the min ratio – leaving variable).

Step 3: Pivoting (this step 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.

Iterate/Finalization Step: Repeat the above steps until the best solution is found or the problem gets unbounded.

IV. BASIC PARALLELIZATON

In the following paragraphs we present in details the algorithmic approach we followed in our basic hybrid parallel implementation. Our approach is based on the most popular and widely used column-based distribution scheme [4] (as opposed to the other relevant alternative of row-based distribution). 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 (both theoretically and experimentally) regarded as the most effective one in the general case.

Following this scheme all the computation parts except step 2 of the basic (sequential) 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. It has also been proved to be the most efficient one (as shown in the literature [4,30]). The basic steps of the algorithm are given below:

***Initialization Step*:** 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 simplex tableau all the calculations required for the global rows pivoting, based on the pivot data received during step 4.

***Iterate/Finalization Step*:** The above steps are repeated until the best solution is found or the problem gets unbounded.

Based on the above step by step decomposition we've designed and implemented our basic hybrid parallelization scheme, assuming a hybrid parallel platform which consists of multiple multicore nodes interconnected via a high-speed communication network. MPI was used for the communication among the network connected nodes, whereas OpenMP was used for the communication among the multiple cores in each node. More concretely, the available constructs, functions and special mechanisms of both the above parallelization frameworks were suitably used as follows:

- Appropriately built OpenMP *parallel for* constructs were used for the efficient thread-based parallelization of the loops implied by steps 1, 3 and 5.
- Especially with regard to the parallelization of steps 1 (in cooperation with step 2) and 3, in order to optimize the parallel implementation of the corresponding procedures (which both involve a reduction operation), we used the *min/max reduction operators* of OpenMP API.
- Also, with regard to the parallelization of step 5, in order to achieve even distribution of computations to the working threads (given that the computational costs of the main loop iterations cannot be regarded a-priori equivalent) we used *collapse-based nested parallelism in combination with dynamic scheduling policy*.
- Beyond the OpenMP-based parallelization inside each node, the well-known MPI *collective communication functions (MPI_Scatter, MPI_Bcast, MPI_Reduce etc.)* were also used for the communication between the network connected nodes as in pure MPI implementation.

V. CPU / GPU COLLABORATION

Furthermore, considering within each multicore node the case of existence of a CUDA-enabled GPU, we've extended our basic parallel approach in such a way that it can exploit concurrently the full power of the provided resources (i.e. using the CPU cores and the GPU concurrently), and thus lead to even better performance. The relevant extension is based on a suitable hybrid multi-threading/GPU offloading scheme, implemented with the combined use of OpenMP and CUDA. In the following paragraphs we briefly present the extended algorithm separately (as an autonomous hybrid approach operating on a single-node multicore environment with a CUDA-enabled GPU), for better understanding. Apparently, it can fit in a straightforward manner to our fully hybrid approach described in the previous section, in the case of multi-node environments (see also section VII). Specifically, we first suppose (upper-level parallel approach) that a global column-based distribution scheme is followed with regard to the distribution of the full simplex tableau among the provided resources (CPU-cores and GPU).

Next, with regard to the required CPU/GPU collaboration we apply a suitable extension of the GPU-oriented parallel approach presented in [19][1], by assigning a portion (a number of columns) of the full simplex tableau to be processed by the GPU and leaving the remaining portion to be processed by the CPU. However, with respect to the internal processing within the GPU-cores, the distribution scheme of the corresponding tableau portion is turned to a block-oriented one, which fits better to the internal architecture and the processing capabilities of an NVIDIA GPU [19].

Based on the above considerations, and assuming that we have a single node with n CPU-cores and one CUDA-enabled

[1] The work of [19] is mostly a GPU-only approach, with the CPU being used only for the reduction operations.

GPU, we've implemented a hybrid CPU+GPU implementation (OpenMP+CUDA) as follows:

Setup: A process with t ($n \geq t \geq 2$) threads is scheduled, with one of them mainly used for GPU handling (offloading / kernel launching) and the remaining t-1 threads kept for CPU assigned computations.

Initialization Step: A portion θ of the simplex tableau is offloaded to the GPU. The remaining portion (1-θ) remains for shared-memory computations among the t-1 CPU threads. The right-hand constraints vector is both offloaded to the GPU and kept in the CPU shared memory too.

Step 1: The t-1 CPU threads and the GPU compute (in parallel) the maximum negative coefficient over their own protion of the first line of the simplex tableau, yielding to two local maximum index values, say k1 and k2 respectively. The GPU local maximum k2 is transferred to the CPU memory, it is then compared to k1, and the global maximum index of the winning column is determined (entering variable).

