CEJOR (2017) 25:967–984
DOI 10.1007/s10100-016-0452-9

ORIGINAL PAPER

Parallel search paths for the simplex algorithm

Péter Tar[1] · Bálint Stágel[1] · István Maros[1]

Published online: 12 August 2016

Abstract It is well known that the simplex method is inherently a sequential algorithm with little scope for parallelization. Even so, during the last decades several attempts were made to parallelize it since it is one of the most important algorithms for solving linear optimization problems. Such parallelization ideas mostly rely on iteration parallelism and overlapping. Since the simplex method goes through a series of basic solutions until it finds an optimal solution, each of them must be available before performing the next basis change. This phenomenon imposes a limit on the performance of the parallelized version of the simplex method which uses overlapping iterations. Another approach can be considered if we think about alternative paths on the n-dimensional simplex polyhedron. As the simplex method goes through the edges of this polyhedron it is generally true that the speed of convergence of the algorithm is not smooth. It depends on the actual part of the surface. If a parallel version of the simplex algorithm simultaneously goes on different paths on this surface a highly reliable algorithm can be constructed. There is no known dominating strategy for pivot selection. Therefore, one can try different pivot selection methods in parallel in order to guide the algorithm on different pathways. This approach can be used effectively with periodic synchronization on shared memory multi-core computing environments to speed up the solution algorithm and get around numerically and/or algorithmically difficult situations throughout the computations.

✉ Péter Tar
tar@dcs.uni-pannon.hu

Bálint Stágel
stagel@dcs.uni-pannon.hu

István Maros
maros@dcs.uni-pannon.hu

[1] University of Pannonia, 10. Egyetem Str., Veszprém 8200, Hungary

Keywords Linear programming · Simplex method · Parallelization

1 Introduction

Linear programming (LP) is one of the most widely used optimization paradigms. There is an essential need for high quality solvers that are capable of solving real life, large-scale LP problems. Solving these problems in themselves is a nontrivial task. Moreover LP solvers are usually the hidden computational engines behind other optimization algorithms thus proving its relevance. Since the solution process of LP problems is sequential there are significant slack capacities of the processor, which can be used to support the solution using parallelization techniques. Our research focuses on the pure LP case, in which these capacities are freely available.

There are solution methods that require the solution of a large number of LP problems. For instance, in most cases mixed-integer linear programming (MILP) models are solved using a series of LP relaxations. An other widely used approach is the local and iterative linearization of non-linear problems which is usually done recursively. As such, it requires the solution of a sequence of LPs as well. Decomposition methods also rely on the repeated solution of a large number of subproblems. Furthermore, the need for fast algorithms is also relevant for modeling because more detailed models can be built for complex problems and more alternatives can be tested if the efficiency of the solution algorithm is increased.

There are two main types of solution algorithms for LP: the simplex (SX) method and the interior point methods (IPMs). If properly implemented, both of them are capable of solving large-scale problems. The limit of state-of-the-art solvers is around hundreds of thousands of constraints and variables. It is worth knowing that the solution time is not just a function of the problem size. The problem structure and data heavily affect the performance of the solution algorithms. Even for small but ill-conditioned problems a solver can fail due to numerical problems which implies that speed and robustness must be considered as the two main factors of efficiency (Maros 2003b).

It is known that the linear optimization software has become 10^5 times faster over the past 50 years (Bixby 2002). This enormous advancement is partly due to the much faster computers ($\approx 10^3$ times) but also due to the continual development of the solution algorithms and implementations ($\approx 10^2$ times improvement) (Maros and Khaliq 2002). At the beginning (early 1950's) only large mainframe computers could be used to solve problems with a hundred or so constraints and variables. Nowadays computers and even small devices like smartphones are way faster than old mainframes. Almost every device can be used to carry out scientific calculations. The hardware industry tends to manufacture parallel processors while the speed improvement of processor cores is almost stalling. This phenomenon leads the software industry to think of the benefits of parallel architectures and develop parallel algorithms. Since the general CPUs on the market usually use 2-8 cores, optimization on parallel architectures can be exploited by almost every user not just institutes with large and powerful computer systems.

The main objective of this research is to exploit the opportunities that parallelism provides on normal computational devices using multi-core CPUs in the field of linear optimization. Former parallelization attempts of the simplex method have focused on the traditional (dense) simplex method using CPU or GPU computations (Bieling et al. 2010; Cvetanovic et al. 1991; Eckstein et al. 1995; Lalami et al. 2011; Yarmish 2001) but they are not suitable for large-scale problems (Hall 2010). Parallelizing the revised simplex method was also attempted using parallelism within the iterations (Helgason et al. 1988) and overlapping iterations from which a few were successful (Bixby and Martin 2000; Hall and Huangfu 2012; Hall and McKinnon 1996, 1998; Ho and Sundarraj 1994; Shu 1995) but these approaches do not affect the robustness of the solution process. A similar approach using multiple search directions has been designed and tried out with promising result (Maros and Mitra 2000) but it was implemented on a specific architecture (hypercube) and never been tried out in multi-core CPU environments. Based on this research our main goal is to introduce the possibility of different search directions on multi-core CPU architectures with defining and validating our concept of using parallel search directions.

The rest of the paper is organized in the following way. In Sect. 2 after a short introduction to the simplex algorithm the obstacles of parallelizing the simplex method are discussed. In Sect. 3 a brief overview is given on former parallelization attempts used for introducing parallelism into the simplex algorithm. In Sect. 4 our novel method is described in detail on parallelizing the search directions of the simplex method. In Sect. 5 we discuss the necessary implementation issues in order to show how the parallelized search method can be adapted in practice using advanced hardware features. Finally in Sect. 6 we show our preliminary computational results using the new approach as a validation of the ideas and show the performance of our method using our simplex solver the Pannon Optimizer (University of Pannonia 2015; Stágel et al. 2015).

