Self-splitting of Workload in Parallel Computation

Matteo Fischetti, Michele Monaci, and Domenico Salvagnin

DEI, University of Padova, Via Gradenigo 6/A, 35131 Padova, Italy
{matteo.fischetti,michele.monaci,domenico.salvagnin}@unipd.it

Abstract. Parallel computation requires splitting a job among a set of processing units called *workers*. The computation is generally performed by a set of one or more master workers that split the workload into chunks and distribute them to a set of slave workers. In this setting, communication among workers can be problematic and/or time consuming. Tree search algorithms are particularly suited for being applied in a parallel fashion, as different nodes can be processed by different workers in parallel. In this paper we propose a simple mechanism to convert a sequential tree-search code into a parallel one. In the new paradigm, called `SelfSplit`, each worker is able to autonomously determine, without any communication with the other workers, the job parts it has to process. Computational results are reported, showing that `SelfSplit` can achieve an almost linear speedup for hard Constraint Programming applications, even when 64 workers are considered.

1 Introduction

Parallel computation requires splitting a job among a set of workers. A commonly used parallelization paradigm is *MapReduce* [1]. According to the MapReduce paradigm, the overall computation is organized in two steps, and performed by two user-supplied operators, namely, `map()` and `reduce()`. The MapReduce framework is in charge of splitting the input data and dispatching it to an appropriate number of map workers, and also of the shuffling and sorting necessary to distribute the intermediate results to the appropriate reduce workers. The output of all reduce workers is finally merged. This scheme 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 [2,3,4]. 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.

H. Simonis (Ed.): CPAIOR 2014, LNCS 8451, pp. 394–404, 2014.

Tree search algorithms are particularly suited for being applied in a parallel fashion, as different nodes can be processed by different workers in parallel. However, traditional schemes can require an elaborate load balancing strategy, in which the set of active nodes is periodically distributed among the workers [5,6,7], in a work stealing fashion. 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 [8,9].

Recently, strategies that try to overcome the traditional drawbacks of the work stealing approach within enumeration algorithms have been proposed. In particular, in [10] 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 [11], a parallelization strategy for LDS [12] 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. These strategies share some similarities with our approach, although some important differences remain.

In the present paper we show how to modify a given deterministic (sequential) tree-search algorithm to let it run on a set of workers. The main features of the approach, that we call `SelfSplit`, are that

1. each worker works on the whole input data and is able to autonomously decide the parts it has to process;
2. almost no communication between the workers is required;
3. the resulting algorithm can be implemented to be deterministic;
4. in most cases, the modification only requires a few lines of codes.

The above features make `SelfSplit` very well suited for those applications in which encoding the input and the output of the problem requires a reasonably small amount of data (i.e., it can be handled efficiently by a single worker), whereas the execution of the job can produce a very large number of time-consuming job parts. This is indeed the case when using an enumerative method to solve an NP-hard problem. As such our method is well suited for, but not limited to, High Performance Computing (HPC) applications including Constraint Programming (CP) and Mixed Integer Programming (MIP), whereas approaches based on the MapReduce paradigm are more suited for Big Data applications.

The outline of the paper is as follows. Section 2 describes the basic self-splitting idea for tree search algorithms, along with possible variants aimed at improving load balancing, while Section 3 describes possible implementation strategies. Section 4 reports computational experiments of the application of the above technique within a CP solver. Finally, in Section 5 we draw some conclusions and outline future research directions.

2 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 into subproblems called *nodes*. In this paper we will only deal with original algorithms of enumeration type (branch-and-bound or alike), but other applications of SelfSplit are possible.

2.1 The Idea

Figure 1 illustrates our self-splitting method to parallelize an enumerative original algorithm.

