Self-split parallelization for Mixed Integer Linear Programming

Matteo Fischetti, Michele Monaci, and Domenico Salvagnin

DEI, University of Padova,
{matteo.fischetti,michele.monaci,domenico.salvagnin}@unipd.it

Abstract. `SelfSplit` is a simple mechanism to convert a sequential tree-search code into a parallel one. In this paradigm, tree-search is distributed among a set of 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. The role of performance variability in limiting scalability is also discussed.

1 Introduction

Parallel computation requires splitting a job among a set of workers. A commonly used parallelization paradigm is *MapReduce* [8], which is very well suited for applications with a very large input that can be processed in parallel by a large number of mappers, while producing a manageable number of intermediate parts to be shuffled. However, the scheme may introduce a large overhead due to the need of heavy communication/synchronization between the map and reduce phases.

A different approach is based on work stealing [18,25,7]. The workload is initially distributed to the available workers. If the splitting turns out to be unbalanced, the workers that have already finished their processing *steal* part of the work from the busy ones. The process is periodically repeated in order to achieve a proper load balancing. Needless to say, this approach can require a significant amount of communication and synchronization among the workers.

Tree search algorithms are particularly suited for being applied in a parallel fashion, as different nodes can be processed by different workers in parallel. Although different strategies have been proposed and developed during the years [30,20], eventually traditional schemes have converged into a work stealing approach, in which the set of active nodes is periodically distributed among the workers [5,16,28], thus requiring an elaborate 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 [23,1].

Recently, strategies that try to overcome the traditional drawbacks of the work stealing approach within enumeration algorithms have been proposed.

A parallelization scheme for branch-and-bound algorithms is given in [24], where tree nodes are first collected by a master solver and then distributed to the available workers so as to balance the worklaod. The effectiveness of the approach was computationally evaluated on simple branch-and-bound codes for three specific optimization problems (quadratic assignment, graph partitioning, and weighted vertex cover).

As to CP solvers, in [27] a master problem enumerates the partial solutions associated with a subset of the variables of the problem to solve, each of which will be later processed by a worker; the number of variables to consider is chosen in such a way to have significantly more subproblems than workers. All subproblems are put into a queue and distributed to workers as needed (usually, a subproblem is assigned to a given worker as soon as the worker is idle). In [26], a parallelization strategy for limited discrepancy search (LDS) [19] is presented, in which 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.

In [12] we showed how to modify a given deterministic (sequential) tree-search algorithm to let it run on a set of, say, K workers. The approach, called `SelfSplit`, is in the spirit of [24] 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 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.

The results reported in [12] show that `SelfSplit` is in fact very well suited for Constraint Programming applications. Indeed, `SelfSplit` was implemented within the CP solver Gecode 4.0 [15], and tested on several instances taken from the repository of modeling examples bundled with Gecode. As the goal of [12] 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 [12]) 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.

Table 1. Speedups for the Constraint Programming solver *Gecode*.

	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

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 in fact very hard to parallelize. We note however that most MILP solvers—including branch-and-cut methods—do in fact 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". The role of performance variability in limiting scalability is also investigated. 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 [13], (ii) the more-sophisticated ATSP branch-and-cut algorithm given in [14,10], and (iii) the general-purpose commercial MILP solver IBM ILOG CPLEX 12.5.1. Computational results are reported, showing that `SelfSplit` performs very well on codes (i) and (ii) above, and has a reasonably good performance on (iii) as well.

The outline of the paper is as follows. Section 2 reviews the basic `SelfSplit` algorithm, along with possible variants aimed at improving load balancing. Sec-

tions 3 and 4 study the effect of different `SelfSplit` parameters on the resulting load balancing. Section 5 reports computational results in a MILP setting. 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 [12]. 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 the self-splitting method.
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.

In its simplest "vanilla" version, step c) is implemented by just assigning each 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 very well suited for those computational environments where communication among workers is time consuming or unreliable as, e.g., in a large computational grid where the workers run in different geographical areas—a relevant context being cloud computing, or a constellation of mobile devices.

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 [24,27] to keep a significant number of open nodes for each worker after sampling. Load balancing is automatically obtained by the modified algorithm in a statistical sense: if the condition that triggers the end of the sampling phase is chosen appropriately, then the number of subproblems to distribute is significantly larger than the number of workers K, thus it is unlikely that a given worker will be assigned much more work to do than any other worker [24,27].

