Noname manuscript No.
(will be inserted by the editor)

Parallelizing Simplex within SMT solvers*

Milan Banković

the date of receipt and acceptance should be inserted later

Abstract The usual approach in parallelizing SAT and SMT solvers is either to explore different parts of the search space in parallel (divide-and-conquer approach) or to run multiple instances of the same solver with suitably altered parameters in parallel, possibly exchanging some information during the solving process (parallel portfolio approach). Quite a different approach is to parallelize the execution of time-consuming algorithms that check for satisfiability and propagations during the search space exploration. Since most of the execution time is spent in these procedures, their efficient parallelization might be a promising research direction. In this paper we present our experience in parallelizing the simplex algorithm which is typically used in the SMT context to check the satisfiability of linear arithmetic constraints. We provide a detailed description of this approach and present experimental results that evaluate the potential of the approach compared to the parallel portfolio approach. We also consider the combination of the two approaches.

Keywords SMT solving · Simplex parallelization within SMT · Parallel SMT portfolio

CR Subject Classification I.2.3 Deduction and Theorem Proving

1 Introduction

In recent years, we have witnessed the rapid progress of SAT and SMT solving technologies. In SAT solving, a problem of interest is represented as a propositional formula which is then checked for satisfiability. On the other hand, in SMT solving problems are represented within a much richer, first-order language, and the obtained formula is then checked for satisfiability with respect to some first order theory which is particularly chosen to be suitable for the problem of interest. The expansion of SAT and SMT technologies is strongly driven by the industrial applications and needs, since most of the problems considered in

* The final publication is available at Springer via `http://link.springer.com/article/10.1007/s10462-016-9495-5`

Milan Banković
Faculty of Mathematics, University of Belgrade, Studentski Trg 16, 11000 Belgrade, Serbia,
E-mail: milan@matf.bg.ac.rs

SAT and SMT arise from the industry. Typical applications include software and hardware verification, planning and scheduling, combinatorial design problems, and so on.

In recent years, the focus of many researchers moves towards development of parallel solvers, in order to exploit multi-processor and multi-core technologies that became available. Most of the state-of-the-art SAT and SMT solvers are DPLL-based (Davis et al. 1962), i.e. they explore the search space by branching and backtracking. Early work on parallelization focused on splitting the search space and then exploring different branches in parallel (Hölldobler et al. 2011). This approach is called *search space partitioning*, or *divide-and-conquer*. The other approach is to take multiple instances of a sequential SAT or SMT solver, tune them in different ways (e.g. with different parameters, heuristics, random seeds and so on) and then run them independently in parallel, until one of them finds a solution. Communication between instances is usually minimal, including exchange of some of the learned clauses (usually only short ones). This approach is known as *parallel portfolio*. Another, quite a different idea is to parallelize the algorithms that check for satisfiability and propagations during the search space exploration and that may be very expensive. Examples of such algorithms are the *two-watched-literals algorithm* (Moskewicz et al. 2001) for detecting conflicts and unit propagations which spends more than 80% of the execution time in SAT solvers, and the *simplex algorithm* (Dutertre and de Moura 2006) which is one of the most expensive procedures in modern SMT solvers. The parallelization of the two-watched-literals algorithm is considered in Manthey (2011). In this paper we consider the parallelization of the simplex algorithm within the SMT solvers.

The simplex algorithm is a very important part of the modern SMT solvers, since it is used for deciding the satisfiability of the conjunctions of linear arithmetic constraints over reals or integers. The original simplex algorithm developed by Dantzig (Dantzig et al. 1955) is used in linear programming (LP) to find a solution that optimizes a linear function under a set of linear constraints. The procedure used in SMT is a simplified version of the original algorithm that considers only the existence of a solution (i.e. the satisfiability of the set of linear constraints). While the problems that arise in LP are usually satisfiable (with many possible solutions) and the goal is to find the optimal solution, the SMT problems are very often unsatisfiable, and the goal is to decide about satisfiability of the problem with absolute reliability. Different goals impose different requirements in implementations: while LP solver may use built-in FPU arithmetic provided by the hardware in all calculations, for SMT solvers it is often necessary[1] to use *multiprecision arithmetic* within the simplex algorithm in order to guarantee the obtained result (which is *satisfiable* or *unsatisfiable*). Multiprecision arithmetic emulated within a software library is usually very time-consuming, making the simplex one of the most expensive procedures in SMT.

The standard simplex algorithm permits parallelization in a quite natural way – since its operations are usually performed independently on all the rows of the *simplex tableau*, they can be executed in parallel. This is known as *data parallelism*. In practice, LP problems usually involve very large and very sparse tableaux, so this naive parallelization approach most often cannot beat the specialized sequential algorithms that operate on sparse matrix implementations (Hall 2010). The tableaux seen in SMT are also sparse in most cases (but not always), but when the expensive multiprecision arithmetic is used, even this straightforward way of parallelization may make sense in SMT. This is certainly the case with dense SMT instances that tend to produce large matrix coefficients. In this paper we consider only this kind of the simplex parallelization, and only within the *shared-memory model*.

[1] For efficiency, most modern SMT solvers use multiprecision arithmetic only when needed, i.e. the built-in hardware arithmetic is used until an overflow is detected.

The aim of this paper is to explore the possibility and effectiveness of the simplex algorithm parallelization on different sets of SMT instances that involve linear arithmetic. We also want to compare this approach to other parallelization approaches – notably the parallel portfolio. Finally, we also consider the hybrid approach – using the simplex algorithm parallelization together with the parallel portfolio. While the simplex parallelization might not be very effective in general, we do expect that it will have the positive effect on the instances that involve a lot of computing within the simplex algorithm. In particular, this might be the case with the classes of instances with the large and dense tableaux, where the most of the execution time is spent in the simplex. We can also expect that the hybrid approach might be beneficial on computers with large number of processor cores. For instance, if the computer has 16 processor cores, we can start four solvers in the parallel portfolio, giving each solver four threads for the internal simplex parallelization. This way, each of the solvers in the portfolio may run faster, speeding up the overall execution of the parallel portfolio. The hybrid approach might exploit the processor resources better, since it employs more cores. In this paper, we try to find the answers to all these questions.

There are two main contributions of this paper. First, the paper shows in details how the simplex algorithm within an SMT solver (Dutertre and de Moura 2006) can be adapted to execute in parallel. Second, the paper provides a detailed experimental evaluation and comparison of different parallelization approaches (simplex parallelization, parallel portfolio and hybrid approach) on different sets of SMT instances (including industrial SMT instances as well as artificially generated random instances). All tests were executed on a multi-processor computer with a shared memory. It is also important to note that the tests were performed with a varying number of execution threads (up to 32) in order to see how different parallelization approaches behave when the number of available processor cores grows.

Related work. Most of the modern SAT solvers are based on so-called *conflict driven clause learning* (CDCL) (Marques-Silva et al. 2009). When first-order solvers are concerned, *lazy* SMT solvers (Barrett et al. 2009) are de-facto standard in the field, and the most notable architecture for lazy SMT solvers is DPLL(**T**) (Ganzinger et al. 2004). Within this architecture, the most notable decision procedure for linear arithmetic is the one described in Dutertre and de Moura (2006) which is based on the simplex procedure. This work was further extended by different authors, and efforts have been made to provide better support for integer arithmetic (Griggio 2012; Dillig et al. 2009), and to develop efficient implementations of the procedure (Griggio 2009; King 2014). On the other hand, fragments of linear arithmetic such as *difference logic* can be solved using more efficient decision procedures which are not simplex-based, and one such notable approach within DPLL(**T**) is given in Nieuwenhuis and Oliveras (2005). There are also successful examples of arithmetic solvers that are not based on DPLL(**T**). The one is described in Sheini and Sakallah (2005). It is also based on integration with a CDCL SAT solver in the way similar to the one found in DPLL(**T**). The solver contains two procedures for solving arithmetic constraints — one is tightly integrated with the SAT engine and handles only so-called *unit-two-variable-per-inequality* (UTVPI) constraints, and the other is a general simplex-based arithmetic solver which is invoked only when a full assignment is reached. Another interesting approach for solving linear arithmetic problems is given in Jovanović and De Moura (2011). Although the procedure is not built on top of a CDCL SAT solver, it is CDCL-like — it includes branching, propagations, conflict analysis, learning and backjumps, but it uses cutting planes instead of clauses.

When parallel solvers are concerned, in past few decades many parallel SAT solvers and few parallel SMT solvers appeared on the scene. Notable parallel SAT solvers that follow the divide-and-conquer paradigm are *PSATO* (Zhang et al. 1996), *PaSAT* (Sinz et al. 2001) and *PSatz* (Jurkowiak et al. 2005). On the other hand, famous solvers that follow the parallel portfolio approach are *ManySAT* (Hamadi et al. 2009) and *Plingeling* (Biere 2013). A more detailed surveys on parallel SAT solvers can be found in Singer (2006) and Hölldobler et al. (2011). When parallelization of SMT solvers is concerned, the parallel portfolio approach is first considered in Wintersteiger et al. (2009), where a parallel version of Z3 solver (de Moura and Bjorner 2008) is described. More recently, within the solver *Picoso* (Kalinnik et al. 2010), both divide-and-conquer and parallel portfolio techniques for parallelization of SMT solvers are studied.

The approach based on parallelization of the time-consuming algorithms for satisfiability and propagation checking in SAT and SMT solvers was also investigated, but only to some extent. In Manthey (2011), the parallelization of the unit propagations based on the mentioned two-watched-literals algorithm (Moskewicz et al. 2001) in a SAT solver is considered. The clause set is partitioned into disjoint subsets of clauses and then multiple working threads are employed to process the watch lists over these partitions. The idea of parallelization of complex and time consuming algorithms in SMT solvers is also stressed in an *OpenSMT* (Bruttomesso et al. 2010) project[2]. Although, up to our knowledge, there is no published work available on this topic yet, it shows the relevance of our research.

Finally, the simplex parallelization in general is well studied area for decades. A good overview on this topic is given in Hall (2010). Most of the effort in that area aims towards efficient parallelization in case of sparse tableaux for which highly optimized sequential algorithms exists that are very hard to improve with parallelization. In our work, we do not try to compete with these approaches, since in SMT we are solving a problem of a quite different nature compared to typical LP problems, as we already discussed.

Outline of the paper. The rest of the paper is organized as follows. In Section 2 we provide some basic concepts and notions used throughout the rest of the paper. For the sake of self containment, in Section 3 we describe the existing variant of the simplex algorithm used in SMT. In Section 4 we explain how the parallelization of the simplex algorithm is achieved. In Section 5 we discuss the parallel portfolio and hybrid solver variants that we used in our experiments. A discussion about the implementation details and the data structures used in the solver is given in Section 6. In Section 7, we show the experimental results obtained by the solver. Finally, in Section 8 we give the conclusions and some possible further work directions.