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 $k \in \{1, \cdots, K\}$ 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* and is illustrated in the figure by the fact that exactly the same enumeration tree is initially built by all workers. No communication at all is involved in this stage. It is assumed that the sampling phase is not a bottleneck in the

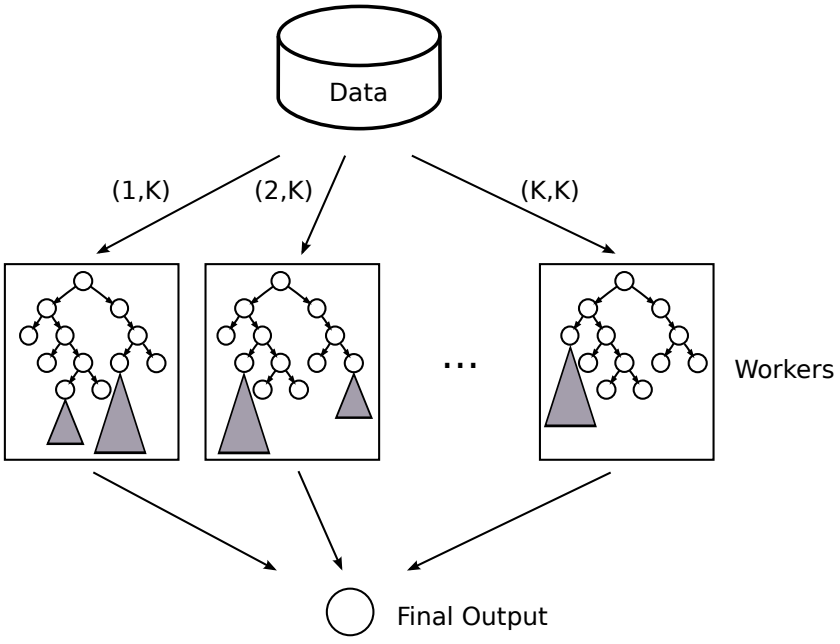

Fig. 1. Illustration of the SelfSplit paradigm

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 identify and solve the nodes that belong to it (gray subtrees in the figure), without any redundancy. 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.

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 finish their computation at very different points in time. To contrast this drawback, our recipe, as in [10], is to keep a significant number of open nodes for each worker after sampling, so as to increase the chances that workload is split in a fair way. More sophisticated rules can also be applied, as described later on.

2.2 Vanilla Algorithm

In its simplest version, our method modifies the original algorithm as follows:

1. Two integer parameters (k, K) are added to the original input: K denotes the number of workers, while k is an index that uniquely identifies the current worker ($1 \leq k \leq K$).
2. A global flag ON_SAMPLING is introduced and initialized to true. The flag becomes false when a given condition is met, e.g., when there are enough open nodes in the branch-and-bound tree. When the flag ON_SAMPLING is set to false we say that the *sampling phase* is over.
3. Each time a node n is created, it is deterministically assigned a *color* $c(n)$ which is a pseudo-random integer in $\{1, \cdots, K\}$ during the sampling phase, and $c(n) = k$ otherwise.

4. Whenever the modified algorithm is about to process a node n, condition

$$(\neg \text{ON\_SAMPLING}) \wedge (c(n) \neq k)$$

is evaluated. If the condition evaluates to true, node n is just discarded, as it corresponds to a subproblem assigned to a different worker; otherwise, the processing of node n continues as usual and no modified action takes place.

Each worker executes exactly the same algorithm, but receives a different input value for k. The above method ensures that each worker can autonomously and deterministically identify and skip the nodes that will be processed by other workers, and each node is covered by (at least) one worker. A similar strategy is exploited in [11], where however the details of the split are very much dependent of the LDS algorithm, and cannot be easily (and efficiently) generalized to an arbitrary tree search algorithm.

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. Static decomposition and statistical load balancing were also at the base of the method proposed in [10], with some key differences:

- a master process is used to generate the subproblems to distribute. Enumeration in the master problem is more static than in `SelfSplit`, and some communication is needed to distribute the subproblems;
- open problems are dynamically assigned to idle workers as soon as they become available, which again requires some communication;
- the algorithm in [10] is non-deterministic because of dynamic scheduling.

`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.

