Building Portfolios for Parallel Constraint Solving by Varying the Local Consistency Applied

Minas Dasygenis and Kostas Stergiou
Department of Informatics and Telecommunications Engineering
University of Western Macedonia, Kozani, Greece
{*mdasygenis, kstergiou*}*@uowm.gr*

***Abstract*—Portfolio based approaches to constraint solving aim at exploiting the variability in performance displayed by different solvers or different parameter settings of a single solver. Such approaches have been quite successful in both a sequential and a parallel processing mode. Given the increasingly larger number of available processors for parallel processing, an important challenge when designing portfolios is to identify solver parameters that offer diversity in the exploration of the search space and to generate different solver configurations by automatically tuning these parameters. In this paper we propose, for the first time, a way to build porfolios for parallel solving by parameterizing the local consistency property applied during search. To achieve this we exploit heuristics for adaptive propagation proposed in [1]. We show how this approach can result in the easy automatic generation of portfolios that display large performance variability. We make an experimental comparison against a standard sequential solver as well as portfolio based methods that use randomization of the variable ordering heuristic as the source of diversity. Results demonstrate that our method constantly outperforms the sequential solver and in most cases it is more efficient than the other portfolio approaches.**

***Keywords*-constraint programming; search; constraint propagation; algorithm portfolios; parallelization**

I. Introduction

Portfolio based approaches to constraint solving aim at exploiting the variability in performance displayed by different solvers or different parameter settings of a single solver. Such approaches have been quite successful in both a sequential and a parallel processing mode. In the former case, machine learning methods are typically used to effectively schedule a rather small number of solvers, or a single solver with different settings, on one machine. In the latter case, which is the focus of this paper, the ever growing availability of processors makes it possible to create portfolios that include very large numbers of different solver configurations.

There are two important challenges when building portfolios for parallel processing. First, one needs to identify solver parameters that offer diversity in the exploration of the search space when they are tuned differently. Second, given the constantly increasing number of available processors, large portfolios with many different solver configurations should be automatically generated. In addition, as argued in [2], the chosen parameters should be *scalable*, *favorable* and *solver-independent*. That is, they should offer increasingly better performance as the number of available processors grows, they should typically outperform the standard sequential approach, and they should be applicable on a variety of solvers.

A growing body of work in the SAT community has developed portfolio approaches for parallel processing [3], [4], [2]. There are fewer works on portfolio approaches for CSPs as the CP community has mainly focused on search space splitting and methods for workload balancing when it comes to parallelization [5].

Constraint propagation is at the core of CP and constitutes one the main reasons for its success. Constraint propagation algorithms typically enforce some local consistency property such as (generalized) arc consistency (GAC) on the constraints of the problem, and in this way prune inconsistent values from the domains of the variables. Local consistencies stronger than GAC have also received attention since they can offer even stronger pruning [6], [7]. Despite the recent advances in algorithms for such consistencies [8], [9], [10], it is widely accepted that there is no local consistency method that is the best choice on all problems. And since GAC achieves quite good performance on average, it is the property that is predominantly applied by CP solvers.

Regarding constraint propagation and parallelization there are works that study the parallelization of specific propagation algorithms as well as the parallelization of solvers' propagation engines. The first task is quite challenging because propagation algorithms are sequential by nature. Existing works have focused on arc consistency and are either theoretical [11], [12] or have failed to show significant speed-ups [13]. [14] considers the parallelization of a modern CP solvers' constraint propagation engine and shows that problems with a large number of (expensive to propagate) global constraints can benefit from parallelization.

In this paper we explore a new way to integrate and exploit different local consistencies within the context of CP solver parallelization. Specifically, we propose the use of the local consistency applied by a solver as a source of diversification for the generation of large portfolios. The solver configurations that appear in such a portfolio are differentiated by the level of local consistency they apply.

To achieve this, we have to address two important challenges:

- Although many local consistencies of varying pruning power have been proposed, we cannot reallistically assume that any solver will have a large number of such consistencies available. Therefore, to achieve solver-independence we need to focus on as few as possible local consistencies.
- Given that we aim at exploiting large numbers of processors, we need to find a way to automatically generate the solver configurations in a portfolio by tuning the local consistency applied to many different settings. This is easy if the source of diversification is the variable ordering heuristic (e.g. by randomizing the tie breaking), but there is no obvious way to do it in the case of propagation.

To address these challenges we exploit existing work on adaptive constraint propagation, and specifically the heuristics proposed in [1]. These heuristics offer ways to automatically switch between two different propagation methods during search by monitoring events such as domain wipeouts and value deletions. We demonstrate how large portfolios that include different settings of a baseline solver can be automatically generated by randomizing certain parameters of the heuristics of [1]. Experiments show that the derived portfolios offer high diversity in the exploration of the search space and high variability in the performance.

We compare our method against a standard sequential solver as well as against portfolio methods that use randomization of the variable ordering heuristic as the source of diversification. Experimental results from benchmark binary problems demonstrate the efficacy of our approach as it outperforms the others methods on most problems, sometimes by very large margins.