Step 2: If the entering variable belongs to the portion θ of the GPU-assigned simplex tableau, the index k is transferred to the GPU memory. The ratio computation is applied to all the elements of the winning column in the GPU. The minimum ratio is also computed in parallel by the GPU cores and the index r of the winning row is determined (leaving variable). The index r of the winning row as well as the elements of the winning column are transferred to the CPU memory.

Step 3: If the entering variable belongs to the portion 1-θ of the CPU-assigned simplex tableau, the t CPU threads apply in parallel the ratio computation to all the elements of the winning column in the CPU memory. Consequently, they also compute in parallel the minimum ratio, and the index r of the winning row is determined (leaving variable). The index r of the winning row as well as the elements of the winning column are transferred to the GPU memory.

Step 4: The t-1 CPU threads and the GPU perform (in parallel) on their own portion of the simplex tableau all the calculations required for the global rows pivoting, based on the pivot data received during the previous step.

***Iterate/Finalization Step*:** The above steps are repeated until the best solution is found or the problem gets unbounded. A suitable synchronization is required in this step between CPU and GPU per iteration.

The tasks implied by steps 1 and 2 of the sequential algorithm (determining the entering and the leaving variables) require finding a max/min within a set of values. In our hybrid approach, part of the corresponding operations are being performed in the GPU, using appropriate reduction techniques. Our experiments showed (as opposed to [19] and [20]) that the performance obtained by sharing these reduction steps in both the CPU and GPU, was at least equivalent (and in any case note worse) to the alternative followed there (of performing the reduction operations totally in the CPU). The relatively large size of the tested problems, the double precision operations, and the limitations of the NVIDIA architecture itself, lead to limited efficiency when the GPU participates in the reduction computations. However, in the more recent NVIDIA GPUs the efficiency of these computations has been improved, thus allowing their proper use in corresponding tasks.

VI. Experimental Results

Our basic parallelization scheme presented in section IV has been implemented with the use of MPI 3.0 message passing library and OpenMP 4.0/4.5 API, and it has been extensively tested (in terms of speed-up and efficiency measures) over a powerful hybrid parallel environment (distributed memory, multi-core nodes)[2]. 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. The efficiency measure actually represents the fraction of the maximum theoretical speed-up that has been achieved. The corresponding results are presented and discussed in the next paragraph, whereas in the rest of the section we give the results of the extensions presented in section V. Our test environment for this set of experiments consists of up to 8 Intel Core 3.0GHz quad-core processors (making a total of 32 cores) with 4GB RAM each, connected via gigabit ethernet (1Gbps) network interface. The relevant computing components were mainly available and accessed through the Okeanos Cyclades cloud computing services [40] and local infrastructure in T.E.I. of Athens and Democritus University of Thrace.

A. Performance of the MPI+OpenMP hybrid scheme

In order to examine and validate the high efficiency and scalability of our basic hybrid MPI+OpenMP parallelization scheme, we've run on our platform a suitable subset of the well known and widely used NETLIB test linear problems of varying (large and very large) sizes that reflect close to the real word practical cases. The corresponding measurements, over all the non-trivial power-of-two numbers of processors/cores (from 4 up to 32), are presented in Table II.

As it can be seen in Table II the achieved speed-up and efficiency values of the hybrid MPI+OpenMP approach are particularly high in all cases, even for large number of cores and very large NETLIB problems. One can also easily observe that the efficiency values decrease with the increase of the number of processors. However, this decrease is quite slow, and the efficiency values remain high even for 16 and 32 processors/cores (no less than 81% and 70% respectively), in all cases. Moreover, particularly high efficiency values (almost linear speedup) are achieved for all the high aspect ratio NETLIB problems (e.g. see the values for problems FIT2P, 80BAU3B and QAP15 where the efficiency even for 16 and 32 processors/cores is over 90% and 85% respectively). This happens because in the case of 16 or 32 processors/cores (4 and 8 nodes respectively), although the required communications progressively increase, as it is shown in [30]: the higher the aspect ratio of the linear problem the better the performance of the column distribution scheme we follow here, with regard to the total communication overhead.

[2] Much more powerful than the one used in [18].