2.3 Paused-Node Queue Algorithm

A slightly more elaborate version, 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.

Because it has access to all the nodes in S, 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.

3 Implementation Details

We will next give more details about the application of our method within an enumerative method for optimization problems. We will focus on the version exploiting the queue S of paused nodes. In this 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 can be obtained 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 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, 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, in our implementation the color $c(n)$ associated with each node n in S is 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.

4 `SelfSplit` for Constraint Programming

We implemented `SelfSplit` within the CP solver Gecode 4.0 [13]. While the most natural option is to implement the scheme as a search engine, we opted for implementing self-splitting as a custom constraint propagator. This is only

because we found the implementation much easier: implementing search engines is somewhat more involved, and requires some expertise in Gecode programming. In addition, our proof of concept implementation shows that the method can be implemented with very limited effort. Of course, different implementation strategies for different solvers can be devised.

Our global constraint, `node_pause`, is implemented as a generic n-ary propagator, that takes on input an array x of variables, a pointer to the queue S to store delayed nodes, a measure of node difficulty V_0 computed from the domains of the variables in x at the root node, and a threshold θ. Each time the `propagate` method of our propagator is called, the node-difficulty measure V is computed based on the local domains of the variables in x, and the resulting value is compared with V_0. If their ratio is greater than θ, then the local domains of the variables x are copied into a custom node class, which is stored in S, and the propagator returns a failure in order to kill the node. Otherwise, we just return. Note that this implementation is compatible with Gecode *copy and recomputation* backtracking model.

We next address the computation of the node-difficulty estimate within NODE_PAUSE, namely, the logarithm of the cardinality of Cartesian product of the variable domains. We provide an implementation both for arrays of integer variables and for arrays of set variables.

In the integer case, for each variable x_j in x we consider its current domain as a list of ranges $\{[l_j^k, u_j^k] | k \in K_j\}$, as implemented in Gecode. As such, the contribution of variable x_j to the difficulty measure can be computed as

$$\log_2 \sum_{k \in K_j} (u_j^k - l_j^k + 1)$$

which is a refinement of the simpler expression $\log_2(u_j - l_j + 1)$.

In the set case, Gecode approximates the domain of a variable x_j with three pieces of information:

i) a set of elements glb_j which is known to be contained in any feasible value for x_j, i.e., $glb_j \subseteq x_j$
ii) a set of elements lub_j which is known to contain any feasible value for x_j, i.e., $x_j \subseteq lub_j$
iii) bounds (m_j, M_j) on the cardinality of x, i.e., $m_j \leq |x_j| \leq M_j$.

With this encoding, we can compute the contribution of variable x_j as

$$\log_2 \sum_{i=m_j-|glb_j|}^{M_j-lub_j} \binom{|lub_j \setminus glb_j|}{i}$$

Note that the above expression can become expensive to compute, and prone to overflow for even modest values of $|lub_j \setminus glb_j|$. For this reason, if the resulting number is greater than 64 we use the valid upper bound $|lub_j \setminus glb_j|$.

The overall scheme is implemented as follows:

- the model to solve is coded into a C++ class (as usually done in Gecode) and the `node_pause` constraint is added to the model with the appropriate parameters. Note that the array x of variables that are considered for computing the measure of difficulty of the current subproblem is possibly a subset of the whole set of variables, chosen by exploiting knowledge about the model. We do not consider this an issue, since modeling a problem in Gecode requires some problem-specific coding in any case;
- the model (with the `node_pause`) propagator is completely enumerated (sampling phase);
- the nodes collected in S that survive the coloring phase are used to construct new models, copying the domains from the nodes, each of which is enumerated by Gecode. Note that the propagator `node_pause` is not used in this phase.

To avoid to have too few nodes in S after the sampling phase, we implemented the following simple adaptive mechanism, along the lines of the previous section. The boolean value returned by NODE_PAUSE(n) is true when the difference between the estimated difficulty of the root node and that of node n, computed as outlined above, is greater or equal to $\theta = 10$ (corresponding to a reduction of the cardinality of the Cartesian product of at least 1,024 times). When the sampling phase is over, if $|S| < 2000$ then θ is doubled and the overall method is just restarted.

