SelfSplit parallelization for mixed-integer linear programming

Self-split parallelization for Mixed-Integer Linear Programming

Matteo Fischetti[a,*], Michele Monaci[b], Domenico Salvagnin[a]

[a] *DEI, University of Padova, Italy*
[b] *DEI, University of Bologna, Italy*

Abstract

`SelfSplit` is a simple static mechanism to convert a sequential tree-search code into a parallel one. In this paradigm, tree-search is distributed among a set of identical workers, each of which is able to autonomously determine—without any communication with the other workers—the job parts it has to process. `SelfSplit` already proved quite effective in parallelizing Constraint Programming solvers. In the present paper we investigate the performance of `SelfSplit` when applied to a Mixed-Integer Linear Programming (MILP) solver. Both ad-hoc and general purpose MILP codes have been considered. Computational results show that `SelfSplit`, in spite of its simplicity, can achieve good speedups even in the MILP context.

Keywords: parallel computing, enumerative algorithms, mixed-integer programming, computational analysis

1. Introduction

Parallel computation uses multiple computing elements simultaneously in order solve a given problem. As such, it usually requires breaking the original task to be executed into smaller parts, that are solved independently and concurrently by different workers. The obvious goal of a parallel algorithm is to be able to solve a given problem faster than its sequential counterpart; ideally, we would like the parallel algorithm to scale linearly with the number of workers that we assign to it.

Tree-search algorithms, as those used in mathematical and combinatorial optimization, appear to be naturally and easily parallelizable, and indeed experiments with parallel branch-and-bound algorithms date back to the early 1970s, see for example [1]. The basic idea is that each worker is assigned a subset of branch-and-bound nodes, so that the complete tree is split among

*Corresponding author
Email addresses: `matteo.fischetti@unipd.it` (Matteo Fischetti), `michele.monaci@unibo.it` (Michele Monaci), `domenico.salvagnin@unipd.it` (Domenico Salvagnin)

the workers. However, it did not take long to realize that branch-and-bound is only deceitfully easy to parallelize, and good scalability is in general very hard to achieve. An excellent and very recent overview of the current state of the art with respect to solution of Mixed-Integer Programming (MIP) problems in parallel is given in [2]. Parallelizing state-of-the-art MIP solvers in particular appears to be extremely challenging, for many different reasons:

- Sophisticated solvers are in general inherently harder to parallelize, as they exploit a wide range of algorithmic techniques whose purpose is to limit the size of the search tree. Most obviously, MIP solvers spend a large amount of time in the shallowest nodes of tree, in particular at the root node.
- MIP solver performance is very dependent on the order in which nodes in the tree are processed. Replicating such order in a parallel algorithm is difficult and requires additional coordination, that limits the scalability of the method.
- The shape of the search tree is not known a-priori as the tree is constructed dynamically. In addition, the shape of the tree can be considerably different from instance to instance and is, most of the times, unbalanced. This poses additional challenges for balancing the workload among workers, i.e., making sure they are all busy doing useful computations.
- MIP solvers collect and exploit a wide amount of global information, which need to be shared among workers in order to avoid the parallel algorithm to perform redundant computations.
- MIP users often rely on the overall algorithm being deterministic, i.e., the solver will produce the very same tree and solution when fed with the same input data and run on the same hardware. This is harder to achieve in general for parallel branch-and-bound codes, and requires additional coordination and synchronization, again at the expense of scalability.

Because of the above, load balancing (i.e., the strategy used to move data around among workers to make sure that they are all busy doing useful work), has been the subject of active research in the last decades. The two main approaches to load balancing for tree-search algorithms[1] are *static load balancing*, in which information is moved only once at the very beginning of the algorithm, and *dynamic load balancing*, in which information is moved throughout the algorithm as needed.

[1] In a recently developed paradigm for parallel computations, *MapReduce* [3], the user is required to define a *map* function to be applied to logical input "records" to produce intermediate key/value pairs, and a *reduce* function to be applied to all objects that share the same key. The method is mostly suited for applications with a very large input and very predictable processing flow. As such, *MapReduce* is not a good candidate for tree-search algorithms, and will not be discussed further in this paper.

In a static load balancing scheme, each worker processes nodes independently and reports the final result when done. If the initial assignment of nodes to workers is unbalanced, the scheme will not redistribute the work, and some workers will stay idle until the whole process is done. The advantages of a static scheme are simplicity and little communication. The downside is that static schemes are in general less efficient, as there is no recourse to the initial assignment of workload even if it turns out to be uneven. Four different classes of static schemes are surveyed in [4], namely:

- *Root initialization:* One worker processes the initial part of the tree. Once enough open nodes are generated, they are broadcasted to the other workers for processing, according to some rules.

- *Enumerative initialization:* The root node is broadcasted to each worker, that then (redundantly) processes the initial part of the tree. When the number of nodes equals the number of workers, the i-th worker keeps the i-th open node and discard the rest.

- *Selective initialization:* The root node is broadcasted again to all workers. Each worker then generates a single path from the root, and then enumerates the corresponding subtree. Note that this requires a sophisticated scheme to ensure workers work on different parts of the tree without overlap and without missing any node (for correctness). This is the approach proposed in [5].

- *Direct initialization:* Each worker directly creates a node from a certain depth of the tree, and then enumerates the corresponding subtree. This is a viable approach only when the search tree structure is known in advance. A similar approach was recently proposed [6] to parallelize limited discrepancy search (LDS, see [7]), where the leaves of the complete LDS tree are deterministically assigned to the workers, and each worker processes a subtree only if it contains a leaf assigned to it.

The last two approaches (selective and direct initialization) cannot be easily added to a state-of-the-art MIP solver, if not at the expense of generating very artificial (and thus potentially much worse than the sequential counterpart) trees. For example, the shape of the tree usually depends on the nodes explored so far, and thus a naïve implementation of selective initialization would not even guarantee correctness of the algorithm. For these reasons, most static variants proposed in the literature for MIP branch-and-bound are somehow based on the first two approaches above, namely root and enumerative initialization. For example, [8] computationally evaluated the root initialization approach on simple branch-and-bound codes for three specific optimization problems (quadratic assignment, graph partitioning, and weighted vertex cover). This is also the approach taken by the GAMS framework described in [9], in which nodes are initially generated according to a best-estimate strategy. Enumerative initialization has been implemented in the framework PEBBL [10].

Different static schemes have also been implemented recently, like *bet-and-run* [11, 12] and the *racing ramp-up* scheme of the UG framework [13].

In a dynamic load balancing scheme, workload is reallocated to the available workers whenever needed. This of course requires communication among the workers, but can be instrumental to keep the work balanced. Dynamic schemes have been the subject of extensive research in the field, and many schemes have been proposed in the literature, with varying degrees of generality and success. Broadly, dynamic strategies can be categorized based on the degree of centralization of the mechanism: centralized schemes involving one or more *managers* are generally called *work-sharing* schemes, whereas schemes in which transfers are initiated by individual workers are usually called *work-stealing*; see, for example, [14, 15, 16]. Clearly, dynamic strategies can require a significant amount of communication and synchronization among the workers.

Different (hybrid) strategies for a parallel implementation of tree search algorithms have been proposed and developed during the years [17], including the work-stealing approach where the set of active nodes is periodically distributed among the workers [18, 19, 13, 10, 20, 21], thus requiring an elaborated load balancing strategy. Depending on the implementation, this may yield a deterministic or a nondeterministic algorithm, with the deterministic option being in general less efficient because of synchronization overhead. In any case, a non-negligible amount of communication and synchronization is needed among the workers, with negative effects on scalability [22, 23].

A notable example of a hybrid strategy for CP solvers was recently developed in [24]. In this approach, a master problem enumerates the partial solutions associated with a subset of the variables of the problem to solve (an approach similar to root initialization). Each such partial solution is then processed by a worker in a second step; the number of variables to consider is chosen so as to have significantly more subproblems than workers. All subproblems are put into a queue and dynamically distributed to workers as needed (usually, a subproblem is assigned to a given worker as soon as the worker is idle).