II. Background

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple (X, D, C) where: $X = \{x_1, \ldots, x_n\}$ is a set of n variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a set of ordered finite domains, and $C = \{c_1, \ldots, c_e\}$ is a set of e constraints. Each constraint c_i is a pair $(var(c_i), rel(c_i))$, where $var(c_i) = (x_{i_1}, \ldots, x_{i_k})$ is an ordered subset of X, and $rel(c_i)$ contains the allowed combinations of values for the variables in $var(c_i)$.

The concept of *local consistency* is central to CP. Local consistencies are used prior to and during search through what is known as constraint propagation to filter domains and discover inconsistencies early. The most widely studied local consistency is arc consistency. A binary CSP is *Arc Consistent* (AC) iff for any constraint $c_{ij} \in C$ and any value $a \in D(x_i)$ there exists at least one value $b \in D(x_j)$ s.t. the assignments $x_i = a$ and $x_j = b$ satisfy c_{ij}. In this case b is called a *support* for a. The generalization of AC to non-binary constraints is known as GAC.

Apart from AC and GAC, numerous other local consistencies have been proposed for binary and non-binary constraints. Here we are particularly interested in local consistencies that are stronger, i.e. can achieve stronger pruning, than AC. Since the experiments included in this paper concern binary problems, we will focus on binary constraints hereafter. However, the technique for designing portfolios presented below is generic and does not depend on the arity of the constraints.

Local consistencies that only prune values from domains and leave the structure of the constraint graph unchanged are called *domain filtering* consistencies [6]. Two of the most efficient and well studied such local consistencies are maxRPC and SAC. A binary CSP is *max Restricted Path Consistent* (maxRPC) iff each constraint c_{ij} is AC and for each value $a \in D(x_i)$, there is a support b for a in $D(x_j)$ s.t. this pair of values is path consistent. That is, this pair of values can be consistently extended to any third variable constrained with x_i and x_j. A binary CSP is *Singleton Arc Consistent* (SAC) [6] iff it has non-empty domains and for any assignment $x_i = a$ of a variable $x_i \in X$, the resulting subproblem can be made AC.

Backtracking tree search is the standard complete method for solving CSPs. This method interleaves branching decisions (e.g. variable assignments) with constraint propagation. A backtracking algorithm searches for a solution in the space of possible variable assignments by gradually extending a partial assignment until it becomes a solution or proves that no solution exists. After each branching decision a constraint propagation phase follows. Typically this consists of iteratively applying local consistency algorithms on the constraints of the problem until a fixpoint is reached. The process of calling and applying an local consistency algorithm on a constraint to filter inconsistent values from domains is known as constraint *revision*. E.g. the so called MAC algorithm always applies AC. If constraint propagation removes all values from the domain of a variable (a *domain wipeout* - DWO) then the algorithm rejects its latest branching decision.

Modern variable ordering heuristics exploit information gathered during search to better direct search [15], [16]. Among such heuristics, dom/wdeg is particularly effective. This heuristic associates a weight, initially set to one, with each constraint. Each time a constraint causes failure (i.e. a DWO), its weight is increased. For each variable, the sum of the weights of the constraints in which it participates (called *weighted degree*) is computed and the heuristic selects the variable having minimum ratio of domain size to weighted degree.

III. Portfolios based on propagation

In this section we describe our proposal for building a portfolio of different solver settings by varying the local consistency applied throughout search. First, we recall the heuristics for adaptive propagator selection proposed in [1] that are at the basis of our approach.

A. Heuristics for Adaptive Propagator Selection

Exploring ways to utilize the pruning power of strong local consistencies without penalizing cpu times, [1] proposed heuristics for dynamically switching between a weak (W) and a strong (S) propagator for each individual constraint during search. The motivation for these heuristics was based on the observation that in structured problems propagation events (DWOs and value deletions) caused by individual constraints are often highly clustered. That is, they occur during consecutive or very close revisions of the constraints. The intuition behind the proposed heuristics is twofold. First to target the application of the strong consistency on areas in the search space where a constraint is highly active so that domain pruning is maximized and dead-ends are encountered faster. And second, to avoid using an expensive propagation method when pruning is unlikely. We now recall the main heuristics of [1].

$H_1(l_{dwo})$: Heuristic H_1 monitors and counts the revisions and DWOs of the constraints in the problem. A constraint c is made S if the number of revisions of c since the last time it caused a DWO is less or equal to a (user defined) threshold l_{dwo}. Otherwise, it is made W.

$H_2(l_{del})$: Heuristic H_2 monitors revisions and value deletions. A constraint c is made S if the number of revisions of c since the last time it caused at least one value deletion is less or equal to a (user defined) threshold l_{del}. Otherwise, it is made W.

H_4 : Heuristic H_4 monitors value deletions. For any constraint c, H_4 applies W until at least one value is deleted from the domain of a variable $x \in var(c)$. Then S is applied on the remaining available values in $D(x)$.