2 Sequentiality of the simplex algorithm

The standard form of the LP problem is as follows:

$$\begin{aligned} \mathit{min}\ \mathbf{c}^{\mathbf{T}}\mathbf{x} \\ \mathbf{A}\mathbf{x} = \mathbf{b} \\ \mathbf{x} \geq \mathbf{0} \\ \mathbf{b} \in \mathbb{R}^m, \quad \mathbf{x}, \mathbf{c} \in \mathbb{R}^n, \quad \mathbf{A} \in \mathbb{R}^{m \times n} \end{aligned}$$

The main concept of the simplex algorithm is the following. First a starting basis must be determined which means m variables with linearly independent columns must be selected in order to form a basis. In each iteration the simplex algorithm moves to a neighboring basis until an optimal solution is found or infeasibility or unboundedness is detected. Denote the index set of the basic variables by $\mathscr{B}$ and the values of basic variables by $\mathbf{x}_{\mathscr{B}}$. A neighboring basis is defined as $\mathscr{B}'$, where $|\mathscr{B}' \setminus \mathscr{B}| = 1$. In order to find the next basis an incoming and an outgoing variable (column) must be chosen that together determine the pivot element. In this logic the simplex algorithm has two main variants: the primal simplex and the dual simplex. In the primal algorithm the pricing operation selects the variable to enter the basis while the ratio test operation selects the outgoing variable. On the other hand, the the dual variant chooses the leaving variable first and then determines the incoming one.

The traditional simplex method maintains the transformed tableau for each basis throughout the computation. The revised simplex method does not compute the entire transformed tableau but maintains only the set $\mathscr{B}$. It only computes a part of the tableau that is necessary for the next iteration. Both the primal and the dual algorithms need the transformed row and transformed column of the pivot element to compute the next basis. These are computed by the two main basic operations of the revised simplex method: the Forward TRANsformation (FTRAN) and the Backward TRANsformation (BTRAN). These are matrix-vector operations so they need heavy computation and their performance significantly affects the solution time (Maros 2003b). They are be defined as:

$$\text{FTRAN:}\alpha = \mathbf{B}^{-1}\mathbf{a},\ \mathbf{a} \in \mathbb{R}^m$$
$$\text{BTRAN:}\alpha^{\mathbf{T}} = \mathbf{a}^T\mathbf{B}^{-1},\ \mathbf{a} \in \mathbb{R}^m,$$

where $\mathbf{B}$ is the actual basis.

Since we consider the solution of large-scale LPs we deal with the revised simplex method where the actual basis or basis inverse can be stored in Lower–Upper triangular (LU) (Markowitz 1957) or in Product Form of the Inverse (PFI) (Dantzig and Orchard-Hays 1954). Both have their advantages and both must be updated in each iteration when the algorithm moves to a neighboring basis. Most of the solvers today rely on the LU form but it is not absolutely superior (Huangfu and Hall 2015a; Tar and Maros 2012). In the following reasoning we follow the primal simplex algorithm but it can be easily adapted to the dual. At basis $\mathbf{B}$ a primal iteration goes through the following steps:

1. Choose one of the improving candidates x_q among the non-basic variables with a given pricing strategy using the reduced cost vector $\mathbf{d}$.
2. Compute $\alpha_{\mathbf{q}} = \mathbf{B}^{-1}\mathbf{a}_{\mathbf{q}}$ for the ratio test.
3. Perform a ratio test and select the outgoing variable $x_{\mathscr{B}_p}$, where $\mathscr{B}_p$ denotes the index of the pth basic variable. The pivot element will be α_q^p.
4. Perform the basis change and update the basis representation.
5. Update $\mathbf{x}_{\mathscr{B}}$ using $\alpha_{\mathbf{q}}$.
6. Update $\mathbf{d}$ using $\alpha^{\mathbf{p}} = \mathbf{e}_{\mathbf{p}}\mathbf{B}^{-1}\mathbf{A}$

The most time consuming part of the iteration is the computation of the $\alpha_{\mathbf{q}}$ and $\alpha^{\mathbf{p}}$ vectors which are computed by an FTRAN and a BTRAN operation (Maros 2003b). If we think of parallelization it must be noted that both operations need the actual basis representation. Since the bases are obtained one after another this implies that two subsequent simplex iterations cannot be done completely in parallel. There are some techniques in the literature that provide some speedup using such methods but none of them is widely used in the implementation of simplex based LP solvers.

3 Parallelization concepts

In this section a brief overview of former parallelization attempts is given. In terms of parallel architectures two different classes can be distinguished (Grama et al. 2003).

The first class is parallelization on shared memory architectures which is also known as symmetric multiprocessing (SMP) using multiprocessor or multi-core processor systems. The common feature of such architectures is that the computational units are physically very close to each other and they share some kind of memory or storage that can be accessed by all the processor cores. Usually there are dedicated memories for multi-level caching for each processor core and a general random-access memory is connected to all of them. This architecture is very advantageous for a small number of cores (usually 2–16) as long as the speed of the memory is not a bottleneck in the system. Nowadays multi-core processors in general purpose computers and smartphones follow this architecture.

The second class of parallel systems is distributed memory architectures. In such systems every processor has its own dedicated memory thus they must use a communication protocol to share data among them. Such systems can be scaled very well but message passing takes some real overhead. Furthermore, it should be remembered that distributed memory systems are usually set up in supercomputers or as a network of stand-alone computers so they are usually not available for everyday users.

In terms of the simplex method our goal is to provide a method that can be applied generally. To achieve this we decided to target the development of our parallel simplex method on SMP since it is available almost everywhere on any modern computer including PCs. In the following, we briefly summarize the parallelization methods published so far (Hall 2010).

3.1 Dense computing approaches

The traditional simplex algorithm using dense data structures can be an ideal candidate to use parallel computing techniques since data parallelization can be extensively used during the computation of the transformed tableau (Cvetanovic et al. 1991; Eckstein et al. 1995). There were several parallelization attempts to develop an effective parallel simplex implementation but none of them succeeded to make their way to be included in leading commercial or open-source LP solvers. Such parallelization can be achieved using CPUs or GPUs (Bieling et al. 2010; Lalami et al. 2011). However, we should consider the fact that if we have an LP problem the complexity of a single iteration will be $O(n^2)$ for the dense implementation since the entire tableau must be updated in each iteration. The performance of such system can be greatly increased by using a modest number of processors but the capabilities of such implementation are limited (Yarmish 2001). Since the number of non-zero elements in the columns of real-life problems is usually independent of the size of the problem (around 10–20) the revised simplex iteration shows a performance of $O(n)$ in practice. This means that whatever speedup someone achieves using the dense simplex it can be overcome by the sparsity exploiting version of the revised method as the problem size increases (Hall 2010).