In [25], we showed how to modify a given deterministic (sequential) tree-search algorithm to let it run on a set of, say, K identical workers. The approach, called `SelfSplit`, is in the spirit of [8] and is based on the following main features:

1. each worker works on the whole input data and is able to autonomously decide the parts it has to process;
2. in the initial *sampling phase*, all workers execute exactly the same search and generate the same tree nodes;
3. after the sampling phase, each worker skips the open nodes that "belong to the other workers" (according to some deterministic rules);
4. no communication between the workers is required (besides the final selection of the best solution);
5. the resulting algorithm can be implemented to be deterministic;
6. in most cases, the modification only requires a few lines of codes.

Note that `SelfSplit` assumes that all workers are identical (i.e., use the same hardware and OS) as the initial sampling phase must be identical for all workers; this situation arises, e.g., in a shared-memory multi-core machine, or in a cluster of identical machines. `SelfSplit` can be considered a case of enumerative initialization, as the initial part of the tree is redundantly constructed in parallel by all workers; a key feature, however, is that it does not stop as soon as the number of open nodes equals the number of workers, but it continues in order to have significantly more nodes, as done in [8, 24].

The results reported in [25] show that `SelfSplit` is in fact very well suited for Constraint Programming applications. Indeed, `SelfSplit` was implemented within the CP solver Gecode 4.0 [26], and tested on several instances taken from the repository of modeling examples bundled with Gecode. As the goal of [25] was to measure the scalability of the method, only some specific instances were addressed, namely instances which are either infeasible or in which one is required to find all feasible solutions – as the parallel speedup for finding a first feasible solution can be completely uncorrelated to the number of workers, making the results hard to analyze. The `SelfSplit` algorithm was ran with number of workers $K \in \{1, 4, 16, 64\}$. Each worker was configured to use only a single thread, to guarantee a deterministic behavior during the sampling phase, and hence the correctness of the algorithm. According to Table 1 (taken from [25]) even on moderately easy instances, that can be solved less than one minute, `SelfSplit` can achieve an almost linear speedup with up to 16 workers, and the speedup is still good for $K = 64$. On harder instances, the method scales almost linearly also with 64 workers. In all cases, the resulting algorithm is deterministic.

	time (s)	speedup		
instance	$K = 1$	$K = 4$	$K = 16$	$K = 64$
`golomb_12`	41.5	3.84	14.31	41.50
`golomb_13`	1,195.8	4.00	15.67	57.49
`golomb_14`	19,051.9	3.97	15.71	61.34
`partition_16`	30.0	3.75	13.64	46.15
`partition_18`	354.8	3.90	14.78	54.58
`partition_20`	4,116.4	3.86	15.64	59.40
`ortholatin_5`	29.3	3.89	13.95	36.63
`sports_10`	98.7	3.91	14.51	44.86
`hamming_7_4_10`	32.3	3.85	14.04	40.38
`hamming_7_3_6`	2,402.4	3.91	15.44	59.76

Table 1: Speedups for the Constraint Programming solver *Gecode*, taken from [25].

Because of lack of communication among workers, one could argue that `SelfSplit` is not suitable for solvers that collect/learn important global information during the search, as this information can be crucial to reduce the search tree. A notable example is Mixed-Integer Linear Programming (MILP), whose

solvers are indeed very hard to parallelize. We note however that most MILP solvers do collect their main global information (cuts, pseudocosts, incumbent, etc.) in their early nodes, i.e., during sampling, thus all such information is automatically available to all workers. Therefore, performing the sampling phase redundantly in parallel by all workers has the advantage of sharing a potentially big amount of global information without communication—a distinguishing feature of `SelfSplit`.

In the present paper we investigate the potential of `SelfSplit` in a MILP context. We first report artificial experiments aimed at studying the `SelfSplit` behavior "in vitro".

We then address the actual parallelization of both ad-hoc and general purpose MILP solvers, namely (i) the branch-and-bound code for the Asymmetric Traveling Salesman Problem (ATSP) of [27], (ii) the more-sophisticated ATSP branch-and-cut algorithm given in [28] and in [29], and (iii) the specific branch-and-cut code for uncapacitated facility location [30] that uses MILP callbacks. In all the above cases, `SelfSplit` appears as the only viable option to exploit a cluster of computers without a complete code redesign. In addition, we address (iv) the general-purpose commercial MILP solver `IBM ILOG CPLEX 12.6.1` and compare its internal distributed versions, as well as some reference static methods, with `SelfSplit`.

Computational results are reported, showing that `SelfSplit` performs very well on codes (i), (ii) and (iii) above, and has a reasonably good performance on (iv) as well. Though better results could probably be obtained by using more sophisticated approaches, considerable speedups can be obtained using a simple scheme such as `SelfSplit`, that requires (almost) no communication among the workers and only very marginal changes to the deterministic algorithm to be parallelized.

The outline of the paper is as follows. Section 2 reviews the basic `SelfSplit` algorithm, along with possible variants aimed at improving load balancing. Sections 3 and 4 study the effect of different `SelfSplit` parameters on the resulting load balancing. Section 5 reports computational results in various MILP settings. Finally, in Section 6 we draw some conclusions and outline possible directions of future work.

2. The `SelfSplit` paradigm

`SelfSplit` addresses the parallelization of a given deterministic algorithm, called the *original algorithm* in what follows, that solves a given problem by breaking it (recursively) into subproblems called *nodes*. The main `SelfSplit` features are outlined below; more details can be found in [25]. The `SelfSplit` scheme is as follows:

a) Each worker reads the original input data and receives an additional input pair (k, K), where K is the total number of workers and $k \in \{1, \ldots, K\}$ identifies the current worker. The input is assumed to be of manageable size, so no parallelization is needed at this stage.

b) The same deterministic computation is initially performed, in parallel, by all workers. This initial part of the computation is called *sampling phase*. No communication at all is involved in this stage. It is assumed that the sampling phase is not a bottleneck in the overall computation, so the fact that all workers perform redundant work introduces an acceptable overhead.

c) When enough open nodes have been generated, the sampling phase ends and each worker applies a deterministic rule to *color* them, i.e., to identify (and skip) the nodes that do not belong to it as they will be processed by other workers. No communication among workers is involved in this stage. It is assumed that processing the subtrees is the most time-consuming part of the algorithm, so the fact that all workers perform non-overlapping work is instrumental for the effectiveness of `SelfSplit`.

d) When a worker ends its own job, it communicates its final output to a *merge worker* that process it as soon as it receives it. The merge worker can in fact be one of the K workers, say worker 1, that merges the output of the other workers after having completed its own job. We assume that output merging is not a bottleneck of the overall computation, as it happens, e.g., for enumerative algorithms where only the best solution found by each worker needs to be communicated.

It is worth stressing here that the `SelfSplit` scheme is not fault tolerant and that it strongly depends on the assumption that all processors are identical and do in fact generate precisely the same tree in the sampling phase. This assumption applies to a shared-memory multi-core machine, or to a tightly controlled cluster of identical machines. In the latter case, it is crucial to ensure that all machines in the cluster are in fact absolutely identical. The described `SelfSplit` framework provides no check to ensure all processes produce identical results in the sampling phase, though some kind of hash function could be implemented to provide such a check. Doing sampling on a single processor and then moving the instances to remote machines afterwards is another possibility which is not addressed in the present paper.

In its simplest "vanilla" version, step c) is implemented by just assigning each open node n a (deterministic) pseudo-random integer $c(n) \in \{1, \ldots, K\}$, with the rule that each worker k will skip all nodes n with $c(n) \neq k$.

Note that all steps but d) requires absolutely no communication among workers. `SelfSplit` is therefore well suited for those computational environments where communication among workers is unreliable or time consuming.

Though very desirable, the absence of communication implies the risk that workload is quite unbalanced, i.e., lucky and unlucky workers can complete their computation at very different points in time. To contrast this drawback, `SelfSplit` follows the recipe of [8, 24] to keep a significant number of open nodes for each worker after sampling.

`SelfSplit` is straightforward to implement if the original algorithm is sequential, and the random/hash function used to color a node is deterministic

and identical for all workers. The algorithm can however be applied even if the original algorithm is itself parallel (and, of course, deterministic), provided that the pseudo-random coloring in Step 3 is done at the proper time.

