A Parametric Framework for Cooperative Parallel Local Search

Danny Munera[1] , Daniel Diaz[1], Salvador Abreu[2] and Philippe Codognet[3]

[1] University of Paris 1-Sorbonne, France
Danny.Munera@malix.univ-paris1.fr, Daniel.Diaz@univ-paris1.fr
[2] Universidade de Évora and CENTRIA, Portugal
spa@di.uevora.pt
[3] JFLI-CNRS / UPMC / University of Tokyo, Japan
codognet@is.s.u-tokyo.ac.jp

Abstract. In this paper we address the problem of parallelizing local search. We propose a general framework where different local search engines cooperate (through communication) in the quest for a solution. Several parameters allow the user to instantiate and customize the framework, like the degree of intensification and diversification. We implemented a prototype in the X10 programming language based on the adaptive search method. We decided to use X10 in order to benefit from its ease of use and the architectural independence from parallel resources which it offers. Initial experiments prove the approach to be successful, as it outperforms previous systems as the number of processes increases.

1 Introduction

Constraint Programming is a powerful declarative programming paradigm which has been successfully used to tackle several complex problems, among which many combinatorial optimization ones. One way of solving problems formulated as a Constraint Satisfaction Problem (CSP) is to resort to *Local Search* methods [13,12], which amounts to the methods collectively designated as *Constraint-Based Local Search* [18]. One way to improve the performance of Local Search Methods is to take advantage of the increasing availability of parallel computational resources. Parallel implementation of local search meta-heuristics have been studied since the early 90's, when multiprocessor machines started to become widely available, see [24]. One usually distinguishes between single-walk and multiple-walk methods. Single-walk methods consist in using parallelism inside a single search process, e.g., for parallelizing the exploration of the neighborhood, while multiple-walk methods (also called multi-start methods) consist in developing concurrent explorations of the search space, either independently or cooperatively with some communication between concurrent processes. A key point is that independent multiple-walk (IW) methods are the easiest to implement on parallel computers and can in theory lead to linear speed-up, cf. [24].

Previous work on independent multi-walk local search in a massively parallel context [2,7,8] achieves good but not ideal parallel speedups. On structured

constraint-based problems such as (large instances of) Magic Square or All-Interval, independent multiple-walk parallelization does not yield linear speedups, reaching for instance a speedup factor of "only" 50-70 for 256 cores. However on the Costas Array Problem, the speedup can be linear, even up to 8000 cores [8]. On a more theoretical level, it can be shown that the parallel behavior depends on the *sequential runtime distribution* of the problem: for problems admitting an exponential distribution, the speedup can be linear, while if the runtime distribution is shifted-exponential or (shifted) lognormal, then there is a bound on the speedup (which will be the asymptotic limit when the number of cores goes to infinity), see [23] for a detailed analysis of these phenomena.

In order to improve the independent multi-walk approach, a new paradigm that includes *cooperation* between walks has to be defined. Indeed, *Cooperative Search* methods add a communication mechanism to the IW strategy, to share or exchange information between solver instances during the search process. However, developing an efficient cooperative method is a very complex task, cf. [6], and many issues must solved: *What information is exchanged? Between what processes is it exchanged? When is the information exchanged? How is it exchanged? How is the imported data used?* [22].

We recently started to work towards a redesigned platform for parallel local search, for which some early results are described in [17]. In the present article we progress towards a general framework, while extending the experimental evaluation to a distributed computing platform.

In this article, we propose a general framework for cooperative search, which defines a flexible and parametric cooperative strategy based on the cooperative multi-walk (CW) scheme. This framework is oriented towards distributed architectures based on clusters of nodes, with the notion of "teams" running on nodes and regrouping several search engines (called "explorers") running on cores, and the idea that all teams are distributed and thus have limited inter-node communication. This framework allows the programmer to define aspects such as the degree of *intensification* and *diversification* present in the parallel search process. A good trade-off is essential to achieve good performance. For instance, a parallel scheme has been developed in [1] with groups of parallel SAT solvers communicating their best configurations on restart, but performance degrades when groups contain more than 16 processes. In [15] another approach is described where a hybrid intensification/diversification is shown to help when scaling into hundreds of cores.

