Advances in the Parallelization of the Simplex Method

Basilis Mamalis(✉) and Grammati Pantziou

Department of Informatics, Technological Educational Institute of Athens, Athens, Greece
{vmamalis,pantziou}@teiath.gr

Abstract. The simplex method has been successfully used in solving linear programming problems for many years. Parallel approaches for the simplex method have been extensively studied in the literature due to the intensive computations required, especially for the solution of large linear problems (LPs). In this paper, first a detailed overview is given of the parallelization attempts concerning the standard and the revised simplex method made to date. Next, some of the most recent and significant relevant attempts are selected and presented in more detail along with experimental results. The latter include some impressive results obtained for the revised simplex method over a modern supercomputer, as well as the recent advances in GPU-based related attempts.

Keywords: Linear programming · Simplex method · Parallel computing

1 Introduction

Linear programming (LP) is probably the most important and well studied optimization technique. The simplex method has been successfully used for solving linear programming problems for many years [1]. Parallel approaches for the simplex method have been extensively studied in the literature due to the intensive computations required [2]. Most research 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 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 implementations of the revised method that scale well [2]. On the other hand, the standard method is more efficient for dense linear problems and it can be easily converted to a parallel version with satisfactory speedup values and good scalability (e.g. [3–5,11–13]). However, lately, some alternative very promising efforts have also been made with regard to the parallelization of the revised method, based either on the block angular structure (or decomposition) of the initially transformed problems or on the dual form of the revised simplex, which is the most preferable one nowadays [6–8]. Two other (a little earlier) valuable attempts over specific variants of the simplex method

DOI: 10.1007/978-3-319-24024-4_17

have also been presented in [14,15], and they have led to quite satisfactory results for large scale problems.

With regard to the hardware architectures used, earlier work focused mainly on more complex, and more tightly coupled, networking structures than a cluster or a network of workstations, which became a quite familiar alternative later on. Hall and McKinnon [9] and Shu [10] worked on parallel revised methods over known supercomputer environments (like Cray T3D). Thomadakis and Liu [11] worked on the standard method utilizing the MP-1 and MP-2 MasPar. Eckstein et al. [12] showed in the context of the parallel connection machine CM-2 that the iteration time for the parallel revised method tended to be significantly higher than for the parallel full tableau method even when the revised method is implemented very carefully. Stunkel [13] found a way to parallelize both the revised and standard methods so that both obtained a similar advantage in the context of the parallel Intel iPSC hypercube. Some more recent works on the standard simplex parallelization [16–18] address the significant influence of the number of columns and rows (aspect ratio) of an LP problem when a distributed-memory architecture is used, and achieve particularly high speedup values over modern (hybrid) cluster architectures, mainly following the column-based data distribution scheme [3,4,16–18].

Studying the literature one can notice that the standard (tableau-based) simplex method has been efficiently parallelized many times in the past with good speedup factors ranging from tens to up to a thousand. However, without using expensive parallel computing resources, its performance is inferior to a good sparsity-exploiting sequential implementation of the revised simplex method. On the other hand, the parallelization of the revised simplex method has not been very efficient and therefore there has been less success in terms of speedup with respect to the standard method. Indeed, since scalable speedup for general large sparse LP problems appears unachievable, the revised simplex method has been considered unsuitable for parallelization. However, since it corresponds to the computationally efficient serial technique, any improvement in performance due to exploiting parallelism in the revised simplex method is a worthwhile goal.

In the above context, throughout this paper we first try to give a detailed overview of the research attempts made till now with respect to the parallelization of the various variants of the simplex method, mainly distinguishing between the standard and the revised form. Next, we describe in more detail three of the most recent and important related attempts: (a) one presenting the efficient parallelization of the dual revised simplex method [6,7] which is considered as the most efficient and preferred variant of the simplex method nowadays, (b) one demonstrating the capability of efficiently solving large-scale stochastic LP problems with the revised simplex method, and gaining particularly high speedups (over the Clp serial solver) when implemented on a modern high-performance supercomputer [8], and (c) one presenting a notably efficient, highly scalable with almost linear speed-up implementation of the standard simplex method over a modern hybrid hardware architecture with high-speed inteconnection network and different alternatives in the software platforms used (MPI and MPI-3 Shared Memory vs. MPI and OpenMP) [16–18]. Finally, we explore separately the corresponding approaches contributed to date with the use of the CPU-GPU model trying to exploit the massive parallelism capabilities offered by the modern graphic processing units; which is one of the hottest topics in almost all the research fields of parallel computing nowadays.

Furthermore, our work can be seen as a thorough update of the survey given in [2], which is the most recent complete survey in simplex parallelization. We first summarize the most valuable works done till the end of 2010's (an interval which is covered in more details in [2]), and then we focus on the significant advances made during the last five years when the emerging technologies in hardware architectures (hybrid supercomputers, modern multicore architectures, hpc clusters, GPU computing etc.) have offered the potential for even greater achievements in response times and speedup performance. Especially, the rapid evolution of multicore technology has pushed the research in designing suitable parallelization schemes that can effectively exploit the increased computation capability of these modern hardware architectures. The first truly parallel commercial simplex solvers over multicore desktop architectures have also appeared recently (e.g. [42]), achieving significant improvements against purely sequential solultions for several kinds of LPs.

The rest of the paper is organized as follows. In Sect. 2 the necessary background with regard to the simplex method and its variants is given. In Sect. 3 the detailed overview of the research work made till now on the parallelization of the simplex method is presented. In Sects. 4, 5 and 6 the main achievements with respect to the three selected recent research attempts described in the previous paragraph, are stated respectively. In Sect. 7 the most recent approaches based on the CPU-GPU model are briefly presented, whereas Sect. 8 concludes the paper.

2 Background

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

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

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

Step 1: *Choice of the entering variable*: Find the winning column i.e., the one having the larger negative coefficient of the objective function.

Step 2: *Choice of the leaving variable*: Find the winning row by appling the min ratio test to the elements of the winning column and choose the row number with the min ratio.

Table 1. Standard simplex method with full tableau representation

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

Step 3: *Pivoting*: Construct the next tableau by performing pivoting in the previous tableau rows based on the new pivot row found in step 2.
Step 4: Repeat the above steps until the best solution is found or the problem gets unbounded.

2.1 The Primal Revised Simplex Method

The revised simplex method performs the same steps as the tableau method but does not keep the tableau as an aid. Rather, whenever the algorithm requires a number from the tableau it is computed from one of several matrix equations, often involving the inverse of the basis. The data for the algorithm are the matrices A, c and b defining the original problem, the number of variables and constraints, n and m, and a record of the current basic and nonbasic variables. The basis matrix B is a square matrix composed of the columns from A corresponding to the m basic variables, whereas the columns of A corresponding to the $n-m$ nonbasic variables form the matrix N. The computational components of the primal revised simplex method are presented in Fig. 1 [2].

```
CHUZC: Scan $\hat{c}_N$ for a good candidate $q$ to enter the basis.
FTRAN: Form the pivotal column $\hat{a}_q = B^{-1}a_q$, where $a_q$ is column $q$ of $A$.
CHUZR: Scan the ratios $\hat{b}_i/\hat{a}_{iq}$ for the row $p$ to leave the basis.
           Update $\hat{b} = \hat{b} - \alpha\hat{a}_q$, where $\alpha = \hat{b}_p/\hat{a}_{pq}$.
BTRAN: Form $\pi_p = B^{-T}e_p$.
PRICE: Form the pivotal row $\hat{a}_p = N^T\pi_p$.
           Update $\hat{c}_N = \hat{c}_N - \beta\hat{a}_p$, where $\beta = \hat{c}_q/\hat{a}_{pq}$.
If {growth in representation of $B$} then
    INVERT: Form a new representation of $B^{-1}$.
else
    UPDATE: Update the representation of $B^{-1}$ due to the basis change.
end if
```

Fig. 1. Operations in an iteration of the primal simplex method

At the beginning of an iteration, it is assumed that the vector of reduced costs $\hat{c}_N$ and the vector $\hat{b}$ of values of the basic variables are known, that $\hat{b}$ is feasible (nonnegative), and that a representation of B^{-1} is available. The first operation is CHUZC (choose column), which scans the (weighted) reduced costs to determine a good candidate q to enter the basis. The pivotal column $\hat{a}_q$ is formed by using the representation of B^{-1} in an operation referred to as FTRAN (forward transformation). The CHUZR (choose row) operation determines the variable to leave the basis, with p being used to denote the index of the row in which the leaving variable occurred, referred to as the pivotal row. As a result, a basis change is said to have occurred and the vector $\hat{b}$ is then updated adequately. Before the next iteration can be performed, one must update also the reduced costs (BTRAN and PRICE) and obtain a representation of the new matrix B^{-1} (UPDATE). Periodically, it is either more efficient or necessary for numerical stability to find a new representation of B^{-1} using the INVERT operation.