2 Background

In this paper we assume the standard syntax and semantics of first order logic *with equality*, adopting the terminology used in Barrett et al. (2009). We also assume that all considered first order formulae are *ground*, i.e. do not contain variables. A formula F over some fixed signature Σ is *satisfiable* if there exists an interpretation M of the function and predicate symbols of Σ such that formula F evaluates to *true* in M (denoted by $M \vDash F$). In practice, we are usually restricted to some particular set $\mathbf{T}$ of interpretations over Σ, called a (first order) *theory*. A formula F is *satisfiable* in $\mathbf{T}$ (or $\mathbf{T}$-*satisfiable*) if there exists $M \in \mathbf{T}$ such that

[2] `http://www.inf.usi.ch/09-urop-11-sharygina-151196.pdf`

$M \vDash F$. The problem of checking **T**-satisfiability of a first order formula is called *Satisfiability Modulo Theory (SMT) problem*. As a special case, a *propositional formula* is a ground formula containing only 0-ary predicate symbols, called *propositional symbols*, that can be interpreted either as *true* or *false*. The problem of checking satisfiability of a propositional formula is known as *SAT problem*.

Procedures for solving SAT and SMT problems are called, respectively, SAT and SMT *solvers*. The most successful modern SAT solvers are CDCL solvers (short for *conflict-driven clause learning* (Marques-Silva et al. 2009)), based on the famous DPLL algorithm (Davis et al. 1962), but with many improvements (both algorithmic and implementational). On the other hand, modern SMT solvers are usually built on top of existing CDCL SAT solvers, following the so-called *lazy approach* (Barrett et al. 2009) (the most famous such architecture is known as DPLL(**T**) (Ganzinger et al. 2004)). In essence, this means that an SMT solver consists of a SAT engine and decision procedures for supported theories (called *theory solvers*). A theory solver for **T** should be able to check the satisfiability of conjunctions of literals over **T** and to provide an *explanation* of unsatisfiability. Beside this, theory solvers may also have other desirable properties such as *incrementality* and detection (and explaining) of *theory propagations*. The SAT engine searches for a satisfying assignment of truth values to literals of the formula in the usual fashion, but consulting the theory solvers when necessary in order to keep the assignment consistent with the theories.

3 The simplex algorithm

In most modern SMT solvers, the theory solver for linear arithmetic is based on the simplex algorithm. A variant of the algorithm commonly used in SMT and also in this paper is described in Dutertre and de Moura (2006). For better understanding, we shortly describe this algorithm first, and then explain how the parallelization is achieved. The basic variant of the algorithm works with linear constraints over rationals. We assume that all the literals that appear in the formula that is checked for satisfiability are in one of the following forms:

$$\sum_j a_j x_j \bowtie b \tag{1}$$

or

$$x \bowtie b \tag{2}$$

where a_j and b are in $\mathbb{Q}$, $\bowtie \in \{\leq, \geq\}$[3], and x and x_js are *unknowns* whose values (in $\mathbb{Q}$) satisfying the formula should be found. The literals of the form (1) are called *polynomial literals*, and the literals of the form (2) are called *elementary literals*. For each polynomial literal, a fresh unknown s is introduced, the literal is replaced by the elementary literal $s \bowtie b$ (with the same relation $\bowtie$ as in the original literal) wherever it appears within the formula, and the literal:

$$s = \sum_j a_j x_j$$

[3] In practice, the set of possible relations is more general, that is $\bowtie \in \{\leq, \geq, <, >, =, \neq\}$. However, the symbols $\neq$ and $=$ can be easily eliminated in the preprocessing stage ($x = y$ is replaced by $x \leq y \land x \geq y$ and $x \neq y$ is replaced by $x < y \lor x > y$). The only non trivial part is the elimination of strict inequalities. This is done by considering the vector space $\mathbb{Q}_\delta$ of pairs of rationals instead of $\mathbb{Q}$. See Dutertre and de Moura (2006) for details.

is added to the formula as a unit clause. After this transformation, the formula does not contain any polynomial literals any more, only the elementary literals and the equations in which some unknowns are represented as linear combinations of other unknowns (called the *equation literals*). The latter appear only within the unit clauses.

The equation literals are unit-propagated and stored within the theory solver – they comprise the *simplex tableau*. Unknowns that are expressed by these linear combinations are called *basic* unknowns, and the unknowns used in the linear combinations to express the basic unknowns are called *non-basic* unknowns. We denote the sets of basic and non-basic unknowns by B and N, respectively (notice that $B \cap N = \varnothing$). On the other hand, the elementary literals can be incrementally asserted to the theory solver and backtracked, if necessary. The elementary literals impose *bounds* on the unknowns. For each unknown x, we denote by $l(x)$ and $u(x)$ its lower and upper bound, respectively. Initially, $l(x) = -\infty$ and $u(x) = +\infty$ for each unknown x. As elementary literals are asserted to the solver, the bounds change accordingly. When the backtracking is performed, the bounds must be reset to the previous values (for that reason we must remember the values of the bounds at each decision).

For each unknown x, we also maintain the *value* of x in the current model, denoted by $\beta(x)$. In order to keep the model consistent with the asserted literals, the solver must ensure that for each unknown it holds that $l(x) \leq \beta(x) \leq u(x)$. For non-basic unknowns, this invariant is constantly maintained. Whenever a bound is changed for some non-basic unknown x (when an elementary literal is asserted to the theory solver), the value $\beta(x)$ is changed if necessary in order to fall into the interval between the bounds (typically, it is set to the bound that was previously violated by the asserted literal). The values of the basic unknowns that depend on x are then updated. If the feasible interval becomes empty (i.e. $u(x) < l(x)$), a conflict is reported.

For basic unknowns this invariant is not maintained constantly. Instead, the appropriate checking procedure is invoked only periodically. If the procedure detects a basic unknown x_i whose value violates its bounds, it must look for a non-basic unknown x_j in the equation that expresses x_i in the tableau, such that the value of x_j can be changed in the appropriate way (towards its lower or upper bound) so that the value of x_i falls into the interval determined by its bounds. If such unknown x_j exists, the operation called *pivoting* is performed on the tableau: x_i becomes non-basic, and x_j becomes basic. This is done by transforming the tableau: we take the equation that expresses x_i, and use it to express x_j. Then, we simply use this transformed equation to replace x_j in all other equations in the tableau (notice that the sets B and N remain disjoint). After the unknowns x_i and x_j are pivoted, we change the value of x_i by setting it to the previously violated bound. The values of the basic unknowns are then changed accordingly. The pivoting step is repeated as long as there exists a basic unknown x_i whose value is inconsistent with its bounds and a non-basic unknown x_j such that x_i can be pivoted with x_j. Therefore, at the end of the procedure we either find a consistent model, or identify a basic unknown whose value cannot be changed to fit the interval between the bounds (in that case, we report a conflict).

It can be proven (Dutertre and de Moura 2006) that this procedure always terminates, provided that a fixed total ordering of unknowns is respected when x_i and x_j are chosen — i.e. we always choose the smallest basic unknown (according to that fixed ordering) that violates its bounds, and then always choose the smallest non-basic unknown among those that can be pivoted with the chosen basic unknown. This is known as *Bland's rule*. While it guarantees termination, it usually does not lead to a quick convergence, so many other heuristic approaches are often considered. For illustration, the approach used by MathSAT SMT solver (Griggio 2009) is as follows: we choose the basic unknown such that the absolute difference between its value and the violated bound is minimal. After that, we choose the non-basic unknown (among those that can be pivoted with the chosen basic unknown) such that it appears in the smallest number of equations (rows). This strategy does not guarantee termination, so it is applied only a given number of times, before the solver switches to Bland's rule. According to Griggio (2009), using this strategy significantly improves the performance of the simplex algorithm. Our solver also uses this strategy.

In case of integer arithmetic, some (or all) unknowns must take integer values, i.e. the problem is extended with a number of *integer constraints* on unknowns, and we refer to these unknowns as *integer unknowns*. First, the same procedure as above is used to find a rational model. If there is no rational models, a conflict is reported. Otherwise, if there is an integer unknown in the current rational model with a non-integer value, the model must be modified in order to satisfy the integer constraint on that unknown. A variant of the *branch-and-bound* method is most frequently used. For illustration, if an integer unknown x has the value $\frac{5}{2}$ in the current model, we try to correct the model by considering two sub-cases: either $x \leq 2$ or $x \geq 3$ must hold. If either of two subproblems has a satisfying model, this will also be a model of the original problem. In practice, the method involves a lot of case-splitting. This method alone does not guarantee termination of the algorithm, and usually does not lead to a quick convergence, so it is combined with other techniques (such as *Gomory cuts* (Dutertre and de Moura 2006), *Diophantine equation* solving (Griggio 2012), and *Cuts-From-Proofs* technique (Dillig et al. 2009)).

4 Parallelizing the simplex

In this section we consider the parallelization of the previously described simplex procedure. We discuss different parts of the algorithm that can be parallelized and describe how this can be achieved. In the following text we assume that a basic unknown $x_i \in B$ is expressed in the tableau by the linear polynomial L_i, that is:

$$x_i = L_i(\mathbf{x}_N) = \sum_{x_j \in N} a_{ij} x_j$$

where $\mathbf{x}_N = (x_j)_{x_j \in N}$ and a_{ij}s are rational coefficients.

Updating the values of basic unknowns. Let us assume that the bound $u(x_j)$ of a non-basic unknown x_j is changed such that $u(x_j) < \beta(x_j)$. In order to satisfy the bound, the value $\beta(x_j)$ is set to $u(x_j)$. Consequently, the values of the basic unknowns that depend on x_j have to be updated, so the following algorithm must be executed:

```
for_each $x_i \in B$ : $\beta(x_i) := \beta(x_i) + a_{ij}(u(x_j) - \beta(x_j))$;
$\beta(x_j) := u(x_j)$;
```

Notice that in each iteration only the value $\beta(x_i)$ is changed. Since this value is not used in subsequent iterations, the iterations of the loop can be executed in parallel:

```
parallel_for_each $x_i \in B$ : $\beta(x_i) := \beta(x_i) + a_{ij}(u(x_j) - \beta(x_j))$;
$\beta(x_j) := u(x_j)$;
```

We assume here that the `parallel_for_each` loop executes as follows: the set of iterations is first partitioned into k subsets, where k is the number of available threads, and then each of the subsets is executed by a different thread. This computational model is implemented by many popular parallel frameworks and libraries, which makes this form of loop parallelization quite easy to achieve. In the above loop, the total number of operations is $3m$, where m is the number of basic unknowns (an addition, a subtraction and a multiplication per unknown). With k threads, each thread processes approximately $3m/k$ operations.