We also propose an implementation of our general cooperative framework and perform an experimental performance evaluation over a set of well-known CSPs. We compare its performance against the Independent Walk implementation and show that in nearly all examples we achieve better performance. Of course, these are just preliminary results and even better performance could be obtained by optimizing the current version. An interesting aspect of the implementation it that we use the X10 programming language, a novel language for parallel processing developed by IBM Research, because it gives us more flexibility than using a more traditional approach, e.g., an MPI communication package.

The rest of the paper is organized as follow. We briefly review the adaptive local search method and present the independent Multi-Walks experiments in section 2. We introduce our Cooperative Search framework in section 3 and, subsequently, present an implementation of this framework in the X10 language in section 4. Section 5 compares the results obtained with both the Independent Multi-Walks implementation and the Cooperative Search implementation. Finally, in section 6, we conclude and propose some ideas for future work.

2 Local Search and parallelism

In this study, we use a generic, domain-independent constraint-based local search method named Adaptive Search [3,4]. This metaheuristic takes advantage of the CSP formulation and makes it possible to structure the problem in terms of variables and constraints and to analyze the current assignment of variables more precisely than an optimization of a global cost function e.g., the number of constraints that are not satisfied. Adaptive Search also includes an adaptive memory inspired in Tabu Search [10] in which each variable leading to a local minimum is marked and cannot be chosen for the next few iterations. A local minimum is a configuration for which none of the neighbors improve the current configuration. The input of the Adaptive Search algorithm is a CSP, for each constraint an error function is defined. This function is a heuristic value to represent the degree of satisfaction of a constraint and gives an indication on how much the constraint is violated. Adaptive Search is based on iterative repair from the variables and constraint error information, trying to reduce the error in the worse variable. The basic idea is to calculate the error function for each constraint, and then combine for each variable the errors of all constraints in which it appears, thus projecting constraint errors on involved variables. Then, the algorithm chooses the variable with the maximum error as a "culprit" and selects it to modify later its value. The purpose is to select the best neighbor move for the culprit variable, this is done by considering all possible changes in the value of this variable (neighbors) and selecting the lower value of the overall cost function. Finally, the algorithm also includes partial resets in order to escape stagnation around local minimum; and it is possible to restart from scratch when the number of iterations becomes too large.

Independent Multi-Walks

To take advantage of the parallelism in Local Search methods different strategies have been proposed like functional parallelism and data parallelism. Functional parallelism aims to parallelize the search algorithm but it generally has too big overheads due to the management of the fine-grained tasks (creation, synchronization and termination) [17]. In contrast, data parallelism tries to parallelize the exploration of the search space. A straightforward implementation of data parallelism is the Independent Multi-Walks (IW) approach. The idea is to use isolated sequential Local Search solver instances dividing the search space of

the problem through different random starting points [24]. This approach has been successfully used in constraint programming problems reaching good performance [2,7,8].

We implemented a IW strategy for the Adaptive Search. This implemention is developed with the PGAS language X10. We tested it on a set of 4 classical benchmarks. Three of them are taken from CSPLib [9]: the All-Interval Problem (AIP, `prob007`) with size 400, Langford's Numbers Problem (LNP, `prob024`) with size 500 and the Magic Square Problem (MSP, `prob019`) with size 200×200. The last benchmark is the Costas Array Problem [14] (CAP) with size 20. For all problems, we select difficult instances involving very large search spaces. These instances are generally out of reach of the traditional complete solvers like Gecode [21].

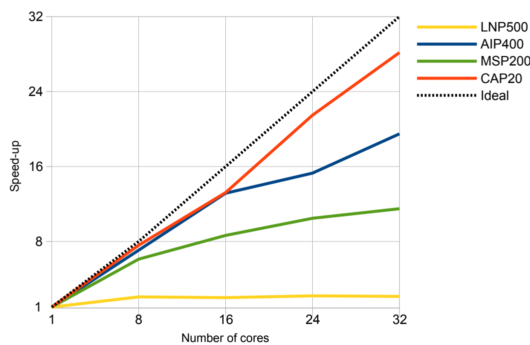

Fig. 1. Speed-ups of Independent Multi-Walks on a distributed system

The testing environment used in each running was a mixed cluster with 5 AMD nodes and 3 Intel nodes. Each AMD node has two quad-core Opteron 2376 processors. Each Intel node has two quad-core Xeon X3450 processors. All systems use a dedicated Gigabit-Ethernet interconnect. Figure 1 shows the speed-ups obtained when increasing the numbers of cores. We solve the instances using 8, 16, 24 and 32 cores.