These heuristics can be combined either disjunctively or conjunctively in various ways. For example, heuristic $H_{124}^{\vee}$ applies S on a constraint whenever the condition specified by either H_1, H_2, or H_4 holds. Heuristic $H_{24}^{\wedge}$ applies S when both the conditions of H_2 and H_4 hold. We can choose a disjunctive or conjunctive combination depending on whether we want S applied more or less frequently respectively.

As explained, heuristics H_1 and H_2, as well as any composite heuristics that include them, require setting the l_{dwo} and l_{del} parameters manually. Low (resp. high) values of these parameters result in few (resp. many) calls to S. As shown in [17] even small changes in the values of these parameters significantly influence the performance of a solver that applies the adaptive propagation heuristics. This makes it very hard, if not impossible, to find "optimal" parameter values that give the best results across a wide range of instances belonging to different problem classes.

At this point we must note that giving larger values to l_{dwo} in H_1 (or l_{del} in H_2) does not necessarily result in the exploration of a smaller search tree, especially when a modern variable ordering heuristic like dom/wdeg is used. Although larger values will result in more invocations of S, the interaction with the variable ordering heuristic may be unpredictable. Recall that dom/wdeg bases its decisions on the weights of the constraints which are influenced by the decteced failures. Given two runs of H_1 on a single instance with different values for l_{dwo}, say a_1 and a_2 with $a_1 < a_2$, it is likely that different failures will be detected, since S will applied more frequently in the second run. This in turn may direct the heuristic to different areas of the search space. Hence, the search trees explored by the two runs of H_1 may vary considerably.

B. Designing a portfolio by varying the local consistency

Since for any adaptive heuristic (except H_4) varying the values of the parameters l_{dwo} and l_{del} results in varying solver performance, the main idea is to build a portfolio of different solver settings by randomly setting the parameters l_{dwo} and l_{del} to different values.

Specifically, the portfolio we used in our experiments was built by first including a small number of manually selected solver configurations. These are the following:

1) Setting l_{dwo}=0 for heuristic H_1 (or equivalently l_{del}=0 for H_2). This results in a solver that always applies W throughout search. That is, a solver that applies AC in our case study with binary constraints.
2) Setting l_{dwo} (or equivalently l_{del}) to a very large value. This results in a solver that always applies S throughout search. That is, a solver that maintains maxRPC in our case study with binary constraints.
3) Using heuristic H_1 with $l_{dwo} = 100$.
4) Using heuristic H_2 with $l_{del} = 10$.
5) Using heuristic H_4 which requires no parameter settings.
6) Using heuristic $H_{12}^{\vee}$ with $l_{dwo} = 100$ and $l_{del} = 10$.
7) Using heuristic $H_{124}^{\vee}$ with $l_{dwo} = 100$ and $l_{del} = 10$.

The values of the parameters for solver settings 3,4 and 6,7 result in good average performance [17]. Also, the disjunctive heuristics $H_{12}^{\vee}$ and $H_{124}^{\vee}$ are quite competitive as they outperform other composite heuristics and are better than individual heuristics quite often [17]. As we will explain in Section IV, the seven manually selected configurations are not necessary to achieve good performance.

After selecting the above seven different parameter settings, the rest of the available processors were utilized by randomly giving values to parameters l_{dwo} and l_{del} using heuristics H_1, H_2, $H_{12}^{\wedge}$, $H_{124}^{\wedge}$. Each different value to one of the parameters, or pair of values to both parameters, results in a different solver configuration. For the experiments presented below, if p is the total number of available processors, we equally distribute the p-7 processors left to the four heuristics. Then for each processor we derive a solver configuration by randomly setting l_{dwo} or l_{del} or both to values in the intervals [1,max$_{dwo}$] and [1,max$_{del}$], where max$_{dwo}$ and max$_{del}$ are predetermined maximum values for l_{dwo} and l_{del} respectively.

As discussed in the Introduction, any solver parameter used to build porfolios should have some necessary qualities. Namely, it should offer diversity in the exploration of the search space, and it should be automatically tunable, scalable, favorable, and solver-independent. We argue that the method we propose has all these qualities.

Scalability and favorability are demonstrated by the experimental results presented below. The randomization of the parameters l_{dwo} and l_{del} allows for automatically producing any desired number of solver configurations. Solver independence is achieved by requiring only two diverse local consistency methods to build a portfolio. Even if most solvers do not incorporate strong local consistency methods such as maxRPC and SAC, they do offer propagators of varying strength for many constraints. For example, many solvers include both a bounds consistency and a generalized arc consistency propagator for alldifferent constraints. Finally, the important property of diversity is achieved through the interaction between the varying propagation strength and the variable ordering heuristic, as explained in Section III-A.