2.2 The Dual Revised Simplex Method

While the primal simplex has been historically more important, it is now widely accepted that the dual variant (the dual simplex method) generally has superior performance. Dual simplex is often the default algorithm in commercial solvers, and it is also used inside branch-and-bound algorithms. Given an initial partition and corresponding values for the basic and nonbasic primal and dual variables, the dual simplex method aims to find an optimal solution of by maintaining dual feasibility and seeking primal feasibility. Thus optimality is achieved when the basic variables $\hat{b}$ are non-negative. The computational components of the dual revised simplex method are illustrated in Fig. 2 [8], where the same data are assumed to be known at the beginning of an iteration. The first operation is CHUZR which scans the (weighted) basic variables to determine a good candidate to leave the basis, with p being used to denote the index of the row in which the leaving variable occurs. The pivotal row $\hat{a}_p^T$ is then formed via BTRAN and PRICE operations. The CHUZC operation determines the variable q to enter the basis. In order to update the vector $\hat{b}$, it is necessary to form the pivotal column $\hat{a}_q$ with an FTRAN operation.

3 Overview of Simplex Parallelization

The approaches contributed till now to the literature with regard to the parallelization of the simplex method can be naturally classified according to the variant of the simplex method that is considered and the extent to which sparsity is exploited. It should be noticed that parallel implementations of the simplex variants that are more efficient as a mean of solving large sparse LP problems are less successful in terms of speed-up. Conversely, the simplex variants that are generally less efficient achieve the best speed-up. A few of the parallel schemes discussed below offered good speed-up relative to efficient coeval serial solvers. However, some parallel techniques are only now seen as being inefficient in the

```
CHUZR: Scan $\hat{b}$ for the row $p$ of a good candidate to leave the basis.
BTRAN: Form $\pi_p = B^{-T} e_p$.
PRICE: Form the pivotal row $\hat{a}_p^T = \pi_p^T N$.
CHUZC: Scan the ratios $\hat{c}_j / \hat{a}_{pj}$ for the row $q$ to enter the basis.
                  Update $\hat{c}_N = \hat{c}_N - \beta \hat{a}_p$, where $\beta = \hat{c}_q / \hat{a}_{pq}$.
FTRAN: Form the pivotal column $\hat{a}_q = B^{-1} a_q$, where $a_q$ is column $q$ of $A$.
                  Update $\hat{b} = \hat{b} - \alpha \hat{a}_q$, where $\alpha = \hat{b}_p / \hat{a}_{pq}$.
If {growth in representation of $B$} then
        INVERT: Form a new representation of $B^{-1}$.
else
        UPDATE: Update the representation of $B^{-1}$ due to the basis change.
end if
```

Fig. 2. Operations in an iteration of the dual simplex method

light of serial revised simplex techniques that either were not sufficiently known at the time or were developed subsequently. In the following two paragraphs the reader may find a brief analysis of the most representative parallelization efforts of the standard and revised simplex methods, whereas Tables 2 and 3 summarize the most significant of these efforts.

3.1 Parallelizing the Standard Simplex Method

There are many parallel implementations of the dense standard simplex method as well as of the revised simplex method with a dense explicit inverse. The simple data structures involved and the potential for linear speed-ups make them attractive implementation exercises either on shared memory machines or over distributed memory parallel environments. However, for solving general large sparse LP problems, the serial inefficiency of these implementations is such that only with a massive number of parallel processes they could conceivably compete with a good sparsity-exploiting serial implementation of the revised simplex method.

Some of the most known early works on parallelizing the standard simplex method can be found in [13,19–22]. Most of them focus on the possible data distribution and communication schemes, with implementations limited to small numbers of processes on distributed memory machines. The work of Stunkel [13] should be considered the most successful among them. He implemented both the dense standard simplex method and the revised simplex method with a dense inverse on a 16-processor Intel hypercube, achieving a speed-up of between 8 and 12 for small problem instances from the Netlib test set.

A few years later, two other valuable approaches were presented [23,24] with similar achievements. Cvetanovic et al. [23] report a speed-up of 12 when solving two small problem instances using the standard simplex method, a result that is notable for being achieved on a 16-processor shared memory machine. Luo and Reijns [24] obtained also satisfactory speedups of more than 12 on 16 transputers when using the revised simplex method with a dense inverse to solve modest Netlib test linear problems.

In the following years until 2000s, the first attempts over massively parallel computers appeared, with Eckstein et al. [12] and Thomadakis and Liu [11] contributing the most known and competent relevant works and implementations. Eckstein et al. [12] parallelized the standard simplex method and the revised simplex method with a dense inverse on the massively parallel Connection Machine CM-2 and CM-5, incorporating the steepest edge pricing strategy directly within their standard simplex implementation. As a consequence of using steepest edge weights and the expand procedure, this implementation is notable for its numerical stability, an issue that has rarely been considered in parallel implementations of the simplex method. Further details can be found in [25]. When solving a range of larger Netlib problems and very dense machine learning problems, speedups of between 1.6 and 1.8 were achieved when doubling the number of processors. They also presented results which indicated that the performance of their implementation was generally superior to MINOS (a well-known serial simplex solver), particularly for the denser problems. Thomadakis and Liu [11] also used steepest edge in their implementation of the standard simplex method on MasPar MP-1 and MP-2 machines. Solving a range of large, apparently randomly-generated problems, they achieved a speed-up of up to three orders of magnitude on the 128×128 processor MP-2.

During the next many years (till now) only a few significant new attempts and implementations were made with regard to the parallelization of the standard simplex method. Among them, a quite valuable theoretical work on parallel implementations of the standard simplex method with steepest edge, and its practical implementation on modest numbers of processors has been presented by Yarmish in [3,40]. The work in [3] has also led to high parallel efficiency and corresponding scalability for large-scale problems, whereas it has also been compared to MINOS, and it has been shown to be highly competitive even for very low density problems. Reports of small-scale parallel implementations also continue to appear. Relatively recently, Badr et al. [4] presented results for an implementation on eight processors, achieving a speed-up of five when solving small random dense LP problems.

A substantial difference between the two above attempts ([3] and [4]) was the way the simplex tableau is distributed among the processors. 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 which is the most popular and widely used for practical problems, were by Yarmish et al. [3] as well as by Qin et al. [5]. On the other hand, the work of Badr et al. [4] referred above, followed the row distribution scheme and presented a well-designed and relatively efficient implementation on eight loosely coupled processors. Furthermore, a comprehensive

Table 2. Standard simplex - representative parallelization efforts

Authors	Hardware platform	Speedup
Stunkel [13] (1988)	16-processor Intel Hypercube	Between 8 and 12 for small Netlib problems
Eckstein et al. [12] (1995)	CM-2 and CM-5 with thousands of processors	Between 1.6 and 1.8 when doubling the number of processors - superior to MINOS
Thomadakis and Liu [11] (1996)	MP-1 and MP-2 (128×128 processor MP-2)	Up to 1000 on large random LPs
Yarmish et al. [3] (2009)	7 workstations - Fast Ethernet	On iteration speed: up to $\sim$ 7 for random high-aspect ratio LPs
Mamalis et al. [16–18] (2011–14)	8-nodes Myrinet-connected Intel Xeon cluster - 4×4 quad-core Intel with gigabit ethernet	On iteration speed: up to $\sim$ 8 for random and Netlib LPs with high-aspect ratio - 11.5 to 15.2 for large LPs of all kinds

study and comparison of both the above data distribution schemes, as well as the corresponding implementations over high-performance cluster (distributed memory or hybrid) environments achieving particularly high speedup values are given in the recent works of Mamalis et al. [16–18]. The key points and achievements of the latter are presented in more details in Sect. 6.