Updating the values before pivoting. Assume that x_i and x_j are, respectively, a basic and a non-basic unknown that are chosen for pivoting. As explained in Section 3, the pivoting is performed because the value of the basic unknown x_i violates one of its bounds. For this reason, x_i must become non-basic (i.e. not bound to the value of a linear polynomial over other unknowns), which enables us to arbitrarily change its value. The selected non-basic unknown x_j swaps its place with x_i and becomes basic. The value of x_i is set to the violated bound v, and the values of x_j and the rest of the basic unknowns are updated accordingly. This is usually done before the pivoting itself, by the following algorithm:

```
$\theta := \frac{v-\beta(x_i)}{a_{ij}}$;
$\beta(x_i) := v$;
$\beta(x_j) := \beta(x_j) + \theta$;
for_each $x_k \in B \setminus \{x_i\}$ : $\beta(x_k) := \beta(x_k) + a_{kj}\theta$;
```

Again, we may notice that the loop in the last line may be executed in parallel:

```
parallel_for_each $x_k \in B \setminus \{x_i\}$ : $\beta(x_k) := \beta(x_k) + a_{kj}\theta$;
```

In this loop, each of k threads process $2m/k$ operations.

Pivoting. After the values are updated, the pivoting operation is performed. As explained in Section 3, this is done by using the equation that expresses x_i in the tableau to express x_j, and this transformed equation is then used to eliminate x_j from all other equations in the tableau. Assume that the basic unknown x_i is expressed in the following form:

$$x_i = L_i(\mathbf{x}_N) = \left(\sum_{x_l \in N \setminus \{x_j\}} a_{il} x_l \right) + a_{ij} x_j = L_i^*(\mathbf{x}_N) + a_{ij} x_j$$

where $L_i^*(\mathbf{x}_N)$ denotes the sum $L_i(\mathbf{x}_N)$ without the expression $a_{ij}x_j$. Now, the non-basic unknown x_j can be expressed in the following form:

$$x_j = \frac{1}{a_{ij}} (x_i - L_i^*(\mathbf{x}_N))$$

Using the above relation, we can transform the tableau in the following way:

```
$L_j(\mathbf{x}_{N^*}) := \frac{1}{a_{ij}} (x_i - L_i^*(\mathbf{x}_N))$
$L_i(\mathbf{x}_{N^*}) := x_i$
for_each $x_k \in B \setminus \{x_i\}$ : $L_k(\mathbf{x}_{N^*}) := L_k^*(\mathbf{x}_N) + a_{kj} L_j(\mathbf{x}_{N^*})$
```

where $N^* = (N \cup \{x_i\}) \setminus \{x_j\}$ (that is, N^* is N after the pivoting operation). Finally, x_j is moved from N to B, and x_i from B to N. Again, we may notice that the transformation of each polynomial L_k (for $x_k \in B \setminus \{x_i\}$) can be done in parallel, so we replace the loop in the last line of the algorithm with:

```
parallel_for_each $x_k \in B \setminus \{x_i\}$ : $L_k(\mathbf{x}_{N^*}) := L_k^*(\mathbf{x}_N) + a_{kj} L_j(\mathbf{x}_{N^*})$
```

Notice the complexity of the polynomial transformations applied in the above loop: first, each coefficient of $L_j(\mathbf{x}_{N^*})$ must be multiplied by a_{kj}, and then the obtained polynomial must be added to the polynomial $L_k^*(\mathbf{x}_N)$ (this, again, involves addition of the corresponding coefficients in these two polynomials). This takes $2(m-1)n$ operations in the worst case, where m and n are, respectively, the numbers of basic and non-basic unknowns. Recall that these operations may be performed within the multiprecision arithmetic, so the pivoting operation may be very expensive. In practice, the complexity depends on the number of non-zero coefficients in the L_k polynomials, so the total number of operations may be significantly less then $2(m-1)n$ in case of sparse tableaux. This means that, together with the size of the tableau, the density of the tableau also contributes to the complexity of the pivoting algorithm and, therefore, makes its parallelization more or less effective. Another important factor is the size of the coefficients in the tableau — if the coefficients are small, the use of the expensive multiprecision arithmetic may be avoided, so the parallelization may be less effective in that case.

Theory propagation. One type of theory propagation used in a typical arithmetic theory solver is based on the detection of the *weaker* constraints (also called *unate propagation*). For instance, if $x \leq c$ is asserted, then for each $c_1 > c$, it must hold $x \leq c_1$. We find and propagate all such literals. This kind of theory propagation is quite cheap and does not need to be parallelized. Another type of theory propagation is based on *calculating bounds* of unknowns from the equations in the tableau. Namely, the bounds of an unknown may be calculated using the asserted bounds of other unknowns that appear in a particular equation. For instance, if we have an equation:

$$x_i = \sum_{x_j \in N} a_{ij} x_j$$

we can calculate the upper bound of the basic unknown x_i by the following expression:

$$\sum_{a_{ij}>0} a_{ij} u(x_j) + \sum_{a_{ij}<0} a_{ij} l(x_j)$$

and the expression for the lower bound is similar. This calculated bound is finite if and only if all relevant asserted bounds of the non-basic unknowns used in the equation are finite. In the similar way we can calculate the bounds of any non-basic unknown that appears in the above equation, provided that the relevant asserted bounds of other non-basic unknowns and the basic unknown x_i are finite. Such bounds calculations can be done for each equation in the tableau.

Since different equations in the tableau may have non-basic unknowns in common, the bounds calculated from one equation may affect the bounds calculations in other equations. This means that we may need to repeat the algorithm multiple times if we want to reach a fixed point. For simplicity, in our implementation we do not try to reach such fixed point for the bounds. Instead, we consider each equation only once: for each equation the calculation of new bounds is based only on *currently asserted* bounds of the unknowns, i.e. the bounds that were established by the asserted literals before the propagation algorithm started. In other words, the newly calculated bounds from one equation are not used in the calculations in other equations. New bounds will be asserted after all the calculations (using the old bounds) are finished, when we propagate the corresponding elementary literals.

Since each basic unknown appears in only one equation in the tableau, its bounds are simply calculated by using that equation. On the other hand, each non-basic unknown may appear in many equations, so we must calculate its bounds from each of these equations and take those bounds that are the strongest. Therefore, the entire algorithm looks like this:

```
for_each $x_i \in B$ : calculate_bounds($e(x_i)$, $bounds$);
propagate_entailed_literals($bounds$);
```

where $e(x_i)$ is the equation that expresses x_i in the tableau and *bounds* is a passed-by-reference data structure[4] that stores calculated lower and upper bounds for all unknowns (initially, it assigns infinite bounds to all unknowns). The procedure `calculate_bounds()` first calculates the bounds for all unknowns that appear in the equation $e(x_i)$, as previously described. For each calculated bound that is finite and stronger than the corresponding bound stored in *bounds*, that stored bound is adjusted. When the loop is finished, the literals entailed from the bounds that are stored in *bounds* are propagated.

It turns out that calculating bounds for all unknowns is quite an expensive procedure. Bounds calculations may require multiplying and summing the values within the multiprecision arithmetic. For this reason, the procedure is a good candidate for parallelization. Unfortunately, simple `parallel_for_each` cannot be applied here, since in that case all calls of the `calculate_bounds()` procedure would share the same instance of *bounds* data structure, so the synchronization between threads would be required. The solution is in the *parallel reduction*:

```
parallel_reduce $x_i \in B$ : calculate_bounds($e(x_i)$, $bounds$);
```

The reduction works as follows. The set of equations is split into two or more partitions (depending on the number of available working threads). Each working thread processes one of these partitions by calling the `calculate_bounds()` procedure for each equation in the partition. The most important detail here is that each working thread has its own private instance of *bounds* data structure which is passed to all calls of the `calculate_bounds()` procedure within the thread. This way, all threads can work independently without the need of synchronization. When two threads finish their jobs, their *bounds* objects are *joined* into one object by collecting all the calculated bounds from both objects and choosing those that are stronger. The join operation is done by one of the two threads. This process continues until all the threads finish their jobs and all the *bounds* objects are joined into one object.

Notice that the parallel execution of the previously described propagation algorithm will be deterministic, i.e. it will not depend on the race conditions. Namely, the procedure `calculate_bounds()` uses only previously asserted bounds, so it does not depend on the newly calculated bounds obtained by other threads. The corresponding literals are propagated sequentially after all the bounds are calculated. Therefore, the result of the parallel execution is always exactly the same as the result of the sequential execution.

5 Parallel portfolio and hybrid approaches

In this section we discuss the variants of the parallel portfolio that we used for comparison (and hybridization) with the simplex parallelization described in the previous section. As said earlier, the main idea behind the parallel portfolio is to take several solvers (or differently tuned instances of the same solver) and run them in parallel on the same input formula, until one of them solves it. If there is no cooperation between the solvers, we refer to it as

[4] The *bounds* data structure is most efficiently implemented as a hash map that assigns a pair of bounds (lower and upper) to each unknown.

simple portfolio. On the other hand, solvers may exchange learnt clauses in order to share the knowledge acquired during the search process. We call this *cooperative portfolio*.

There are many ways we can form a parallel portfolio, depending on which solvers or solver parameters we choose for different solver instances in the portfolio. In our work, we consider the portfolios consisting of n different instances of the same SMT solver that use different random seeds that affect the branching strategy. For such portfolios, both cooperative and simple variants are considered.

When many processors are available, the *hybrid approach* may be considered: here a parallel portfolio of n solvers is run, but each solver in the portfolio also has the simplex parallelization enabled (as described in Section 4). The entire portfolio is given m processors, where $m > n$, so additional processors may be used in simplex parallelization (typically $m = n \cdot k$, which permits each solver to execute simplex on k processors).

Generally, each solver configuration considered can be described as a pair (n,k), where n is the number of solvers in the portfolio, and $k = m/n$, where m is the number of available processors. Configurations $(1,k)$ correspond to single solvers with only simplex parallelization enabled (not portfolio). Configurations $(n,1)$ correspond to portfolios without simplex parallelization. All other configurations are hybrid configurations. Notice that hybrid configurations may also be simple and cooperative. In Section 7 we present an experimental evaluation of different configurations with the number of available processors $m \leq 32$.

6 Implementational details

The solver that is used in the experiments is the author's own SMT solver (called `argosmt`[5]) implemented from scratch as a part of this research. The solver is implemented in `C++` programming language. Its SAT engine uses VSIDS (Moskewicz et al. 2001) branching strategy with 5% randomly chosen literals. The SMT formulae accepted by the solver are in SMT-LIB 2.0 syntax (Barrett et al. 2010). The solver uses *GNU multiprecision library* (GMP)[6] for exact calculations in the arithmetic theory solver. For efficiency, GMP is used only when needed, i.e. the solver uses hardware types as long as it is possible, and switches to GMP when a potential overflow is detected (a similar optimization is implemented within MathSAT SMT solver (Griggio 2009)). Strategies for choosing basic and non-basic unknowns for pivoting in the arithmetic solver are also borrowed from MathSAT (as described in Section 3). When integer constraints are concerned, our solver implements only a naive strategy based on the *branch-and-bound* (as described in Section 3). A case split is expressed as a clause and passed to the SAT engine to deal with it.