A more elaborate version (see [25] for details), aimed at improving workload balancing among workers, can be devised using an auxiliary queue S of *paused nodes* for each worker (so, there is no need for communication). The modified algorithm reads as follows:

1. As before, two integer parameters (k, K) are added to the original input.
2. A paused-node queue S is introduced and initialized to empty.
3. Whenever the modified algorithm is about to process a node n, a boolean function NODE_PAUSE(n) is called: if NODE_PAUSE(n) is true, node n is not solved and it is moved into S and the next node is considered; otherwise the processing of node n continues as usual and no modified action takes place.
4. When there are no nodes left to process, the *sampling phase* ends. All nodes n in S, if any, are popped out and assigned a color $c(n)$ between 1 and K, according to a deterministic rule. Again, coloring is done independently by each worker.
5. All nodes n whose color $c(n)$ is different from the input parameter k are just discarded. The remaining nodes are processed (in any order and possibly in a nondeterministic way) till completion.

With respect to its vanilla counterpart, the coloring phase in Step 4 has more chances to determine a balanced workload split among the workers than its vanilla counterpart, at the expense of a slightly more elaborate implementation. In the paused-node version, both the decision of moving a node into S as well as the color actually assigned to a node are based on an estimate of the computational difficulty of the node. The idea is to move a node in S if it is expected to be significantly easier than the root node (original problem), but not too easy as this would lead to an exceedingly time-consuming sampling phase. Estimating the difficulty of a (sub-)problem is a topic well-studied in the literature, both for specific classes of models, as done for example for QAP in [31], and for general MIPs, as done in [32, 33, 34]. However, most methods are rather computationally intensive, while for the purposes of `SelfSplit` we need something cheap and that can be computed using node-local information only. For these reasons, within NODE_PAUSE, a rough estimate of the difficulty of a node is obtained in [25] by computing the logarithm of the cardinality of the Cartesian product of the current domains of the integer variables, to be compared with the same measure computed at the end of the root node. For problems involving binary variables only, this figure coincides with the number of free variables at the node. To cope with the intrinsic approximation involved in this estimate, the following adaptive scheme can be used to improve `SelfSplit` robustness. At the end of the sampling phase, if the number of nodes in S is considered too small for the number K of available workers, then the internal parameters of NODE_PAUSE are updated in order to make the move into the queue S less likely. The sampling procedure is then continued after

putting the nodes in S back into the branch-and-bound queue—or the overall method is just restarted from scratch.

As to node coloring, the color $c(n)$ associated with each node n in S can be obtained in three steps: (1) compute a score estimating the difficulty of each node n, (2) sort the nodes by decreasing scores, and (3) assign a color c between 1 and K, in round-robin, so as to split node scores evenly among workers.

3. Statistical Load Balancing

As `SelfSplit` is intended to be a communication-free scheme, it can achieve a proper load balancing only in a statistical sense. Intuitively, if all subproblems have a similar level of difficulty, we can hopefully balance the workload assigned to the workers. In addition, the larger the number of subproblems that are assigned to each worker, the higher the chances that all workers will have to perform a comparable work. Although basic probability theory guarantees that this happens for very large numbers only, in practice we need to check whether a reasonable load balancing can be obtained (on average) even with "small" numbers – the method would clearly be unpractical if millions of subproblems had to be assigned to each worker to obtain a good balancing.

In order to computationally evaluate whether statistical load balancing is a viable option, we performed a preliminary artificial experiment, randomly generating a number of *virtual* subproblems with different difficulties and assigning them to workers. The experiment works as follows:

- randomly generate N (say) integer numbers from a given distribution. In our scheme we fixed $N = K \times M$, where K is the number of workers and M is a parameter counting the number of subproblems that we would like to assign to each worker. These random numbers measure the computational difficulty of each subproblem expressed, e.g., in terms of enumeration nodes;
- randomly assign M subproblems to each worker; and
- evaluate the speedup that we would obtain with this assignment, w.r.t. a single worker scenario in which a single worker has to process all the nodes of all subproblems.

Note that this is admittedly a gross simplification of a real-world scenario for many reasons:

- We are completely ignoring the overhead needed to generate the subproblems (sampling phase).
- We are assuming that all nodes require the same computational effort, regardless of their location in the tree and the order in which a worker processes them. Thus, we are also implicitly assuming that no interaction whatsoever exists between the subproblems assigned to the same worker.

- We are assuming that the set of nodes evaluated would be same in both parallel and sequential cases. Almost no practical algorithm achieves this.

On the other hand, our simple experiment is computationally cheap (hence it can be repeated several times with different random seeds) and side-effects free, so it might provide meaningful insights into the problem. For each choice of the parameters (namely, type of random distribution, K, and M), we repeated the simulation $10,000$ times and recorded the population of speedups, so as to evaluate the speedup as a random variable for which we can compute classical statistical measures, e.g., average, standard deviation and percentiles.

The most natural choice for the initial distribution would be an "ideal" distribution that yields subproblems of (almost) the same difficulty; however, such a policy can only be a very rough approximation of real situations, mainly for two reasons:

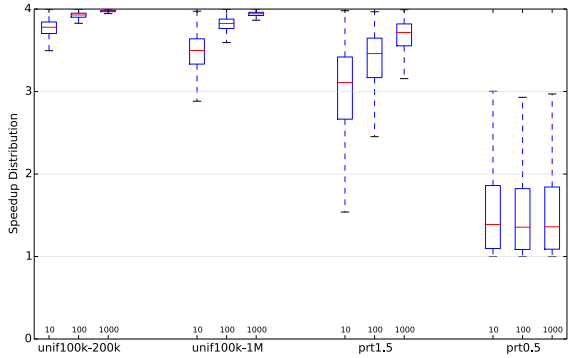
1. it is very difficult to accurately predict the size of the search tree associated to a given subproblem starting from the pieces of information available before solving it;
2. even if we were able to predict the "inherent" difficulty of a given subproblem, the underlying solver may still considerably vary in performance, because of performance variability.

Thus, we chose two random distributions, namely a uniform distribution and a Pareto distribution.

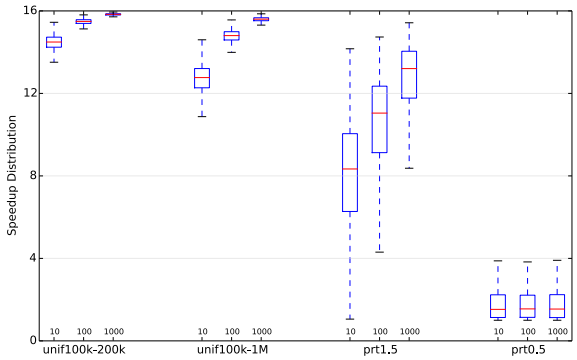
The uniform distribution models the case in which subproblems are *well-behaved* and their difficulty varies uniformly in a given range. In our experiments, we considered two possible ranges, namely $[100k, 200k]$ and $[100k, 1M]$, which correspond to the cases in which our estimates can be wrong by a factor of 2 and 10, respectively.

The Pareto distribution, on the other hand, models the more realistic case in which subproblems follow an heavy-tail distribution, which does arise for some classes of problem solved by enumerative algorithms (see [35]). In this case, there is no bound on how wrong our estimates can be (even worse, arbitrarily bad subproblems show up with non negligible probability). The general Pareto (type I) distribution has a probability density function defined as $\alpha/x^{\alpha+1}$, where parameter α controls the heavy-tailedness of the distribution. In particular, the distribution has no finite moments of order $n \geq \alpha$ (thus, the lower the α, the more heavy-tail the distribution). In accordance with [35], in our experiments we chose $\alpha = 0.5$ and $\alpha = 1.5$. Note that we are not claiming that all MILP instances fall into one of the distributions that we have chosen, nor that those distributions are comprehensive, in the sense that they cover all possible cases. We have chosen those two as extreme cases of behaviours that can actually happen in practice.

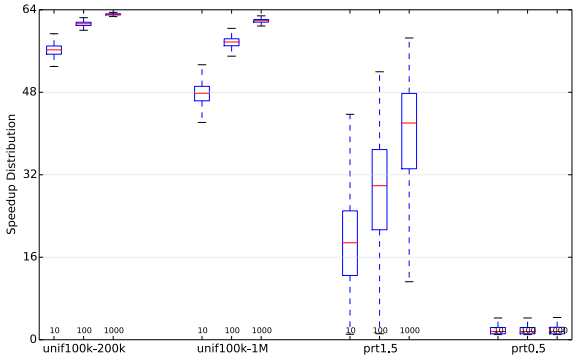
The results of our tests are depicted in Figure 1, where we show boxplots [36] for the chosen four distributions, for $M = 10,100$ and $1,000$, and for $K \in \{4, 16, 64\}$. Each box extends from the lower to the upper quartile, while the horizontal line within each box shows the median of the corresponding population. The whiskers extend from the box to show the range of the data.