3.2 Parallelizing the Revised Simplex Method

Undoubtedly, the real challenge in developing a parallel simplex implementation of general practical value is to exploit parallelism when using a variant of the revised method with sparse matrix algebra techniques. Only then the resulting solver could be competitive to a good serial implementation when solving general large sparse LP problems using a realistic number of processors. Efficient serial simplex solvers are based on the revised simplex method with a factored inverse since the practical LP problems whose solution poses a computational challenge are large and sparse. For such problems, the superiority of the revised simplex method over the known serial standard simplex schemes is clear and obvious. It follows that the only scope for developing a really worthwhile parallel simplex solver is to identify how the revised simplex method with a factored inverse may be parallelized. The natural data parallelism of the PRICE operation has been exploited by most authors. Several researchers have also considered the data parallelism in other computational components, whereas others have studied the extent to which task parallelism can be exploited by overlapping operations that are then executed either in serial or, in the case of PRICE, by exploiting data parallelism.

The first worth telling research attempts were contributed by Pfefferkorn and Tomlin [26] and Helgason et al. [27]. In [26] the authors were the first to discuss how parallelism could be exploited in each computational component of the revised simplex method. On the other hand, in [27] the authors trying to contribute new ideas, beyond the classics, discussed the scope for parallelizing FTRAN and BTRAN operations based on the relatively unknown quadrant interlocking factorization technique. Further, McKinnon and Plab [28] considered how data parallelism could be exploited in FTRAN operation for both dense and sparse right-hand sides. They also investigated how the Markowitz criterion could be modified in order to increase the potential for exploiting data parallelism in subsequent FTRAN operations.

The first attempt to exploit task parallelism in the revised simplex method was reported by Ho and Sundarraj [29]. In addition to the natural data parallelism of the PRICE operation, Ho and Sundarraj identified that the INVERT operation can be overlapped with simplex iterations. The performance of Ho and Sundarrajs implementation, on an Intel iPSC/2 and Sequent Balance 8000, was quite promising, however limited in accordance with Amdahls law (since only some parts of the whole algorithm were parallelized). On a set of small Netlib and proprietary problems, they report an average saving of 33 % over the corresponding serial solution time.

The next ambitious implementation that attempted to exploit full data parallelism over the revised simplex method was that of Shu [10]. It was based on a parallel triangularization phase for the INVERT operation, a distributed sparse LU decomposition of the basis matrix for parallel FTRAN and BTRAN operations, as well as a typical parallelization of the PRICE operation. However, no significant speed-up was achieved in the corresponding experimental tests.

In the following years till now, the most known and notable corresponding atttempts were contributed by Hall and McKinnon [9,41]. Their first parallel revised simplex scheme was ASYNPLEX [9]. This corresponds to a variant of the revised simplex method in which reduced costs are computed directly. ASYNPLEX was implemented on a Cray T3D and it was tested using four modest but representative Netlib test problems. Using between 8 and 12 processors to perform simplex iterations, the iteration speed was increased by a factor of about 5 in all cases. However, the increase in the number of iterations required to solve the problem led to a speed-up in solution time ranging from 2.4 to 4.8. Hall and McKinnons second parallel scheme was PARSMI [41]. This was developed in an attempt to address the deficiencies of ASYNPLEX. In order to make realistic comparisons with good serial solvers, PARSMI updates the reduced costs and uses Devex pricing. As the authors state, the implementation of PARSMI was a highly complex programming task, magnified by the fact that communication times are not deterministic and the order of arrival of messages determined the operation of the program. Programming difficulties, together with numerical instability meant that the implementation was never reliable. In the very limited results that were obtained on modest Netlib problems, using eight processors, the speed-up in iteration speed was between 2.2 and 2.4 (leading to a speedup in solution time between 1.7 and 1.9).

As it can be seen, all attempts to exploit parallelism in the revised simplex method have focused on the primal simplex method. The dual simplex method, which is much more efficient for many LP problems has not been addressed a lot in terms of parallelization. However, as it is explained in more detail in [2,30], there is one important distinction between the primal and dual simplex methods which has the potential to cause significant differences in parallel performance. In the primal simplex method the PRICE operation is restricted to just a subset of the non-basic variables, whereas, in the dual simplex method, especially for problems with very large column/row ratios the completely dominant cost is due to the PRICE operation. Hence its parallelization can be expected to yield a significant improvement in performance over that of the efficient serial dual simplex solvers. Bixby and Martin [30] were the first to investigate the scope for parallelism in the dual revised simplex method and chose to parallelize only those operations whose cost is related to the number of columns in the problem, that is the PRICE operation, the dual ratio test and the update of the dual variables. They implemented the dual simplex method on several architectures; however using the full Netlib set, no significant gain in performance was obtained. The latter was a normal behavior since few problems in the Netlib set have significantly more columns than rows. Focusing on this kind of problems and using up to four processors on an IBM SP2, a quite satisfactory speedup was observed, ranging from 1 to 3.

Till recently, no other valuable attempts have been made to parallelize the dual revised simplex method, thus making the one that immediately follows (Huangfu and Hall [6,7], Sect. 4) a distinguished one. Another significant parallel implementation using the revised simplex method that deserves to be presented separately (see Sect. 5) has also been recently achieved by Lubin et al. [8], demonstrating the capability of solving large scale stochastic LP problems in acceptable times within the computing environment of a supercomputer.

4 Recent Advances on the Parallelization of the Dual Revised Simplex Method

As mentioned in the previous sections, for sparse LP problems the revised (either primal or dual) simplex method is generally preferred against the simplex method

Table 3. Revised simplex - representative parallelization efforts

Authors	Hardware platform	Speedup
Ho and Sundarraj [29] (1994)	Intel iPSC/2 and Sequent Balance 8000 (primal simplex)	1.5 on average for medium sized Netlib and other LPs
Hall and McKinnon [9] (1998)	Cray T3D (8 to 12 processors) (primal simplex)	Between 2.5 and 4.8 on medium sized Netlib LPs ($\sim$ 5 on iteration speed)
Bixby and Martin [30] (2000)	Different platforms - 4 pr. on a IBM SP2 (dual simplex)	Between 1 and 3 on high-aspect ratio Netlib LPs
Huangfu and Hall [6,7] (2012–14)	8 cores of a 16xIntel Xeon E5620 (dual suboptimization)	$>$ 2 on average for test LPs of all kinds (max of 3.5), 1.5 on average over regular dual simplex, comparable to Cplex & Clp
Lubin et al. [8] (2013)	a 320x8-node cluster with InfiniBand and a Blue Gene/P supercomputer with 40960 nodes (both primal and dual)	On iteration speed: $\sim$ 100 over Clp when using 16 nodes (128 cores) for large-scale two-stage stochastic LP problems

in its standard form since it permits the sparsity of the problem to be exploited. This is achieved using techniques for decomposing sparse matrices and solving hyper-sparse linear systems. Also important for the dual revised simplex method are advanced algorithmic variants introduced in the 1990s, particularly dual steepest-edge (DSE) pricing and the bound flipping ratio test (BFRT). These led to significant performance improvements and resulted in the dual simplex algorithm being preferred. However, for many years now, although the dual revised method is regarded as the most preferable and efficient, no practical parallel implementations appeared in this context. Towards the above direction, the authors in [6,7] introduce two novel parallel dual simplex solvers for general large scale sparse linear programming problems, over standard desktop architectures. The first approach extends a relatively unknown pivoting strategy called suboptimization and exploits parallelism across multiple iterations. The second approach exploits purely single iteration parallelism. Computational results show that the performance of the first approach is comparable with the world-leading commercial simplex solvers, and that the second approach complements the first one, in achieving speedup when it results in slowdown.