IV. Experiments

We have experimented with benchmark binary CSPs taken from C. Lecoutre's repository and used in CSP solvers competitions. Specifically, we experimented with instances belonging to the following classes: radio links frequency assignment (rlfap), graph coloring (gc), driver (dr), quasigroup completion (qcp), quasigroups with holes (qwh), using 150 instances in total.

All experiments were performed on our computing cluster consisting of four rack mounted servers FUJITSU Server PRIMERGY RX200 S7 R2, interconnected by a gigabit Ethernet switch. Every server had 2 sockets of Intel(R) Xeon(R) CPU E5-2667 clocked at 2.90GHz, each one having 6 physical cores supporting 12 threads, with 48 GB of ECC RAM and 16MB cache. Thus, we could utilize up to 96 threads. The cluster was managed by the Torque Resource Manager 4.2.7[1], supporting the OpenMPI 1.4.5 programming interface and was using the Debian GNU/Linux 7 (wheezy) 64bit operating system. The cluster was using a shared storage area accessible via the Network Filesystem (NFS) mechanism by any process.

[1]http://www.adaptivecomputing.com/products/open-source/torque/

A. Methods compared

We have compared the following methods:

- A baseline sequential solver that runs the MAC search algorithm with dom/wdeg for variable ordering and lexicographic tie breaking.
- A portfolio of algorithms generated using MAC as basis and randomizing the tie breaking of the variable ordering heuristic. That is, every member of the portfolio is essentially MAC with a different seed for the randomization of the tie breaking. This method is called *rand tie break* hereafter.
- A portfolio of algorithms generated using MAC as basis and randomizing the variable selection in the following way. After ranking the variables according to the value of dom/wdeg, one of the top three variables is selected at random. Again each member of the portfolio uses a different random seed. This method is called *rand top 3* hereafter.
- A portfolio of algorithms generated by varying the local consistency enforced in the way described in Section III. The weak (resp. strong) local consistency used was AC (resp. maxRPC). This method is called *LC* hereafter.

To achieve an evaluation that is as fair as possible, all algorithms in the portfolios employ lexicographic value ordering.

B. Variability

In Figure 1 we show the variability in runtimes of the LC portfolio method described above. We show the maximum and minimum runtimes obtained when running a portfolio of 96 solver settings, plus the runtime of MAC, on a sample of 25 instances.

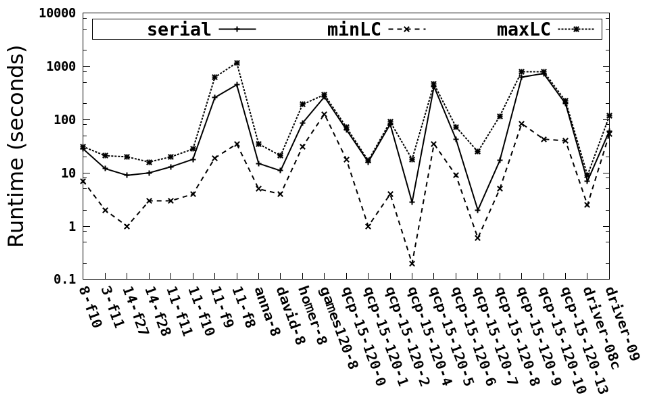

Figure 1. Variability of run times for a portfolio generated by varying the local consistency applied.

Figure 1 demonstrates that the variability of our method is quite high since we obtain run times that can be significantly lower and (sometimes) quite higher than the runtimes of MAC. Importantly, the methods also displays favorability since its minimum run times are constantly lower than those of MAC while in many cases the maximum run times are very close to MAC.

C. Comparing different porfolio schemes

In Figure 2 we compare the three portfolio methods and MAC by displaying the number of solved instances as the time limit imposed is increased from 10 to 800 seconds. These results were obtained using the maximum number of processors (96).

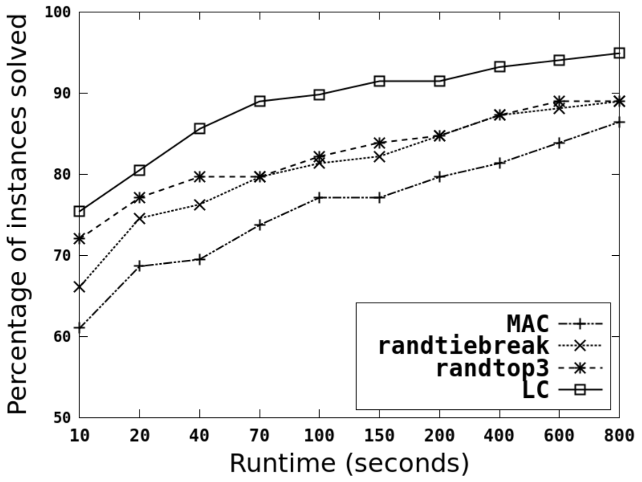

Figure 2. Solved instances as the cpu time limit increases.