(a) $K = 4$



(b) $K = 16$



(c) $K = 64$

Figure 1: Boxplots for statistical load balancing for different values of the number of workers.

Our results show that the distribution of the speedup is highly influenced by the distributions from which the subproblems sizes are drawn, whereas marginal effects are experienced when changing the values of M, provided the latter be "large enough". Indeed, while almost linear speedups can be obtained (on average) starting from a uniform distribution or from a very mildly heavy-tailed one, that is not the case with strongly heavy-tailed distributions, even assigning $1,000$ subproblems to each worker. Interestingly, the subproblem distributions also affect the variance of the speedup population, with the most heavy-tailed ones leading again to the worst results. Indeed, for these distributions, parallelizing a single enumeration tree via a static load balancing method seems to be problematic, while a bet-and-run approach [11] is likely to provide better results. Finally, the number of workers K is a marginal factor when "well-behaved" distributions are concerned, but turns into a very important one with heavy-tailed distributions. This shows that, even in this simplified experiment, the average speedup that can be obtained does not scale very well with the number of workers in the heavy-tailed case.

The main conclusion of this artificial experiment is a further confirmation that a static load balancing method like `SelfSplit` can yield reasonable speedups only on relatively well-behaved classes of problems, while it is most likely to fail for instances coming from strongly unbalanced distributions. Because of the gross simplifications of the artificial experiment, however, it is not possible to predict from it at which point a static method will stop providing reasonable speedups. Note that more sophisticated analyses can be found in the literature (e.g., in [37]), but none of them provides a close-enough approximation of a real MILP solver. Hence we decided to not pursue this approach further.

4. Node-pausing strategies comparison

Different criteria can be used to implement the NODE_PAUSE function. Preliminary computational tests showed that measuring the effectiveness of such a criterion can be very difficult (or even misleading) if directly implemented within a solver. The reason is that many state-of-the-art implementations of enumerative methods exhibit a high performance variability, and differences in the shape of the tree can be caused by changes that are supposed to be performance neutral. The issue is particularly severe in the MILP case [38], but any solver that bases some of its decisions on pieces of information collected during the search itself—such as pseudocosts, nogood clauses, or primal bounds—is sensitive to this phenomenon.

To compare different strategies in a clean environment, we set up a further artificial experiment as follows. Using the callback mechanism of a commercial MILP solver, namely `IBM ILOG CPLEX 12.5.1`, we collected the branch-and-cut trees enumerated when solving all the instances in the MIPLIB 2010 testbed. The solver was run with default options, in single-thread mode; for each instance we provided the optimal (or best known) solution on input to the solver, while disabling primal heuristics to minimize their variability effect. For each run,

we used callback functions to store the associated enumeration tree, saving for each node some relevant informations, e.g., indices of descendant nodes, domain of integer variables, depth and alike. Then we discarded all the trees with less than $500,000$ nodes, remaining with 32 instances. In this way we collected and stored a number of *frozen enumeration trees* that are significant (both in terms of size and shape) for our experiments. We want to stress that, in these artificial experiments, we are not interested in computing times, and that we take no action that may interfere with the behaviour of the solver, thus keeping the frozen trees as close as possible to the default ones.

We then *simulated* the behavior of different NODE_PAUSE criteria on the frozen trees. This experimental environment has several advantages:

- all strategies are compared on the same set of trees;
- with most node-pausing strategies, the final partitioning is independent of the order of visit of the nodes;
- it is computationally very cheap to test many alternative strategies, as no real enumeration is taking place—only tree visits.

On the other hand, the environment is admittedly artificial, and relies on assumptions that are likely to be violated in practical implementations. In particular, as in the previous section, we are assuming that both sequential and parallel codes will enumerate the very same tree for a given instance. In addition, speedup is computed by considering the size of the subtrees assigned to the different workers, again implicitly assuming that all nodes take the same amount of time to process. As such, the speedups we measured are only a rough approximation of the real speedup in computing time. Still, this is a finer approximation than the one given in the previous section, as we are considering "real" enumeration trees.

We compared two simple criteria for estimating the difficulty of a given node n, to be used within the boolean function NODE_PAUSE():

- `volume`: we compute the volume of the Cartesian product of the domains of the integer variables in the model, i.e.,

$$V(n) = \prod_{j \in J} (u_j - l_j + 1) \tag{1}$$

  where J denotes the set of integer variables and $\{l_j, \ldots, u_j\}$ is the domain of variable j at node n. This measure is compared against the one computed at the end of the root node, say $V(1)$. Value NODE_PAUSE=`true` is returned if the ratio $V(1)/V(n)$ is above a given threshold, say ρ. To avoid overflow, in the implementation we consider the logarithms of the above expressions.

- `depth`: value NODE_PAUSE=`true` is returned if the node depth in the tree is larger than a given threshold θ.

Note that we do not need to maintain timestamps for the individual nodes: as the trees are fixed, and the strategies we compare are based on node-local information only, we just need to visit the trees (in any order) to simulate the effect of the different NODE_PAUSE strategies. Different NODE_PAUSE strategies will generate different frontiers S, i.e., open nodes n to be assigned to workers. Before actually assigning them, we sort the nodes according to their expected difficulty. We considered the following options:

- `ideal`: Sorting (by decreasing order) with respect to the actual number of nodes in the subtree rooted at n. Of course, this number is not available in a real implementation, but it is nevertheless interesting to consider what would happen if we had a perfect oracle able to predict the size of the subtrees.

- `score`: Sorting (by increasing order) with respect to the score

$$\text{score}(n) = 10^3 \cdot \text{dualBound}(n) + \text{depth}(n)$$

- `volume`: Sorting (by decreasing order) with respect to the volume given by the Cartesian product of the domains of the integer variables in the model, defined by (1).

- `random`: nodes are just shuffled according to a random permutation.

Note that, in this phase, the difficulty measure has a different purpose than in the NODE_PAUSE function. Indeed, within the NODE_PAUSE function we are not interested in ranking nodes, but rather to assess whether a given node has become "easy" enough. Conversely, here we need a ranking among nodes, hence the different strategies.

Finally, we considered two strategies for coloring the nodes:

- `offline`: nodes are assigned to the workers in round-robin, as in the `SelfSplit` paradigm, and no communication or synchronization is needed.

- `online`: nodes are assigned to the first available worker by a master scheduler. Note that, again, we use the size of the subtrees as a proxy for time, so the next subtree is assigned to the worker whose sum of nodes assigned to it so far is a minimum. This is supposed to yield a better load balancing, at the expenses of communication between the workers. Note that this is exactly the kind of communication that we would like to avoid when using our self-splitting framework; nevertheless it is interesting to measure how much of the theoretical speedup is lost when forbidding any synchronization among workers.

As to the parameter ρ or θ used by NODE_PAUSE() in its `volume` or `depth` case, respectively, for each instance we tried several candidates and kept the best one. This is equivalent to assuming that NODE_PAUSE() is able to automatically determine the best parameter for each instance (the so-called"Virtual

	offline		online	
	volume	depth	volume	depth
ideal	10.77	10.20	12.51	12.65
score	10.25	9.78	12.31	12.44
volume	9.87	9.32	11.91	12.06
random	9.92	9.42	11.57	11.31

Table 2: Comparison of the two methods (speedups with 16 workers).

Best Case" scenario). The outcome of this preliminary experiment is reported in Table 2 for the case of $K = 16$ workers.

A first comment about the results in Table 2 is that all options produce a speedup of about 10 or more (out of 16 workers), which is an extremely good figure. Of course, those numbers must be taken with great care, as they are the result of an artificial experiment on a limited set of instances chosen to need big enumeration trees. Still, they indicate that *if* we could enumerate the very same tree in parallel that we would do in the sequential case, then we could reasonably achieve good speedups even with a static load balancing method like `SelfSplit`. Note also that, for both `volume` and `depth`, the internal parameter (ρ or θ) is chosen *a posteriori* in an optimal way, instance by instance, while a practical implementation can only guess it (or use a dynamic update policy): this also explains the somewhat optimistic figures obtained in the experiment.