Moreover, in the past, parallel implementations generally used dedicated high performance computers to achieve the best performance. Now that every desktop computer is a multi-core machine, any speedup is desirable in terms of solution time reduction for daily use. In this direction, the authors have chosen to use relatively standard architecture to perform computational experiments with very good results. It should certainly be considered as one of the most valuable practical attempts during the last decade with respect to the general-purpose parallelization of the revised simplex method.[1]

4.1 Design and Implementation (Key Issues)

As reported in Sect. 2, the dual simplex algorithm solves an LP problem iteratively by seeking primal feasibility while maintaining dual feasibility. Considering the operations within each iteration (as given in Fig. 2), there is immediate scope for data parallelization within CHUZR, PRICE, CHUZC and most of the update operations since they require independent operations for each (nonzero) component of a vector. Additionally, the scope for task parallelism by overlapping the execution of the sub-operations within FTRAN was considered by Bixby and Martin but rejected as being disadvantageous computationally. On the other hand, Huangfu and Hall [6,7] have based their implementation on the technique of suboptimization, which is is one of the oldest variants of the revised simplex method and consists of a major-minor iteration scheme [7]. Within the primal revised simplex method, suboptimization performs minor iterations of the standard primal simplex method using small subsets of columns from the reduced

[1] Note also that the techniques applied in the proposed approaches, have been the basis for the integration of FICO Xpress parallel solver [42], which was the first commercial parallel simplex solver and has been regarded quite faster than the pre-existing ones in various kinds of large-scale LPs.

coefficient matrix $B^{-1}A$. Suboptimization for the dual simplex method was first set out by Rosander [31]. It performs minor operations of the standard dual simplex method, applied to small subsets of rows from $B^{-1}A$. Originally, suboptimization was proposed mainly as a pivoting scheme for achieving better pivot choices and advantageous data affinity. In modern revised simplex implementations, the DSE and BFRT are together regarded as the best pivotal rules and the idea of suboptimization has been naturally forgotten. However, in terms of parallelization, suboptimization is attractive because it certainly provides more scope for parallelization.

In the design and implementation of the first proposed approach, the authors extend the suboptimization scheme of [31], incorporating (serial) algorithmic techniques and exploiting parallelism across multiple iterations. The main suboptimization steps (the exact mathematical formulation can be found in [7]) in each iteration include (a) the major optimality test, (b) the minor initialization step, (c) the minor iterations step consisting of of three basic sub-operations i.e., the minor optimality test, the minor ratio test and the minor update step, and (d) the major update step. Based on the above decomposition, the authors apply extensively data parallelism (mainly with respect to the various vector-based operations met) in almost all steps. Specifically, vector-based operations are met (and can be efficiently parallelized) on both the major optimality test and the major update step, whereas the minor initialization step offers a good opportunity for task parallelization. With regard to the composite minor iterations step, data parallelism is applied on the minor ratio test with respect to the PRICE operation and the first part of CHUZC operations, as well as on the minor update step (vector-based update operations).

In their second parallel implementation the authors introduce a relative simple approach to exploiting parallelism within a single iteration of the dual revised simplex method. The relevant approach is a significant development of the work of Bixby and Martin [30] who parallelized only the PRICE, CHUZC and update-dual operations, having rejected the task parallelism of FTRAN sub-operations as being computationally disadvantageous. The mixed parallelization scheme of this implementation can be found in more details in [7].

4.2 Experimental Results

The experimental performance of the two parallelization approaches described above has been tested using a reference set consisting of 30 problems. Most of these LP problems are taken from a comprehensive list of various representative LP problems maintained by Mittelmann [32]. The problems in this reference set reflect the wide spread of LP properties and revised simplex characteristics, including the dimension of the linear systems (number of rows and columns), the density of the coefficient matrix (average number of non-zeros per column), and the extent to which they exhibit hyper-sparsity.

The performance of both approaches has been measured using experiments performed on a workstation with 16 (Intel Xeon E5620, 2.4 GHz) cores, using 8 of the cores for the parallel calculations. With respect to the results obtained for the first approach, the main observations can be summarized as follows:

– For most of the problems included in the reference set, the speedup compared to its sequential version is more than 2 (with a geometric mean of 2.23). The best speedup obtained was equal to 3.50.
– When compared to the regular dual simplex method, the sequential version of the proposed implementation is generally less efficient (about 30 % slower). As a consequence, the overall (true) speedup is somewhat restricted, resulting in a mean speedup of about 1.5.
– It is also worth mentioning that the instances of better speedup (greater than the average) correspond largely to the sparse LP problems.

The above results are satisfactory since they refer to a general-case reference set with all kinds of problems. Furthermore, the worst performances are associated with dense LP problems, whereas the achieved performance when solving hyper-sparse LP problems is moderate but relatively stable. As it is also clearly indicated in the original paper [7], the performance of the proposed approach is comparable with the dual simplex implementation of CPLEX, a world-leading commercial dual revised simplex solver, and clearly superior to that of CLP, the world's leading open-source solver. On the other hand, the results obtained for the second approach can be summarized as follows:

– The overall performance is quite worse than that of the first approach. An average speedup of 1.13 has been achieved, with a maximum value of 2.05.
– The worst cases are associated with the hyper-sparse LP problems where, in most cases, it results in a slowdown.
– However, when applied to dense LP problems, the performance is moderate and relative stable. This is especially so for those instances where the first approach exhibits a slowdown.

In summary, the second approach is a straightforward parallelization approach which exploits purely single iteration parallelism and achieves relatively poor speedup for general LP problems. However, it is frequently complementary to the first approach in achieving speedup when the latter results in slowdown. Overall, the authors in this paper have introduced the design and development of a novel parallel dual revised simplex method implementation framework, which has been measured to provide an average speedup of 1.5 for general large scale sparse linear programming problems, over standard desktop architectures. Although this is not particularly high, the resulting performance of the first approach is comparable to the dual simplex implementation of CPLEX.

5 Parallel Distributed-Memory Simplex for Large-Scale Stochastic LP Problems

The parallel implementation of the revised simplex method for several special cases of linear programming problems may often be implemented quite more efficiently than in the general case due to the special structure of these problems [2]. Seeking towards that direction (special-purpose parallel revised simplex solvers) one of the most impressive works made recently was the one presented in [8]. In this work the authors present a parallelization of the revised simplex method for large extensive forms of two-stage stochastic linear programming (LP) problems, which can be considered one of the most interesting special cases of linear programming problems (due to their very large size as well as their significance in real life). These problems have been considered too large to solve with the simplex method; instead, decomposition approaches based on Benders decomposition or, more recently, interior point methods are generally used. However, these approaches do not provide optimal basic solutions. The present approach exploits the dual block-angular structure of these problems inside the linear algebra of the revised simplex method in a manner suitable for high-performance distributed-memory clusters or supercomputers. While this paper focuses on stochastic LPs, the work is applicable to all problems with a dual block-angular structure. The whole implementation is competitive in serial with highly efficient simplex solvers and achieves significant relative speed-ups when executed in parallel. Additionally, very large problems with hundreds of millions of variables have been successfully solved to optimality.

Moreover, as the authors claim, this is the largest-scale parallel sparsity-exploiting revised simplex implementation that has been developed to date and the first truly distributed solver. It is built on novel analysis of the linear algebra for dual block-angular LP problems when solved by using the revised simplex method and a novel parallel scheme for applying product-form updates.[2]

5.1 Design and Implementation (Key Issues)

More concretely, the proposed parallelization approach is based on the revised simplex method for linear programming (LP) problems with a special structure which is known as dual block angular or block angular with linking columns [8]. This structure commonly arises in stochastic optimization as the extensive form or deterministic equivalent of two-stage stochastic linear programs [33]. Linear programs with block-angular structure, both primal and dual, occur in a wide array of applications, and this structure can also be identified within general LPs. They are typically met in the form given below:

$$\begin{array}{rllllll}
\text{minimize} & c_0^T x_0 & + \; c_1^T x_1 & + \; c_2^T x_2 & + \; \ldots & + \; c_N^T x_N & \\
\text{subject to} & A x_0 & & & & & = b_0, \\
& T_1 x_0 & + W_1 x_1 & & & & = b_1, \\
& T_2 x_0 & & + W_2 x_2 & & & = b_2, \\
& \vdots & & & \ddots & & \vdots \\
& T_N x_0 & & & & + W_N x_N & = b_N, \\
& \multicolumn{6}{l}{x_0 \geq 0, \;\; x_1 \geq 0, \;\; x_2 \geq 0, \;\; \ldots, x_N \geq 0}
\end{array}$$

[2] Note also that this paper has received recently the COAP (Computational Optimization and Applications) journal Best Paper Award for year 2013.