From Figure 2 it is evident that our method achieves a considerably better performance than the other portfolio methods as well as the sequential method. For example, when the time limit was set to 70 our method solved 15% more instances than MAC, and 9.3% more instances than both *rand tie break* and *rand top 3*. The portfolio methods that use the variable ordering heuristic as the source of variability were closely matched, with *rand top 3* being slightly more efficient. As expected, all methods become increasingly more efficient as the time limit is increased.

To give a better indication of the differences in run times between the various methods, Table I compares the performance of the tested methods on a selection of problem instances. Apart from run times, we also include the numbers of node visits. The results were obtained using 96 processors and a time limit of 800 seconds.

The data given in Table I demonstrates that LC can be significantly faster than the other methods. The differences in favour of LC are particularly evident in RLFAPs where LC is orders of magnitude faster than the other methods on many instances. Importantly, this problem class includes 7 instances which all methods apart from LC failed to complete within the maximum time limit. Two of these instances (s11-f7 and s11-f16) are included in Table I. Four of these 7 instances were completed by LC in less than 50 seconds (s11-f16 is one of them), with two of them taking only 5 and 9 seconds respectively. Notably, these are all insoluble problems. In qcp and qwh instances the results are rather mixed. There are some instances where *rand tie break* and *rand top 3* were much faster (e.g. qcp-15-0) while in other cases LC is significantly faster (e.g. qwh-20-2). Importantly, most of these are soluble instances. All portfolio methods exhibit high variability in their performance on these instances since different searches may discover different solutions.

Table I

NODES (N) AND CPU TIMES (T) IN SECONDS. A TIME LIMIT OF 800 SECS WAS IMPOSED. AS SLASH (-) INDICATES THAT THE INSTANCE WAS NOT COMPLETED WITHIN THE TIME LIMIT. THE BEST CPU TIME FOR EACH INSTANCE IS HIGHLIGHTED WITH BOLD.

instance	MAC		*tie break*		*top 3*		*LC*	
	(n)	(t)	(n)	(t)	(n)	(t)	(n)	(t)
rlfap								
s11-f9	101.525	259	97.688	245	77.091	221	6.974	**19**
s11-f8	179.025	450	169.043	418	151.166	423	12.992	**35**
s11-f7	-	-	-	-	-	-	9.688	**50**
s11-f6	-	-	-	-	-	-	3.381	**28**
s02-f25	12.688	9	12.463	9	769	**0,5**	1.036	1
s03-f11	9.486	12	9.625	12	3.211	5	1.593	**2**
g08-f10	19.590	28	9.057	14	9.496	18	4.620	**7**
g14-f27	13.833	9	3.304	2	2.988	3	926	**1**
gc								
anna-8	69.321	15	69.321	15	69.312	17	29.772	**5**
david-8	69.280	11	69.280	11	69.280	11	29.944	**4**
homer-8	69.280	86	69.280	88	69.280	99	29.692	**31**
ga120-8	3.208K	267	2.746K	251	2.330K	246	1.383K	**127**
lei450-8	106.517	708	106.663	761	69.819	375	45.496	**256**
driver								
driver-8	3.872	7	1.858	3	514	**1**	711	2
driver-9	14.129	61	10.401	56	6.493	**37**	8.426	54
qcp,qwh								
qcp-15-0	102.136	65	472	**0,2**	355	**0,2**	21.187	18
qcp-15-2	125.130	82	19.047	13	1.063	**0,5**	6.750	4
qcp-15-5	536.056	418	60.141	52	11.591	**11**	28.038	35
qcp-15-9	851.950	625	206.840	173	203.683	160	74.681	**84**
qcp-15-10	1.058.477	732	255.166	169	179.549	131	51.645	**43**
qcp-15-13	269.980	198	164.253	136	159.234	138	37.318	**40**
qwh-20-0	94.013	177	31.054	66	12.656	23	4.075	**8**
qwh-20-2	-	-	168.550	361	235.357	506	22.758	**61**
qwh-20-3	-	-	37.228	75	38.356	85	5.925	**14**
qwh-20-4	231.087	429	11.679	23	7.373	**13**	8.640	20
qwh-20-5	89.151	173	24.202	51	13.063	26	8.052	**19**
qwh-20-6	-	-	76.946	161	35.479	**76**	110.486	328

In Figures 3, 4, 5 we make pairwise comparisons between our method and the other three tested methods. Data points that are situated below the diagonal in each of the figures correspond to instances that were solver faster by LC, while points above the diagonal correspond to instances that were solver faster by one the competing methods (i.e. MAC in Figure 3, *rand tie break* in Figure 4 and *rand top 3* in Figure 5).

Importantly, LC is almost always much faster than MAC. Comparing it to the alternative portfolio methods, we can see that it is very rarely outperformed by *rand tie break*. *Rand top 3* is more competitive but in most of the instances it is inferior to LC. Clearly, a portfolio that combines the strengths of varying the local consistency enforced and diversifying the variable ordering through randomization will be even more competitive. We leave the exploration of efficient ways to build such portfolios as future work.