The results show quasi-linear speed-ups for the CAP instance, in accordance with [8]. However, for MSP and AIP the speed-up tends to flatten out when increasing the number of cores. For instance, for MSP the speed-up is only improved by 1 unit when going from 16 to 32 cores. Finally for LNP the performance is very poor with a speed-up of 2 using 32 cores.

3 Cooperative Search Framework

As seen above, the speed-ups obtained with the IW strategy are good with few compute instances, however when the number of cores increases the performance tends to taper off and the gain is not significant. To tackle this problem, Cooperative Search methods add a communication mechanism to the IW strategy, in order to share information between solver instances while the search is running. Sharing information can improve the probability to get a solution faster than a parallel isolated search. However, all previous experiments indicate that it is very hard to get better performance than IW [16,22].[1] Clearly, this may be explained by the overhead incurred in performing communications, but also by

[1] Sometimes it even degrades performance!

the uncertainty of the benefits stemming from abandoning the current state in favor of another, heuristics-based information which may or may not lead to a solution.

In this work we propose a parametric cooperative local search framework aimed at increasing the performance of parallel implementations based on the Independent Multi-Walks strategy. This framework allows the programmer to define, for each specific problem, a custom trade-off between *intensification* and *diversification* in the search process. Intensification directs the solver to explore deeply a promising part of the search space, while diversification helps to extend the search to different regions of the search space [13].

3.1 Framework Design

Figure 2 presents the general structure of the framework. All available solver instances (*Explorers*) are grouped into *Teams*. Each team implements a mechanism to ensure intensification in the search space, swarming to the most promising neighborhood found by the team members. Simultaneously, all the teams implement a mechanism to collectively provide diversification for the search (outside the groups). The expected effect is that different teams will work on different regions of the search space. Inter-team communication is needed to ensure diversification while intra-team communication is needed for intensification. This framework is oriented towards distributed architectures based on clusters of nodes: teams are mapped to nodes and explorers run on cores. For efficiency reasons it will be necessary to limit inter-node (ie. inter-team) communication.

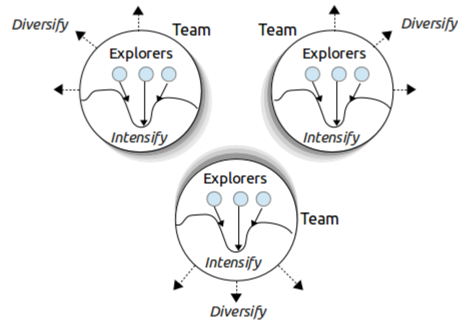

Fig. 2. Cooperative Framework Overview

The first parameter of the framework is the number of nodes per team (`nodes_per_team`), which is directly related to the trade-off between intensification and diversification. This parameter takes values from 1 to the maximum number of nodes (frequently linked to maximum number of available cores for the program in IW). When `nodes_per_team` is equal to 1, the framework coincides with the IW strategy, it is expected that each 1-node team be working on a different region of the search space, without intensification. When the `nodes_per_team` is equal to the maximum number of nodes, the framework has the maximum level of intensification, but there is no diversification at all (only 1 team available). Both diversification and intensification mechanisms are based on the use of communication between nodes. We will explain the precise role of each one in the following section.

Although we presented the framework with Local Search, it is clearly applicable to other metaheuristics as well, such as Simulated Annealing, Genetic Algorithms, Tabu Search, neighboring search, Swarm Optimization, or Ant-Colony

optimization. It is also possible to combine different algorithms in a portfolio approach. For instance a team could implement a local search method, a second team could use a pure tabu search heuristics and another team could try to find a solution using a genetic algorithm. Inside a same team it is also possible to use different versions of a given metaheuristics (e.g. with different values for control parameters).

3.2 Ensuring Diversification

To provide diversification we propose a communication mechanism between teams. The teams share information to compute their current *distance* to other teams (distance between current configurations in the search space). Thus, if two groups are too close, a corrective action is executed. The parameters of this mechanism are defined as follows.