Data structures of the arithmetic theory solver are designed to permit the parallelization techniques described in Section 4. This means that the tableau implementation must be *row-independent*, that is, the solver must be able to perform the operations on multiple rows in parallel. Basic unknowns are stored in a vector, and all the loops described in Section 4 actually iterate over this vector. Each equation (row) in the tableau that expresses some basic unknown x_i is implemented as a *hash map* assigned to x_i that maps non-basic unknowns appearing in the equation to its coefficients (only the unknowns with non-zero coefficients are stored). Thus, we can efficiently iterate over each row, check whether a non-basic unknown exists in the row and retrieve any coefficient from the row. The tableau implementation knows nothing about the columns, i.e. the appearances of a non-basic unknown x_j in

[5] `http://www.matf.bg.ac.rs/~milan/argosmt-5.21/`

[6] `http://www.gmplib.org`

different rows are not connected by any means. Therefore, we cannot iterate over columns efficiently. Iteration over the column of a non-basic unknown x_j can be simulated by iterating over the vector of all basic unknowns (i.e. all the rows of the tableau), skipping the rows that do not contain x_j (which can be efficiently checked). Such row-independent tableau implementation enables efficient parallelization: the basic unknown vector is divided into disjoint ranges and each of them is processed by a separate thread. Notice that the described data structures are very similar to those used by MathSAT SMT solver (Griggio 2009), which confirms that there are state-of-the-art SMT solvers which may benefit from the described parallelization technique. On the other hand, solvers that do not use a row-independent tableau implementation cannot directly use the parallelization approach described in this paper (that is, significant modifications of their data structures may be needed).

For parallelization, the solver relies on Intel's *Threading Building Blocks (TBB)*[7], which is a multi-threaded library that enables different ways of parallelization within the shared memory model. The library aims to provide an efficient load balancing and enables parallelization of loops described in Section 4 in a simple and straightforward way. Most of the loops are rewritten using the `tbb::parallel_for` algorithm from the library that works exactly as the `parallel_for_each` loop described in Section 4. For parallel reduction (that is used for parallelization of bounds calculation), the library algorithm `tbb::parallel_reduce` is used. It works exactly as the `parallel_reduce` concept described in Section 4.

When backtracking is applied, the arithmetic theory solver only has to reset the bounds $l(x_i)$ and $u(x_i)$ of each unknown x_i to the previous values that were valid at the end of the appropriate decision level. For this reason, the state of the bounds must be stored at the end of each decision level. However, some optimizations are possible. For instance, at each decision level, many of the bounds remain unchanged. It would be wasteful to store all the bounds whenever a new level is established, and later backtrack to these stored bounds even if the values of many of them actually did not change. For this reason, in our solver a *copy-on-write* technique is used for storing the bounds — a bound inherited from the previous level is stored for backtracking only if it is changed at the current level. On the other side, the values $\beta(x_i)$ of the unknowns do not need to be reset when the backtracking is applied, since the backtracking may only make the bounds $l(x_i)$ and $u(x_i)$ weaker (the values $\beta(x_i)$ that satisfied stronger bounds will certainly satisfy weaker bounds).

7 Experimental results

In this section we present the experimental results. All tests were performed on a computer with four AMD Opteron 6168 1.6GHz 12-core processors (that is, 48 cores in total), and with 94GB of RAM. The main goal of the experimental evaluation is to measure the effectiveness of the simplex parallelization on different sets of instances. Another goal is to compare the simplex parallelization approach to the parallel portfolio approach and to explore the potential of the hybrid approach (as described in Section 5). Finally, we also want to compare our implementation to some of the state-of-the-art sequential SMT solvers. In order to achieve all these goals, the presented experimental results include evaluation of the following solvers:

[7] `http://www.threadingbuildingblocks.org`

- argosmt solver[8] — our solver with different configurations denoted by (n,k), including the sequential solver (the configuration $(1,1)$), configurations with parallel simplex ($(1,k)$ configurations), simple and cooperative parallel portfolio ($(n,1)$ configurations) and hybrid configurations (both simple and cooperative).
- mathsat5 – the sequential SMT solver MathSAT5[9] (Cimatti et al. 2013).
- cvc4 – the sequential SMT solver CVC4[10] (Barrett et al. 2011).

The tests were performed on several sets of SMT instances from SMT-LIB[11], and also on some randomly generated dense instances. The selected benchmarks and the obtained results are described in more details in the following text. For all tests, we used an appropriate cutoff time per instance. If an instance is not solved within the cutoff time, we considered that cutoff time as its solving time.

7.1 Random dense instances

In this section we present the results obtained by the simplex parallelization on two sets of randomly generated *dense* instances. The first set (denoted by *dense* QF_LRA) consists of 50 randomly generated dense linear real arithmetic instances. These instances are actually satisfiable conjunctions of dense linear equalities, disequalities and inequalities of different sizes. There are no other boolean connectives, so the boolean structure is trivial, and the tableaux are completely filled in. The second set (denoted by *dense* QF_LIA) is very similar, but this time the linear *integer* arithmetic is considered. Again, there are 50 instances in the set and all of them are generated as satisfiable. Although such dense instances are quite artificial and not of significant practical importance, the experimental evaluation of the simplex parallelization on such simple instances may help us to measure the impact of pure simplex parallelization, isolated from any other influence that arise from the boolean structure or the structure of the tableau of a particular instance. Therefore, the speedup[12] obtained on these instances may be considered as the upper bound of the simplex parallelization speedup that we can expect in practice.

		argosmt configuration					
		(1,1)	**(1,2)**	**(1,4)**	**(1,8)**	**(1,16)**	**(1,32)**
dense QF_LRA	**Solved (of 50)**	*50*	*50*	*50*	*50*	*50*	*50*
	Avg. time	820.0	410.9	210.8	115.6	63.7	36.2
dense QF_LIA	**Solved (of 50)**	*43*	*47*	*47*	*47*	*47*	*47*
	Avg. time	303.1	232.0	197.0	170.8	156.9	153.8

Table 1: Simplex parallelization on dense instances using argosmt solver with different number of threads (processor cores) available (1 thread means — no parallelization). Solving times are given in seconds. For unsolved instances, the cutoff time is used.

The cutoff time per instance used for both sets is 1200s. The results in Table 1 show almost linear speedup on dense QF_LRA instances and very good speedup on dense QF_LIA

[8] http://www.matf.bg.ac.rs/~milan/argosmt-5.21/
[9] http://http://mathsat.fbk.eu/
[10] http://cvc4.cs.nyu.edu/web/
[11] http://www.smt-lib.org/
[12] The *speedup* is defined as the ratio of the times consumed by the sequential and parallel executions.

instances. Such great results can be explained by the portion of the execution time spent in the parallelized parts of the simplex algorithm: it is almost 100% for `QF_LRA` and about 95% on `QF_LIA` instances. This is a clear consequence of the density of the tableau (since all coefficients are non-zero) and the trivial boolean structure of the input formula (in case of `QF_LIA`, a small portion of the execution time is spent in case-splitting within the SAT engine).

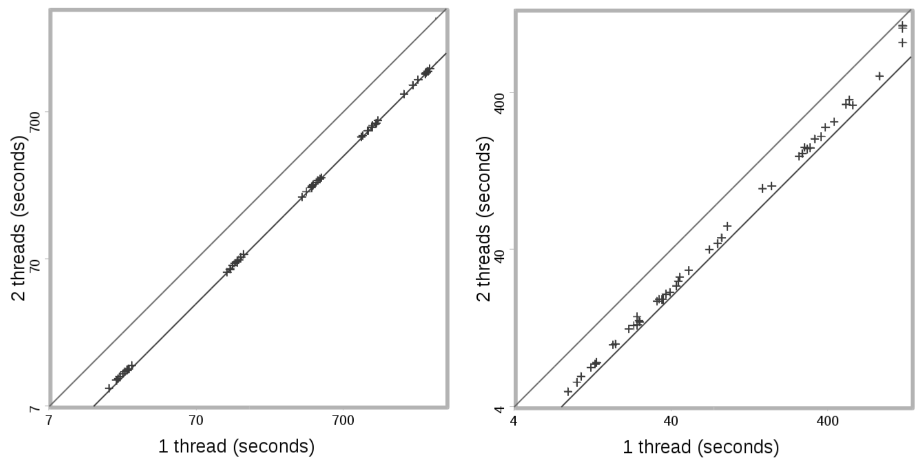

Fig. 1: Simplex parallelization on dense instances (1 thread vs. 2 threads): dense `QF_LRA` (left), dense `QF_LIA` (right)

The charts in Figure 1 show the relation between the execution times (1 thread against 2 threads) on particular instances for dense `QF_LRA` (left) and dense `QF_LIA` sets (right). The diagonal line represents the ratio 1 (i.e. no speedup), and the sub-diagonal line represents the ratio 2 (i.e. the ideal speedup). As expected, most of the points are near the sub-diagonal line (especially in case of dense `QF_LRA`, where almost all instances have nearly ideal speedup). The best speedup for dense `QF_LRA` is 2.0, and for dense `QF_LIA` is 1.86.

7.2 SMT-LIB `QF_LRA` benchmarks

The main part of the experimental evaluation presented in this paper considers `QF_LRA` instances from different sets obtained from the SMT-LIB benchmark library. Initially, we took 11 different instance sets with 785 instances in total. Due to the resource limitations, we wanted to eliminate very hard instances that our solver probably would not be able to solve. For this reason, the preliminary tests were performed using `mathsat5` and `cvc4` solvers on all instances. Based on these results, we selected exactly the instances that both solvers could solve within 10 minutes at most (we assumed that such instances are easy enough, so that sufficient portion of them could be solved by our solver). All subsequent tests are performed on this subset of instances and all the results presented in this section concern this subset. Notice that the instances are selected *a priori*, using the consistent criterion, and not based on the results obtained by our solver, so we think that this gives us a fair setup. The details about the selected instance sets are summarized in Table 2. The first six sets are `QF_LRA` instances, while other five sets are actually `QF_RDL` instances (i.e. instances in *real difference logic*, which is, syntactically, a subset of `QF_LRA`). It is reasonable to expect that the specific structure of `QF_RDL` instances may induce different tableaux structures than in case of general `QF_LRA` instances (for instance, they may result in a sparser tableaux on average), which may result in a different behaviour of the simplex parallelization on such instances. [13] This is the main reason why these instances are included in the evaluation.