We tested our implementation on several instances taken from the repository of modeling examples bundled with Gecode. Since we are interested in measuring the scalability of our method, we considered only instances which are either infeasible or in which we are required to find all feasible solutions (the parallel speedup for finding a first feasible solution can be completely uncorrelated to the number of workers, making the results hard to analyze). On some instances we added some form of symmetry breaking constraints in order to make the search for all solutions more efficient. We ran our method with number of workers $K \in \{1, 4, 16, 64\}$. Each worker is configured to use only a single thread: this is because a deterministic behavior is needed for correctness in the sampling phase, and it also makes the results more easily reproducible.

Detailed results are available in Table 1. According to the table, even on moderately easy instance requiring half a minute to solve, `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. Note that, despite the fact that our instances are very different, we used exactly the same parameter tuning on all of them, showing that the method is quite robust.

5 Conclusions and Future Work

We have presented `SelfSplit`, a new deterministic and (almost) communication free parallelization paradigm for tree search methods. The idea is that a given deterministic algorithm can easily be parallelized by letting each processing unit

Table 1. Measuring scalability with Gecode

instance	time (s) $K = 1$	speedup $K = 4$	$K = 16$	$K = 64$
golomb_12	41.5	3.84	14.31	41.50
golomb_13	1195.8	4.00	15.67	57.49
golomb_14	19051.9	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	4116.4	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	2402.4	3.91	15.44	59.76

autonomously decide which are the parts of work it can skip as they will be performed by other units. This is achieved without any communication among the units, and only requires a same deterministic selection rule be applied by all units in the early part of the computation.

A main feature of the method is that exactly the same code is run independently by all units, without the need of any external coordination nor master-slave hierarchy—the work gently "splits itself" over the units. As a consequence, `SelfSplit` is very well suited for HPC on a computational grid (or cloud) where the processing units are geographically distributed and communication is expensive or unreliable, and task synchronization becomes a bottleneck of the overall computation.

Two different implementations of `SelfSplit` have been outlined, that only require minor changes of the deterministic algorithm to be parallelized. Computational results on a Constraint Programming implementation on top of an open-source solver show that an almost linear speedup can be achieved, even on 64 processing units, without any communication among them—besides the final merge of the solution(s) returned by each unit, and still with deterministic behavior.

A practically very important feature of `SelfSplit` is that it often requires just minor code changes. As an exercise, we took the Asymmetric TSP sequential codes in [14] (pure branch-and-bound) and in [15] (branch-and-cut code), which are optimized yet legacy FORTRAN codes. We parallelized them by using the `SelfSplit` approach, adding about 10 new lines of codes in the both cases, obtaining surprisingly good speedups.

Because of lack of communication among workers, `SelfSplit` can turn out to be not suitable for solvers that collect/learn important global information during the search, as this information can be crucial to reduce the search tree. Observe however that some solvers—notably, MIP 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 any case, for those solvers a limited amount of communication (e.g., of the incumbent value) can be advisable.

Possible `SelfSplit` variants to be addressed in future work are outlined below:

a) `SelfSplit` can be run with just $K' \ll K$ workers, with input pairs $(1, K)$, $(2, K), \cdots, (K', K)$. In this case the overall procedure is heuristic in nature, meaning that some nodes will not be explored by any worker (namely, those with color $k = K' + 1, \cdots, K$). This setting is particularly attractive for the parallelization of heuristics for optimization/feasibility problem, as it guarantees that the solution spaces explored (exactly or heuristically) by the K' workers after sampling is non-overlapping—though their union does not necessarily cover the whole solution space.