Inter-team communication Topology: This parameter defines the way in which the communications between teams is done. For instance, in the *All-to-All* Scheme each team shares information with every other team; in the *Ring* Scheme each team only shares information with the "adjacent" teams, i.e. the previous and the next teams (e.g., team 5 only communicates with teams 4 and 6). In the *Random* scheme two teams are selected randomly to communicate each other.

Inter-team communication interval: This parameter indicates how frequently the communication between teams occurs. One possible approach is to measure the communication interval in terms of number of iterations elapsed in the main loop of the algorithm.

Distance function: this function is used to check the closeness of two teams (in order to detect if they are exploring a similar region of the search space). For this, the teams compare their current configurations using the distance function. A simple function can count the number of different values in both configurations (i.e. vectors). But, depending on the problem, it is possible to elaborate more complex functions (e.g., taking into account the values and/or the indexes in the vector, using weighted sums...) When a computed distance is lower than the *minimum_permissible_distance* parameter, the two teams are declared too close.

Corrective action: this parameter controls what to do when two teams are too close. In that case one team must correct its trajectory (this can be the "worst" one, i.e. the team whose configuration's cost is the higher). As possible corrective action the team's head node can decide to update its internal state, e.g., clearing its Elite Pool (see below). It can also restart a percentage of the team's explorers, in order to force them to explore a different portion of the search space.

3.3 Ensuring Intensification

We provide intensification by means of a communication mechanism. Here also, it is possible to have different communication topologies between the explorers of

a team. In our framework, we select a topology in which each node communicates with a single other node, thereby constituting a *team* (see Figure 3).

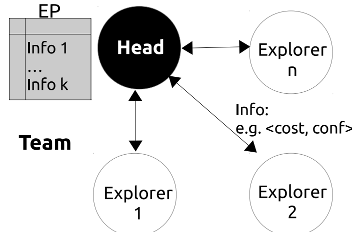

Fig. 3. Structure of a Team

The team is composed of one *head node* and `n` *explorer nodes*. Explorer nodes implement a solver instance of a Local Search method. Each Explorer node periodically reports to the head node, conveying some practical information about its search process (e.g., its current configuration, the associated cost, the number of iterations reached, the number of local minimum reached, etc.). The *head node* then processes the messages and makes decisions to ensure the intensification in the search [2]. Moreover, the head node stores the configurations with the best costs in its Elite Pool (EP) and provides it on demand to Explorers. The head node can also decide to do some actions on its elite pool (e.g., combining configurations to create a new one, similarly to what is done in genetic algorithms).

Explorer Node Explorer nodes periodically ask the head node for an elite configuration from the EP. If the cost of the new EP configuration is lower than its current configuration, the worker node switches to the EP one. Thus the nodes in a group intensify the search process, progressing towards the most promising neighbor found by their group. This is mainly intensification, but if an explorer node luckily finds a better configuration, then the whole team moves to this new promising neighborhood of the search space. The team is not bound to any specific region. The parameters of this mechanism are as follows:

- Report interval: The report interval parameter indicates how frequently the explorer nodes communicate information to the head node.
- Update interval: The update interval parameter indicates how frequently the explorer nodes try to obtain a new configuration from the EP in the head node.

Head Node The head node receives and processes all the information from the explorer nodes in the team. Because of this, it has a more global vision about what is happening in the team and it can make decisions based in the comparative performance of the explorers nodes.

The main duty of the head node is to provide an *intensification* mechanism for the explorer nodes, resorting to the EP. The EP has different tuning parameters that must be defined at design time: First, the *size of the pool* which is the maximum number of configurations that the EP can store. Second, the *entry policy*, which defines the rules to accept or reject the incoming configurations. Finally the *request policy*, which defines which configuration is actually delivered to an explorer node when it makes a request.

[2] Recall, the head node also ensures the diversification by inter-team communication as explained above.

One possible entry policy for this framework is described below. When a message from the explorer node is received by the head node, the algorithm discards the configuration instead of storing it in the EP, in the following situations: (1) if the cost of the configuration is greater than the current worst cost in the EP, (2) if the configuration is already stored in the EP. If the configuration is not discarded, the algorithm then looks for a free slot in the EP. If there is one, the incoming configuration is stored. If not, the algorithm selects a victim configuration (e.g., random, worst, etc.) to be replaced by the new one.