Borrowing the terminology from stochastic optimization, the vector x_0 is supposed to contain the first-stage variables and the vectors x_1, ... , x_N the second-stage variables. On the other hand, the matrices W_1, W_2, ..., W_N contain the coefficients of the second-stage constraints, and the matrices T_1, T_2, ..., T_N those of the linking constraints.

With regard to the distribution of data across the parallel processes, the authors have adopted a carefully designed allocation scheme as follows: Given a set of P MPI processes and $N \geq P$ scenarios or second-stage blocks, on the initialization, each second-stage block is assigned to a single MPI process. All data, iterates, and computations relating to the first stage are duplicated in each process. The second-stage data (i.e., W_i, T_i, c_i, and b_i), iterates, and computations are only stored in and performed by their assigned process. If a scenario is not assigned to a process, this process stores no data pertaining to the scenario, not even the basic/nonbasic states of its variables.

Then, the authors proceed with the first (and probably the most important) step of their solution, i.e., the factorizing of the basis matrix. Accordingly, one has to form an invertible representation of the basis matrix B. This is performed in efficient sparsity-exploiting codes by forming a sparse LU factorization of the basis matrix. These factors are formed in parallel via the following steps:

– Perform partial sparse Gaussian elimination on each second-stage block.
– Collect and duplicate the necessary terms across processes.
– In each process, form and factor the first-stage block.

The first step may be performed in parallel for each second-stage block (since they have been evenly distributed to the multiple processes). In the second step the results are collected and duplicated in each parallel process by using MPI_Allgather() function, and in the third step, each process factors its local copy of the first-stage block. If then an LU factorization of the first-stage block is performed, this entire procedure could be viewed as forming an LU factorization of B through a restricted sequence of pivot choices. The remaining linear system is now trivial to solve towards the final solution. Appropriate parallelization is being applied in all the following necessary steps (solving linear systems. matrix-vector product, updating the inverse etc.) by the means of some of the well-known collective communication functions of MPI (MPI_Bcast(), MPI_Allgather(), MPI_Allreduce() etc.) [8].

5.2 Experimental Results

The proposed approach (PIPS-S) was evaluated experimentally with the use of two powerful distributed memory architectures available at Argonne National Laboratory (ANL):

– *Fusion* is a 320-node cluster with an InfiniBand QDR interconnect; each node has two 2.6 GHz Xeon processors (total 8 cores). Most nodes have 36 GB of RAM, while a small number of them offer 96 GB of RAM.

- *Intrepid* is a Blue Gene/P (BG/P) supercomputer with 40,960 nodes with a custom high-performance interconnect. Each BG/P node has a quad-core 850 MHz PowerPC processor with 2 GB of RAM.

Using suitable stochastic LP test problems, the authors present results of three different scales by varying the number of scenarios. Part of the measurements i.e. the ones for SSN and Storm problems which exhibit the best performance, are presented in Table 4. First, the authors consider instances that could be solved on a modern desktop from scratch, that is, from an all-slack starting basis. These large-scale instances (with 110 million total variables) serve both to compare the serial efficiency of PIPS-S with that of a modern simplex code and to investigate the potential for parallel speedup on problems of this size. The main observations can be summarized as follows:

- Clp is faster than PIPS-S in serial on all instances; however, the total number of iterations performed by PIPS-S is consistent with the number of iterations performed by Clp.
- Significant parallel speedups are observed in all cases; as an example PIPS-S is 5 and 8 times faster than Clp for SSN and Storm respectively when using four nodes (32 cores).
- The speedups obtained on some other instances are smaller, possibly because of the smaller number of scenarios and the larger dimensions of the first stage.

Next, some quite larger instances with 20–40 million total variables are considered. The high memory nodes of the Fusion cluster with 96 GB of RAM were required for these tests. Given the long times to solution for the smaller instances solved in the previous section, it is impractical to solve these larger instances from scratch. Instead, the authors proceed using advanced or near-optimal starting bases in two different contexts. The corresponding results can be summarized as follows:

- Clp remains faster in serial than PIPS-S on these instances, although by a smaller factor than before.
- The parallel scalability of PIPS-S is almost ideal (>90 % parallel efficiency) up to 4 nodes (32 cores) and continues to scale well up to 16 nodes (128 cores). Scaling from 16 nodes to 32 nodes is poor.
- On 16 nodes, the iteration speed of PIPS-S is about 100 times better than that of Clp for Storm and 70 times better than that of Clp for SSN.

Finally, the authors report on the solution of a very large instance with 8,192 scenarios. This instance has 463,113,276 variables and 486,899,712 constraints. An advanced starting basis was generated from 4,096 scenarios, not included in the execution time. This problem requires approximately 1 TB of RAM to solve, requiring a minimum of 512 Blue Gene/P nodes; however, results are only available for runs with 1,024 nodes or more because of execution time limits. The derived solution time was around 6 h on 1024 nodes (2048 cores), 5 h on 2048 nodes (4096 cores), and 4.5 h on 4096 nodes (8192 cores). While scaling performance is poor on these large numbers of nodes, this test demonstrates the

Table 4. Part of the experimental results of [8]

Test problem	Solver	Nodes	Cores	Time (sec.)	Iter/sec.
Solves from scratch using dual simplex					
Storm	Clp	1	1	133,047	50.4
	PIPS-S	1	1	385,825	16.5
		1	8	52,948	119.8
		4	32	15,667	405.2
SSN	Clp	1	1	12,619	93.1
	PIPS-S	1	1	58,425	17.5
		1	8	7,788	135.5
		4	32	1,931	542.1
Solves from advanced starting basis using primal simplex					
Storm	Clp	1	1	7,537	2.2
	PIPS-S	1	1	7,184	1.3
		2	16	137	47.6
		16	128	35.5	216.6
		32	256	25.2	260.4
SSN	Clp	1	1	50,737	2.0
	PIPS-S	1	1	427,648	0.8
		2	16	9,550	22.9
		16	128	1,481	143.3
		32	256	1,117	180.0

capability of PIPS-S to solve instances considered far too large to solve today with commercial solvers.

6 Revisiting the Parallelization of Standard Full Tableau Simplex Method

The fact that a parallel simplex solver based on the standard (full tableau) representation may be practical only for dense LP problems and may not easily be competitive to the fast serial revised simplex solvers of nowadays unless it uses expensive parallel computing resources, has naturally led to less corresponding efforts in the literature during the last years. However, any new intuitive corresponding study and relevant implementation would still be worthwhile as far as it achieves either particularly high speedups in absolute values (which also usually means competitive solution times to the ones of the serial solvers) or particularly high efficiency values combined with correspondingly high scalability. The latter has the potential to lead to even higher speedups and competitive solution times when executing to architectures with larger number of processors/cores.

One of the most recent and worth mentioning relevant attempts combining sufficient theoretical study and results with a relevant particularly efficient parallel implementation of the standard simplex method, was the one presented by Yarmish et al. [3]. The corresponding implementation has led to very satisfactory speedup values, whereas it has also been compared to MINOS (a well-known serial revised simplex solver), and it has been shown to be highly competitive, even for very low density problems. Moreover, together with the work of Badr et al. [4], they are the most recent works that put on the table the significant influence of the number of columns and rows of an LP problem when a distributed memory architecture is used. Being inspired by the motivation and the results of the above two research attempts, as well as by the means of the current technology (either in terms of high-speed network connections or in terms of powerful hybrid hardware architectures and corresponding hybrid software solutions), in [16–18] the authors present two very promising relevant approaches with regard to simplex parallelization in its standard (full tableau) form.

First, in [16,17] the authors present a highly scalable parallel implementation framework designed for distributed memory (message passing) environments. Two basic data distribution schemes have been implemented, a column-based one and a row-based distribution scheme, in order to measure the influence of each distribution method over LP problems with different aspect ratio and compare their performance with other works in the literature. They have experimentally evaluated their implementations over a considerably powerful parallel environment; a linux-cluster of 8 (16 threads) Xeon processors connected via a dedicated (low latency) Myrinet network interface. They have tested and compared the two implementation schemes among each other, as well as to the corresponding implementations of [3,4] referred above. Both schemes lead to particularly high speed-up and efficiency values for typical test LPs, that are considerably better in all cases than the ones achieved by the corresponding implementations of [3,4].