Instance set	All instances	Selected instances	SMT-LIB Category
clock_synchro	36	36	`QF_LRA`
cooperatingt2	219	113	`QF_LRA`
latendresse	18	10	`QF_LRA`
miplib	42	16	`QF_LRA`
svcomp	94	43	`QF_LRA`
ultimate	123	60	`QF_LRA`
sal	60	54	`QF_RDL`
scheduling	106	58	`QF_RDL`
skdmxa2	32	27	`QF_RDL`
skdmxa	4	3	`QF_RDL`
temporal	51	45	`QF_RDL`
ALL SETS	**785**	**465**	

Table 2: SMT-LIB QF_LRA benchmarks selection summarized

First tests were performed using the sequential version of our `argosmt` solver. The results are compared to the state-of-the-art sequential solvers `mathsat5` and `cvc4` (Table 3). We used the cutoff time of 1800 seconds per instance. We can see that our solver, while certainly not as efficient as the other two solvers, still can solve a large portion of the selected instances (about 69%), which makes it comparable to the state-of-the-art solvers. The solver is about six time slower than `cvc4`, but this is also influenced by the fact that the cutoff time was used as the solving time for the instances that our solver was not able to solve. The average solving time on solved instances only is 275.9 seconds, which is only about two times slower than `cvc4`, and only about three times slower than `mathsat5`.

Simplex parallelization tests. The separated tests are performed with 2, 4, 8, 16 and 32 threads. The multi-threaded solvers use the same default branching seed as the sequential solver, so they behave identically to the sequential solver, i.e. they follow the same search path during solving. The cutoff time per instance was 1800 seconds. The results are given in Table 4.

The results in Table 4 show that the simplex parallelization gives a significant speedup on average on `QF_LRA` instances, although the speedup varies across different instance sets. Table 5 shows the average speedup for each of the instance sets (1-thread execution against 2-threads execution). The average speedup on all instances is 1.26, while the maximal speedup is 1.74. The table also shows the average percent of execution time spent in parallelized parts of the algorithm. It follows from the table that a larger portion of time spent in parallelized parts of the algorithm leads to a greater speedup. For instance, `clock_synchro` and

[13] Interestingly, based on the results we obtained in the experiments, tableaux in `QF_RDL` instances are, actually, not sparser on average than tableaux in `QF_LRA` instances.

		mathsat5	cvc4	argosmt
clock_synchro	**Solved (of 36)**	*36*	*36*	*28*
	Avg. time	44.4	50.1	479.0
cooperatingt2	**Solved (of 113)**	*113*	*113*	*59*
	Avg. time	180.4	233.0	1130.8
latendresse	**Solved (of 10)**	*10*	*10*	*8*
	Avg. time	1.2	11.5	362.7
miplib	**Solved (of 16)**	*16*	*16*	*10*
	Avg. time	97.7	69.6	725.0
svcomp	**Solved (of 43)**	*43*	*43*	*21*
	Avg. time	80.2	237.6	1159.0
ultimate	**Solved (of 60)**	*60*	*60*	*37*
	Avg. time	108.7	171.2	979.0
sal	**Solved (of 54)**	*54*	*54*	*40*
	Avg. time	50.2	45.1	503.4
scheduling	**Solved (of 58)**	*58*	*58*	*54*
	Avg. time	7.4	56.5	331.1
skdmxa2	**Solved (of 27)**	*27*	*27*	*19*
	Avg. time	149.0	127.4	974.4
skdmxa	**Solved (of 3)**	*3*	*3*	*2*
	Avg. time	93.2	30.1	718.9
temporal	**Solved (of 45)**	*45*	*45*	*45*
	Avg. time	14.8	2.3	22.9
ALL INSTANCES	**Solved (of 465)**	***465***	***465***	***323***
	Avg. time	**89.5**	**126.9**	**741.3**
	Avg. time on solved	**89.5**	**126.9**	**275.9**

Table 3: Results on the selected SMT-LIB `QF_LRA` instances, using the sequential solvers `mathsat5`, `cvc4` and our solver `argosmt`. Solving times are given in seconds. For unsolved instances, the cutoff time is used. The average time on solved instances is also reported.

`latendresse` sets have good speedups on average, since the time spent in parallelized loops for these sets is 96% and 87% on average, respectively. On the other hand, instances of the `sal` set do not have any speedup, since the portion of the execution time spent in parallelized loops is only 17% on average.

The above observation is a natural consequence of Amdahl's law (Amdahl 1967). Namely, if the portion of the execution time spent in parallelized parts of the algorithm is denoted by p, then the theoretical upper bound of the speedup (with two threads) is $2/(2-p)$. In practice, this speedup is usually lower than this, due to the additional expenses that multithreaded execution introduces. Figure 2 provides details about speedups on particular instances (1-thread execution against 2-threads execution). The left chart shows the relation between the solving times. The main diagonal line represents the ratio 1 (i.e. no speedup), and the sub-diagonal line represents the ratio 2 (i.e. the ideal speedup). Crosses represent particular instances. The figure includes all instances from all instance sets with solving time greater than 5 seconds in the sequential execution (228 instances in total). The right chart shows the relation between the speedup and the portion of the execution time in parallelized loops. The black circles represent the upper bound of the speedup, according to Amdahl's law. The figure confirms that in practice the obtained speedups are in accordance with Amdahl's law, with some additional overhead.

Another important question is whether there exists some structural property of an instance (or its tableau) that determines the behaviour of the simplex parallelization on that particular instance. In the following text we investigate how the effectiveness of the simplex

		argosmt configuration					
		(1,1)	**(1,2)**	**(1,4)**	**(1,8)**	**(1,16)**	**(1,32)**
clock_synchro	**Solved (of 36)**	*28*	*28*	*29*	*30*	*31*	*31*
	Avg. time	479.0	448.2	433.6	405.0	383.0	401.9
cooperatingt2	**Solved (of 113)**	*59*	*67*	*71*	*73*	*75*	*75*
	Avg. time	1130.8	1065.1	1016.7	980.7	928.5	924.5
latendresse	**Solved (of 10)**	*8*	*8*	*8*	*8*	*8*	*8*
	Avg. time	362.7	361.7	361.3	361.4	361.5	361.5
miplib	**Solved (of 16)**	*10*	*10*	*10*	*10*	*10*	*10*
	Avg. time	725.0	727.0	728.0	727.2	729.0	733.6
svcomp	**Solved (of 43)**	*21*	*22*	*22*	*23*	*25*	*25*
	Avg. time	1159.0	1099.6	1067.9	1034.5	1014.2	1001.9
ultimate	**Solved (of 60)**	*37*	*40*	*40*	*41*	*41*	*41*
	Avg. time	979.0	914.6	881.9	857.0	821.9	819.3
sal	**Solved (of 54)**	*40*	*40*	*40*	*40*	*40*	*40*
	Avg. time	503.4	503.5	504.0	504.3	505.2	506.7
scheduling	**Solved (of 58)**	*54*	*54*	*54*	*54*	*54*	*54*
	Avg. time	331.1	272.5	256.4	263.3	307.7	326.9
skdmxa2	**Solved (of 27)**	*19*	*21*	*21*	*21*	*21*	*21*
	Avg. time	974.4	901.0	833.3	822.7	770.0	772.5
skdmxa	**Solved (of 3)**	*2*	*2*	*2*	*2*	*2*	*2*
	Avg. time	718.9	709.3	705.2	710.0	700.0	704.4
temporal	**Solved (of 45)**	*45*	*45*	*45*	*45*	*45*	*45*
	Avg. time	22.9	19.3	19.3	14.7	13.9	13.0
ALL INSTANCES	**Solved (of 465)**	***323***	***337***	***342***	***347***	***352***	***352***
	Avg. time	**741.3**	**697.2**	**671.3**	**653.9**	**635.6**	**637.4**
	Avg. time on solved	**275.9**	**278.4**	**265.4**	**264.2**	**261.8**	**264.2**

Table 4: Simplex parallelization on SMT-LIB `QF_LRA` instances, using `argosmt` solver with different number of available threads (processor cores). The results obtained by the sequential solver are also repeated for comparison (the configuration $(1,1)$). Solving times are given in seconds. For unsolved instances, the cutoff time is used. The average time on solved instances is also reported.

	Avg. speedup	**Avg. parallel %**
clock_synchro	1.54	96
cooperatingt2	1.23	69
latendresse	1.58	87
miplib	1.18	44
svcomp	1.27	74
ultimate	1.29	73
sal	0.98	17
scheduling	1.39	90
skdmxa2	1.18	63
skdmxa	1.08	30
temporal	1.20	70
ALL INSTANCES	**1.26**	**70**

Table 5: Average speedups and percents of execution time in parallel loops for different instance sets.

parallelization relates to the following instance property, called the *weight* of the instance I: $weight(I) = nc(I) \cdot mp(I)$, where $nc(I)$ is the total number of non-zero coefficients in the tableau of the instance I, and $mp(I)$ is equal to 10 if the instance I requires the use of the

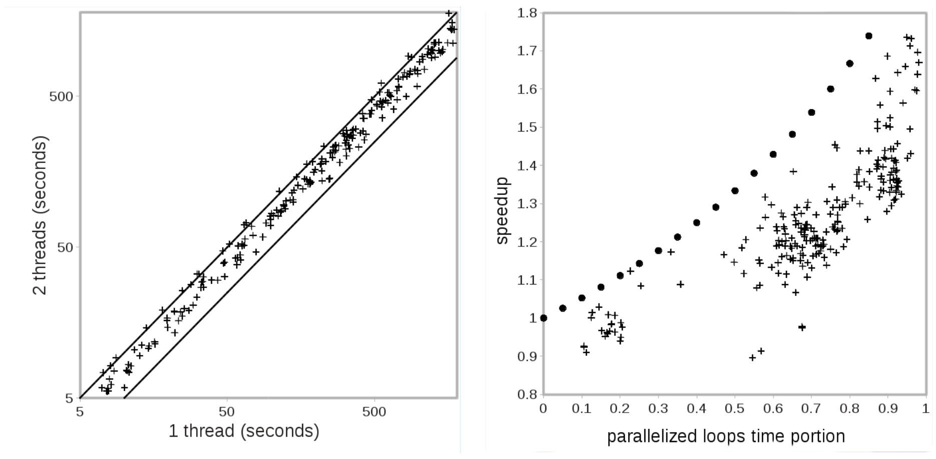

Fig. 2: Execution time ratios (left) and speedups expressed as the function of the portion of the execution time spent in parallelized parts of the algorithm (right) for `QF_LRA` instances (1 thread vs. 2 threads)

multiprecision arithmetic within the solver for exact calculations, and 1 otherwise. Motivation for introducing such a parameter is based on several straightforward assumptions. First, the effectiveness of the parallelization should greatly depend on the tableau size, i.e. the numbers of rows and columns. Since most of the parallelized parts of the simplex algorithm are actually loops that iterate over rows, having a great number of rows means more work to be parallelized. On the other hand, since we are using sparse matrix implementation, it is of a great importance how many non-zero coefficients are there in rows on average, i.e. what is the average density of the tableau rows. Since the both tableau properties (size and density) directly affect the total number of non-zero coefficients in the tableau, it seems natural to expect a significant correlation between this number and the effectiveness of the simplex parallelization. Finally, the important fact is whether the instance requires the multiprecision arithmetic. Since the multiprecision arithmetic may be very expensive, the parallelization of the algorithms that employ such calculations may be much more effective than parallelizing the algorithms that use plain hardware arithmetic. For this reason, we multiply the number of non-zero coefficients $nc(I)$ by 10 for such instances, i.e. we assume that the use of the multiprecision arithmetic increases the cost of the parallelized parts of the algorithm for an order of magnitude.[14] In Table 6 we can see how many instances in each of the instance sets actually require the multiprecision arithmetic during solving. We can see that it is a relatively small portion of the selected instance sets.