`SelfSplit` is straightforward to implement if the original deterministic 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 deterministic algorithm is itself parallel, provided that the pseudo-random coloring at Step 3 is done right after a synchronization point.

A more elaborate version (see [12] for details), aimed at improving workload balancing among workers, can be devised using an auxiliary queue S of *paused nodes*. 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 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.
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 at 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.

Within NODE_PAUSE, a rough estimate of the difficulty of a node is obtained in [12] by computing the logarithm of the cardinality of the Cartesian product of the current domains of the 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 was 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, and the sampling procedure is 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 three main reasons. First, we are completely ignoring the overhead needed to generate the subproblems (sampling phase). In addition, we are assuming that all nodes require the same computational effort. Finally, we are implicitly assuming that no interaction whatsoever exists between the subproblems assigned to the same worker. 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 should 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 to 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 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.

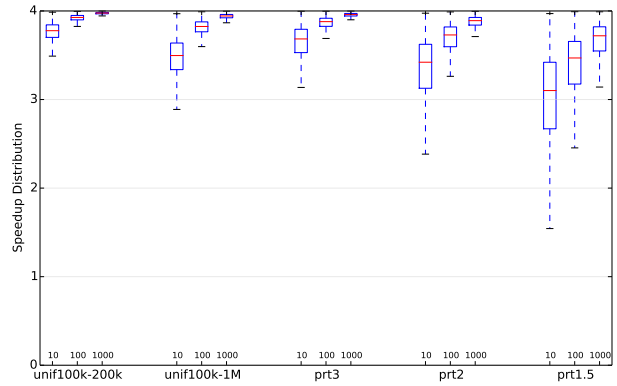
The Pareto distribution, on the other hand, models the more realistic case in which subproblems follow an heavy-tail distribution, which is the most common case for enumerative algorithms (see [17]). In this case, there is no bound on how wrong our estimates can be (even worse, arbitrarily bad subproblems show up with non negligible probability). The general Pareto (type I) distribution has a probability density function of

$$\frac{\alpha}{x^{\alpha+1}}$$

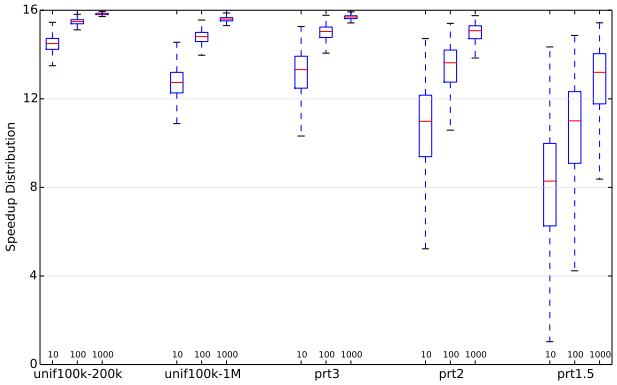
where parameter α controls the heavy-tailedness of the distribution. In particular, the distribution has no finite moments of order $n \geq \alpha$ (thus, the lower the α, the more heavy-tail the distribution). In our experiments, we chose $\alpha \in \{1.5, 2, 3\}$.

The results of our experiments are depicted in Figure 1, where we show boxplots [29] for the 5 chosen 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 red line shows the median of the corresponding population. The whiskers extend from the box to show the range of the data.

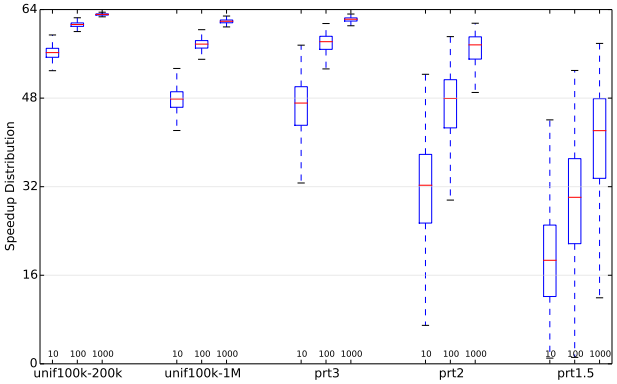
Our experiments show that the distribution of the speedup is highly influenced by the distributions from which the subproblems sizes are drawn, whereas marginal effects are experienced when changing the values of M, provided the latter be "large enough". Indeed, while almost linear speedups can be obtained (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



(a) $K = 4$



(b) $K = 16$



(c) $K = 64$

Fig. 1. Boxplots for statistical load balancing.

ones leading again to the worst 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.