Next, in [18] the authors focus on the modern hybrid hardware architectures (distributed memory/cluster environments with multicore nodes) of nowadays, and they involve several different software alternatives on the parallelization of the standard simplex method with the column-based data distribution scheme. Specifically, they present relevant implementations combining pure MPI, OpenMP and MPI 3.0 Shared Memory support. They compare their approaches among each other for variable number of nodes/cores and problem size, as well as to the approach presented in [3]. The experiments have been performed over a hybrid parallel environment which consists of up to 4 quad-core processors (making a total of 16 cores) connected via Gigabit ethernet interface. All the evaluated parallelization schemes have led to particularly high speed-up and efficiency values, whereas the corresponding values for the hybrid MPI+OpenMP based scheme (which is proved to be the most efficient) are considerably better in all cases than the ones achieved in the work of [3].

6.1 Design and Implementation (Key Issues)

As mentioned above the most efficient implementations presented in [16–18] have followed the column-based distribution for spreading the initial tableau to all the processors. This is a relatively straightforward parallelization scheme within the standard simplex method which involves dividing up the columns of the simplex table among all the processors and it is theoretically regarded as the most effective one in the general case. Following this scheme all the computation parts except step 2 of the basic (sequential) algorithm (presented in Sect. 2), are fully parallelized. Additionally, this form of parallelization looks as the most natural choice since in most practical problems the number of columns is larger than the number of rows. The basic steps of the algorithm are given below:

Step 0: The simplex table is shared among the processors by columns. Also, the right-hand constraints vector is broadcasted to all processors.
Step 1: Each processor searches in its local part and chooses the locally best candidate column the one with the larger negative coefficient in the objective function part (local contribution for the global determination of the entering variable).
Step 2: The local results are gathered in parallel and the winning processor i.e., 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 columns elements.
Step 4: The same (winning) processor then broadcasts the winning column as well as the winning rows id to all processors.
Step 5: Each processor performs (in parallel) on its own part (columns) of the table all the calculations required for the global rows pivoting, based on the pivot data received during step 4.
Step 6: The above steps are repeated until the best solution is found or the problem gets unbounded.

A relevant, row-based, distribution scheme has also been implemented and studied in comparison to the one stated above (see [17] for more details). Based on the above step by step decomposition three different parallelization schemes were designed and implemented as follows:

a. Pure MPI implementation.
The well-known MPI collective communication functions MPI_Scatter, MPI_Bcast and MPI_Reduce/Allreduce (with or without MAXLOC/MINLOC operators) were appropriately used for the efficient implementation of the data communication required by steps 0, 2 and 4 of the parallel algorithm.

b. Hybrid MPI+OpenMP implementation.
Appropriately built 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, the newly added min/max reduction operators of OpenMP API specification for C/C++ were used. Also, with regard to the parallelization of step 5, in order to achieve even distribution of computations to the working threads, collapse-based nested parallelism is used in combination with dynamic scheduling policy. Beyond the OpenMP-based parallelization inside each node, the well-known MPI collective communication functions were also used for the communication between the network connected nodes as in pure MPI implementation.

c. Hybrid MPI+MPI Shared Memory implementation.
The corresponding shared memory support functions of MPI 3.0 (mainly: MPI_Comm_split_type, MPI_Win_allocate_shared and MPI_Win_shared_query) as well as the syncronization primitives MPI_Win_fence, MPI_Win_lock/unlock and MPI_Accumulate were used for the efficient implementation of all the data communication (the initial and intermediate data sharing as well as the computation of minimum/maximum values) required by steps 0, 2, 3 and 4 of the parallel algorithm over the multiple cores of each node. The well-known MPI collective communication functions were used for the communication between the network connected nodes as in pure MPI implementation.

6.2 Experimental Results

First, in order to compare their approach to the one of Yarmish et al. [3] (which is also based on the column-based distribution scheme) the authors have run on their Myrinet-connected linux-cluster platform their basic implementation over the large size (1000x5000) linear problem presented there [3], with the same characteristics, and they have measured the execution time per iteration for 1, 2 up to 8 processors. This problem is a large-scale problem with many more columns than rows, so it is expected to have good speedup with the use of the column distribution scheme. Note also that the parallel platform used in the experiments of [3] consisted of 7 dedicated processors (the exact configuration is not mentioned) connected via Fast Ethernet network interface. The corresponding results (in terms of speedup and efficiency measures based on the execution time per iteration) for varying number of processors are presented in Table 5. Observing the results of Table 5, firstly it can easily be noticed that the execution times (in one or more processors) of the algorithm in [17] are much better than the ones of [3], which however was expected due to the fact that the test platform is quite more powerful than the platform of [3]. The most important, the achieved speedup values are also better than the ones achieved in [3]. Furthermore, observing the corresponding efficiency values in the last column someone can easily notice the high scalability (higher and smoother than in [3]) achieved. Note also that the achieved speedup remains very high (close to the maximum/speedup = 7.92, efficiency = 99.0 %) even for 8 processors.

Additionally, in other experimental measurements in the same platform (as presented in [17]), the authors compare the performance of their two different

Table 5. Comparing to the implementation of [3]

P	Yarmish et al. [3]			Mamalis et al. [17]		
#proc	Time/iter	$S_p = T_1/T_P$	$E_p = S_p/P$ (%)	Time/iter	$S_p = T_1/T_P$	$E_p = S_p/P$ (%)
1	0.61328	1.00	100.0	0.27344	1.00	100.0
2	0.31150	1.97	98.4	0.13713	1.99	99.7
3	0.21724	2.82	94.1	0.09225	2.96	98.8
4	0.15496	3.96	98.9	0.06877	3.98	99.5
5	0.13114	4.68	93.5	0.05592	4.89	97.8
6	0.10658	5.75	95.9	0.04636	5.90	98.3
7	0.09128	6.72	96.0	0.03958	6.91	98.7
8				0.03453	7.92	99.0

data distribution schemes (column-based vs. row-based) among each other (with the use of a suitable subset of NETLIB test linear problems of varying sizes), concluding to two basic remarks: (a) the column-based distribution scheme is clearly superior in most cases, however (b) there are several cases that one should choose the row-based distribution scheme instead mainly for small sized problems with almost equal number of rows and columns or greater number of rows. The high scalability of the column-based distribution scheme over sixteen processors (threads) is also demonstrated in relevant experiments for very large-scale problems. Furthermore, the authors notice 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 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.

Similarly, the authors in [17] have also run corresponding exeperiments on their hybrid hardware platform (4x4 quad-core processors connected with Gigabit Ethernet), in order to compare the performance of their two different hybrid parallelization schemes among each other (MPI+OpenMP vs. MPI+MPI 3.0 Shared Memory support), with use of another suitable subset of the NETLIB test linear problems of varying sizes. The corresponding results (speedup and efficiency values for 8 and 16 processors/cores - two quad-core processors and four quad-core processors, respectively) are presented in Table 6, and they can be summarized as follows:

- The achieved speed-up and efficiency values of the hybrid MPI+OpenMP implementation are better than the ones of the hybrid MPI+MPI 3.0 Shared Memory implementation, in all cases.
- However the achieved values for the MPI+MPI 3.0 Shared Memory implementation are also particularly high and competitive (up to 90% efficiency for medium-sized problems on eight cores).
- More concretely, for linear problems of small size the corresponding measurements are almost the same (slightly better for the MPI+OpenMP approach), whereas for problems of larger size the difference is quite clear.

Overall, one can say that the shared window allocation mechanism of MPI 3.0 offers a very good alternative (with almost equivalent results to the MPI+OpenMP approach) for shared memory parallelization when the shared data are of relatively small/medium scale, however it cannot scale up the same well (for large windows and large number of cores) due to internal protocol limitations and management costs, especially in applications where some kind of synchronization is required. Finally, in order to further validate the high efficiency and scalability of the hybrid MPI+OpenMP parallelization scheme, in more representative (closer to the real word) cases, the authors have also performed corresponding experiments for large and very large NETLIB problems. The corresponding measuremnets are shown for 2 up to 16 processors/cores in Table 7. One can easily observe the following:

– The efficiency values decrease with the increase of the number of processors. However, this decrease is quite slow, and both the speedup and efficiency values remain high (≥80%) even for 16 cores, in all cases.
– Particularly high efficiency values (almost linear speedup) are achieved for all the high aspect ratio problems (e.g. see the values for problems FIT2P, 80BAU3B and QAP15 where the efficiency even for 16 processors/cores is over 90% - a particularly high value for realistic problems).