Also we can consider that a $100{,}000 \times 100{,}000$ matrix contains 10 billion elements. Using double precision arithmetic only the storage of the coefficient matrix takes $\sim$80 GB of memory. If the dense algorithm only stores the original matrix and a working copy which is used for computation it takes $\sim$160 GB. Such need is completely unacceptable (because not satisfiable) on traditional computers, thus it is absolutely not possible to use that algorithm for such problems. Since the revised simplex can exploit sparsity if we consider 20 elements in each column then it takes only $\sim$16 MB to store the coefficient matrix and also the inverse or the factorized form of the basis can be stored in such way thus $\sim$32 MB of memory is necessary for the computation. These values are rough estimates since we do not consider the working vectors of the simplex algorithm but they clearly show that only the revised simplex can be used for solving large-scale real-life problems.

3.2 Iteration parallelization techniques

Most of the computational effort and time of the simplex algorithm are spent on FTRAN and BTRAN which are matrix-vector operations. If the basis is managed using the PFI an elementary step of the FTRAN operation is:

$$\left[\mathbf{e_1} \cdots \mathbf{e_{p-1}} \boldsymbol{\eta} \mathbf{e_{p+1}} \cdots \mathbf{e_m}\right] \mathbf{a} = \mathbf{a} + a_p \boldsymbol{\eta} \tag{1}$$

while an elementary step of the BTRAN operation is:

$$\left[a_1, \ldots, a_p, \ldots, a_m\right] \left[\mathbf{e_1} \ldots \mathbf{e_{p-1}} \boldsymbol{\eta} \mathbf{e_{p+1}} \ldots \mathbf{e_m}\right] = \left[a_1, \ldots, \sum_{i=1}^{m} a_i \boldsymbol{\eta}^i, \ldots, a_m\right]. \tag{2}$$

Both elementary operations can be parallelized well. The vector of the elementary FTRAN can be decomposed into blocks and each computing unit can perform the computations for the decomposed part. The elementary BTRAN operation needs a dot product that can be decomposed as well. On SMP architectures these decompositions can work well but the gain is limited in the speedup if only these elementary operations are parallelized (Helgason et al. 1988). Neither the number of iterations nor the robustness of the algorithm is improved if such parallelization is used. Furthermore such method is relevant only if the dimension of the vector is high and one elementary transformation needs a lot of computations. Due to sparsity it is common that the trivial decomposition of the vector (equal parts in terms of the dimension) does not lead to significant speedup since most of the non-zeros fall into 1 or 2 blocks thus this method works well for dense vectors only.

Other time consuming operation is the pricing if the list of improving variables is long. In such case (depending on the pricing rule) the set of candidates can be split into blocks and they can be evaluated at the same time by different processor cores. The sparsity of the variable list also affects this partitioning thus maximal efficiency cannot be guaranteed. Since the ratio test takes only a very little amount of time (usually $<1\,\%$) it is not worth to parallelize the computation of the ratios.

As a conclusion it can be seen that the effect of the iteration level parallelization is very limited and there is no way to achieve big improvements in these fields. For the most powerful sequential simplex solvers exploiting hyper-sparsity (sparsity of the transformed vectors) is more efficient than applying a straightforward parallelization scheme on data parallelism (Hall and McKinnon 2005).

3.3 Overlapping iterations

If we consider the whole optimization process when talking about parallelizing the simplex method it is worth considering the computation of multiple iterations at the same time. The implementation of such system can follow a synchronous or asynchronous logic. The first has an advantage that the working processors follow a strict order, while the asynchronous system provides more freedom for the working threads but needs a comprehensive background logic (Hall and McKinnon 1998; Shu 1995).

If we consider the primal simplex the order of the main operations is: pricing, computation of the transformed column (FTRAN), ratio test, computation of the transformed row (BTRAN) and finally the update of the basis, the solution and the reduced costs. Since the pivot selection logic needs a moderate amount of computation an other approach can be constructed if some pivot positions are prepared in parallel, which means that more candidates are transformed using FTRAN with the actual basis (Ho and Sundarraj 1994). After the pivoting is done on the first candidate checking whether the other candidates are still improving is much cheaper than their computation since the transformed vectors can be updated easily using the PFI or LU representations of the basis (Hall and McKinnon 1996). The same logic can be adapted for the dual as well (Bixby and Martin 2000; Hall and Huangfu 2012) which is called suboptimization. This overlaps computational components of the dual simplex algorithm across multiple iterations. This approach is implemented as part of the FICO Xpress solver (Huangfu and Hall 2015b) as the first commercial parallel simplex implementation. The limit of this approach is that there is a chance that a candidate turns out to be non-improving thus the work spent on this column is lost.

Another important point is that the inversion or factorization operations of the basis can overlap the simplex iterations. The PFI update form can use the updated transformation vectors while the basis is being reinverted. Since the nature of the PFI is nothing but a sequence of transformation matrices the fresh inverse can shorten this sequence and also improve the numerical accuracy of the inverse (Hall and McKinnon 1998).

4 Parallelizing the search directions during simplex method

The pivoting step of the simplex algorithm can be greatly customized by the large number of algorithmic parameters and tolerances. The basic concept behind our parallelization is that we start different simplex logics in multiple directions on the polyhedron each running on a different core or thread. This was investigated in the past on a hypercube architecture with promising results but it was limited to pricing techniques only and the selected architecture has not survived during the development of computing architectures (Maros and Mitra 2000). After a given number of iterations a synchronization is done between the actual states of these directions, during which we decide what is currently the best direction based on some criteria (e.g. objective value, feasibility) and let the other logics continue from this best progressing basis. This kind of realization of the simplex algorithm is very robust, because if the solution process is cycling or stalling (due to numerical error, degeneracy, etc.) there is a better chance that the solution can be continued in one of the search directions.