As expected, the `online` approach works better than its `offline` counterpart, the speedup difference being interpretable as the "price of communication" that `SelfSplit` has to pay to avoid any communication among workers. This difference is of about 20%, which is not negligible, but reasonable considering that we are also not paying the overhead incurred by a deterministic parallel method with synchronization points.

A surprising result is instead that the sorting option `ideal` is only slightly better than its `score` counterpart, meaning that a perfect estimate of the node difficulty is not really required in our setting. In addition, using the same formula to evaluate node difficulty in function NODE_PAUSE() and for sorting of nodes seems not to be effective, and suggests the adoption of two distinct criteria. An explanation is that a statistical workload balancing among workers occurs in any case, making the node difficulty estimation less critical. This is confirmed by the fact that the `random` option leads to a performance deterioration of just 10%, which is much less than what we would have expected.

5. `SelfSplit` for Mixed-Integer Linear Programming

We next address three different applications of `SelfSplit` to parallelize enumerative MILP codes of different degree of complexity (and performance variability). All the experiments of this section were executed on a cluster of identical Intel Xeon E3-1220V2 machines running at 3.10 GHz, with 16GB of RAM each, in single thread mode.

5.1. Parallelization of a sequential ATSP branch-and-bound code

Our first exercise was to apply `SelfSplit` to the sequential branch-and-bound code of [27] for the Asymmetric Traveling Salesman Problem (ATSP). This is an optimized yet legacy FORTRAN code of about 3,000 lines based on the following main ingredients: (i) lower bounds are computed by a very efficient parametrized code for the assignment problem relaxation, (ii) branching is based on subtours, and can produce more than two children per node, (iii) a best-bound-first tree exploration strategy is used; see [27] for details. We implemented two variants of `SelfSplit` where:

a) The optimal solution value is provided on input to the solver;

b) The value of the overall best incumbent is periodically written/updated on a single global file; each worker periodically reads it and only uses it to possibly abort its own run (just 46 new lines of code added to the original code).

In all cases, we implemented `SelfSplit` in its simplest variant, ending the sampling phase when the number of open nodes in the branch-and-bound tree was equal to $1,000$.

According to our preliminary computational tests, variant b) often achieves a more-than-linear speedup in that `SelfSplit` is able to find the optimal ATSP solution much quicker than the sequential version, thus saving a considerable number of nodes. This is because, after sampling, diversified solution subspaces are searched in parallel by the K workers, increasing the chances of early finding good feasible solutions. The improved behavior of `SelfSplit` as a heuristic is of course a positive side effect of our method, however it is not strictly related to parallelization as it could be simulated by a sequential implementation not using the rigid best-bound search strategy of the original ATSP code. To better evaluate the speedup coming from the parallel processing of tree nodes, we therefore decided to concentrate on variant a) above.

Our experiments are aimed at evaluating the speedup that `SelfSplit` can achieve on difficult instances requiring a large number of tree nodes, given for granted that for easy instances no parallelization technique working at the tree-search level can produce interesting speedups. Our testbed then contains instances randomly generated as in class B in [27], which were the hardest instances for our code among those considered by [27]. Namely, the cost of each arc (i, j) in the graph is defined as $c_{ij} = \sigma_{ij} + \alpha_{ij}$ where $\sigma_{ij} = \sigma_{ji}$ and α_{ij} are uniformly random integers in $[1, \ldots, 1,000]$ and in $[1, \ldots, 20]$, respectively. We randomly generated 100 instances with 200 vertices and 100 instances with 250 vertices, and solved all of them with the branch-and-bound code of [27] in sequential mode, providing the optimal solution value on input. Then, we disregarded all instances that turned out to be "too easy" with these settings, i.e., that could be solved in less than 1,000 wall-clock seconds on our machine. The resulting benchmark was finally composed of 20 instances—6 with 200 nodes and 14 with 250 nodes. Note that our instance filtering policy (though based on the performance of the sequential solver alone) is not unfair, as we are not

comparing the behavior of different solvers but the speedups resulting from the parallelization of a single solver.

Table 3 gives the outcome of our experiments and reports:

- the average (in geometric mean) computing time for the sequential version of the code, and
- the average speedups (geometric mean) that were obtained with different values of K with respect to the sequential version of the code, again in terms of computing time.

time (sec)	time speedup			
$K = 1$	$K = 4$	$K = 16$	$K = 32$	$K = 64$
1,504	7.76	13.37	21.56	30.55

Table 3: Parallelization of a sequential ATSP branch-and-bound code.

Figures of Table 3 show that an almost linear speedup can be obtained with up to 16 workers, whereas some saturation occurs for larger values of K. Taking into account that only a very minor code change was implemented, the `SelfSplit` performance on these instances can be considered very good.

Our results confirm the findings of [8], namely, that simple branch-and-bound codes for specific applications can effectively be parallelized by no-communication paradigms like `SelfSplit`. Indeed, the experiments reported in Section 3 suggest that this is mainly due to the low performance variability experienced by simple enumeration codes—that by the way makes them rather appealing in a massively distributed setting.

Finally, we observe that parallelization of branch-and-bound algorithms for ATSP has already been tested in the literature. The first attempt was presented in [39] with a synchronous master-slave approach that yields poor results. A parallel branch-and-bound based on the assignment problem lower bound and on a subtour elimination branching rule was presented in [40, 41]. In their algorithm, multiple workers can cooperate either in exploring a single node of the enumeration tree or in simultaneously solving different nodes; in any case a considerable communication is required among workers. Using an architecture with $K = 14$ processors on randomly generated instances, this algorithm reported an average speedup from 0.85 to 11.24 depending on the size of the instance (from 50 to 500 cities).

5.2. Parallelization of a sequential ATSP branch-and-cut code

We then addressed a more sophisticated ATSP code, namely the sequential branch-and-cut code of [28] and of [29]. This is FORTRAN code of about 10,000 lines where node bounds are computed through an LP solver (`IBM ILOG CPLEX 12.6.1` in our case). Various classes of facet-defining ATST cuts—including SEC's, SD's, and DK's (see, [42] and [43, 44, 45])—are separated at each

tree node, and variables are dynamically generated and/or fixed according to a Lagrangian pricing mechanism. A best-bound tree exploration is implemented, and primal heuristics are applied for an early update of the incumbent.

In this more challenging setting, a paused-node version of `SelfSplit` was implemented, according to the following trivial scheme that required just 11 new lines of code to implement. Each time a node is popped from the active-node queue, we count the number of open nodes. As soon as this number becomes larger than NO (say), the sampling phase ends and the node lower bound is used as the node-difficulty measure used for node coloring. In our implementation we set $NO = \max\{160, 5K\}$, where parameters 160 and 5 (not tuned to avoid overfitting) are intended to guarantee a reasonable number of nodes for each worker—which is nontrivial, as branch-and-cut codes for specific problems tend to produce a small number of open nodes due to the effectiveness of their ad-hoc cutting planes.

We considered randomly generated instances with $n = 200$ and $n = 250$ nodes. According to [28], the most challenging instances for our sequential branch-and-cut code correspond to instances of type C in [27], in which the coordinates of each node are randomly generated in $[1, \ldots, 1,000]$ and the cost of each arc (i, j) is given by $c_{ij} = d_{ij} + \alpha_{ij}$, where $d_{ij} = d_{ji}$ is the Euclidean distance between i and j (rounded down) and α_{ij} is a uniformly random integer in $[1, \ldots, 20]$.

As in the previous experiments, we ran our sequential code on all instances, providing the optimal ATSP solution value on input, and removed all those problems that were solved within 1,000 seconds. This produced a benchmark of 44 instances—22 with 200 nodes, and 22 with 250 nodes. Table 4 reports the same information as Table 3 for both the sequential and the `SelfSplit` version of our code.

time (sec)	time speedup			
$K = 1$	$K = 4$	$K = 16$	$K = 32$	$K = 64$
2,465	6.74	10.89	14.54	17.91

Table 4: Parallelization of a sequential ATSP branch-and-cut code.

In this case too, `SelfSplit` produces considerable speedups for $K \leq 16$, though the current figures are less impressive that those of Table 3. This behavior was not unexpected, as the branch-and-cut ATSP code has more variability than the branch-and-bound one addressed in the previous subsection, and we know from the previous discussions that variability can interfere with scalability. In addition, as already mentioned, branch-and-cut codes for specific problems typically produce a limited number of nodes, making `SelfSplit` less attractive due to the sampling-phase overhead incurred when several workers are available.