7 GPU-Based Simplex Parallelization Efforts

The computational power provided by the massive parallelism of modern graphics processing units (GPUs) has moved increasingly into focus over the past few years. However, in the area of simplex parallelization for several reasons (similar to the ones discussed for the conventional CPU-only parallelization) there have

Table 6. Comparing the two hybrid schemes

Linear Problems	MPI+OpenMP				MPI+MPI 3.0 SM			
	2×4 cores		4×4 cores		2×4 cores		4×4 cores	
	S_p	E_p (%)	S_p	E_p (%)	S_p	E_p (%)	S_p	E_p (%)
SC50A (50×48)	4.89	61.1	6.50	40.6	4.85	60.6	6.40	40.0
SHARE2B (96×79)	5.59	69.8	8.24	51.5	5.49	68.6	8.00	50.0
SC105 (105×103)	5.81	72.6	8.78	54.9	5.69	71.1	8.50	53.1
BRANDY (220×249)	7.00	87.5	12.17	76.1	6.60	82.5	10.63	66.5
AGG (488×163)	6.76	84.5	12.11	75.7	6.38	79.8	11.27	70.4
AGG2 (516×302)	7.00	87.5	12.89	80.5	6.58	82.3	11.99	74.9
BANDM (305×472)	7.42	92.8	13.61	85.0	6.82	85.3	12.03	75.2
SCFXM3 (990×1371)	7.60	95.0	14.38	89.9	7.16	89.5	13.05	81.5

Table 7. Speed-up & efficiency for large problems

Linear Problems	2×1 cores		2×2 cores		2×4 cores		4×4 cores	
	S_p	E_p (%)	S_p	E_p (%)	S_p	E_p (%)	S_p	E_p (%)
FIT2P (3000×13525)	1.977	98.9	3.94	98.5	7.80	97.5	15.24	95.3
80BAU3B (2263×9799)	1.969	98.5	3.91	97.8	7.72	96.5	14.92	93.3
QAP15 (6330×22275)	1.963	98.2	3.89	97.3	7.62	95.3	14.47	90.5
MAROS-R7 (3136×9408)	1.957	97.9	3.87	96.8	7.54	94.3	14.12	88.3
QAP12 (3192×8856)	1.953	97.7	3.86	96.5	7.50	93.8	13.97	87.3
DFL001 (6071×12230)	1.945	97.3	3.85	96.3	7.50	93.8	14.04	87.8
GREENBEA (2392×5405)	1.949	97.5	3.84	96.0	7.40	92.5	13.59	84.9
STOCFOR3 (16675×15695)	1.925	96.3	3.79	94.8	7.23	90.4	12.80	80.0

not been noticed as many relevant attempts as one would expect. As a consequence, 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 LPs (sparse or dense, randomly generated or benchmark etc.). Certainly, some quite significant progress (and corresponding comparative results, satisfactory speedup values etc.) has been achieved at least for dense LP problems.

The quite strict model of parallelization, the limited development tools, and the limited processing element/core speed are some of the basic disadvantages comparing to the conventional model. The relatively slow memory transfer between CPU and GPU is also a significant drawback when a fully combined processing model is to be adopted. So, although the modern GPUs offer thousands of processing cores, the revised simplex method remains difficult to be efficiently parallelized and give satisfactory/competitive results compared to the existing serial solvers, whereas a GPU-based parallel version of the standard simplex method remains to be practical only for dense problems and with multiple computing resources (i.e. multiple GPUs). Furthermore, no corresponding GPU-accelerated implementation has been reported on a supercomputer. In the above context, we present the most recent and worth telling corresponding works in the next paragraphs.

Spampinato and Elster have proposed in [34] a parallel implementation of the revised Simplex method for LP on GPU with NVIDIA CUBLAS and NVIDIA LAPACK libraries. Tests were carried out on randomly generated LP problems of at most 2000 variables and 2000 constraints. The implementation showed a maximum speedup of 2.5 on a NVIDIA GTX 280 GPU as compared with sequential implementation on CPU with Intel Core2 Quad 2.83 GHz. Bieling, Peschlow and Martini have proposed in [35] another implementation of the revised Simplex method on GPU. This implementation permits one to speed up solution with a maximum factor of 18 in single precision on a NVIDIA GeForce 9600 GT GPU card as compared with GLPK solver run on Intel Core 2 Duo 3 GHz CPU.

Lalami et al. [36] have presented a parallel implementation via CUDA of the standard Simplex algorithm on CPU-GPU systems for dense LP problems. Experiments carried out on a CPU with 3 GHz Xeon Quadro INTEL processor and a GTX 260 GPU card have shown substantial speedup of 12.5 in double precision. Double precision implementation is used in order to improve the quality of solutions. The authors have also extended their work on a multi-GPU implementation [37] and their computational results on randomly generated dense problems showed a maximum speedup of 24.5. The experiments were performed with use of two Tesla C2050 boards.

Meyer et al. [38] proposed a mono- and a multi-GPU implementation of the tableau simplex algorithm and compared their implementation with the serial Clp solver. Their implementation outperformed Clp solver on large sparse LPs. Both these papers [37,38] that extend their approach to multiple GPUs, have dealt with a complete implementation of the simplex algorithm on the GPUs including the pivoting and the selection of the entering and leaving variables in order to avoid extra communication between the CPU and the GPUs. In multi-GPU computing several decomposition schemes can be adopted. An horizontal decomposition distributes the constraints on the different GPUs. A vertical decomposition distributes the variables of the LP problem on the GPUs. Finally, one may consider also tiles. The choice of a decomposition scheme has important consequences on the resulting communication pattern and multi-GPU efficacy. A decomposition based on tiles may appear scalable; it nevertheless necessitates many communications between GPUs. In [38], the authors have adopted a vertical decomposition in order to have less communication between GPUs. An horizontal decomposition has been adopted in [37].

Finally, Ploskas and Samaras [39] propose two efficient GPU-based implementations of the revised simplex algorithm and a primal-dual exterior point simplex algorithm. Both parallel algorithms have been implemented in MATLAB using MATLABs Parallel Computing Toolbox. Computational results on randomly generated sparse and dense linear programming problems and on a set of benchmark problems (netlib, kennington, meszaros) are also presented. The results show that the primal-dual exterior point simplex implementation achieves a quite satisfactory speedup (2.3 on average) over MATLABs interior point method for the set of benchmark LPs and much greater speedups for the randomly generated LPs. However, the corresponding results (and the speedups obtained) for the revised simplex implementation, although quite good for randomly generated dense LPs, they were significantly inferior in the general case to the ones referred above for the exterior point method.

As it can be seen, although the general feeling is that the overall research work in GPU-based simplex parallelization has not yet led to significant achievements in the general case (especially for sparse problems in comparison with serial solvers), worth telling improvements have been noticed in the case of dense LP problems. Also as GPUs are rapidly evolving, we can certainly expect for such implementations a great improvement of performances in the near future.

8 Conclusion

A number of valuable recent works in the parallelization of the simplex method are presented throughout this paper. A detailed overview is also given, including the recent advances in GPU-based simplex parallelization efforts. Naturally, most of the parallelization attempts made the last years refer to the revised simplex method, however a parallel implementation based on the standard simplex method could also be practical for dense problems if powerful/expensive computing resources are used. The difficulty of implementing a parallel simplex solver that could be significantly faster than (or at least highly competitive to) the existing commercial serial solvers in all cases remains an issue. Indeed the most impressive recent work in the literature refers to the utilization of the revised simplex method for solving large-scale stochastic LP problems, achieving speed-up values more than 100 over the Clp serial simplex solver when implemented on a supercomputer. The efficient parallelization of the dual revised method exploiting the scope of parallelization offered by the technique of suboptimization, should also be considered a significant contribution. Moreover, all the corresponding results (either the older ones or the most recent ones) definitely outline the fact that there isn't an appropriate parallel solver for all kinds of LPs. As a consequence, a valuable piece of future research should probably be the design of some kind of metasolver, that given an LP would automatically propose/select the most efficient parallel solver.