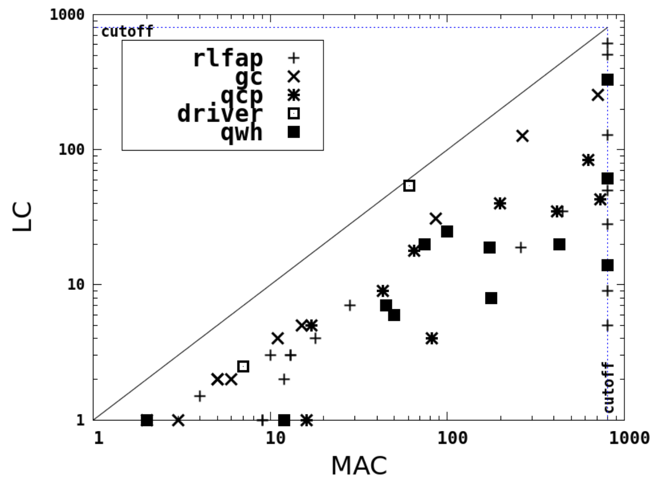

Figure 3. LC vs. MAC

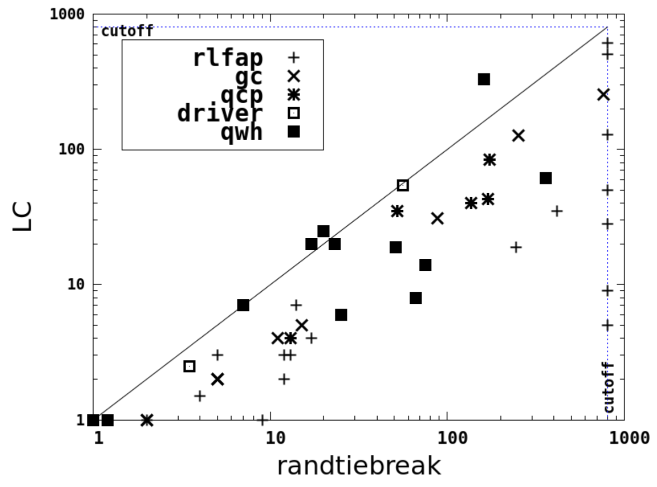

Figure 4. LC vs. random tie breaking

Finally, in Table II we give the percentage of instances where each of the baseline methods in the portfolio reached a solution or proved insolubility first. For example, in 42.1% of the instances a solver setting that used heuristic H_1, with some value for parameter l_{dwo}, was the winner. In brackets we give the percentage of instances where the winning method was one of the eight baseline ones. That is, in 0.9% of the instances the winning method was H_1 with the baseline setting $l_{dwo} = 100$. Absense of brackets for a heuristic indicates that the baseline setting was never the winner.

We can observe that the baseline methods that apply a

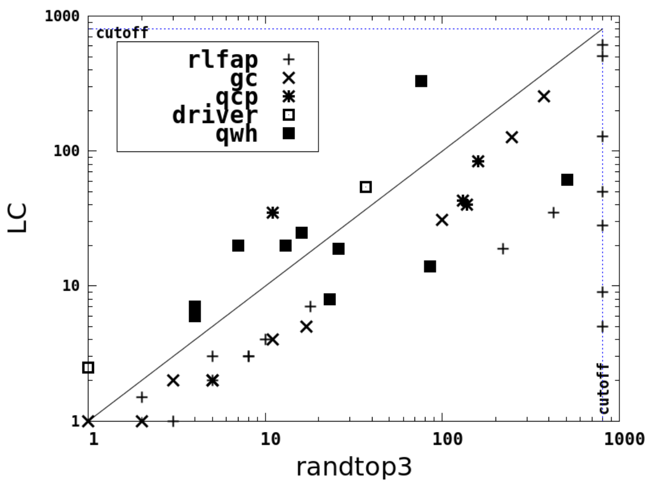

Figure 5. LC vs. random top 3 selection

Table II
PERCENTAGE OF INSTANCES WHERE EACH BASELINE METHOD WAS THE WINNER IN THE PORTFOLIO.

AC	maxRPC	H_1	H_2	H_4	H_{12}	H_{124}
1.9%	2.9%	42.1% (0.9%)	1.9%	0%	21.5%	28.4% (0.9%)

fixed propagation method (AC or maxRPC) do not contribute much to the success of the LC portfolio method. Actually, the instances in which they were the winning configurations are very easy. Hence, they can be omitted when building a portfolio using our methodology, given that a reasonably large number of processors is available. The same holds for heuristics H_2 and H_4. Heuristic H_1 with random settings for its patameter l_{dwo} was the configuration that was most commonly the winner. This is due to the dominance of this heuristic on RLFAPs and graph coloring. On the other hand, the composite heuristics H_{12} and H_{124} with random values for their parameters were dominant on the quasigroup instances.

D. Scalability

Finally, regarding the scalability of the LC portfolio method, Table III gives the run times of the method as the number of used processors is increased for some instances. The results were obtained with the maximum time limit of 800 seconds.