TABLE II. SCALABILITY OF MPI+OPENMP FOR VERY LARGE PROBLEMS

Linear Problems	Speed-up & Efficiency / MPI+OpenMP							
	2x2=4 cores		2x4=8 cores		4x4=16 cores		8x4=32 cores	
	Sp	Ep	Sp	Ep	Sp	Ep	Sp	Ep
FIT2P (3000x13525)	3.94	98.50%	7.80	97.50%	15.25	95.30%	29.50	92.20%
80BAU3B (2263x9799)	3.91	97.80%	7.72	96.50%	14.93	93.30%	28.64	89.50%
QAP15 (6330x22275)	3.89	97.30%	7.62	95.30%	14.48	90.50%	27.10	85.20%
MAROS-R7 (3136x9408)	3.87	96.80%	7.54	94.30%	14.13	88.30%	26.27	82.10%
QAP12 (3192x8856)	3.86	96.50%	7.50	93.80%	13.97	87.30%	25.70	80.30%
DFL001 (6071x12230)	3.85	96.30%	7.50	93.80%	14.05	87.80%	25.95	81.10%
GREENBEA (2392x5405)	3.84	96.00%	7.40	92.50%	13.58	84.90%	24.38	76.20%
STOCFOR3 (16675x15695)	3.79	94.80%	7.23	90.40%	12.96	81.00%	22.50	70.30%

Note also that the proposed hybrid scheme has been shown [18] to perform better than the alternative of MPI+MPI3.0 Shared Memory approach, as well as than the coresponding implementation of [4] which is one of the most competing relevant approaches in the literature. Furthermore, consider that the implementation of [4] 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.

B. Performance of the CPU/GPU collaboration scheme

The complementary single-node hybrid parallellization scheme presented in section V (that assumes the existence of a CUDA-enabled GPU as well) has also been implemented with the use of OpenMP 4.5 API and CUDA 7.0 Toolkit, and it has been extensively tested over a real hybrid hardware platform (much more powerful than the one in [20]). More concretely, for this set of experiments we've used an Intel Dual Quad Core 3.0GHz Xeon system (8 cores in total), as well as one GTX 760 and one GTX TitanX NVIDIA GPUs, which are of different technologies (Kepler and Maxwell respectively). These desktop-level GPUs have relatively low double precision (DP) performance (the GTX 760 gives ~95 GFLOPS with 1152 cores, whereas the GTX TitanX gives ~192 GFLOPS with 3072 cores), however as it can be seen they can lead to quite significant improvements. This emphasizes the capability of using GPUs for scientific computing on desktop environments too. Note also that we've chosen to use the above referred 8-core Xeon system instead of one of the quad-core machines used in our first set of experiments in order to have more available cores in a single machine, and conclude to more representative, conveincing and sufficiently reliable results. However, it should be noticed that the per core performance of the two different test platforms is approximately the same.

1) Performance of the GPU offloading only scheme

First, we briefly present our initial experiments, in which our CPU-only and GPU-only schemes are compared to each other for varying number of CPU-cores. The performance gains achieved by our GPU-only approach are shown in Tables III,IV as well as in Fig. 1. Later on we present the additional performance gains achieved by our CPU/GPU collaboration (OpenMP+CUDA) scheme over the GPU-only approach, thus demonstaring the really high level of improvements that can be offered by the use of a combined CPU/GPU computing approach in hybrid (multi-node, multi-core) environments that involve CUDA-enabled GPUs too. The measurements presented in Tables III and IV have been taken over a dense randomly generated LP problem of size equal to 10000x10000 and similar properties as in [19,33].

The specific problem size is the larger one in our experiments and leads to the best speedup values for all the tested cases. It's also near the maximum LP problem size that can fit and be processed conveniently within the available memory of the GTX 760 card (2GB), which is the main card used in the experiments made over our fully hybrid platform (presented in section VII). Further experiments involving quite larger LP problems over the GTX Titan X card (which offers a substantially larger amount of memory, i.e. 12GB) are of high priority in our future work.

In Table III, in the first columns the performance measurements for our OpenMP (CPU-only) implementation are shown. Specifically, the execution time per iteration is given for varying number of cores (from 1 up to 8) as well as the corresponding speedup (Sp) values achieved in each case. On the other hand, in the last two columns we give the execution time per iteration achieved by the GPU-only implementation with the GTX 760 GPU, as well as the corresponding speedup achieved over the CPU-only implementation for each different number of cores. In Table IV the relevant measurements are presented for the GTX TitanX GPU, in an equivalent manner.