There are many options to implement the request policy in the head node. A simple one is to always return the best configuration in the EP, or any (randomly chosen) configuration of the EP. Also, it is possible to implement more sophisticated algorithms. For instance, a mechanism where the probability of a configuration being selected from the EP is tied to its cost. We may even create a mutation mechanism on the EP, inspired in genetic algorithms [25,5], aspiring to improve the quality of the current configurations.

Although the main function of the head node is to provide intensification within the team, there exist many smart activities that the head node can carry out based in the collected information. For example, it can improve the efficiency of all the nodes in the team by comparing its performance and take corrective decisions, even before an event happens in the explorer node. Also, path relinking techniques [11] can be applied when different local minima have been detected.

4 An X10 Implementation

In order to verify the performance of our cooperative search strategy, we implemented a prototype of the framework using the Adaptive Search method, written in the X10 programming language.

X10 [20] is a general-purpose language developed at IBM, which provides a PGAS variant, Asynchronous PGAS (APGAS), which makes it more flexible and usable even in non-HPC platforms [19]. With this model, X10 supports different levels of concurrency with simple language constructs.

There are two main abstractions in the X10 model: *places* and *activities*. A *place* is the abstraction of a virtual shared-memory process, it has a coherent portion of the address space. The X10 construct for creating a place in X10 is the `at` operation, and is commonly used to create a place for each physical processing unit. An *activity* is the mechanism which abstracts the single threads that perform computation within a place. Multiple activities may be simultaneously active in one place.

Regarding communication, an *activity* may reference objects in other *places*. However, an activity may synchronously access data items only in the place in which it is running. If it becomes necessary to read or modify an object at some other place, the place-shifting operation `at` may be used. For more explicit communication, the `GlobalRef` construct allows cross-place references. `GlobalRef` includes information on the place where an object resides, therefore an activity may locally access the object by *moving* to the corresponding place.

A detailed examination of X10, including tutorials, language specification and examples may be found at `http://x10-lang.org/`.

To implement our framework in X10, we mapped each explorer node to one X10 place, using a solver instance of the Adaptive Search method as in the Independent Multi-Walks strategy. In this implementation, the head nodes also act as explorer nodes in their "spare" time.

The parameter `nodes_per_team` is passed to the main program as a external value. The program reads this value and creates all the instances of the solver together with the necessary references to perform the communication between nodes within a team and between the different teams in the execution.

We used the construct `GlobalRef` to implement communication in X10. Every *head node* reference is passed to the relevant *explorer nodes* of the team, and to the other head nodes in the program. The main loop of the solver has code to trigger all the events in the framework: *Inter-Team Communication event* (between teams), *Report event* (between the explorer nodes and head node into a team) and *Update event* (explorer nodes request a new configuration from the head node).

In the initial implementation, we opted for each explorer node only communicating its current configuration and cost pair $\langle \text{configuration}, \text{cost} \rangle$ to its head node. In the request event, we chose to send a random configuration from the EP to the explorer nodes. For simplicity, this first implementation does not communicate between head nodes of different teams, so diversification is only granted by the randomness of the initial point and the different seeds in each node.

5 Results and Analysis

In this section we compare our X10 implementation[3] of our framework to the independent Multi-Walks version in order to see the gain in terms of speed-ups. For this experiment, we the set of problems presented in section 2.

We used different values for parameter *nodes_per_team:* 2, 4, 8 and 16. The *Report Interval* and the *Update Interval* parameters were set to 100 iterations, finally we tried values from 1 to 4 as the size of the EP. We only retained the results for the best performing parameters. For all cases, we ran 100 samples and averaged the times.

Table 1 compares the Independent Multi-Walks implementation (IW) to our Cooperative Multi-Walks implementation (CW) for each of the problems (LNP, AIP, MSP and CAP). For each problem, a pair of rows presents the speed-up factor of the cooperative strategy CW w.r.t. the independent strategy IW (the best entry in each column is in **bold** fold).