4.1 Search directions

A set of neighboring bases defines a path in the n dimensional search space. The simplex algorithm reaches an optimal solution traversing these search directions. Usually there is a large number of search paths that can be very different in nature. Some of them can face extreme numerical difficulties or very deep degeneracy. In such cases these phenomena can prevent the solver from reaching the optimum. The chance of successfully coping with numerical challenges is increased if multiple search directions are investigated. Multiple search paths can be evaluated using multi-core parallel CPU architectures with an efficient and wisely implemented algorithm. This concept makes it possible to run customized primal and dual simplex logics in parallel thus greatly increasing the robustness of the algorithm while reducing the solution time.

Choosing one direction from a given basis is done during the pivoting procedure. This part of the algorithm can have quite a lot of variants which are available due to some freedom of choice in certain algorithmic steps. The choices are controlled by run parameters. As mentioned before, pivoting consists of two steps: choosing a variable to enter and one to leave the basis. In the primal simplex algorithm the first one is called the pricing the second is the ratio test. Both of these procedures greatly rely on heuristic methods. Pricing can offer multiple improving candidates, each leading to different directions, while the choice of the pivot in the ratio test can choose among degenerate positions.

Our first parallelization concept is based on the Dantzig pricing method (Dantzig 1963) but it can be generalized to any full pricing techniques. Since a full pricing method always chooses the variable with the largest improvement (reduced cost) considering all variables, partial pricing can be used if the matrix $\mathbf{A}$ is divided into separate clusters and the improving candidate is chosen after scanning some of the clusters. The most important parameters of this partial pricing framework are the number of separate clusters, the number of clusters to consider, and the number of variables to choose the improving candidate from (Maros 2003a). Different clusters lead to different candidates thus different search paths.

The second step of pivoting is the ratio test. Several anti-degeneracy algorithms are available. Most of them use perturbation heuristics and numerical anti-degeneracy methods in the ratio test. Though they are heuristic in nature these procedures are effective in many cases but their efficiency depends on the problem. Thus running multiple simplex logics using different anti-degeneracy methods increases the chance of successfully avoiding degeneracy and cycling. This approach can be adapted to both the primal and the dual simplex with minimal effort.

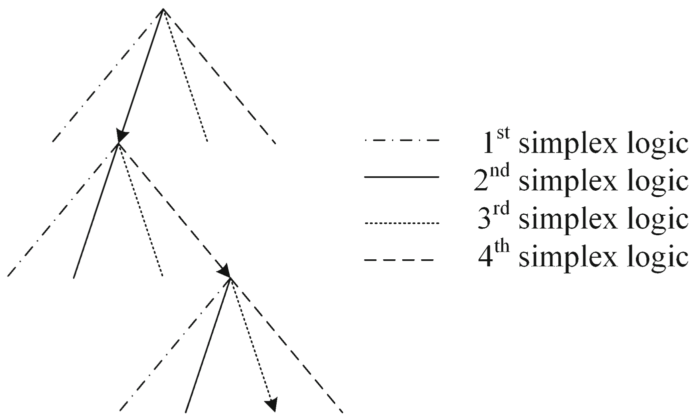

Fig. 1 Evaluating multiple search directions in the simplex method

Our approach makes it possible to run differently parametrized simplex logics in parallel. During the iterations of the simplex method only the neighboring bases of the actual basis can be investigated. In our parallel framework the simplex algorithm is able to search in multiple directions on the polyhedron. An example for a four core CPU can be seen in Fig. 1. One branch of this tree corresponds to some iterations performed with a given logic denoted by different lines. After a certain number of iterations the search directions are synchronized and the best progressing basis is chosen. In the example the followed sequence is 2nd–4th-3rd.

4.2 Basis handling

In our solver the basis is handled with the PFI (Tar and Maros 2012). After an inversion is completed the basis inverse is represented by a sequence (product) of elementary transformation matrices (ETMs) $\mathbf{B_0^{-1}} = \mathbf{E_m E_{m-1}} \ldots \mathbf{E_1}$, where $\mathbf{E_i}$ is an ETM. During the iterations the actual basis inverse can be obtained easily using the pivot column and the previous inverse. In every iteration a new ETM is constructed from the pivot column thus the length of the list of ETMs increases. The inverse of the $k + 1$th iteration can be obtained by $\mathbf{B_{k+1}^{-1}} = \mathbf{E_{m+k} B_k^{-1}}$. Once the inverse is constructed this update formula should be applied in every iteration. After a given number of iterations it is necessary to reconstruct the inverse by a reinversion in order to preserve sparsity and minimize the impact of numerical errors and also reduce the number of ETMs in the PFI. Even though numerical precision can be maintained during the solution with minimal overhead this periodic reinversion is inevitable to prevent the algorithm from slowing down. In our parallel procedure the reinverted basis is stored in the shared memory and the different simplex logics only have to maintain a list of ETM matrices (eta file) that is constructed during updates (basis changes). This idea can be applied to the LU form of the basis, too, if the LU formed basis is updated with the PFI update formula.