As it can be seen the speedup achieved with the GTX 760 GPU ranges from 1.94 (compared to the 8-core CPU-only implementation) to 14.06 (compared to the 1-core/sequential implementation), whereas the speedup achieved with the GTX TitanX GPU ranges from 3.20 to 23.22 respectively. These speedup values are quite satisfactory and they validate the worth of using desktop-level GPUs for this kind of scientific computations, although their DP performance is relatively low. They are also comparable to other relevant approaches in the literature, and quite better than the ones of [20]. For example in

[19] a speedup of 12.5 is achieved over the sequential execution, with a GTX 260 GPU (which has a DP performance of ~90 GFLOPS). Note also that our CPU-only implementation is a highly efficient/scalable one, since the efficiency values (obtained if we divide Sp by the corresponding number of processors in each case) are over 90% in all cases and the speedup remains particularly high (7.26) even for 8 cores.

TABLE III. SPEEDUP FOR GTX 760 GPU IMPLEMENTATION

P	CPU (multi-threaded)		GPU (760)	
#cores	Time/iter	Sp	Time/iter	Sp
1	2.8915	1.00	0.2056	14.06
2	1.5139	1.91	0.2056	7.37
3	1.0135	2.85	0.2056	4.93
4	0.7674	3.77	0.2056	3.73
5	0.6245	4.63	0.2056	3.04
6	0.5250	5.51	0.2056	2.56
7	0.4519	6.40	0.2056	2.20
8	0.3985	7.26	0.2056	1.94

TABLE IV. SPEEDUP FOR GTX TITANX GPU IMPLEMENTATION

P	CPU (multi-threaded)		GPU (TitanX)	
#cores	Time/iter	Sp	Time/iter	Sp
1	2.8915	1.00	0.1245	23.22
2	1.5139	1.91	0.1245	12.16
3	1.0135	2.85	0.1245	8.14
4	0.7674	3.77	0.1245	6.17
5	0.6245	4.63	0.1245	5.02
6	0.5250	5.51	0.1245	4.21
7	0.4519	6.40	0.1245	3.62
8	0.3985	7.26	0.1245	3.20

TABLE V. EXECUTION TIMES FOR CPU+GPU IMPLEMENTATION

portion	CPU+GPU(760)		CPU+GPU(TitanX)	
(θ)	4 cores	8 cores	4 cores	8 cores
0.0	0.7674	0.3985	0.7674	0.3985
0.1	0.6817	0.3591	0.6753	0.3526
0.2	0.6070	0.3177	0.5953	0.3136
0.3	0.5288	0.2793	0.5167	0.2703
0.4	0.4572	0.2370	0.4493	0.2326
0.5	0.3909	0.1984	0.3824	0.2015
0.6	0.3337	0.1602	0.3250	0.1707
0.7	0.2598	0.1765	0.2482	0.1339
0.8	0.1853	0.1883	0.1788	0.1015
0.9	0.2022	0.2027	0.1142	0.1176
1.0	0.2056	0.2056	0.1245	0.1245

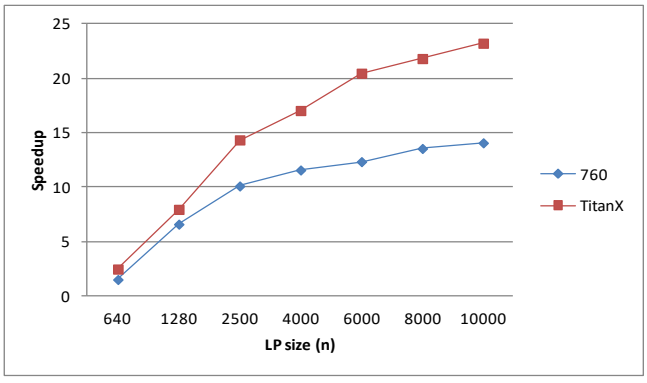

Fig. 1. Speed-up curves for different LP sizes

Additionally, in Fig.1 the behavior of our GPU-only approach over different sizes of LP problems is shown, in terms of the corresponding speedup curves). The experiments have been performed over randomly generated dense LP problems ranging in size from 640x640 to 10000x10000, of similar properties as in [19,33], and with double precision arithmetic. Note also that the speedup values have been computed (without loss of generality) comparing to the 1-core/sequential implementation. As it was expected the speedup increases with the increase of the problem size. Moreover, the speedup reaches a sufficiently high value (near the maximum) for LP problems greater or equal to 2500x2500, whereas it decreases sharply for smaller LP problems. This happens because as the problem size decreases the shared computational load also decreases a lot, and the total CPU-GPU communication overhead (and/or the corresponding reduction overhead) naturally becomes the dominant factor with regard to the total processing time.

2) Performance of the Hybrid OpenMP+CUDA Scheme

In our second set of experiments we measure the performance of our hybrid CPU/GPU implementation and we show its superiority over the GPU-only implementation, which was the faster among the other two. The measurements were taken by varying the load distribution factor (portion θ) of the simplex tableau, from 0 (equivalent to the CPU-only approach) to 1 (equivalent to the GPU-only approach) by steps of 0.1.