As a concluding remark, we observe that our analysis clearly shows the importance of reducing performance variability in a distributed setting—even if this reduction implies a significant performance deterioration in a sequential framework.

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 [22], 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 [21], 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. 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 then *simulated* the behavior of different NODE_PAUSE criteria on the frozen trees. This experimental environment has several advantages:

- it is side-effects free;
- 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 the speedup we measured, which can only be computed by considering the size of the subtrees assigned to the different workers, are only a rough approximation of the real

speedup in computing time. As such, a final validation of the results in a *live* environment is needed, as given in the next section.

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. 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.

For sorting the nodes n in S according to their expected difficulty, we considered the following two 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

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

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

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

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

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

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. This confirms the effectiveness of our self-splitting method, at least in an enumerative context with no (or little) variability.

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 however of just 20%, which is likely to be smaller than 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. 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.

Finally, for the `offline` setting (which is the only that can be actually implemented within `SelfSplit`) the performance of `volume` is (slightly) better that that of `depth`. Note however that in both cases 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).

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 24 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 [13] 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 [13] 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 to possibly abort its own run (just 46 new lines of code added to the sequential 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 partly due to the very rigid best-bound search strategy implemented in the sequential 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 any parallelization technique working at the tree-search level cannot produce interesting speedups. Our testbed then contains instances randomly generated as in class B in [13], which were the hardest instances for our code among those considered in [13]. 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 [13] 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 CPU 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 CPU time.

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

time (sec.s)	time speedup			
$K = 1$	$K = 8$	$K = 16$	$K = 32$	$K = 64$
1,504	7.76	13.37	21.56	30.55

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 [24], 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.

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 [14,10]. This is FORTRAN code of about 10,000 lines where node bounds are computed through an LP solver (IBM ILOG CPLEX 12.5.1 in our case). Various classes of facet-defining ATST cuts—including SEC's, SD's, and DK's [9,2,3,4]—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 [14], the most challenging instances for our sequential branch-and-cut code correspond to instances of type C in [13], 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.

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

time (sec.s)	time speedup			
$K = 1$	$K = 8$	$K = 16$	$K = 32$	$K = 64$
2,465	6.74	10.89	14.54	17.91

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 general-purpose MILP solver

Our third set of experiments concerns the applicability of `SelfSplit` to a general MILP solver. This is indeed the most challenging setting for our 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 12.5.1 through callback functions. In particular, at each node n, we compute the ratio $V(1)/V(n)$ where $V(1)$ and $V(n)$ 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. As explained in Section 2, during the sampling phase we always extract from the queue a non-paused node, if any. 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, and function NODE_PAUSE is re-executed for each node.

In our implementation we used the same parameters for all instances, namely $\rho = 1$, $N = 2,000$ and $\delta = 10$.

As in the previous experiments, we decided to provide the optimal solution on input to the MILP solver and to disable all heuristics. In our view this setting is not too far from a production implementation involving some limited amount of communication, in which the incumbent value is shared among workers.As `SelfSplit` is intended to be used for problems with a large enumeration tree, we selected all the 32 instances from MIPLIB 2010 [22] that were selected for the experiments described in Section 4, namely all those instances for which CPLEX took more than 500,000 nodes.

Table 5 compares the performances of CPLEX 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 hit the time limit.

Table 5. CPLEX vs `SelfSplit` in a single-worker setting.

	CPLEX		`SS(1)`	
seed	#opt	time	#opt	time
1	22	2,673	21	3,284
2	22	2,679	21	2,814
3	24	2,515	21	3,184
4	22	2,688	22	2,633
5	25	2,603	24	3,106

The results of Table 5 show that the slowdown incurred when deviating from CPLEX defaults node selection policy (that `SelfSplit` changes during the sampling phase to improve workload balancing) was just 10-20% on average; this indicates that `SS(1)` is a good approximation of a state-of-the-art commercial solver in its default settings.

We then considered the availability of a system with 16 single-thread machines (each acting as a worker) and compared two ways to exploit this architecture without communication:

- running algorithm `RND(16)`, i.e., executing `SS(1)` with 16 different random seeds and taking the best run for each instance; and
- running algorithm `SS(16)`, i.e., executing `SelfSplit` with 16 workers.