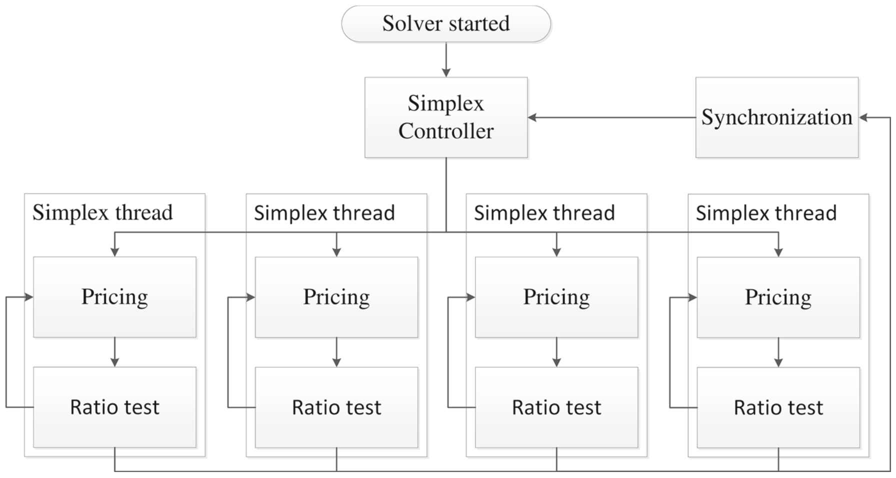

Fig. 2 The control logic of the parallel search paths framework

4.3 Thread logic

One given simplex logic is called a simplex thread or worker thread. During a synchronization the best progressing thread is chosen which is called the master simplex. In such an arrangement a supervisor thread must control the solution process which, in our Pannon Optimizer, is called Simplex Controller. First, it searches for a starting basis and initializes the different simplex logics. Then it spawns the simplex threads with different parameterizations. After performing a given number of iterations a synchronization is done during which the best progressing worker thread is chosen and a reinversion or refactorization is performed for the actual basis of the master simplex. It will serve as a starting point for the next batch of iterations for the workers. After the reinversion of the basis of the master simplex the inverse of the actual basis is shared with all other simplex threads. The behavior of the threads can be seen in Fig. 2 .

The initial parameters of the simplex threads are read from a parameter file but our parallel framework is also capable of adaptive parametrization, which means that the Simplex Controller can change the parameterization of the simplex threads during the solution process. A protocol of the adaptive parametrization can be worked out to change the parametrization if needed (e.g. the solution process is terminated, deep degeneracy). Introducing such a protocol means the integration of some level of artificial intelligence into the solver. It seems to be a promising future research topic for our framework.

After a working thread is finished its state can be classified as iterated, finished or terminated threads. A working thread is called an iterated thread if it progressed a given number of iteration since the last synchronization without any irregular events. If an optimal solution is found by one of the working threads it is called a finished thread. A thread can be a terminated one if serious numerical error occurs or infeasibility or

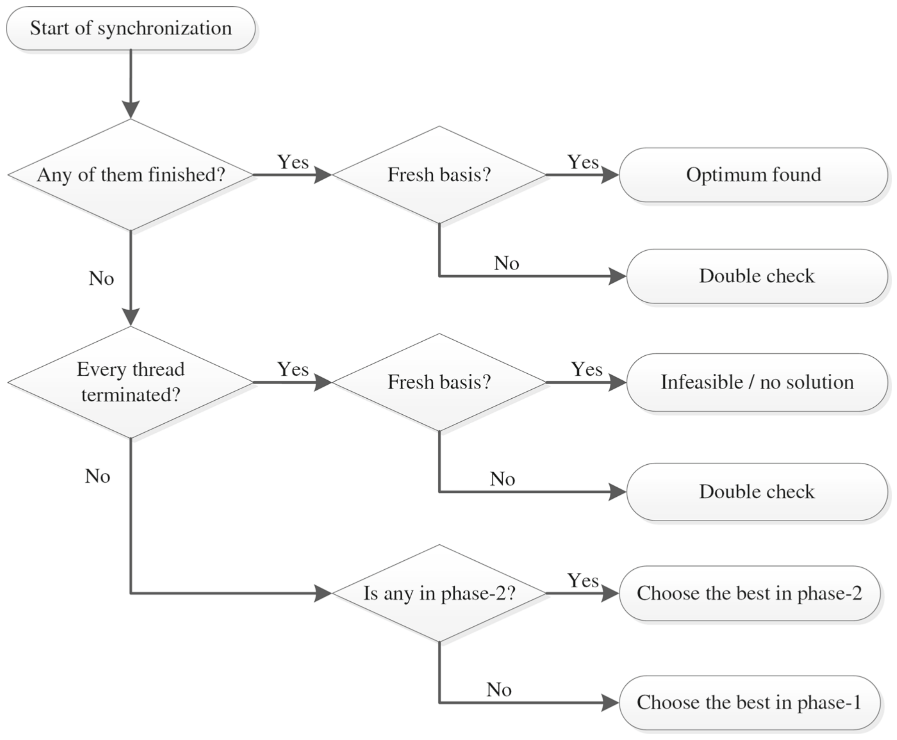

Fig. 3 Synchronization protocol of the parallel search paths

unboundedness is detected. The choice of the master simplex is realized as it is shown in Fig. 3.

The two phase simplex method searches for a feasible solution in phase one and, while keeping feasibility, it looks for an optimal solution in phase two. Therefore the first step of the synchronization protocol is to investigate whether there is any thread in phase-2 if all of the threads are iterated. If there is any thread finished or every thread is terminated a double check is performed. This means the triggering of a reinversion or refactorization. This is to prevent the algorithm from a false conclusion of the solution process caused by numerical inaccuracies. If the given basis is fresh (no iterations done after inversion or factorization) then the real solution of the problem is found.

As described before the basis needs to be reinverted or refactorized after a given number of iterations in order to maintain numerical accuracy and sparsity. Our synchronization logic selects only one of the best progressing threads. It is clear that only the basis of this thread should be reinverted because all the worker threads will continue from the same basis after synchronization. Thus the synchronization is done before reinversion, which implies that synchronization happens according to the reinversion frequency.