In Table V, the corresponding execution times are given for both the tested GPUs, supposing they share the computational load (according to the varying value of θ) with 4 and 8 CPU cores, over the randomly generated 10000x10000 LP problem; which gives the better performance. As it can be seen, in all cases there is at least one value of θ that leads to better execution time than the GPU-only implementation. This clearly validates the worth of using both resources (CPU and GPU) concurrently, instead of the GPU alone. Moreover, the maximum improvement is achieved for 8 CPU-cores, where we have an improvement of 22.1% (from 0.2056 to 0.1602) for the GTX 760 GPU and 18.5% (from 0.1245 to 0.1015) for the GTX TitanX GPU. In terms of speedup values the above improvements imply an additional speedup improvement of 1.28 and 1.23 for our two different test cards respectively. These achievements are comparable to the ones presented in other recent works in the literature [14,16-17].

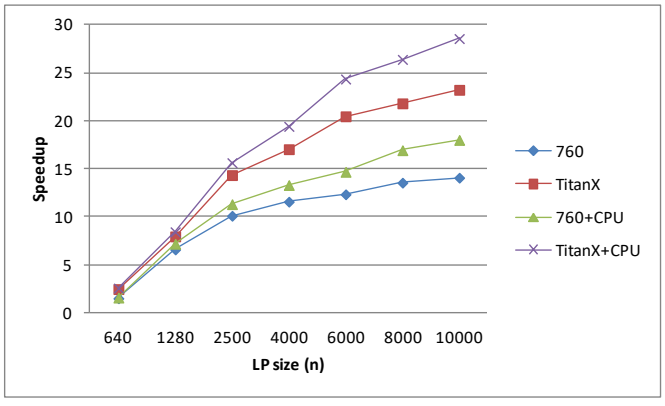

Fig. 2. Speedup curves for CPU/GPU collaboration

Furthermore, the value θ that leads to the larger improvement for 4 CPU-cores is 0.8 for the GTX 760 and 0.9 for the GTX TitanX GPU, whereas the corresponding values for 8 CPU-cores become 0.6 and 0.8 respectively. The fact that are all greater than 0.5 is not surprising since the GPU-only implementation leads to significant speedup over the CPU-only implementation, itself; near or greater than 2 in any case (even for 8 cores). Thus, it's naturally expected that in order to gain some additional improvement from CPU/GPU collaboration, the larger portion of the simplex tableau should be offloaded to the GPU. Intuitively, we can say that if the GPU-only implementation leads to a speedup of 's' compared to the corresponding CPU-only implementation, in order to improve the execution time by applying CPU/GPU collaboration, we should assign to the CPU certainly less than the '1/s' portion (practically much less) of the total computational load. Also, as the number of the CPU-cores increase the maximum improvement that can be achieved naturally increases too, since a greater portion of computational load can be assigned to the CPU with equivalent performance as it was in GPU.

Also, in Fig. 2, the behavior of the hybrid CPU/GPU approach for different sizes of LP problems is shown. Specifically, the execution times and the speedup values (over the 1-core/sequential implementation) have been measured for the randomly generated LP problems ranging in size from 640x640 to 10000x10000, and the corresponding speedup curves have been drawn along with the speedup curves of Fig. 1. All the measurements have been taken for 8 participating CPU cores and for the value θ that maximizes the performance in each case. As it can be seen the maximum achieved speedup decreases with the decrease of the problem size, for both the tested GPUs. This is naturally explained again by the fact that as the problem size decreases the shared computational load decreases analogously, and the CPU-GPU communication overhead becomes more crucial for the total processing time. Note that in the CPU/GPU collaboration scheme there is additional CPU-GPU communication (i.e. per iteration) beyond the initial offloading overhead.