5.3. Parallelization of a branch-and-bound code based on callbacks

Our third set of experiments concerns the applicability of `SelfSplit` to a specific branch-and-cut algorithm implemented within a generic MILP solver through callbacks—an approach that became customary in recent years. In this setting, `SelfSplit` appears to be the only viable option to exploit a cluster of identical computers. Indeed, to the best of our knowledge, the presence of callbacks prevents the usage of the standard distributed version of a MILP solver.

We considered the state-of-the-art branch-and-cut code of [30] for the Uncapacitated Facility Location (UFL) problem with linear costs, that uses Benders decomposition and ad-hoc heuristics. The code is written in C language (about 5,000 lines including comments) and is built on top of `IBM ILOG CPLEX 12.6.1`. It includes a number of callbacks to derive Benders cuts (within the so-called `lazyconstraint` and `usercut callabacks`) and to implement ad-hoc heuristics (within the so-called `heuristic callback`). The code itself is multi-thread and exploits all the cores detected in the single computer where it is run.

We implemented a very general vanilla `SelfSplit` framework to be applied when enumeration takes place, i.e., after some preliminary deterministic heuristics are executed. To better exploit all the cores of each computer, each worker runs the MILP solver in its multi-thread mode. The overall framework is implemented to be deterministic only in the sampling phase and nondeterministic afterwards. This is consistent with the original code, which is nondeterministic, the only changes being the definition of parameters associated with the current worker, the total number of workers (K), the number of nodes in the sampling phase (say NSAMPLE), and the name of a syncronization file used to share the incumbent.

Our `SelfSplit` implementation works according to the following three passes.

We initially save the value of the MILP solver parameters that we need to change, and execute pass 1 (i.e., the sampling phase) running the MILP solver in deterministic mode and imposing the given node limit and a best-bound strategy. Note that this phase is fully deterministic, so it will be executed identically and independently by each worker. When the node limit is reached, the second pass can start.

In the second pass, we activate a dummy callback that just removes from the MIP tree all the open nodes that are not to be processed on the given worker, i.e., with a wrong color. When all such nodes are killed, we are ready for the third pass.

In the third (and last) pass, we deactivate the above `SelfSplit` additional callback, restore the original MILP solver parameters (including the node limit, reduced by the number of nodes already explored), and execute the MILP solver to process the open nodes left after the second pass. During this last run, from time to time the incumbent solution is written/updated/read on the synchronization file by using appropriate callbacks, i.e., we allow for a minimal communication among workers. As the original code is anyway nondeterministic, we actually use the read solution to possibly update the incumbent of each worker.

As a technical observation, it is worth mentioning that the above implementation can install its own callbacks only once, before pass 1, as (un)installing them at a later time would reset the internal problem and cancel the tree. Therefore we pass to the callbacks the pass number (1, 2, or 3) so each callback function knows when it has to be applied or it must be skipped. In addition, all `SelfSplit`-specific callbacks are installed on top of the callbacks already installed in the original code, meaning that each `SelfSplit`-specific callback eventually calls the original callback function (if not undefined) before its final `return` statement.

To evaluate the speedup obtained by the above `SelfSplit` implementation, we addressed a class of hard UFL instances from the literature. To be specific, we considered the Koerkel-Ghosh [46] instances `ga500c` and `gs500c`, some of which have been solved by [30] for the first time. Table 5 compares the performance of three codes:

- `ORIG`: the original (nondeterministic) code as implemented by [30], run on a single 4-core computer;
- `SS(1)`: our `SelfSplit` variant with just one worker ($K = 1$), run on a single 4-core computer;
- `SS(16)`: our `SelfSplit` variant with 16 workers ($K = 16$), run on a Blade cluster with 16 identical 4-core computers.

For all codes, the multi-thread version of `IBM ILOG CPLEX 12.6.1` was run on each 4-core computer. Both `SelfSplit` codes were run with NSAMPLE=1,000. For the first two codes and for each instance, the table reports the required number of nodes and the computing time (in wall-clock seconds). For `SS(16)`, we report the minimum and maximum computing time among all workers, and the total number of nodes (computed as the sum of the nodes for all workers). Note that the total number figures includes the 16,000 unavoidable nodes spent in sampling phase.

Reported speedups sp1 and sp2 are with respect to `ORIG` and `SS(1)`, respectively, the latter being in our opinion the most relevant one as it measures the efficacy of our node-distribution scheme. The final lines of the table report averages and geometric means.

As expected, `SS(1)` is significantly slower than `ORIG` as the former deviates from the default (best-tuned) strategy and cannot use the more efficient opportunistic mode during sampling. As to `SS(16)`, in all cases it is much faster than both `ORIG` and `SS(1)`, with speedups ranging from about 5 to 12. In particular, the speedup with respect to `SS(1)` is 9.07 on average, and 8.66 in geometric mean. Thus `SelfSplit` performs rather well in this setting.

5.4. Parallelization of a general-purpose MILP solver

Our last set of experiments concerns the applicability of `SelfSplit` to a fully general MILP solver. This is indeed the most challenging setting for our

	ORIG		SS(1)		SS(16)				
instance	time	nodes	time	nodes	time (min)	time (max)	nodes (tot)	speedup wrt ORIG	speedup wrt SS(1)
ga500c-1	4,330	65,643	4,175	67,085	275	875	103,002	4.9	4.7
ga500c-2	11,235	187,799	14,422	142,936	336	1,832	194,332	6.1	7.8
ga500c-3	11,214	156,055	25,542	222,400	403	2,088	222,263	5.3	12.2
ga500c-4	10,327	153,690	19,498	179,123	375	1,828	180,629	5.6	10.6
ga500c-5	10,558	197,417	17,777	203,943	330	1,964	246,407	5.3	9.0
gs500c-1	5,047	75,105	4,891	76,779	310	1,031	115,060	4.8	4.7
gs500c-2	5,296	81,622	9,919	99,203	292	1,053	126,971	5.0	9.4
gs500c-3	12,501	193,123	21,273	195,316	364	2,199	254,205	5.6	9.6
gs500c-4	8,247	127,432	18,066	170,671	374	1,529	180,073	5.3	11.8
gs500c-5	16,959	276,844	27,809	277,546	380	2,650	343,940	6.3	10.4
average	9,571	151,473	16,337	163,500	343	1,704	196,688	5.4	9.1
geo.mean	8,798	137,626	13,936	149,264	341	1,608	184,435	5.4	8.7

Table 5: Comparison among different versions of the UFL code.

method, due to the large performance variability experienced in many difficult MILP instances.

We implemented `SelfSplit` in its node-pause version using `IBM ILOG CPLEX 12.6.1` through callback functions. This implementation does not require an explicit queue S, as only the queue of the solver is used. In particular, it is enough to attach a flag to each node to indicate whether it is paused or not. In order to interfere as little as possible with the default solver strategies, while keeping the overall goal of having an effective and fast sampling step, we implemented a more sophisticated sampling phase for this general case. At each node n we compute the ratio $V(1)/V(n)$, where $V(n)$ and $V(1)$ give a measure of the domain at the current node and after the root node, respectively, as defined by (1): if this ratio is above a given threshold, say ρ, the node is paused. However, we actually pause a node only if its depth is above a given threshold (equal to 10 in our implementation) and in any case we do not pause any node during the first $2,000$ nodes. As explained in Section 2, during the sampling phase we always extract from the queue a non-paused node, if any. One additional detail is that, when selecting a node during sampling, we always follow the solver's default strategy if we are in a dive (unless we stumble upon a paused node), while if we are backtracking, we pick a non-paused node of minimal depth (i.e., we follow a breadth-first-with-diving policy). If the tree contains only paused nodes, and there is a sufficiently large number of nodes, say N, the sampling phase is terminated and colors are assigned to the nodes. Otherwise, the current value of ρ is increased by a quantity equal to δ (say), and function NODE_PAUSE is re-executed for each node. In our implementation we used the same parameters for all instances, namely $\rho = 5$, $N = 2,000$ and $\delta = 10$.