Note that the first algorithm corresponds to the *No Communication* configuration for workers considered in [6], and 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 do not investigate possible alternative schemes, obtained either with different level of communications (as proposed, e.g., in [6]) or through mixed strategies in which different strategies are used for different sets of workers.

Table 6 reports, for each instance, the computing time for algorithms `SS(1)`, `RND(16)` and `SS(16)`; for the last two algorithms, the speedup, in terms of computing time, with respect to `SS(1)` is also given. The last line of the table reports the average results (in geometric mean) over all instances. According to Table 6, both `RND(16)` and `SS(16)` solved to proven optimality 29 instances within the given time limit. The geometric means of the speedup with respect to `SS(1)` were 2.01 and 5.29, respectively, indicating that algorithm `SS(16)` is about 2.6 times faster than `RND(16)` in geometric mean.

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 [12].

Artificial experiments have been reported, aimed at evaluating the behavior of `SelfSplit` "in vitro". Our experiments highlight the role of performance variability in limiting scalability in a low-communication distributed setting—even if variability reduction would require a severe performance deterioration in a sequential framework, it can be instrumental for massively distributed computation.

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.

Our experiments with a branch-and-bound ATSP code confirm the findings [24], 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 tend to produce smaller trees with heavy nodes, which is not the most appropriate setting for our method due to sampling-phase overhead.

Finally we have shown that, contrarily to our own expectations, `SelfSplit` can achieve a significant speedup even for a very sophisticated general-purpose MILP solver such as IBM ILOG CPLEX.

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

Table 6. Parallelization of a state-of-the-art MILP solver.

	SS(1)	RND(16)		SS(16)	
instance	time	time	speedup	time	speedup
beasleyC3	10,000.0	1,601.2	6.25	10,000.0	1.00
csched007	10,000.0	2,166.2	4.62	1,445.5	6.92
csched010	5,471.8	1,183.6	4.62	475.6	11.50
danoint	2,579.5	1,767.1	1.46	234.8	10.99
enlight16	272.4	154.5	1.76	10.3	26.32
iis-bupa-cov	10,000.0	10,000.0	1.00	1,762.8	5.67
k16x240	3,526.5	3,526.5	1.00	365.0	9.66
mcsched	4,744.5	3,735.3	1.27	371.8	12.76
mik-250-1-100-1	1,131.1	1,129.1	1.00	1,543.8	0.73
momentum1	8,730.2	3,476.1	2.51	2,224.6	3.92
neos-1426662	5,591.1	590.3	9.47	2,980.3	1.88
neos-1442657	639.1	180.0	3.55	83.9	7.61
neos-1616732	2,792.7	1,410.8	1.98	549.9	5.08
neos-1620770	10,000.0	1,356.7	7.37	208.0	48.07
neos-942830	1,626.5	258.6	6.29	3,662.1	0.44
neos15	5,096.1	5,094.5	1.00	3,081.1	1.65
neos16	10,000.0	2,041.4	4.90	127.8	78.21
neos858960	2,043.9	794.9	2.57	173.6	11.77
newdano	10,000.0	9,820.6	1.02	1,406.5	7.11
nobel-eu-DBE	10,000.0	5,641.8	1.77	7,577.2	1.32
noswot	147.1	23.9	6.13	17.3	8.46
ns1766074	89.4	85.8	1.04	10.3	8.68
ns2081729	6,588.3	6,588.3	1.00	10,000.0	0.66
nu60-pr9	10,000.0	5,088.1	1.97	3,248.6	3.08
pg5_34	9,416.2	9,407.7	1.00	826.2	11.40
pigeon-10	421.6	395.4	1.07	65.1	6.47
ran14x18	3,163.1	2,438.4	1.30	236.9	13.35
ran14x18-disj-8	1,709.9	1,709.7	1.00	201.2	8.50
reblock166	10,000.0	10,000.0	1.00	10,000.0	1.00
rmine6	10,000.0	4,763.8	2.10	1,716.6	5.83
rococoB10-011000	10,000.0	10,000.0	1.00	5,092.3	1.96
timtab1	1,675.1	868.8	1.93	171.3	9.78
average (geo.mean)	3,284.1	1,630.2	2.01	620.6	5.29

Also of interest is to study how to reduce performance variability in general-purpose branch-and-cut MILP solvers, given for granted that even a severe slow-down in the sequential case (of 2 times or more) can be welcome in a massive-parallel setting if it leads to improved scalability. In this context, it would be interesting to start a systematic study of the degree of variability injected in a MILP solver by the various sophistications that have introduced in the recent years—including preprocessing, pseudo-costs, propagation, branching, cutting planes, primal heuristics, etc.