In figure 4 we visualize same results, in a more directly perceptible form. The speed-ups obtained with IW (dotted line) and CW (continuous line) clearly show that in most cases, we are getting closer to a "ideal" speedup. It is worth noticing that AIP is unaffected by the cooperative solver when using a small numbers of

[3] Source at `https://github.com/dannymrock/CSP-X10.git`, branch `teamplaces`.

Problem	time (s) 1 core	Strategy	Speed-up with k cores 8	16	24	32	time (s) 32 cores
AIP-400	280	IW	7.1	13.1	15.3	**19.5**	**14.3**
		CW	**7.3**	**15.3**	**17.3**	19.2	14.6
speedup gain			3.5 %	17 %	13 %	-1.6 %	
LNP-500	19.1	IW	2.1	2.0	2.2	2.1	8.95
		CW	**2.5**	**3.1**	**3.4**	**3.4**	**5.7**
speedup gain			22 %	56 %	54 %	57 %	
MSP-200	274	IW	6.1	8.6	10.5	11.5	23.9
		CW	**8.5**	**14.6**	**15.7**	**18.9**	**14.6**
speedup gain			39 %	69 %	50 %	64 %	
CAP-20	992	IW	7.6	13.2	21.5	28.2	35.2
		CW	**8.9**	**16.8**	**27.6**	**32.2**	**30.8**
speedup gain			18 %	27 %	28 %	15 %	

Table 1. Timings and speed-ups for IW and CW on a distributed system

cores and worse when using 24 or 32 cores. However, For the LNP, MSP and CAP the results clearly show that the cooperative search significantly improves on the performance of the Independent Multi-Walks approach. For instance, in CAP the cooperative strategy actually reaches super linear speed-ups over the entire range of cores (speed-up of 32.2 with 32 cores). The best gain reaches 69% in the MSP.

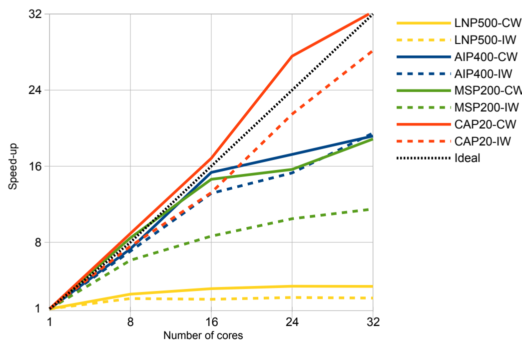

Fig. 4. Speed-Up CW vs IW

This experiments we carried out show that our cooperative framework can improve the time to find a solution for challenging instances in three of four benchmarks. It is clear that the overall performance of our cooperative teams strategy is better than the Independent Multi-Walks implementation. The main source of improvement can be attributed to the search intensification achieved within each team. Intensification ensures that the search always stays in the best neighborhood found by the team. However, diversification is also necessary to ensure the entire set of cores does not get stuck in a local minimum.

6 Conclusion and Further Work

Following up on previous work on parallel implementations, in this paper we are concerned with the design of a general cooperative framework for parallel exe-

cution of local search algorithms, enabling a wide range of experimentation. We decided to work with X10 as the implementation language, because it abstracts over many interesting parallel architectures while retaining a general-purpose stance. The general organization of the proposed framework entails structuring the workers as teams, each with the mission of intensifying the search in a particular region of the search space. The teams are then expected to communicate among themselves to promote search diversification. The concepts and entities involved are all subject to parametric control (e.g., trade-off between intensification and diversification, the team communication topology,...).

The initial experimentation which we carried out with an early prototype already proved to outperform the independent Multi-Walks parallel approach, even with very incomplete parameter tuning. We find these results very encouraging, suggesting that we proceed along this line of work, by defining new organizational and operational parameters as well as extending the experimentation with the ones already introduced.

This being only a preliminary work, and looking forward, we will continue to explore different communication patterns and topologies. The framework we presented relies on Local Search but it is not limited to it. We therefore plan on experimenting with other meta-heuristics or a portfolio search scheme. This is also made convenient by X10's object-oriented setting. It is also important to figure out why problems such as the All-Interval Series (AIP) do not benefit from cooperation among solvers: is it intrinsic to a certain class of problems? Which problems? Can we improve performance with different settings of the framework parameters?

Acknowledgments

The authors wish to acknowledge the Computer Science Department of UNL (Lisbon) for granting us access to its computing resources.