Results demonstrate the scalability of the method. In most cases, as the number of processors rises, so does the efficiency of the method. This is to be expected, especially in the case of soluble problems. However, it seems that with a reasonable number of processors LC can achieve a reasonably good performance level. Thereafter, larger numbers of processors do not always make a significant difference.

Table III
SCALABILITY OF LC'S PERFORMANCE ON SELECTED INSTANCES. A SLASH (-) INDICATES THAT THE INSTANCE WAS TIMED OUT.

instance	processors				
	8	**16**	**32**	**64**	**96**
s11-f6	74	73	65	43	28
s11-f5	599	354	325	154	129
s11-f4	-	-	403	414	607
s11-f3	-	-	232	390	503
qcp-15-5	51	50	42	48	35
qcp-15-9	110	100	101	88	84
qcp-15-10	45	45	40	44	43
qcp-15-13	64	39	33	43	40
qwh-20-1	25	28	25	24	25
qwh-20-2	276	89	71	60	61
qwh-20-6	-	489	357	323	328

V. RELATED WORK

Parallel constraint solving is attracting increasing attention as a result of the advances in parallel computing. [5] presents a recent review of relevant research works. The main approaches to parallel constraint solving can roughly be divided into four categories as detailed in [2].

1) Search Space Splitting explores the parallelism provided by the search space and is the approach that has been most commonly studied. In a few words, when a branching decision is made, the different branches of the search tree that are created can be explored in parallel. A significant challenge that such approaches need to tackle is load balancing: the branches of a search tree are typically extremely imbalanced and therefore there is a non-negligible overhead of communication for work stealing among the different processors. Some works based on this approach are [18], [19], [20], [21], [2], [22]. As is the case with our algorithms, these approaches explore a single search tree and exploit parallelization to speed up this search.
2) A different approach, called *multi-agent search* in [5], that has been studied extensively, especially by the SAT community, is to explore different search trees in parallel. For instance, a portfolio of different solvers, heuristics, or parameter tunings of a single solver can be run in parallel on the same problem. The parallel searches may be completely independent [2], [23] or they may communicate useful information, such as learned clauses, to each other [3], [4], [24]. Diversity in search tree exploration is achieved by communicating learned clauses, by randomizing crucial solver parameters, such as the variable ordering heuristic and the restart schedule, or by other means (e.g. by permuting the variable ordering).
There are fewer works on portfolio approaches for CSPs as the CP community has mainly focused on search space splitting and methods for workload balancing when it comes to parallelization [5]. Notable exceptions are various works by Yun and Epstein who proposed a hybrid approach that uses both a portfolio of different solver settings and search space splitting [25], [26]. The same authors explored the use of case-based reasoning as a tool to build portfolios for parallel constraint solving [27].
3) Problem Splitting is another relevant idea. In this case a problem is split into subproblems and each subproblem is assigned to a different processor. Hence no processor has complete knowledge of the problem. The distributed CSP framework is a typical representative of problem splitting.
4) Finally, one or more components of the solver, e.g. the constraint propagation engine, can be parallelized. Parallelizing constraint propagation algorithms is a challenging task since most such algorithms are sequential by nature [11]. Existing works have focused on AC and have either been purely theoretical [12], or any experiments that were conducted failed to show significant speed-ups [13] or were limited to very few processors [28]. [14] considers the parallelization of a modern CP solvers' constraint propagation engine and shows that problems with a large number of (expensive to propagate) global constraints can benefit from parallelization.

The method proposed in the paper obviously falls in multi-agent search approach, and it specifically concerns portfolios where the parallel searches are completely independent.

VI. CONCLUSION

Portfolio based approaches to constraint solving have been quite successful in both a sequential and a parallel processing mode. Given the increasingly larger number of available processors for parallel processing, an important challenge when designing portfolios is to identify solver parameters that offer diversity in the exploration of the search space and to generate different solver configurations by automatically tuning these parameters.

In this paper we proposed a way to build porfolios for parallel solving by parameterizing the local consistency property applied during search. To our knowledge this is the first time that the propagation method applied is considered as the basis to generate large numbers of different solver configurations. To achieve this we exploited heuristics for adaptive propagation proposed in [1].

We showed how this approach can result in the easy automatic generation of portfolios that display large performance variability. We made an experimental comparison against a standard sequential solver as well as portfolio based methods that use randomization of the variable ordering heuristic as the source of diversity. Results demonstrated that our method constantly outperforms the sequential solver and in most cases it more efficient than the other portfolio approaches.

In the future we first intend to evaluate the proposed method using a wider variety of benchmarks, that include problems with non-binary constraints, and to compare it against alternative parallelization techniques for CSPs (e.g. search space splitting methods). Also, we would like to explore ways of building porfolios by combining randomization in the variable ordering and the local consistency method.