In Figure 3 we provide the relation between the speedup and the instance weight (as defined in the previous paragraph). The chart mostly confirms that the instance weight significantly influences the simplex parallelization behaviour. The chart depicts all QF_LRA instances from all instance sets that was solved in at least 5 seconds with sequential execution (228 instances in total). Only 18 of these instances have extremely large weights (from

[14] The factor 10 is arbitrarily chosen, i.e. we did not measure exactly how many times the multiprecision arithmetic is slower than the hardware arithmetic. We have also tried to use values other than 10, but the conclusions were mostly the same.

	Use multiprecision arith.	**Use hardware arith.**
clock_synchro	31	0
cooperatingt2	13	62
latendresse	8	0
miplib	0	10
svcomp	5	20
ultimate	10	31
sal	0	40
scheduling	0	54
skdmxa2	0	21
skdmxa	0	2
temporal	0	45
ALL INSTANCES	**67**	**285**

Table 6: The numbers of solved instances that use multiprecision arithmetic and that do not use it

150000 to about 1300000), while the weights of all other instances are almost evenly distributed from 700 to 150000. The average speedup for the former instances is 1.42, while for the latter is 1.25. However, there are instances that do not follow the observed pattern. For instance, notice the cluster of instances that is the nearest to the upper left corner of the chart in Figure 3. Most of these instances belong to the `scheduling` instance set. Although these instances do not have large weights (most weights are between 3000 and 7000), the simplex parallelization behave quite well on them (the average speedup is 1.39). Further investigation is needed to discover the reason for such behaviour. But even for the instances from the `scheduling` instance set the weight parameter still plays a strong role — only 3 of 54 `scheduling` instances have weights above 50000, and the average speedup for these instances is 1.7, while the average speedup for all other instances from this set (whose weights are below 7000) is 1.34. This means that the instance weight, although maybe not the only property that affects the behaviour of the simplex parallelization, is certainly one of the most important properties that influence the parallelization effectiveness.

Notice that neither of the properties discussed in the previous text (the portion of the execution time spent in the parallelized loops and the instance weight) can help us to actually *predict* the behaviour of the simplex parallelization in advance. On one side, since we do not know in advance what portion of the execution time will be spent in the parallelized parts of the simplex algorithm for some particular instance, we cannot use this fact to *a priori* decide whether it would be useful to employ the simplex parallelization on that instance. On the other side, despite the fact that the instance weight *can* be calculated in advance (at the beginning of the solving process), it is usually not a good idea, since the instance weight tends to be variable in time (the tableau density usually changes during solving, and the multiprecision arithmetic may not be employed at the beginning, but it can be introduced later in the solving process). For this reason, the initial instance weight may be quite different from the instance weight measured later in the solving process, so it may not be a good parameter for predicting the simplex parallelization behaviour. A possible workaround in both cases is to start solver in the sequential mode for a short interval of time, only to collect some data that can help us with the prediction. For instance, we can measure the portion of time spent in the parallelized loops within that interval, and then decide whether the simplex parallelization should be employed in the rest of the solving based on that information. A similar idea can be used in case of the instance weight. Namely, based on our experiments, the tableau density tends to stabilize after a small number of pivotings. Furthermore, if the multiprecision arithmetic is needed, it is usually first used very early, again after a small number of pivotings. This means that the instance weight usually stabilizes after a short period of time and its value does not change much in the rest of the solving. Therefore, we can calculate the instance weight after a sufficient number of pivotings (i.e. when the weight stabilizes) and then decide whether to employ the simplex parallelization in the rest of the solving, based on that calculated weight. We can conclude that the described technique enables both discussed instance properties to be used in practice to effectively decide whether to employ the simplex parallelization on a particular instance.

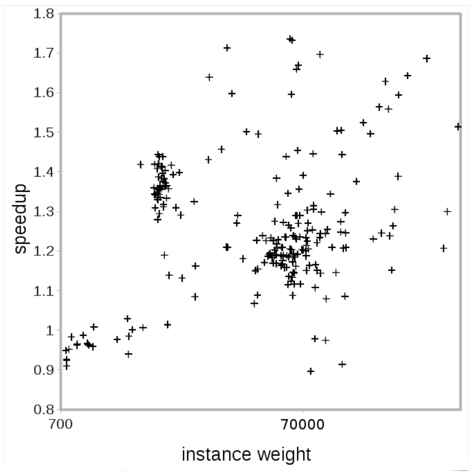

Fig. 3: The speedups on particular `QF_LRA` instances expressed as the function of the instance weight (1 thread vs. 2 threads)

Parallel portfolio tests. Simple parallel portfolio tests were performed with the solver configurations that consist of, respectively, 4, 8, 16 and 32 instances of the `argosmt` solver that use different branching seeds. For the sake of fair comparison, the first solver in the portfolio always uses the same branching seed as the sequential solver, while others use different seeds. The results are shown in Table 7.

The cooperative portfolio configurations are exactly the same as the simple portfolio configurations described above, except that the solvers in the cooperative portfolios exchange all learnt clauses whose length is at most 25 (this length is chosen based on the preliminary tests performed on a subset of the instance sets used in our experiments). Each solver imports the clauses shared by other solvers every time it reaches the decision level 0 (either by restarting or by backjumping to the level 0). The results are shown in Table 8.

It can be immediately noticed that the parallel portfolio gives better results than the simplex parallelization even without cooperation between the solvers. For example, the 4-threaded simple portfolio solves 348 of 465 instances in 645 seconds on average, while the 4-threaded solver with the parallel simplex solves 342 instances in 671 seconds on average. The results are (as expected) even better with the cooperative portfolio (the 4-threaded solver solved 369 instances in 595 seconds on average).

		argosmt configuration				
		(1,1)	**(4,1)**	**(8,1)**	**(16,1)**	**(32,1)**
clock_synchro	**Solved (of 36)**	*28*	*29*	*30*	*30*	*29*
	Avg. time	479.0	453.5	442.7	433.5	448.9
cooperatingt2	**Solved (of 113)**	*59*	*69*	*71*	*73*	*71*
	Avg. time	1130.8	1026.6	996.8	996.0	1005.3
latendresse	**Solved (of 10)**	*8*	*9*	*9*	*9*	*9*
	Avg. time	362.7	184.0	184.2	184.5	185.4
miplib	**Solved (of 16)**	*10*	*10*	*10*	*10*	*10*
	Avg. time	725.0	699.0	697.8	698.0	698.6
svcomp	**Solved (of 43)**	*21*	*28*	*28*	*28*	*29*
	Avg. time	1159.0	903.4	839.2	814.2	814.8
ultimate	**Solved (of 60)**	*37*	*40*	*41*	*41*	*41*
	Avg. time	979.0	879.8	839.6	847.3	860.2
sal	**Solved (of 54)**	*40*	*40*	*40*	*40*	*40*
	Avg. time	503.4	495.4	495.3	499.0	503.2
scheduling	**Solved (of 58)**	*54*	*56*	*56*	*56*	*56*
	Avg. time	331.1	175.0	134.8	118.9	114.3
skdmxa2	**Solved (of 27)**	*19*	*20*	*20*	*20*	*19*
	Avg. time	974.4	865.8	844.1	870.7	903.5
skdmxa	**Solved (of 3)**	*2*	*2*	*2*	*2*	*2*
	Avg. time	718.9	717.6	716.9	728.4	747.5
temporal	**Solved (of 45)**	*45*	*45*	*45*	*45*	*45*
	Avg. time	22.9	21.2	20.0	19.9	21.8
ALL INSTANCES	**Solved (of 465)**	***323***	***348***	***352***	***354***	***351***
	Avg. time	**741.3**	**645.9**	**620.3**	**618.2**	**625.5**
	Avg. time on solved	**275.9**	**258.0**	**241.6**	**247.7**	**244.0**

Table 7: Simple portfolio results (the results for the sequential solver, i.e. the configuration $(1,1)$ are repeated for comparison). Solving times are given in seconds. For unsolved instances, the cutoff time is used. The average time on solved instances is also reported.