VII. Fully Hybrid Parallelization

Finally (standing as the most important contribution of our work), we've appropriately integrated our basic hybrid schemes into one fylly hybrid design, and we've performed a number of compementary experiments in order to demonstrate the worth of use of our parallel approaches in a fully hybrid parallel platform, i.e. a multi-node environment with multicore nodes and also equipped with one CUDA enabled GPU. As it has already been mentioned in Section V, our CPU/GPU collaboration approach can fit in a straightforward manner to our MPI+OpenMP approach, in the case of multi-node environments. More concretely, we first have to share the whole simplex tableau and broadcast the right-hand constraints vector to all the processors, according to the algorithmic approach presented in Section IV. Then each processor can execute the CPU/GPU collaboration algorithm presented in Section V on the local part of the simplex tableau that it has received. Furthermore, additional (MPI-based) communication steps between all processors have been introduced in each iteration.[3] Our test equipment for this set of experiments is the same as for our first set of experiments (up to 8 Intel 3.0GHz quad-core processors with 4GB RAM and gigabit ethernet network connection), with each node also equipped with a GTX 760 NVIDIA GPU. Based on the above platform, we've run experiments over the set of very large NETLIB problems used for the experiments in paragraph VI-A; where the MPI+OpenMP scheme is used on the same platform, however without the GTX 760 GPUs. The corresponding results are presented in Table VI, for the cases of 16 (4 nodes/processors) and 32 (8 nodes/processors) cores in total, together with the corresponding results of Table II for comparative reasons.

As it can be seen in Table VI, the combined use of the GTX 760 GPUs in each node introduces a substantial improvement on the speedup and efficiency values for both the cases of 4 and 8 nodes/processors[4]. This improvement increases as the size of the problem gets larger, and it rises up to 15,9% (the efficiency increases from 70.3% to 81.5%) for STOCFOR3 (which is the larger LP problem in our experiments) over 8 nodes/32 cores. This is naturally explained by the fact that in larger problems the computational load is much larger within each node/processor, so a substantial improvement on their execution time can influence significantly the total performance, especially in the case of prior downgrade due to increased communication overhead. For smaller problems (e.g. FIT2P and 80BAU3B) the improvement is quite smaller (however clear in all cases) since the speedup and efficiency values for these problems is quite high even without the use of the GPUs. Also the corresponding improvement decreases a little as the number of nodes decreases, e.g. for STOCFOR3 it becomes equal to 11.7% over 4 nodes/16 cores (the efficiency increases from 81% to 90.5%). This happens for similar reasons as above, i.e. because for less number of nodes the communication overhead is quite smaller, so the total speedup and efficiency values are quite higher and the substantial improvement in the execution times of the computational tasks cannot influence the total performance in the same degree.

[3] In order the globally max negative coefficient be computed and the winning column/row be broadcasted.

[4] Note that as it comes out from Table III (comparing to the use of 4 CPU cores), we can have a speedup of 3.73 (0.2056 vs. 0.7674) by using a GTX 760 GPU, whereas we can have an additional 10% (0.1853 vs. 0.2056, see Table 5) by using the 4 CPU cores and the GTX 760 GPU in collaboration.

TABLE VI. PERFORMANCE OF THE FULL HYBRID SCHEME WITH GTX 760 GPUS

Linear Problems	MPI+OpenMP				MPI+OpenMP+Cuda (GTX 760)			
	4x4=16 cores		8x4=32 cores		4x4=16 cores		8x4=32 cores	
	Sp	Ep	Sp	Ep	Sp	Ep	Sp	Ep
FIT2P (3000x13525)	15.25	95.30%	29.50	92.20%	15.66	97.90%	30.43	95.10%
80BAU3B (2263x9799)	14.93	93.30%	28.64	89.50%	15.47	96.70%	30.18	94.30%
QAP15 (6330x22275)	14.48	90.50%	27.10	85.20%	15.25	95.30%	29.50	92.20%
MAROS-R7 (3136x9408)	14.13	88.30%	26.27	82.10%	15.18	94.90%	28.86	90.20%
QAP12 (3192x8856)	13.97	87.30%	25.70	80.30%	15.06	94.10%	28.54	89.20%
DFL001 (6071x12230)	14.05	87.80%	25.95	81.10%	15.06	94.10%	28.61	89.40%
GREENBEA (2392x5405)	13.58	84.90%	24.38	76.20%	14.85	92.80%	27.58	86.20%
STOCFOR3 (16675x15695)	12.96	81.00%	22.50	70.30%	14.48	90.50%	26.08	81.50%