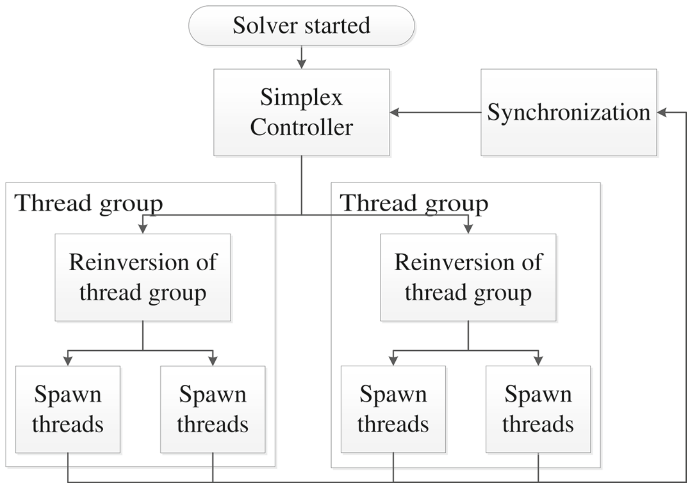

Fig. 4 Maintaining thread groups

4.4 Thread groups

As an extension of the introduced concept the spawned simplex threads can be divided into separate groups. This way all groups can operate on their own, the basis is reinverted separately for each group. This approach is vital in case if both primal and dual logics are running in parallel, thus two separate bases are required. These bases are shared with the worker threads inside the thread group and a master simplex is also chosen in every thread group. The worker threads can be moved among the thread groups dynamically. Using thread groups a more sophisticated and complex search can be performed on the polyhedron.

Without using thread groups the proposed synchronization protocol chooses only one master simplex. If the corresponding current best basis is degenerate, all the other threads will fall into this degenerate solution. Although the chance of leaving the degenerate vertex of the polyhedron is higher using the parallel framework, avoiding degenerate vertices is a much more rewarding approach. For this purpose alternative synchronization protocols can be worked out as well (Fig. 4).

5 Implementation issues

The parallelization of the search directions has been implemented in the Pannon Optimizer linear programming solver developed by the Operations Research Laboratory at the University of Pannonia. Pannon Optimizer implements the revised simplex method including the primal and the dual algorithms. Both implementations rely on uniquely designed data structures that support the efficient solution of large-scale linear optimization problems. The solver is equipped with anti-degeneracy methods and numerical stabilization techniques. It can solve the entire *netlib* (Gay 1985) and *ken-*

nington (Carolan et al. 1990) libraries. In this section the implementation issues of the parallelization are explained in detail.

Since Pannon Optimizer is a C++ application following the C++11 standard, parallelization is implemented using the C++11 threading tools. As described in the previous section, one of the threads is the supervisor thread that manages basis inversion and instructs the worker threads to perform the simplex iterations. Altogether our framework uses $N + 1$ threads where N is the number of CPU cores of the system. Since the operation of the supervisor thread and the worker threads are disjoint in time only at most N threads are running in parallel at any given time. While the worker threads are iterating the supervisor thread is waiting for the next synchronization and while the supervisor is working the workers are waiting to get the next starting point.

5.1 Data localization

Since the system is designed for SMP computers the communication between threads is done through the shared memory. Each worker thread manages its local instances using thread local instances of the data structures which involve the ETMs and the instantiated simplex related objects that manage the working vectors e.g. the prices or the solution.

The original mathematical model (that is parsed from the input file) is handled as a constant reference between all the worker threads since it is usually not modified throughout the solution process. If perturbation is applied the perturbation of the cost vector or the RHS vector is enabled on the workers with different perturbation logic or parameterization then these values must be localized without changing the model.

ETMs of the PFI are split in order to implement the different search paths. This means that there is a common part of the ETM sequence of different bases. This is shared among the threads using a common memory address. The local part of the ETM sequence generated while the iterations are performed within the threads must be individual for each thread. Each inversion recreates the common part and clears the local eta file of the threads. An efficient implementation must handle thread local data transparently without significant computational overhead.

To implement the local eta file thread local variables of high-level programming languages can be used, but they have to be addressed carefully to maintain performance. Such variables have global names (identifiers) but they point to different memory addresses for each thread that uses these variables. The built-in low-level implementations of thread local variables (e.g. C++11 thread_local keyword) depends on the compiler but they operate the same way. They create a hash table between thread identifiers and the corresponding memory addresses for each thread local variable in order to access thread local data. This means that accessing a thread local variable takes the time of searching a hash map for a memory address plus memory addressing while accessing a single variable only takes the time of addressing. Since local data (e.g. reduced cost vector, solution vector, auxiliary vectors, etc.) are being used extensively in loops in the solver algorithm, these thread local variables are accessed frequently thus revealing the hidden performance cost of using thread locals. This can be overcome if the address of every thread local variable is accessed once and the given memory address is stored as long as the thread is running since these addresses are constant in the memory for the life-cycle of each thread. Since the indirection can be cached this method can dismiss the cost of using thread local variables and the threaded version can give the same performance as the sequential one.

5.2 Simplex state

Throughout the solution process synchronization is done using the protocol defined in Sect. 4.3. After the synchronization step decided which simplex path should be selected as the master simplex an inversion is done and the worker threads are initialized with the state of the master simplex. This is an extraction of the thread local data of the master simplex that shall be passed over to the other workers. The state consists of the following:

- The basis head $\mathscr{B}$ describing the basic variables.
- The values (states) of the nonbasic variables $\mathbf{x}_{\mathscr{B}}$ describing whether they are at their lower or upper bounds.
- The solution vector $\mathbf{x}_{\mathscr{B}}$ of the selected node is computed only once after inversion and it is distributed among the workers.
- The reduced cost vector $\mathbf{d}$ is copied from the master simplex if the reduced costs are not recomputed after every inversion. Such method is applied in Pannon Optimizer to enhance the efficiency of the EXPAND procedure (to handle degeneracy) (Gill et al. 1989). If the reduced costs are recomputed they are distributed like the solution vector.
- Perturbation vectors used on the model (if any) (Koberstein 2005).
- The value of the objective function z.

Since the size of these data structures is linear in terms of the problem size and the number of threads and state distribution among the threads happens only after reinversions this does not cause even a minor overhead in the solution process.

5.3 FPU arithmethic

Another implementational issue is that the Floating Point Unit (FPU) of the CPU can use different precisions according to the operating system or the programming language used. The default setting for a sequential program is that floating point operations are performed using 80-bit registers from which the double precision values are extracted as the most significant 64 bits. This means that small numerical errors can be eliminated by the FPU if 80-bit arithmetic is used. The implementation of the threading in C++11 is not covered by the standard therefore the precision of the computations is not completely under control. If `std::thread` is being used it automatically turns off the 80-bit computation and uses 64-bit arithmetic for the floating point operations.