We decided to provide the optimal solution on input to the MILP solver and to disable all heuristics. We used this setting to have deterministic results, i.e., that do not depend on the actual instant in which each worker updates its own incumbent solution. On the other hand, in our view this is not too far from a production implementation involving some limited amount of communication,

in which the incumbent value is shared among workers.

In this set of experiments we addressed all the problems contained in the MIPLIB 2010 [38] library of instances. This testbed includes 358 problems arising from different sources and with different characteristics. In particular, observe that we may encounter both instances that require millions of nodes to be solved, as well as instances that can be solved with "small" enumeration trees.

Table 6 compares the performances of `IBM ILOG CPLEX 12.6.1` in its default settings (with empty callbacks) and of algorithm `SS(1)`, i.e., `SelfSplit` with 1 worker. The comparison is performed for 5 different random seeds separately, to mitigate the effects of performance variability of the MILP solver (see, e.g., [11]). For each random seed and algorithm, we report the number of instances solved to proven optimality and the geometric mean of the associated computing time—a time equal to $10,000$ was counted for those instances that were not solved to optimality for any reason (e.g., memory requirements or alike).

	`CPLEX(1)`		`SS(1)`	
seed	#opt	time	#opt	speedup
0	174	965.74	174	1.00
1	175	909.64	177	0.99
2	175	965.47	176	0.99
3	175	964.66	177	0.98
4	177	961.22	175	0.98

Table 6: `IBM ILOG CPLEX 12.6.1` vs `SelfSplit` in a single-worker setting.

The results of Table 6 show that the slowdown incurred when deviating from the MILP solver default node-selection policy (that `SelfSplit` changes during the sampling phase to improve workload balancing) was just 1-2% on average; this indicates that `SS(1)` achieves the fundamental goal of not interfering too much with the fine-tuned solver's strategies during the sampling phase, and is thus a very close approximation of a state-of-the-art commercial solver in its default setting.

We then considered the availability of a system with 16 single-thread machines (each acting as a worker) and compared different ways to exploit this architecture without communication. Using the default random seed of the solver (namely, seed 0), we then compared the reference solver, `IBM ILOG CPLEX 12.6.1` with one thread, which we will denote as `CPLEX(1)`, against `SelfSplit` with $K = 16$ (denoted by `SS(16)`), and against a simpler static method inspired by the literature. The simpler static method behaves as follow:

- It forces a breadth-first node selection strategy during the sampling phase.
- It stops as soon as N open nodes are available, where N is an input parameter.

- When sampling is done, nodes are assigned to the available workers in a round-robin fashion.

We tested the scheme above using $N = 16$ and $N = 2000$. In the following, the corresponding algorithms will be denoted as `STATIC_16` and `STATIC_2000`, respectively.

We report aggregated results on this comparison in Table 7. As usual, for each method, we report the number of instances solved to optimality. In addition, the table gives the shifted geometric mean of the computing time for the reference solver, and the speedups for the remaining methods. To evaluate how the different methods behave as the instances get harder to solve, we divided the instances into different subsets, based on their hardness. To avoid any bias in the comparison, the level of difficulty is defined by taking all the solvers under comparison into account. First, the set `all` is defined by keeping all the models but those for which one of the solvers encountered a failure of some sort or where numerical difficulties led to different optimal objective values for the two solvers (both values being actually correct due to feasibility tolerances). Then, we considered all the instances that were solved by at least one solver, and further divided the testbed into 5 subclasses $[n, 10k]$ (for $n \in \{0, 1, 10, 100, 1k\}$), each containing the subset of models for which at least one of the solvers took at least n seconds.

According to the table, the more sophisticated sampling strategy of `SS(16)` indeed pays off, and `SS(16)` clearly outperforms the simpler static methods. In addition, the gap between `SS(16)` and the static methods increases significantly as we consider the subset of harder models.

	`CPLEX(1)`		`SS(16)`		`STATIC_16`		`STATIC_2000`	
class	#opt	time	#opt	speedup	#opt	speedup	#opt	speedup
all	81	1399.52	89	1.30	83	1.11	86	1.07
$[0, 10k]$	81	172.09	89	1.76	83	1.26	86	1.15
$[1, 10k]$	74	221.37	82	1.83	76	1.28	79	1.16
$[10, 10k]$	59	421.24	67	2.06	61	1.34	64	1.20
$[100, 10k]$	38	1214.74	46	2.62	40	1.38	43	1.28
$[1k, 10k]$	25	2110.34	33	2.87	27	1.26	30	1.30

Table 7: Parallelization of a state-of-the-art MILP solver: comparison among static methods.

Finally, we compared `SS(16)` against the distributed algorithms implemented in `IBM ILOG CPLEX 12.6.1`. In particular, we compared it against the following schemes:

- algorithm `RAMPUP`, i.e., the default version of the distributed version of `IBM ILOG CPLEX 12.6.1` that applies an infinite rampup phase;
- algorithm `DYNAMIC`, which is a more sophisticated implementation of the distributed version of `IBM ILOG CPLEX 12.6.1` that applies an initial rampup and eventually applies some dynamic load balancing techniques.

Note that the first algorithm corresponds to the *No Communication* configuration for workers considered in [12], which is actually considered one of best options to parallelize a MILP solver in a distributed environment (i.e., without communication).

To keep our computational settings as clean as possible, we did not investigate possible alternative schemes, obtained either with different levels of communication (as proposed, e.g., in [12]) or through mixed methods in which different strategies are used for different sets of workers.

Computational results are reported in Table 8, whose rows and columns have the same meaning as in Table 7.

	`CPLEX(1)`		`SS(16)`		`DYNAMIC`		`RAMPUP`	
class	#opt	time	#opt	speedup	#opt	speedup	#opt	speedup
all	174	959.34	193	1.32	208	1.20	192	1.12
$[0, 10k]$	174	244.18	193	1.57	208	1.35	192	1.21
$[1, 10k]$	174	244.18	193	1.57	208	1.35	192	1.21
$[10, 10k]$	153	341.90	172	1.63	187	1.43	171	1.27
$[100, 10k]$	85	1391.51	104	2.05	119	2.16	103	1.60
$[1k, 10k]$	53	2981.75	72	2.15	87	3.23	71	1.83

Table 8: Parallelization of a state-of-the-art MILP solver.

According to Table 8, the performance of `SS(16)` is competitive with the distributed strategies implemented by the commercial solver `IBM ILOG CPLEX 12.6.1`. Surprisingly, it is even the fastest method if considering the whole testbed. However, it cannot match the number of instances solved to optimality by `DYNAMIC`. In addition, when we consider harder and harder models, `DYNAMIC` takes the lead, which is not unexpected given its dynamic load balancing capabilities. Still, it is worth noting that `SS(16)` slightly dominates `RAMPUP`, both in runtime and number of instances solved, on all subsets.

6. Conclusions and future work

We have investigated the performance on MILP problems of `SelfSplit`, the deterministic and (almost) communication free parallelization paradigm for tree search methods presented in [25].

Artificial experiments have been reported, aimed at evaluating the behavior of `SelfSplit` "in vitro". Our experiments highlight the role of performance variability and heavy-tail distributions in limiting scalability in a low-communication distributed setting.

Two different implementations of `SelfSplit` have been considered, that only require very minor changes of the deterministic algorithm to be parallelized. Computational results on both ad-hoc and general-purpose MILP solvers have been reported, showing that a nontrivial speedup can sometimes be obtained with these simple changes, though more sophisticated approaches could lead to even better results.

Our experiments with a branch-and-bound ATSP code confirm the findings of [8], and show that simple ad-hoc enumerative codes can effectively be parallelized by no-communication paradigms like `SelfSplit`. In our view, this is mainly due to their low performance variability—a property that makes them rather appealing in a distributed setting.

Reasonable speedups were also obtained for a more sophisticated branch-and-cut ATSP code, though the effectiveness of its cutting planes tends to produce smaller trees with heavy nodes, which is not the most appropriate setting for our method due to sampling-phase overhead.

We have also addressed the parallelization of a specific branch-and-cut algorithm working on top of a generic MILP solver through callbacks that prevent the use of the standard distribution schemes. To be specific, we have considered the state-of-the-art code of [30] for the Uncapacitated Facility Location (UFL) problem with linear costs, and parallelized it through a simple `SelfSplit` scheme producing a very satisfactory speedup of 5 to 8 when using 16 computers in a cluster.