References

1. Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In *Facets of Combinatorial Optimization*, pages 449–481, 2013.
2. Egon Balas and Matteo Fischetti. A lifting procedure for the asymmetric traveling salesman polytope and a large new class of facets. *Mathematical Programming*, 58(1-3):325–352, 1993.
3. Egon Balas and Matteo Fischetti. Lifted cycle inequalities for the asymmetric traveling salesman problem. *Mathematics of Operations Research*, 24(2):273–292, 1999.
4. Egon Balas and Matteo Fischetti. Polyhedral theory for the asymmetric traveling salesman problem. In *The traveling salesman problem and its variations*, pages 117–168. Springer US, 2007.
5. Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. Experiments with massively parallel constraint solving. In Craig Boutilier, editor, *IJCAI 2009*, pages 443–448, 2009.
6. Rodolfo Carvajal, Ahmed Shabbir, George Nemhauser, Kevin Furman, Vikas Goel, and Yufen Shao. Using diversification, communication and parallelism to solve mixed-integer linear programs. *Operations Research Letters*, 42:186–189, 2014.
7. Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In Ian P. Gent, editor, *CP 2009*, volume 5732, pages 226–241. Springer, 2009.
8. Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.
9. Matteo Fischetti. Facets of the asymmetric traveling salesman polytope. *Mathematics of Operations Research*, 16(1):42–56, 1991.
10. Matteo Fischetti, Andrea Lodi, and Paolo Toth. Exact methods for the asymmetric traveling salesman problem. In *The traveling salesman problem and its variations*, pages 169–205. Springer US, 2007.
11. Matteo Fischetti and Michele Monaci. Exploiting erraticism in search. *Operations Research*, 62:114–122, 2014.
12. Matteo Fischetti, Michele Monaci, and Domenico Salvagnin. Self-splitting of workload in parallel computation. In *CPAIOR14*, pages 394–404. Springer US, 2014.
13. Matteo Fischetti and Paolo Toth. An additive bounding procedure for the asymmetric travelling salesman problem. *Mathematical Programming*, 53:173–197, 1992.
14. Matteo Fischetti and Paolo Toth. A polyhedral approach to the asymmetric traveling salesman problem. *Management Science*, 43(11):1520–1536, 1997.
15. Gecode Team. Gecode: Generic constraint development environment, 2012. Available at `http://www.gecode.org`.

16. Ian P. Gent, Chris Jefferson, Ian Miguel, Neil C.A. Moore, Peter Nightingale, Patrick Prosser, and Chris Unsworth. A preliminary review of literature on parallel constraint solving. In *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*, 2011.
17. Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
18. Ananth Grama and Vipin Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Trans. Knowl. Data Eng*, 11(1):28–35, 1999.
19. William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI 1995*, pages 607–615, 1995.
20. Dominik Henrich. Initialization of parallel branch-and-bound algorithms, 1994.
21. IBM ILOG. *CPLEX 12.5 User's Manual*, 2013.
22. Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010 - Mixed Integer Programming Library version 5. *Mathematical Programming Computation*, 3:103–163, 2011.
23. Thorsten Koch, Ted K. Ralphs, and Yuji Shinano. Could we use a million cores to solve an integer program? *Mathematical Methods of Operations Research*, 76(1):67–93, 2012.
24. Per S. Laursen. Can parallel branch and bound without communication be effective. *SIAM Journal on Optimization*, 4(2):288–296, 1994.
25. Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS Journal on Computing*, 21(3):363–382, 2009.
26. Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper. Parallel discrepancy-based search. In Christian Schulte, editor, *CP*, volume 8124 of *Lecture Notes in Computer Science*, pages 30–46. Springer Berlin Heidelberg, 2013.
27. Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 596–610. Springer Berlin Heidelberg, 2013.
28. Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler. Fiberscip – a shared memory parallelization of scip. Technical report, ZIB, 2013.
29. John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.
30. Oliver Vornberger. Implementing branch-and-bound in a ring of processors. In *CONPAR 86*, volume 237, pages 157–164. Springer, 1986.