b) The previous variant of running $K' \ll K$ workers can also be used to obtain a lower bound on the amount of computing time needed to solve the problem with K workers (just take the maximum computing time among the K' workers) as well as an estimate of the amount of computing time T_1 needed to solve the problem with the original (unmodified) algorithm by a single worker, e.g., through the simple formula

$$T_1 = T_s + K \cdot \overline{T}$$

where T_s is the sampling time and $\overline{T}$ is the average time spent by a worker after sampling.

c) `SelfSplit` can also be used to split the overall workload into K chunks to be solved at different points in time by a single (or few) worker(s), thus implementing a simple strategy to pause and resume the overall computation. This is also beneficial in case of failures, as it allows one to re-execute the affected chunks only.

d) A limited amount of communication may be introduced between the workers after the sampling and coloring phases. This communication is meant to exchange globally valid information, such as the primal bound in an enumerative scheme, which can be used to avoid unnecessary work. For example, if a feasibility problem is addressed, as soon as a worker finds the first feasible solution all the other workers can be interrupted as the overall problem is solved. Similarly, if the incumbent is periodically shared among the workers, each worker can be interrupted in case its own best bound is worse than the incumbent value. If the incumbent is not used in any other way, the search path followed by each workers is not affected by communication and each worker behaves deterministically till its own abort point.

e) Workers can be allowed to (periodically) communicate to deal with failures in the computational environment that require re-running a certain (k, K) pair.

f) After sampling, each worker can decide not to discard the nodes that have two or more colors $c_1, c_2, ..., c_m$, where $c_1 = k$ and the other colors $c_2, \cdots, c_m$ are selected according to some rules. In this case some redundant work is performed by the workers, e.g., with the aim of coping with failures in the computational environment. The final merge worker can stop the overall computation when all colors have been processed by some worker, even if other workers are still running or were aborted for whatever reason. Alternatively, two or more workers with the same (k, K) pair can be run, in parallel, making the event that all of them fail very unlikely, and still keeping the communication overhead negligible.

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. CACM 51(1), 107–113 (2008)
2. Grama, A., Kumar, V.: State of the art in parallel search techniques for discrete optimization problems. IEEE Trans. Knowl. Data Eng. 11(1), 28–35 (1999)
3. Michel, L., See, A., Hentenryck, P.V.: Transparent parallelization of constraint programming. INFORMS Journal on Computing 21(3), 363–382 (2009)
4. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 226–241. Springer, Heidelberg (2009)
5. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Boutilier, C. (ed.) IJCAI 2009, pp. 443–448 (2009)
6. Gent, I.P., Jefferson, C., Miguel, I., Moore, N.C., Nightingale, P., Prosser, P., Unsworth, C.: A preliminary review of literature on parallel constraint solving. In: Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving (2011)
7. Shinano, Y., Heinz, S., Vigerske, S., Winkler, M.: Fiberscip – a shared memory parallelization of scip. Technical report, ZIB (2013)
8. Koch, T., Ralphs, T.K., Shinano, Y.: Could we use a million cores to solve an integer program? Mathematical Methods of Operations Research 76(1), 67–93 (2012)
9. Achterberg, T., Wunderling, R.: Mixed integer programming: Analyzing 12 years of progress. In: Facets of Combinatorial Optimization, pp. 449–481 (2013)
10. Régin, J.C., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 596–610. Springer, Heidelberg (2013)
11. Moisan, T., Gaudreault, J., Quimper, C.-G.: Parallel discrepancy-based search. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 30–46. Springer, Heidelberg (2013)
12. Harvey, W.D., Ginsberg, M.L.: Limited discrepanc search. In: IJCAI 1995, pp. 607–615 (1995)
13. Gecode Team: Gecode: Generic constraint development environment (2012), Available at `http://www.gecode.org`
14. Fischetti, M., Toth, P.: An additive bounding procedure for the asymmetric travelling salesman problem. Mathematical Programming 53, 173–197 (1992)
15. Fischetti, M., Lodi, A., Toth, P.: Exact methods for the asymmetric traveling salesman problem. In: The traveling Salesman Problem and Its Variations, pp. 169–205. Springer, US (2004)