References

[1] K. Stergiou, "Heuristics for Dynamically Adapting Propagation," in *Proceedings of ECAI'08*, 2008, pp. 485–489.

[2] L. Bordeaux, Y. Hamadi, and H. Samulowitz, "Experiments with Massively Parallel Constraint Solving," in *IJCAI*, 2009, pp. 443–448.

[3] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a Parallel SAT Solver," *JSAT*, vol. 6, no. 4, pp. 245–262, 2009.

[4] A. Johannes Hyvärinen, T. Junttila, and I. Niemelä, "Incorporating Clause Learning in Grid-Based Randomized SAT Solving," *JSAT*, vol. 6, no. 4, pp. 223–244, 2009.

[5] I. Gent, C. Jefferson, I. Miguel, N. Moore, P. Nightingale, P. Prosser, and C. Unsworth, "A preliminary review of literature on parallel constraint solving," in *PMCS'11 Workshop on Parallel Methods for Constraint Solving*, 2011.

[6] R. Debruyne and C. Bessière, "Domain Filtering Consistencies," *JAIR*, vol. 14, pp. 205–230, 2001.

[7] C. Bessière, K. Stergiou, and T. Walsh, "Domain filtering consistencies for non-binary constraints," *Artificial Intelligence*, vol. 172, no. 6-7, pp. 800–822, 2008.

[8] C. Bessiere, S. Cardon, R. Debruyne, and C. Lecoutre, "Efficient Algorithms for Singleton Arc Consistency," *Constraints*, vol. 16, pp. 25–53, 2011.

[9] T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh, "New algorithms for max restricted path consistency," *Constraints*, vol. 16, no. 4, pp. 372–406, 2011.

[10] R. Woodward, S. Karakashian, B. Choueiry, and C. Bessiere, "Revisiting Neighborhood Inverse Consistency on Binary CSPs," in *CP*, 2012, pp. 688–703.

[11] S. Kasif, "On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks," *Artif. Intel.*, vol. 45, no. 3, pp. 275–286, 1990.

[12] S. Kasif and A. Delcher, "Local Consistency in Parallel Constraint Satisfaction Networks," *Artif. Intel.*, vol. 69, no. 1-2, pp. 307–327, 1994.

[13] A. Ruiz-Andino, L. Araujo, F. Saenz, and J. Ruz, "Parallel Arc-Consistency for Functional Constraints," in *Workshop on Implementation Technology for Programming Languages based on Logic, ICLP*, 1998, pp. 86–100.

[14] C. Rolf and K. Kuchcinski, "Combining parallel search and parallel consistency in constraint programming," in *TRICS workshop at CP*, 2010, pp. 38–52.

[15] F. Boussemart, F. Heremy, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proceedings of ECAI'04*, 2004, pp. 482–486.

[16] P. Refalo, "Impact-based search strategies for constraint programming," in *Proceedings of CP'04*, 2004, pp. 556–571.

[17] K. Stergiou, "Heuristics for dynamically adapting propagation in constraint satisfaction problems," *AI Communications*, vol. 22, no. 3, pp. 125–141, 2009.

[18] L. Perron, "Search Procedures and Parallelism in Constraint Programming," in *CP*, 1999, pp. 346–360.

[19] J. Jaffar, A. Santosa, R. Yap, and K. Zhu, "Scalable Distributed Depth-First Search with Greedy Work Stealing," in *ICTAI*, 2004, pp. 98–103.

[20] L. Michel, A. See, and P. Van Hentenryck, "Transparent Parallelization of Constraint Programming," *INFORMS Journal on Computing*, vol. 21, no. 3, pp. 363–382, 2009.

[21] G. Chu, C. Schulte, and P. Stuckey, "Confidence-Based Work Stealing in Parallel Constraint Programming," in *CP*, 2009, pp. 226–241.

[22] T. Menouer and B. Le Cun, "A parallelization mixing or-tools/gecode solvers on top of the bobpp framework," in *Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2013*, 2013, pp. 242–246.

[23] M. Aigner, A. Biere, C. Kirsch, A. Niemetz, and M. Preiner, "Analysis of portfolio-style parallel SAT solving on current multi-core architectures," in *POS-13. Fourth Pragmatics of SAT workshop, a workshop of the SAT 2013 conference*, 2013, pp. 28–40.

[24] G. Audemard and L. Simon, "Lazy clause exchange policy for parallel SAT solvers," in *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, 2014, pp. 197–205.

[25] X. Yun and S. Epstein, "A Hybrid Paradigm for Adaptive Parallel Search," in *CP*, 2012, pp. 720–734.

[26] ——, "Adaptive Parallelization for Constraint Satisfaction Search," in *SOCS*, 2012.

[27] ——, "Learning Algorithm Portfolios for Parallel Execution," in *LION*, 2012, pp. 323–338.

[28] T. Nguyen and Y. Deville, "A Distributed Arc-Consistency Algorithm," *Sci. Comput. Program.*, vol. 30, no. 1-2, pp. 227–250, 1998.