VIII. CONCLUSION

A highly scalable parallel implementation framework of the standard full tableau simplex method on a hybrid experimental platform has been presented and evaluated throughout the paper, in terms of typical performance measures. Specifically, we have designed, implemented and evaluated a highly efficient hybrid parallelization scheme over a real hybrid parallel (distributed memory, multicore) platform, with use of the standard Netlib test linear problems. The proposed hybrid scheme involves the use of OpenMP for the parallelization over the cores of each node and MPI for the communication between the nodes themselves, and as shown in all the experiments it leads to particularly high speedup and efficiency values even for very large / huge LP problems. It has also been shown to perform considerably better than other competitive approaches of the literature. Further, our hybrid MPI+OpenMP scheme is then suitably extended in order to gain improved performance when a CUDA-enabled GPU is also involved in the hybrid environment. A robust hybrid CPU multithreading/ GPU offloading scheme is proposed that can efficiently use the CPU cores and the GPU concurrently. A GPU-offloading only scheme has also been implemented for comparison purposes. In the corresponding experiments the proposed CPU/GPU-collaboration scheme proves to be superior to the GPU-only scheme, with an additional speedup of up to 1.28 being achieved for randomly generated LP problems of size 10000x10000. The performance of both the CPU/GPU-collaboration and the GPU-only schemes are comparable to other relevant implementations in the literature, whereas they are also clearly superior to the OpenMP-based CPU-only implementation (even for 8 cores). Finally, and the most important, we have integrated our proposed schemes into a *fully hybrid* one, and we present some very encouraging experiments which lead to quite significant improvements (up to 15,9%); when this scheme is used over a *fully hybrid* multi-node platform, for large and very large LP problems. In all the experiments we've used desktop-level GPUs in order to emphasize the capability of using the GP-GPU computing model for scientific applications on desktop environments too.

REFERENCES

[1] J.A. Hall. Towards a Practical Parallelization of the Simplex Method. Computational Management Science, Springer, 7(2), 2010, pp. 139-170.

[2] E.S. Badr, M. Moussa, K. Paparrizos, N. Samaras and A. Sifaleras. Some Computational Results on MPI Parallel Implementation of Dense Simplex Method. World Acad. of Science, Engineering, Technology, 23, 2008, pp. 778-781.

[3] J. Qin and D.T. Nguyen. A Parallel-vector Simplex Algorithm on Distributed-Memory Computers. Structural Optimizations, 11(3), 1996, pp. 260-262.

[4] G. Yarmish and R.V. Slyke. A Distributed Scaleable Simplex Method. Journal of Supercomputing, Springer, 49(3), 2009, pp. 373-381.

[5] 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, pp. 571-596.

[6] K.K. Sivaramakrishnan. A Parallel Interior Point Decomposition Algorithm for Block Angular Semidefinite Programs. Computational Optimization and Applications, 46(1), 2010, pp. 1-29.

[7] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W.D. Gropp, V. Kale and R. Thakur. MPI + MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory. Computing J, Springer, 95, 2013, pp. 1121-1136.

[8] B. Chapman, G. Jost and R. van der Pas. Using OpenMP. MIT Press, 2008.

[9] R. Rabenseifner, G. Hager and G. Jost. Hybrid MPI and OpenMP Parallel Programming. Supercomputing 2013 Conference, Nov 17-22, Denver, USA, Tutorial, http://openmp.org/ wp/sc13-tutorial-hybrid-mpi-and-openmp-parallel-programming, 2013.

[10] R. Rabenseifner, G. Hager and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. Proceedings of 17th Euromicro Intl. Conf. on Parallel, Distributed and Network-based Processing, 2009, pp. 427-436.

[11] Rabenseifner, R., and Wellein, G. Comparison of Parallel Programming Models on Clusters of SMP Nodes. Proceedings of the Intl. Conf. on High Performance Scientific Computing, March 10-14, Hanoi, Vietnam, 2004, pp. 409-426.

[12] B. Mamalis and G. Pantziou. Advances in the Parallelization of the Simplex Method. Proceedings of ALGO 2015 Annual Event, Springer, LNCS 9295 Festschrift (P. Spirakis), September 14-18, Patras, Greece, 2015, pp. 281-307.

[13] O. Adenikinju, J. Gilyard, J. Massey and T. Stitt. Concurrent Solutions to Linear Systems using Hybrid CPU/GPU Nodes, SIAM Undergraduate Research Online, vol. 8, 2015, pp. 1-10.

[14] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier and J. Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing, Parallel Computing, 38 (1-2), 2012, pp. 37-51.

[15] G.F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems, in 17th International Symposium On High performance distributed computing, HPDC'08, ACM, 2008, pp. 197-200.

[16] M. Fatica. Accelerating linpack with CUDA on heterogenous clusters, in 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, ACM, New York, NY, USA, 2009, pp. 46-51.

[17] J. Lang and G. Runger. Dynamic distribution of workload between CPU and GPU for a parallel conjugate gradient method in an adaptive FEM, ICCS 2013 Conference, in Procedia Computer Science, 18, 2013, pp. 299-308.

[18] B. Mamalis and M. Perlitis. Hybrid Parallelization of Standard Full Tableau Simplex Method with MPI and OpenMP. Proceedings of the 18th Panhellenic Conference in Informatics, ACM ICPS, 2014, pp. 1-6.