References

1. Murty, K.: Linear Programming. Wiley, New York (1983)
2. Hall, J.A.: Towards a practical parallelization of the simplex method. Comput. Manag. Sci. **7**(2), 139–170 (2010)
3. Yarmish, G., Slyke, R.V.: A distributed scaleable simplex method. J. Supercomput. **49**(3), 373–381 (2009)
4. Badr, E.S., Moussa, M., Paparrizos, K., Samaras, N., Sifaleras, A.: Some Computational Results on MPI Parallel Implementation of Dense Simplex Method. World Academy of Science, Engineering and Technology (WASET), vol. 23, pp. 778–781 (2008)
5. Qin, J., Nguyen, D.T.: A parallel-vector simplex algorithm on distributed-memory computers. Struct. Optim. **11**(3), 260–262 (1996)
6. Hall, J., Huangfu, Q.: A high performance dual revised simplex solver. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 143–151. Springer, Heidelberg (2012)
7. Huangfu, Q., Hall, J.A.: Parallelizing the dual revised simplex method. Technical report ERGO-14-011 (2014). http://www.maths.ed.ac.uk/hall/Publications.html
8. Lubin, M., Hall, J.A., Petra, C.G., Anitescu, M.: Parallel distributed-memory simplex for large-scale stochastic LP problems. Comput. Optim. Appl. **55**(3), 571–596 (2013)
9. Hall, J.A., McKinnon, K.: ASYNPLEX an asynchronous parallel revised simplex algorithm. Ann. Oper. Res. **81**, 27–49 (1998)
10. Shu, W.: Parallel implementation of a sparse simplex algorithm on MIMD distributed memory computers. J. Parallel Distrib. Comput. **31**(1), 2540 (1995)

11. Thomadakis M.E., Liu, J.C.: An efficient steepest-edge simplex algorithm for SIMD computers. In: Proceedings of the International Conference on Supercomputing (ICS 96), Philadelphia, pp. 286–293 (1996)
12. Eckstein, J., Boduroglu, I., Polymenakos, L., Goldfarb, D.: Data-parallel implementations of dense simplex methods on the connection machine CM-2. ORSA J. Comput. **7**(4), 402–416 (1995)
13. Stunkel, C.B.: Linear optimization via message-based parallel processing. In: Proceedings of International Conference on Parallel Processing (ICPP), Pennsylvania, pp. 264–271 (1988)
14. Klabjan, D., Johnson, L.E., Nemhauser, L.G.: A parallel primal-dual simplex algorithm. Oper. Res. Lett. **27**(2), 47–55 (2000)
15. Maros, I., Mitra, G.: Investigating the sparse simplex method on a distributed memory multiprocessor. Parallel Comput. **26**(1), 151–170 (2000)
16. Mamalis, B., Pantziou, G., Dimitropoulos, G., Kremmydas, D.: Reexamining the parallelization schemes for standard full tableau simplex method on distributed memory environments. In: Proceedings of the 10th IASTED PDCN (Parallel and Distributed Computing and Networks) Conference, Innsbruck, Austria, pp. 115–123 (2011)
17. Mamalis, B., Pantziou, G., Dimitropoulos, G., Kremmydas, D.: Highly scalable parallelization of standard simplex method on a myrinet connected cluster platform. ACTA Intl. J. Comput. Appl. **35**(4), 152–161 (2013)
18. Mamalis, B., Perlitis, M.: Hybrid parallelization of standard full tableau simplex method with MPI and OpenMP. In: Proceedings of the 18th Panhellenic Conference in Informatics (PCI 2014), ACM ICPS, October 2–4, Athens, Greece, pp. 1–6 (2014)
19. Finkel, R.A.: Large-grain parallelism: three case studies. In: Jamieson, L.H., Gannon, D., Douglas, R.J. (eds.) The Characteristics of Parallel Algorithms, pp. 21–63. MIT Press, Cambridge (1987)
20. Boffey, T.B., Hay, R.: Implementing parallel simplex algorithms. In: CONPAR 88, p. 169176. Cambridge University Press, Cambridge (1989)
21. Babaev, D.A., Mardanov, S.S.: A parallel algorithm for solving linear programming problems. ZhVychislitelnoi Matematiki Matematicheskoi Fiziki **31**(1), 8695 (1991)
22. Agrawal, A., Blelloch, G.E., Krawitz, R.L., Phillips, C.A.: Four vectormatrix primitives. In: ACM Symposium on Parallel Algorithms and Architectures, pp. 292–302 (1989)
23. Cvetanovic, Z., Freedman, E.G., Nofsinger, C.: Efficient decomposition and performance of parallel PDE, FFT, Monte-Carlo simulations, simplex, and sparse solvers. J. Supercomput. **5**, 1938 (1991)
24. Luo, J., Reijns, G.L.: Linear programming on transputers. In: van Leeuwen, J. (ed.) Algorithms, Software, Architecture. IFIP Transactions A (Computer Science and Technology), pp. 525–534. Elsevier, Amsterdam (1992)
25. Boduroglu, I.: Scalable massively parallel simplex algorithms for block-structured linear programs. Ph.D. thesis, GSAS, Columbia University, New York (1997)
26. Pfefferkorn, C.E., Tomlin, J.A.: Design of a linear programming system for the ILLIAC IV. Technical report SOL 76–8. Systems Optimization Laboratory, Stanford University (1976)
27. Helgason, R.V., Kennington, L.J., Zaki, H.A.: A parallelisation of the simplex method. Ann. Oper. Res. **14**, 1740 (1988)
28. McKinnon, K., Plab, F.: An upper bound on parallelism in the forward transformation within the revised simplex method. Technical report, Department of Mathematics and Statistics, University of Edinburgh (1997)

29. Ho, J.K., Sundarraj, R.P.: On the efficacy of distributed simplex algorithms for linear programming. Comput. Optim. Appl. **3**(4), 349363 (1994)
30. Bixby, R.E., Martin, A.: Parallelizing the dual simplex method. INFORMS J. Comput. **12**, 4556 (2000)
31. Rosander, R.: Multiple pricing and suboptimization in dual linear programming algorithms. Math. Program. Study **4**, 108–117 (1975)
32. Mittelmann, H.: Benchmarks for optimization software (2014). http://plato.la.asu.edu/bench.html. Accessed 30 July 2014
33. Birge, J., Louveaux, F.: Introduction to Stochastic Programming. Springer Series in Operations Research and Financial Engineering, 2nd edn. Springer, New York (2011)
34. Spampinato, D.G., Elster, A.C.: Linear optimization on modern GPUs. In: Proceedings of the 23rd IEEE IPDPS09 Conference, Rome, Italy (2009)
35. Bieling, J., Peschlow, P., Martini, P.: An efficient GPU implementation of the revised simplex method. In: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium, (IPDPS 2010), Atlanta (2010)
36. Lalami, M.E., Boyer, V., El-Baz, D.: Efficient Implementation of the simplex method on a CPU-GPU system. In: IEEE International Parallel and Distributed Processing Symposium, pp. 1994–2001 (2011)
37. Lalami, M.E., El-Baz, D., Boyer, V.: Multi GPU implementation of the simplex algorithm. In: Proceedings of the 2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC), Banff, pp. 179–186 (2011)
38. Meyer, X., Albuquerque, P., Chopard, B.: A multi-GPU implementation and performance model for the standard simplex method. In: Proceedings of the 1st International Symposium and 10th Bal-kan Conference on Operational Research, Thessaloniki, Greece, pp. 312–319 (2011)
39. Ploskas, N., Samaras, N.: Efficient GPU-based implementations of simplex type algorithms. Appl. Math. Comput. **250**, 552570 (2015)
40. Yarmish, G.: A distributed implementation of the simplex method. Ph.D. thesis, Polytechnic University, Brooklyn (2001)
41. Hall, J.A., McKinnon, K.: PARSMI: a parallel revised simplex algorithm incorporating minor iterations and Devex pricing. In: Madsen, K., Olesen, D., Waśniewski, J., Dongarra, J. (eds.) PARA 1996. LNCS, vol. 1184, pp. 67–76. Springer, Heidelberg (1996)
42. FICO Xpress Optimization Suite, A parallel simplex solver (2014). http://www.fico.com/en/products/fico-xpress-optimization-suite