It is fair to notice that the overall speedup obtained by parallelizing the simplex algorithm is limited by Amdahl's law (Amdahl 1967). The upper limit depends on the portion of the execution time spent in the parallelized parts of the simplex algorithm. For instance, if the solver spends 60% of its overall execution time in the parallelized parts of the simplex algorithm, with two threads this part of the execution time may shrink to its half at its best, thus obtaining overall speedup of 1.43 only. With four threads, the best we can obtain is to reduce this part of the execution time four times, obtaining the overall speedup of 1.82, and so on. With the growing number of threads, the speedup converges to 2.5, but only in theory. In practice, we can expect slightly lower speedup, due to the overhead induced by the thread scheduling algorithm. On the other hand, the parallel portfolio does not have such limitations. This is especially the case when the solvers within the portfolio exploit some form of randomness (e.g. solvers that use different random seeds in branching heuristics). The best solver in the portfolio may give a super-linear speedup, even with no cooperation between the solvers. Therefore, we cannot expect that the simplex parallelization will beat the parallel portfolio on average. However, on some particular instances the cooperative parallel portfolio may actually take much longer time than the sequential solver, since its execution is non-deterministic and highly unpredictable due to the race conditions. This phenomenon is illustrated in Figure 4 (the top, middle and bottom lines represent the speedups 1, 2, and 4, respectively). It can be seen that there are instances with super-linear speedups (i.e. greater than 4 with 4-threaded portfolio), but also the instances with the speedups significantly

		argosmt configuration				
		(1,1)	**(4,1)**	**(8,1)**	**(16,1)**	**(32,1)**
clock_synchro	**Solved (of 36)**	*28*	*33*	*34*	*34*	*34*
	Avg. time	479.0	292.6	214.0	184.7	164.9
cooperatingt2	**Solved (of 113)**	*59*	*82*	*87*	*93*	*99*
	Avg. time	1130.8	957.6	801.9	665.0	570.6
latendresse	**Solved (of 10)**	*8*	*9*	*9*	*9*	*9*
	Avg. time	362.7	184.5	183.8	184.2	184.4
miplib	**Solved (of 16)**	*10*	*7*	*7*	*7*	*7*
	Avg. time	725.0	1014.4	1013.7	1013.7	1013.3
svcomp	**Solved (of 43)**	*21*	*32*	*33*	*37*	*38*
	Avg. time	1159.0	754.9	659.6	551.9	470.3
ultimate	**Solved (of 60)**	*37*	*42*	*48*	*50*	*51*
	Avg. time	979.0	796.8	693.6	612.0	565.4
sal	**Solved (of 54)**	*40*	*40*	*40*	*40*	*39*
	Avg. time	503.4	527.2	518.7	509.7	531.5
scheduling	**Solved (of 58)**	*54*	*55*	*56*	*56*	*56*
	Avg. time	331.1	137.7	144.8	118.3	112.8
skdmxa2	**Solved (of 27)**	*19*	*22*	*23*	*23*	*23*
	Avg. time	974.4	752.8	689.5	685.3	693.3
skdmxa	**Solved (of 3)**	*2*	*2*	*3*	*3*	*3*
	Avg. time	718.9	651.6	497.1	443.6	384.1
temporal	**Solved (of 45)**	*45*	*45*	*45*	*45*	*45*
	Avg. time	22.9	19.1	16.9	17.7	18.7
ALL INSTANCES	**Solved (of 465)**	***323***	***369***	***385***	***397***	***404***
	Avg. time	**741.3**	**595.0**	**523.9**	**463.0**	**427.0**
	Avg. time on solved	**275.9**	**281.5**	**258.8**	**234.1**	**219.7**

Table 8: Cooperative portfolio results (the results for the sequential solver, i.e. the configuration $(1,1)$ are repeated for comparison). Solving times are given in seconds. For unsolved instances, the cutoff time is used. The average time on solved instances is also reported.

lower than 1. In case of simplex parallelization, on the other hand, the solver runs in a deterministic fashion following the same search path as the sequential solver, and the obtained speedup can be roughly estimated in advance (e.g. using the instance weight introduced in the previous section).

Hybrid approach tests. The hybrid approach is tested with $(2,16)$, $(4,8)$, $(8,4)$ and $(16,2)$ configurations (that is, all hybrid configurations that use 32 threads). Both simple and cooperative variants are considered. For cooperative variant, the solvers are exchanging learnt clauses with length at most 25. In case of simple hybrid configurations (Table 9), the results show that the the best configuration on average is $(8,4)$. Notice that this configuration beats both the configuration with only simplex parallelization enabled (i.e. the configuration $(1,32)$) and the configuration with 32 solvers in the portfolio without the simplex parallelization (i.e. the configuration $(32,1)$). This means that if the cooperation between the solvers in the portfolio is not an option, than the hybridization with the simplex parallelization might be the best choice. On the other hand, the results for the cooperative hybrid configurations (Table 10) show that the best configuration is $(32,1)$. It means that if the implementation permits the cooperation between the solvers in the portfolio, then it is usually better to simply use more solvers in the portfolio than to enable the simplex parallelization within the solvers in order to exploit more processor cores. However, if we look at the results across different instance sets, we can notice that there are instance sets (such as `temporal` and

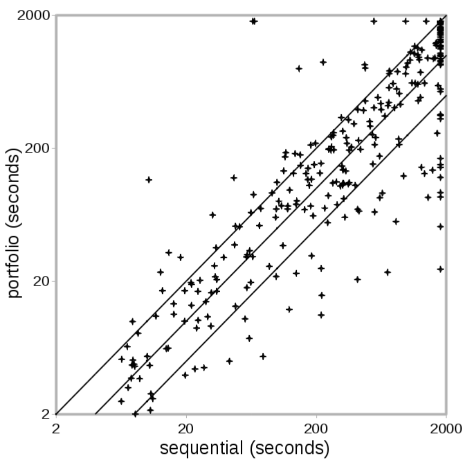

Fig. 4: Cooperative parallel portfolio (the configuration $(4,1)$) vs sequential solver on `QF_LRA` instances

`skdmxa2`) for which the hybrid configurations ($(4,8)$ and $(8,4)$, respectively) give the best average solving times. This tells us that the hybrid approach might be beneficial for some instances, but further investigation is needed in order to better understand in which cases we can expect the best results from such approach.

7.3 SMT-LIB `QF_LIA` benchmarks

Recall that our `argosmt` solver implements only a simple branch-and-bound technique for dealing with the integer arithmetic, so it is still not optimized for solving problems that include integer constraints. For this reason, we do not provide a detailed evaluation of the parallelization techniques described in this paper on `QF_LIA` instances. In this section we only present some preliminary results of the simplex parallelization experiments on a small set of `QF_LIA` instances from SMT-LIB. The tests were performed on the instances selected from two SMT-LIB `QF_LIA` instance sets, and the selection criterion is the same as for `QF_LRA` instances — we choose exactly the instances that can be solved with both `mathsat5` and `cvc4` solvers in at most 10 minutes. The details about the selected instances are shown in Table 11.

The results for sequential executions are shown in Table 12. Our solver seems much slower than the other two solvers, but this is again influenced by the fact that we are using the cutoff time of 1800 seconds as the solving time for unsolved instances. The average solving time on solved instances is 7 seconds for the `slacks` instance set, and 538 seconds for the `tropical-matrix` instance set, which is more comparable with the other two solvers.

The results of the simplex parallelization tests are shown in Table 13. The results show a significant speedup on the `tropical-matrix` instance set, but almost no speedup on the `slacks` instance set. It is hard to tell the exact reason for such behaviour. On one hand, the `tropical-matrix` instance set consists of only 19 instances (of which only 16 are solved with the parallel `argosmt` solver). It is hard to draw some strong conclusions from such a

		argosmt configuration					
		(1,32)	**(2,16)**	**(4,8)**	**(8,4)**	**(16,2)**	**(32,1)**
clock_synchro	**Solved (of 36)**	*31*	*32*	*30*	*30*	*29*	*29*
	Avg. time	401.9	342.8	367.6	397.6	449.5	448.9
cooperatingt2	**Solved (of 113)**	*75*	*79*	*84*	*84*	*79*	*71*
	Avg. time	924.5	888.4	862.4	872.3	940.6	1005.3
latendresse	**Solved (of 10)**	*8*	*9*	*9*	*9*	*9*	*9*
	Avg. time	361.5	182.7	182.5	183.9	185.8	185.4
miplib	**Solved (of 16)**	*10*	*10*	*10*	*10*	*10*	*10*
	Avg. time	733.6	728.8	703.3	701.4	700.9	698.6
svcomp	**Solved (of 43)**	*25*	*31*	*31*	*32*	*29*	*29*
	Avg. time	1001.9	808.4	800.8	767.1	816.5	814.8
ultimate	**Solved (of 60)**	*41*	*42*	*40*	*41*	*40*	*41*
	Avg. time	819.3	784.5	795.4	791.4	847.4	860.2
sal	**Solved (of 54)**	*40*	*40*	*40*	*40*	*40*	*40*
	Avg. time	506.7	504.6	502.8	499.6	501.6	503.2
scheduling	**Solved (of 58)**	*54*	*56*	*56*	*56*	*56*	*56*
	Avg. time	326.9	208.6	163.3	150.7	177.3	114.3
skdmxa2	**Solved (of 27)**	*21*	*20*	*21*	*22*	*20*	*19*
	Avg. time	772.5	777.2	742.2	757.0	830.0	903.5
skdmxa	**Solved (of 3)**	*2*	*2*	*2*	*2*	*2*	*2*
	Avg. time	704.4	697.5	706.6	710.1	738.1	747.5
temporal	**Solved (of 45)**	*45*	*45*	*45*	*45*	*45*	*45*
	Avg. time	13.0	14.3	14.6	16.0	19.4	21.8
ALL INSTANCES	**Solved (of 465)**	***352***	***366***	***368***	***371***	***359***	***351***
	Avg. time	**637.4**	**583.0**	**570.7**	**570.8**	**611.5**	**625.5**
	Avg. time on solved	**264.2**	**253.8**	**246.7**	**259.3**	**260.6**	**244.0**

Table 9: Hybrid approach results (simple variant). Results for configurations $(1,32)$ and $(32,1)$ are repeated for comparison. Solving times are given in seconds. For unsolved instances, the cutoff time is used. The average time on solved instances is also reported.

small number of instances, but it seems that these instances are "hard enough" to spend a significant portion of time in the parallelized loops of the simplex algorithm, so the parallelization is effective. On the other hand, since the average solving time on solved instances of the `slacks` instance set is only 7 seconds, we cannot expect that the parallelization of the simplex algorithm on such easy instances may produce any substantial effect. Unfortunately, we were strongly limited in selection of the benchmarks by the lack of more efficient integer constraint solving techniques in our solver. The final conclusion is that the parallelization of the simplex on `QF_LIA` instances may be as effective as with `QF_LRA` instances, but more detailed evaluation is needed on different instance sets and with the solver that implements more efficient techniques for solving integer constraints in order to be able to solve harder `QF_LIA` instances.