Finally we have shown that `SelfSplit` can achieve a significant speedup even for a very sophisticated general-purpose MILP solver such as `IBM ILOG CPLEX 12.6.1`.

Future research can be devoted to verify the effectiveness of `SelfSplit` for parallelizing different types of enumerative codes.

Acknowledgements

This research was supported by the University of Padova (Progetto di Ateneo "Exploiting randomness in Mixed-Integer Linear Programming"), by MiUR, Italy (PRIN project "Mixed-Integer Nonlinear Optimization: Approaches and Applications"), and by the Vienna Science and Technology Fund (WWTF) through project ICT15-014. Thanks are also due to two anonymous referees for their constructive comments.

References

[1] B. Gendron, T. G. Crainic, Parallel branch-and-branch algorithms: Survey and synthesis, Operations Research 42 (1994) 1042–1066.

[2] T. Ralphs, Y. Shinano, T. Berthold, T. Koch, Parallel Solvers for Mixed Integer Linear Optimization, Technical Report, COR@L, 16T-014-R3, 2017.

[3] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, CACM 51 (2008) 107–113.

[4] D. Henrich, Initialization of parallel branch-and-bound algorithms (1993).

[5] O. I. El-Dessouki, W. H. Huen, Distributed enumeration on between computers, IEEE Transactions on Computers (TOC) C-29 (1980) 818–825.

[6] T. Moisan, J. Gaudreault, C.-G. Quimper, Parallel discrepancy-based search, in: C. Schulte (Ed.), CP, volume 8124 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2013, pp. 30–46.

[7] W. D. Harvey, M. L. Ginsberg, Limited discrepancy search, in: IJCAI 1995, 1995, pp. 607–615.

[8] P. S. Laursen, Can parallel branch and bound without communication be effective, SIAM Journal on Optimization 4 (1994) 288–296.

[9] M. R. Bussieck, M. C. Ferris, A. Meeraus, Grid-enabled optimization with GAMS, INFORMS Journal on Computing 21 (2009) 349–362.

[10] J. Eckstein, C. A. Phillips, W. E. Hart, Pebbl 1.0 user guide, 2007.

[11] M. Fischetti, M. Monaci, Exploiting erraticism in search, Operations Research 62 (2014) 114–122.

[12] R. Carvajal, A. Shabbir, G. Nemhauser, K. Furman, V. Goel, Y. Shao, Using diversification, communication and parallelism to solve mixed-integer linear programs, Operations Research Letters 42 (2014) 186–189.

[13] Y. Shinano, S. Heinz, S. Vigerske, M. Winkler, FiberSCIP – a shared memory parallelization of SCIP, INFORMS Journal on Computing 30 (2018) 11–30.

[14] A. Grama, V. Kumar, State of the art in parallel search techniques for discrete optimization problems, IEEE Trans. Knowl. Data Eng 11 (1999) 28–35.

[15] L. Michel, A. See, P. V. Hentenryck, Transparent parallelization of constraint programming, INFORMS Journal on Computing 21 (2009) 363–382.

[16] G. Chu, C. Schulte, P. J. Stuckey, Confidence-based work stealing in parallel constraint programming, in: I. P. Gent (Ed.), CP 2009, volume 5732, Springer, 2009, pp. 226–241.

[17] O. Vornberger, Implementing branch-and-bound in a ring of processors, in: CONPAR 86, volume 237, Springer, 1986, pp. 157–164.

[18] L. Bordeaux, Y. Hamadi, H. Samulowitz, Experiments with massively parallel constraint solving, in: C. Boutilier (Ed.), IJCAI 2009, 2009, pp. 443–448.

[19] I. P. Gent, C. Jefferson, I. Miguel, N. C. Moore, P. Nightingale, P. Prosser, C. Unsworth, A preliminary review of literature on parallel constraint solving, in: Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving, 2011.

[20] Y. Xu, T. K. Ralphs, L. Ladányi, M. J. Saltzmann, Computational experience with a software framework for parallel integer programming, INFORMS Journal on Computing 21 (2009) 383–397.

[21] Q. Chen, M. C. Ferris, J. Linderoth, FATCOP 2.0: Advanced features in an opportunistic mixed integer programming solver, Annals OR 103 (2001) 17–32.

[22] T. Koch, T. K. Ralphs, Y. Shinano, Could we use a million cores to solve an integer program?, Mathematical Methods of Operations Research 76 (2012) 67–93.

[23] T. Achterberg, R. Wunderling, Mixed integer programming: Analyzing 12 years of progress, in: Facets of Combinatorial Optimization, 2013, pp. 449–481.

[24] J.-C. Régin, M. Rezgui, A. Malapert, Embarrassingly parallel search, in: C. Schulte (Ed.), Principles and Practice of Constraint Programming, volume 8124 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2013, pp. 596–610.

[25] M. Fischetti, M. Monaci, D. Salvagnin, Self-splitting of workload in parallel computation, in: CPAIOR14, volume 8451 of *Lecture Notes in Computer Science*, Springer US, 2014, pp. 394–404.

[26] Gecode Team, Gecode: Generic constraint development environment, 2012. Available at `http://www.gecode.org`.

[27] M. Fischetti, P. Toth, An additive bounding procedure for the asymmetric travelling salesman problem, Mathematical Programming 53 (1992) 173–197.

[28] M. Fischetti, P. Toth, A polyhedral approach to the asymmetric traveling salesman problem, Management Science 43 (1997) 1520–1536.

[29] M. Fischetti, A. Lodi, P. Toth, Exact methods for the asymmetric traveling salesman problem, in: The traveling salesman problem and its variations, Springer US, 2007, pp. 169–205.

[30] M. Fischetti, I. Ljubić, M. Sinnl, Redesigning Benders Decomposition for Large Scale Facility Location (2015). Submitted.

[31] K. M. Anstreicher, N. W. Brixius, J.-P. Goux, J. Linderoth, Solving large quadratic assignment problems on computational grids, Mathematical Programming 91 (2002) 563–588.

[32] D. E. Knuth, Estimating the efficiency of backtrack programs, Mathematics of Computation 29 (1975) 121–136.

[33] G. Cornuéjols, M. Karamanov, Y. Li, Early estimates of the size of branch-and-bound trees, INFORMS Journal on Computing 18 (2006) 86–96.

[34] O. Y. Özaltin, B. Hunsaker, A. J. Schaefer, Predicting the solution time of branch-and-bound algorithms for mixed-integer programs, INFORMS Journal on Computing 23 (2011) 392–403.

[35] C. P. Gomes, B. Selman, N. Crato, H. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, Journal of Automated Reasoning 24 (2000) 67–100.

[36] J. W. Tukey, Exploratory Data Analysis, Addison-Wesley, 1977.

[37] P. Sanders, Randomized receiver initiated load-balancing algorithms for tree-shaped computations, The Computer Journal 45 (2002) 561–573.

[38] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, K. Wolter, MIPLIB 2010 - Mixed Integer Programming Library version 5, Mathematical Programming Computation 3 (2011) 103–163.

[39] J. Mohan, A Study in Parallel Computation: The Traveling Salesman Problem, Technical Report, Computer Science Department. Carnagie-Mellon University, Pittsburgh, Penn. CMU-CS-82-136(R), 1982.

[40] D. Miller, J. Pekny, Results from a parallel branch and bound algorithm for the asymmetric traveling salesman problem, Operations Research Letters 8 (1989) 129–135.

[41] J. Pekny, D. Miller, A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems, Mathematical Programming 55 (1992) 17–33.

[42] M. Fischetti, Facets of the asymmetric traveling salesman polytope, Mathematics of Operations Research 16 (1991) 42–56.

[43] E. Balas, M. Fischetti, A lifting procedure for the asymmetric traveling salesman polytope and a large new class of facets, Mathematical Programming 58 (1993) 325–352.

[44] E. Balas, M. Fischetti, Lifted cycle inequalities for the asymmetric traveling salesman problem, Mathematics of Operations Research 24 (1999) 273–292.

[45] E. Balas, M. Fischetti, Polyhedral theory for the asymmetric traveling salesman problem, in: The traveling salesman problem and its variations, Springer US, 2007, pp. 117–168.

[46] M. Krkel, On the exact solution of large-scale simple plant location problems, European Journal of Operational Research 39 (1989) 157–173.