Since simplex pivot selection logic is very sensitive to the numerical values, different candidates can be selected in case of degeneracy if the arithmetic operations are not the same in the sequential and the parallel implementations. In order to be comparable with the parallel version the sequential implementation must use the same FPU precision settings. In Pannon Optimizer we have developed a small assembly routine that switches the FPU to 64-bit when the program is initialized thus the results provided in our computational study are comparable.

6 Preliminary results

In order to get an insight into the efficiency of the introduced parallel search framework we have implemented it in Pannon Optimizer. Since it is capable of solving the entire *netlib* (Gay 1985) and *kennington* (Carolan et al. 1990) test sets with very reasonable speed it is a high-performance solver. Our solver has been designed for research purposes thus its modular structure provides the opportunity to try out modular improvements and new ideas. In our brief study some instances of *netlib* problem set have been used as input to Pannon Optimizer on an Intel Core i5-3210M 2.5 GHz CPU architecture. The solver can be parameterized using parameter files that are parsed at the beginning. Parameter values can be overwritten anytime during the solution process. This limited study does not aim to provide the best possible directions or give a recommendation on how to parameterize the search directions. We simply want to demonstrate that using this approach can be effective which supports the viability of our concept. As an immediate extension we will implement all reasonable pricing strategies and perform a comprehensive computational study in order to work out criteria of how to guide the search directions in the most efficient way.

Our solver is parameterized in two parts. The common part is shared among all the worker threads while the customized part defines the different logics for the worker threads. In the following we do not provide all the parameters since the solver uses $\sim$50 parameters depending on the algorithm. During the brief study the most relevant items of the common parametrization were the following:

- Four parallel simplex worker threads were set.
- PFI basis representation was used with reinversion frequency set to 100.
- No presolving, scaling, and no advanced starting basis techniques were applied.
- Dantzig pricing was used with the partial pricing framework.

The following parameters of the worker threads are different. These parameters generate the different search paths on the n dimensional polyhedron:

- Partial pricing parameterization:
  - Number of clusters which determines the division of the set of variables.
  - Number of investigated clusters which is used by the pricing to define how many clusters should be investigated to find the best candidate depending on the pricing rule. If no candidate is found pricing can involve more clusters.
  - The number of candidates which can optionally stop the pricing after the defined number of candidates is found in order to speed up the pricing.
- Applied anti-degeneracy techniques and numerical tolerances:
  - Algorithmic parameters customizing the pivoting rule.
  - Numerical parameters to define the minimal step length.
  - The random seed is guaranteed to be different for each of the threads.

Table 1 Iteration numbers to optimality of some *netlib* problems using sequential or parallel search paths

Problem name	Row	Col.	Nonz.	Sparsity ratio (%)	Sequential search	Parallel search	Gain (%)
25FV47	822	1571	11,127	0.86	8798	7078	18.41
CYCLE	1904	2857	21,322	0.39	9640	3409	64.64
GREENBEA	2393	5405	31,499	0.24	10,287	8919	13.30
GREENBEB	2393	5405	31,499	0.24	14,035	8864	36.84
MAROS	847	1443	10,006	0.82	7850	5578	28.94
PEROLD	626	1376	6026	0.7	4224	1800	57.39
PILOT	1442	3652	43,220	0.82	9084	8153	10.25
PILOT_JA	941	1988	14,706	0.79	6882	5589	18.79
PILOT_WE	723	2789	9218	0.46	8723	6826	21.75
CRE-A	3517	4067	19,054	0.13	11,150	5276	52.68
CRE-C	3069	3678	16,922	0.15	13,230	4883	63.09
KEN-11	14,695	21,349	70,354	0.02	22,263	21,670	2.66
OSA-14	2338	52,460	367,220	0.30	2379	1912	19.63
OSA-30	4351	100,024	700,160	0.16	4307	4077	5.34
OSA-60	10,281	23,2966	1,630,758	0.07	9703	8246	15.02
AIR-04	823	8904	72,965	1.00	16,716	11,195	33.03
BEASLEYC3	1750	2500	5000	0.11	1110	1013	8.74
N4-3	1236	3596	14,036	0.32	1791	857	52.15
NEOS-1109824	28,989	1520	89,528	0.20	270	226	16.30
NEOS-1337307	5687	2840	30,799	0.19	17,114	12,966	24.24
NEOS-686190	3664	3660	18,085	0.13	2504	1465	41.49
RAN16x16	288	512	1024	0.69	358	300	16.20
SP98IR	1531	1680	71,704	2.79	5019	3173	36.78

Since we want to compare the parallel search path with the sequential one we have to define a performance measure. If we consider that a basis change corresponds to an edge on the search path then two paths can be compared by the number of edges which is nothing but the number of iterations. Thus in the following table the basis of comparison is the iteration number using the different logics.

In Table 1 some examples are shown for running differently parameterized anti-degeneracy techniques with differently parameterized partial pricing strategies in parallel. In our results the number of clusters are set using relative primes to ensure that the partitions are not overlapping among the threads.

The general reduction using the parallel approach against the sequential is around 20–25 %. This is a notable improvement considering that we are using slack capacities that are already present without additional resource investment. These capacities can be used in case of pure LP problems or root LP relaxations of MILP problems. In our tests we have used numerically and computationally hard problems from the *netlib* LP test set that are widely known in the scientific literature. Furthermore we investigated the *kennington* LP test set as well as some root relaxations of MIPLIB2010 problems.

As the results clearly show our approach provides significantly better search paths for problems with heavy degeneracy like CYCLE or PEROLD. On problems that are likely to fall into numerical cycling or stalling our robust approach performs exceedingly well thus validating the expectations. For the other problems the parallel framework also eliminates some degenerate situations but since they are not so ill-conditioned the gain is smaller. From our future research on the best recommendation of the parameterization of the threads and the implementation of the adaptive control logic we expect that the efficiency of this framework will be further increased. Our results correspond to the solution of pure LP problems or root LP relaxations of other instances. Since the aim of our paper is to highlight that our method is preferable on SMP systems the combined effect with parallel MILP of NLP solvers is the topic of our further research.