8 Conclusions and further work

In this paper we evaluated the effectiveness of different approaches for parallelizing SMT solvers when solving problems within the linear arithmetic is concerned. The first approach is the parallelization of the simplex algorithm on which the procedure for deciding the satisfiability of linear arithmetic constraints within SMT solvers is based. This is one of the most time-consuming procedures in SMT solvers and its parallelization may have positive impact on the performance. This parallelization approach is based on execution of the same

		argosmt configuration					
		(1,32)	**(2,16)**	**(4,8)**	**(8,4)**	**(16,2)**	**(32,1)**
clock_synchro	**Solved (of 36)**	*31*	*34*	*34*	*34*	*34*	*34*
	Avg. time	401.9	204.9	174.5	173.6	181.0	164.9
cooperatingt2	**Solved (of 113)**	*75*	*82*	*87*	*91*	*94*	*99*
	Avg. time	924.5	843.6	776.7	703.7	636.3	570.6
latendresse	**Solved (of 10)**	*8*	*9*	*9*	*9*	*9*	*9*
	Avg. time	361.5	182.9	183.1	184.0	184.8	184.4
miplib	**Solved (of 16)**	*10*	*9*	*7*	*7*	*7*	*7*
	Avg. time	733.6	950.0	1015.1	1014.2	1013.8	1013.3
svcomp	**Solved (of 43)**	*25*	*31*	*35*	*35*	*36*	*38*
	Avg. time	1001.9	790.5	710.3	558.9	556.4	470.3
ultimate	**Solved (of 60)**	*41*	*44*	*44*	*50*	*50*	*51*
	Avg. time	819.3	743.8	691.5	631.7	582.5	565.4
sal	**Solved (of 54)**	*40*	*39*	*40*	*40*	*40*	*39*
	Avg. time	506.7	539.3	529.4	534.2	534.6	531.5
scheduling	**Solved (of 58)**	*54*	*53*	*53*	*56*	*56*	*56*
	Avg. time	326.9	227.6	205.4	140.6	153.1	112.8
skdmxa2	**Solved (of 27)**	*21*	*23*	*22*	*23*	*23*	*23*
	Avg. time	772.5	704.6	652.8	613.3	622.3	693.3
skdmxa	**Solved (of 3)**	*2*	*2*	*2*	*2*	*2*	*3*
	Avg. time	704.4	670.5	656.3	654.7	644.7	384.1
temporal	**Solved (of 45)**	*45*	*45*	*45*	*45*	*45*	*45*
	Avg. time	13.0	14.6	13.0	13.9	16.2	18.7
ALL INSTANCES	**Solved (of 465)**	***352***	***371***	***378***	***392***	***396***	***404***
	Avg. time	**637.4**	**564.2**	**526.5**	**477.2**	**457.1**	**427.0**
	Avg. time on solved	**264.2**	**251.1**	**233.4**	**230.9**	**223.2**	**219.7**

Table 10: Hybrid approach results (cooperative variant). Results for configurations $(1,32)$ and $(32,1)$ are repeated for comparison. Solving times are given in seconds. For unsolved instances, the cutoff time is used. The average time on solved instances is also reported.

Instance set	**All instances**	**Selected instances**	**SMT-LIB Category**
slacks	233	143	`QF_LIA`
tropical-matrix	108	19	`QF_LIA`
ALL SETS	**341**	**162**	

Table 11: SMT-LIB QF_LIA benchmarks selection summarized

		mathsat5	**cvc4**	**argosmt**
slacks	**Solved (of 143)**	*143*	*143*	*131*
	Avg. time	15.1	20.2	157.5
tropical-matrix	**Solved (of 19)**	*19*	*19*	*13*
	Avg. time	78.5	159.4	937.1
ALL INSTANCES	**Solved (of 162)**	***162***	***162***	***144***
	Avg. time	**22.5**	**36.5**	**248.9**
	Avg. time on solved	**22.5**	**36.5**	**55.0**

Table 12: Results on the selected SMT-LIB `QF_LIA` instances, using the sequential solvers `mathsat5`, `cvc4` and our solver `argosmt`. Solving times are given in seconds. For unsolved instances, the cutoff time is used. The average time on solved instances is also reported.

		argosmt configuration					
		(1,1)	**(1,2)**	**(1,4)**	**(1,8)**	**(1,16)**	**(1,32)**
slacks	**Solved (of 143)**	*131*	*132*	*132*	*132*	*132*	*131*
	Avg. time	157.5	154.1	153.7	153.7	153.7	154.4
tropical-matrix	**Solved (of 19)**	*13*	*15*	*16*	*16*	*16*	*16*
	Avg. time	937.1	858.0	784.3	718.9	677.9	665.6
ALL INSTANCES	**Solved (of 162)**	***144***	***147***	***148***	***148***	***148***	***147***
	Avg. time	**248.9**	**236.6**	**227.6**	**220.0**	**215.2**	**214.3**
	Avg. time on solved	**55.0**	**77.1**	**78.9**	**70.5**	**65.3**	**52.5**

Table 13: Simplex parallelization on SMT-LIB `QF_LIA` instances, using `argosmt` solver with different number of available threads (processor cores). The results obtained by the sequential solver are also repeated for comparison (the configuration $(1,1)$). Solving times are given in seconds. For unsolved instances, the cutoff time is used. The average time on solved instances is also reported.

operations on multiple rows of the simplex tableau in a parallel fashion (known as the data-parallelism). The paper describes in details how this parallelization is achieved. The second approach is running the parallel portfolio of solvers that use different branching seeds. The third approach is the combination of the previous two: each solver in the portfolio is permitted to use additional processor cores to parallelize the execution of the simplex algorithm.

The paper contains a detailed evaluation and comparison of the proposed parallelization approaches. The tests were performed on a large set of instances (both industrial and artificially generated random instances), using the computer with a great number of processors (the approaches are evaluated using different numbers of threads, up to 32). The paper also contains the comparison with the state-of-the-art SMT solvers — MathSAT5 and CVC4. The entire research took many months of developing and testing the implementation, as well as almost three months (about 80 days) of running the tests in order to obtain the results presented in the paper. We hope that the results given in the paper can provide useful information to the developers of parallel SMT solvers.

The experimental results confirmed the potential of the simplex parallelization on some classes of instances. The general conclusion is that the effectiveness of the simplex parallelization directly depends on the portion of the execution time spent in the parallelized parts of the simplex algorithm. When structural properties of instances are concerned, the parallelization behaviour depends on the number of non-zero coefficients of the simplex tableau, which, in turn, depends on the size and the density of the tableau. Another important factor is whether the instance requires the use of the multiprecision arithmetic. In Section 7, these properties are combined into the unique property called the instance *weight*, which is shown to be helpful in estimating the potential of the simplex parallelization on a particular instance.

The results also show that the parallel portfolio is better than the simplex parallelization on average (due to the theoretical limitations of the simplex parallelization imposed by the Amdahl's law), but the deterministic behaviour of the simplex parallelization makes it more reliable and predictable than the parallel portfolio. Finally, experiments with the hybrid approach tell us that in some cases there is the possibility to improve the parallel portfolio by using it in conjunction with the simplex parallelization described in this paper.

For future work, we plan to investigate in more details some of the phenomena noticed in this research. For instance, it seems that there are other instance properties (that we are not aware of) that affect the simplex parallelization, beside the instance weight. Moreover, a similar analysis may be done for the portfolio approach, i.e. we may try to discover the instance properties that affect the behaviour of the parallel portfolio. It may be interesting to compare the results of such analysis with those obtained here for the simplex parallelization and to discuss the similarities and differences. For instance, if we look at the results obtained for the `sal` instance set in this paper, we can see that these instances parallelize badly in all approaches, while the instances of the `temporal` instance set parallelize better using the simplex parallelization than with the parallel portfolio. The question which parallelization approach to employ (if any) on a particular instance and other interesting questions regarding the parallelization of SMT solvers may be addressed in the further work.

Disclosure of potential conflicts of interest

Funding: This work was partially supported by the Serbian Ministry of Science grant 174021 and by the SNF grant SCOPES IZ73Z0_127979/1.

Conflict of interest: The author declares that he has no conflict of interest.

References

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. (2011). Cvc4. In *Computer aided verification*, pages 171–177. Springer.

Barrett, C., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009). Satisfiability Modulo Theories. In *Handbook of Satisfiability*, chapter 26, pages 825–885. IOS Press.

Barrett, C., Stump, A., and Tinelli, C. (2010). The SMT-LIB Standard: Version 2.0. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf`.

Biere, A. (2013). Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In *Proceedings of SAT Competition 2013*, pages 51–52. University of Helsinki.

Bruttomesso, R., Pek, E., Sharygina, N., and Tsitovich, A. (2010). The OpenSMT Solver. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer.

Cimatti, A., Griggio, A., Schaafsma, B., and Sebastiani, R. (2013). The MathSAT5 SMT Solver. In Piterman, N. and Smolka, S., editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer.

Dantzig, G. B., Orden, A., Wolfe, P., et al. (1955). The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

de Moura, L. and Bjorner, N. (2008). Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.

Dillig, I., Dillig, T., and Aiken, A. (2009). Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *Computer Aided Verification*, pages 233–247. Springer.

Dutertre, B. and de Moura, L. (2006). Integrating simplex with DPLL(T). Technical report, CSL, SRI International.

Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., and Tinelli, C. (2004). DPLL(T): Fast Decision Procedures. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer.

Griggio, A. (2009). *An effective SMT engine for Formal Verification*. PhD thesis, University of Trento.

Griggio, A. (2012). A practical approach to satisfiability modulo linear integer arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:1–27.

Hall, J. (2010). Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7(2):139–170.

Hamadi, Y., Jabbour, S., and Sais, L. (2009). ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262.

Hölldobler, S., Manthey, N., Nguyen, V. H., Steinke, P., and Stecklina, J. (2011). Modern Parallel SAT-Solvers. Technical report, TR 2011-6, Knowledge Representation and Reasoning Group, TU Dresden, Germany.

Jovanović, D. and De Moura, L. (2011). Cutting to the chase solving linear integer arithmetic. In *Automated Deduction–CADE-23*, pages 338–353. Springer.

Jurkowiak, B., Li, C. M., and Utard, G. (2005). A Parallelization Scheme Based on Work Stealing for a Class of SAT Solvers. *Journal of Automated Reasoning*, 34(1):73–101.

Kalinnik, N., Abraham, E., Schubert, T., Wimmer, R., and Becker, B. (2010). Exploiting Different Strategies for the Parallelization of an SMT Solver. In *MBMV*, pages 97–106. Fraunhofer Verlag.

King, T. (2014). *Effective Algorithms for the Satisfiability of Quantifier-Free Formulas Over Linear Real and Integer Arithmetic*. PhD thesis, NEW YORK UNIVERSITY.

Manthey, N. (2011). Parallel SAT Solving-Using More Cores. In *Pragmatics of SAT Workshop*.

Marques-Silva, J., Lynce, I., and Malik, S. (2009). Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, chapter 4, pages 131–155. IOS Press.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Annual ACM IEEE Design Automation Conference*, pages 530–535. ACM.

Nieuwenhuis, R. and Oliveras, A. (2005). DPLL (T) with exhaustive theory propagation and its application to difference logic. In *Computer Aided Verification*, pages 321–334. Springer.

Sheini, H. M. and Sakallah, K. A. (2005). A scalable method for solving satisfiability of integer linear arithmetic logic. In *Theory and Applications of Satisfiability Testing*, pages 241–256. Springer.

Singer, D. (2006). *Parallel Resolution of the Satisfiability Problem: A Survey*, pages 123–147. Wiley.

Sinz, C., Blochinger, W., and Kchlin, W. (2001). PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. *Electronic Notes in Discrete Mathematics*, 9:205–216.

Wintersteiger, C. M., Hamadi, Y., and de Moura, L. (2009). A Concurrent Portfolio Approach to SMT Solving. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 715–720. Springer.

Zhang, H., Bonacina, M. P., and Hsiang, J. (1996). PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21:543–560.