References

1. A. Arbelaez and P. Codognet. Massively Parallel Local Search for SAT. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 57–64, Athens, November 2012. IEEE.
2. Y. Caniou, P. Codognet, D. Diaz, and S. Abreu. Experiments in Parallel Constraint-Based Local Search. In *Evolutionary Computation in Combinatorial Optimization - 11th European Conference, EvoCOP 2011*, pages 96–107, Torino, Italy, 2011. Springer.
3. P. Codognet and D. Diaz. Yet another local search method for constraint solving. In *Stochastic Algorithms: Foundations and Applications*, pages 342–344. Springer Berlin Heidelberg, London, 2001.
4. P. Codognet and D. Diaz. An Efficient Library for Solving CSP with Local Search. In *5th international Conference on Metaheuristics*, pages 1–6, Kyoto, Japan, 2003.
5. O.A.C. Cortes and J. C. da Silva. A Local Search Algorithm Based on Clonal Selection and Genetic Mutation for Global Optimization. In *2010 Eleventh Brazilian Symposium on Neural Networks*, pages 241–246. Ieee, 2010.

6. T. G. Crainic, M. Gendreau, P. Hansen, and N. Mladenovic. Cooperative parallel variable neighborhood search for the p-median. *Journal of Heuristics*, 10(3):293–314, 2004.
7. D. Diaz, S. Abreu, and P. Codognet. Targeting the Cell Broadband Engine for constraint-based local search. *Concurrency and Computation: Practice and Experience (CCP&E)*, 24(6):647–660, 2011.
8. D. Diaz, F. Richoux, Y. Caniou, P. Codognet, and S. Abreu. Parallel Local Search for the Costas Array Problem. In *PCO'12,Parallel Computing and Optimization*, Shanghai, China, May 2012. IEEE.
9. I. P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, 1999.
10. F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, July 1997.
11. F. Glover, M. Laguna, and R. Martí. Fundamentals of Scatter Search and Path Relinking. *Control and Cybernetics*, 29(3):653–684, 2000.
12. T. Gonzalez, editor. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall / CRC, 2007.
13. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann / Elsevier, 2004.
14. S. Kadioglu and M. Sellmann. Dialectic Search. In *Principles and Practice of Constraint Programming (CP)*, volume 5732, pages 486–500, 2009.
15. R. Machado, S. Abreu, and D. Diaz. Parallel local search: Experiments with a pgas-based programming model. *CoRR*, abs/1301.7699, 2013. Proceedings of PADL 2013, Rome, Italy.
16. R. Machado, S. Abreu, and D. Diaz. Parallel Performance of Declarative Programming using a PGAS Model. In *Practical Aspects of Declarative Languages (PADL 2013)*. Springer Berlin / Heidelberg, 2013. (forthcoming).
17. D. Munera, D. Diaz, and S. Abreu. Towards Parallel Constraint-Based Local Search with the X10 Language. In *20th International Conference on Applications of Declarative Programming and Knowledge Management (INAP)*, Kiel, Germany, 2013.
18. V. H. Pascal and M. Laurent. *Constraint-Based Local Search*. The MIT Press, 2005.
19. V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space Model. In *The First Workshop on Advances in Message Passing*, pages 1–8, Toronto, Canada, 2010.
20. V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification - Version 2.3. Technical report, 2012.
21. C. Schulte, G. Tack, and M. Lagerkvist. Modeling and Programming with Gecode, 2013.
22. M. Toulouse, T. Crainic, and M. Gendreau. Communication Issues in Designing Cooperative Multi-Thread Parallel Searches. In *Meta-Heuristics: Theory&Applications*, pages 501–522. Kluwer Academic Publishers, Norwell, MA., 1995.
23. C. Truchet, F. Richoux, and P. Codognet. Prediction of parallel speed-ups for las vegas algorithms. In *ICPP'13, 43rd International Conference on Parallel Processing*. IEEE Press, October 2013.
24. MGA Verhoeven and EHL Aarts. Parallel local search. *Journal of Heuristics*, 1(1):43–65, 1995.
25. Q. Zhang and J. Sun. Iterated Local Search with Guided Mutation. In *IEEE International Conference on Evolutionary Computation*, pages 924–929. Ieee, 2006.

View publication stats