7 Conclusions

In this paper we have introduced a novel parallelization method for the simplex method. The concept of parallel search paths has been compared to former parallelization methods in this field. Our method is designed to improve not just the speed but also the robustness of the simplex algorithm. This provides the ability of overcoming numerically and/or algorithmically difficult situations throughout the solution process. The key implementational features including the utilization of modern processor architecture have also been discussed that give answers to practice oriented questions. We also provided computational results to prove the effectiveness, efficiency and robustness of the new approach. The results, obtained by using Pannon Optimizer, seem to validate our expectations.

Acknowledgements This publication/research has been supported by the European Union and Hungary and co-financed by the European Social Fund through the Project TÁMOP-4.2.2.C-11/1/KONV-2012-0004—National Research Center for Development and Market Introduction of Advanced Information and Communication Technologies.

References

Bieling J, Peschlow P, Martini P (2010) An efficient GPU implementation of the revised simplex method. In: 2010 IEEE international symposium on parallel distributed processing, workshops and PhD Forum (IPDPSW), pp 1–8

Bixby R (2002) Solving real-world linear programs: a decade and more of progress. Oper Res 50(1):3–15

Bixby R, Martin A (2000) Parallelizing the dual simplex method. INFORMS J Comput 12(1):45–56

Carolan W, Hill J, Kennington J, Niemi S, Wichmann S (1990) An empirical evaluation of the KORBX algorithms for military airlift applications. Oper Res 38(2):240–248

Cvetanovic Z, Freedman E, Nofsinger C (1991) Efficient decomposition and performance of parallel PDE, FFT, Monte Carlo simulations, simplex, and Sparse solvers. J Supercomput 5(2–3):219–238

Dantzig G (1963) Linear programming and extensions. Princeton University Press, Princeton

Dantzig G, Orchard-Hays W (1954) The product form for the inverse in the simplex method. Math Tables Other Aids Comput 8(46):64–67

Eckstein J, Boduroğlu I, Polymenakos L, Goldfarb D (1995) Data-parallel implementations of dense simplex methods on the connection machine CM-2. ORSA J Comput 7(4):402–416

Gay D (1985) Electronic mail distribution of linear programming test problems. Math Program Soc COAL Newsl 13:10–12

Gill P, Murray W, Saunders M, Wright M (1989) A practical anti-cycling procedure for linearly constrained optimization. Math Program 45(1–3):437–474

Grama A, Karypia G, Gupta A, Kumar V (2003) Introduction to parallel computing: design and analysis of algorithms. Addison-Wesley, Reading

Hall J (2010) Towards a practical parallelisation of the simplex method. Comput Manag Sci 7(2):139–170

Hall J, Huangfu Q (2012) A high performance dual revised simplex solver. In: Wyrzykowski R, Dongarra J, Karczewski K, Waśniewski J (eds) Parallel Processing and Applied Mathematics. Lecture Notes in Computer Science, vol 7203. Springer, Berlin, pp 143–151

Hall J, McKinnon K (1996) PARSMI, a parallel revised simplex algorithm incorporating minor iterations and Devex pricing. In: Waśniewski J, Dongarra J, Madsen K, Olesen D (eds) Applied parallel computing industrial computation and optimization. Lecture notes in computer science, vol 1184. Berlin, Springer, pp 359–368

Hall J, McKinnon K (1998) ASYNPLEX, an asynchronous parallelrevised simplex algorithm. Ann Oper Res 81:27–50

Hall J, McKinnon K (2005) Hyper-sparsity in the revised simplex method and how to exploit it. Comput Optim Appl 32(3):259–283

Helgason R, Kennington J, Zaki H (1988) A parallelization of the simplex method. Ann Oper Res 14(1):17–40

Ho J, Sundarraj R (1994) On the efficacy of distributed simplex algorithms for linear programming. Comput Optim Appl 3(4):349–363

Huangfu Q, Hall J (2015a) Novel update techniques for the revised simplex method. Comput Optim Appl 60(3):587–608

Huangfu Q, Hall J (2015b) Parallelizing the dual revised simplex method. Technical report, Cornell University

Koberstein A (2005) The dual simplex method, techniques for a fast and stable implementation. PhD thesis, Universität Paderborn, Paderborn

Lalami M, Boyer V, El-Baz D (2011) Efficient implementation of the simplex method on a CPU-GPU system. In: Proceedings of the 2011 IEEE international symposium on parallel and distributed processing workshops and PhD Forum, IPDPSW'11, Washington, DC, USA. IEEE Computer Society, pp 1999–2006

Markowitz HM (1957) The elimination form of the inverse and its application to linear programming. Manag Sci 3(3):255–269

Maros I (2003a) A general pricing scheme for the simplex method. Ann Oper Res 124(1–4):193–203

Maros I (2003b) Computational techniques of the simplex method. Kluwer Academic Publishers, Norwell

Maros I, Khaliq M (2002) Advances in design and implementation of optimization software. Eur J Oper Res 140(2):322–337

Maros I, Mitra G (2000) Investigating the sparse simplex algorithm on a distributed memory multiprocessor. Parallel Comput 26(1):151–170

Shu W (1995) Parallel Implementation of a sparse simplex algorithm on MIMD distributed memory computers. J Paral Distrib Comput 31(1):25–40

Stágel B, Tar P, Maros I (2015) The Pannon Optimizer - a linear programming solver for research purposes. In: Proceedings of the 5th international conference on recent achievements in mechatronics, automation, computer science and robotics, vol 1(1), pp 293–301

Tar P, Maros I (2012) Product form of the inverse revisited. In: 3rd student conference on operational research, Ravizza S, Holborn P (eds) OpenAccess Series in Informatics (OASIcs), Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, vol 22, pp 64–74

University of Pannonia (2015) Pannon optimizer. http://sourceforge.net/projects/pannonoptimizer/

Yarmish G (2001) A distributed implementation of the simplex method. PhD thesis, Polytechnic University, Brooklyn, NY, USA. AAI3006399