[19] M.E. Lalami, V. Boyer and D. El-Baz. Efficient Implementation of the Simplex Method on a CPU-GPU System, IEEE International Parallel & Distributed Processing Symposium, 2011, pp. 1994-2001.

[20] B. Mamalis and M. Perlitis. A Hybrid Parallelization Scheme for Standard Simplex Method based on CPU/GPU Collaboration, Proceedings of the 20th Panhellenic Conference in Informatics, ACM ICPS, 2016, pp. 12.

[21] J.A. Hall and K. McKinnon. ASYNPLEX an Asynchronous Parallel Revised Simplex Algorithm. Annals of Operations Research, 81, 1998, pp. 27-49.

[22] W. Shu and M.Y. Wu. Sparse Implementation of Revised Simplex Algorithms on Parallel Computers. Proceedings of the 6th SIAM Conference in Parallel Processing for Scientific Computing, Norfolk, VA, USA, 1993, pp. 501-509.

[23] M.E. Thomadakis and J.C. Liu. An Efficient Steepest-edge Simplex Algorithm for SIMD Computers. Proceedings of the Intl. Conference on Supercomputing, Philadelphia, PA, USA, 1996, pp. 286-293.

[24] 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, Vol. 7 (4), 1995, pp. 402-416.

[25] C.B. Stunkel. Linear Optimization via Message-based Parallel Processing. Proceedings of Intl. Conf. on Parallel Processing, Pennsylvania, PA, USA, 1998, pp. 264-271.

[26] D. Klabjan, L.E. Johnson and L.G. Nemhauser. A Parallel Primal-dual Simplex algorithm.. Operations Research Letters, Vol. 27 (2), 2000, pp. 47-55.

[27] I. Maros and G. Mitra. Investigating the Sparse Simplex Method on a Distributed Memory Multiprocessor, Parallel Computing, 26(1), 2000, pp. 151-170.

[28] J.A. Hall and Q. Huangfu. A high performance dual revised simplex solver. Parallel Processing and Applied Mathematics, LNCS 7203, Springer, 2012, pp. 143-151.

[29] Q. Huangfu and J.A. Hall. Parallelizing the dual revised simplex method. Technical Report ERGO-14-011, http://www.maths.ed.ac.uk/hall/Publications.html, 2014.

[30] B. Mamalis, G. Pantziou, D. Kremmydas and G. Dimitropoulos. Highly Scalable Parallelization of Standard Simplex Method on a Myrinet Connected Cluster Platform. Acta Intl. Journal of Computers and Applications, 35(4), 2013, pp. 152-161.

[31] D.G. Spampinato and A.C. Elster. Linear optimization on modern GPUs, in Proc. of the 23rd IEEE IPDPS'09 Conference, 2009, pp. 1-8.

[32] J. Bieling, P. Peschlow and P. Martini. An efficient GPU implementation of the revised Simplex method, in Proc. of IEEE 24th International Symposium on the Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, pp. 1-8.

[33] M.E. Lalami, D. El-Baz and V. Boyer. Multi GPU implementation of the simplex algorithm, in Proc. of the 2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC), Banff, Canada, 2011, pp. 179-186.

[34] X. Meyer, P. Albuquerque and B. Chopard. A multi-GPU implementation and performance model for the standard simplex method, in Proc. of the 1st Intl. Symposium and 10th Balkan Conference on Operational Research, Thessaloniki, Greece, 2011, pp. 312–319.

[35] N. Ploskas and N. Samaras. Efficient GPU-based implementations of simplex type algorithms. Applied Mathematics and Computation, 250, 2015, pp. 552–570.

[36] V. Boyer and D. El-Baz. Recent Advances on GPU Computing in Operations Research. In Proc. of IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPSW), 2013, pp. 1778-1787.

[37] A. Gurung, B. Das and R. Rajarshi. Simultaneous Solving of Linear Programming Problems in GPU, in Proc. of IEEE HIPC 2015 Conference: Student Research Symposium on HPC, Vol. 8, Bengaluru, India, 2015, pp. 1-5.

[38] C.T. Yang, C.L. Huang and C.F. Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU Clusters, Computer Physics Communications, 182, 2011, pp. 266–269.

[39] J. Lee, M. Samadi, Y. Park and S. Mahlke. Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems, in Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT '13, 2013, pp. 245-256.

[40] Okeanos Cyclades Cloud Services. Greek Ministry of Education, General Secretariat for Research and Technology (G.S.R.T.), https://okeanos.grnet.gr/services/cyclades, 2015.