Experiments in Parallel Constraint-Based Local Search

Yves Caniou[1], Philippe Codognet[2], Daniel Diaz[3], and Salvador Abreu[4]

[1] JFLI, CNRS / NII, Japan
[2] JFLI, CNRS / UPMC / University of Tokyo, Japan
[3] University of Paris 1-Sorbonne, France
[4] Universidade de Évora and CENTRIA FCT/UNL, Portugal
`Yves.Caniou@ens-lyon.fr`, `codognet@jfli.itc.u-tokyo.ac.jp`,
`Daniel.Diaz@univ-paris1.fr`, `spa@di.uevora.pt`

Abstract. We present a parallel implementation of a constraint-based local search algorithm and investigate its performance results on hardware with several hundreds of processors. We choose as basic constraint solving algorithm for these experiments the "adaptive search" method, an efficient sequential local search method for Constraint Satisfaction Problems. The implemented algorithm is a parallel version of adaptive search in a multiple independent-walk manner, that is, each process is an independent search engine and there is no communication between the simultaneous computations. Preliminary performance evaluation on a variety of classical CSPs benchmarks shows that speedups are very good for a few tens of processors, and good up to a few hundreds of processors.

1 Introduction

Constraint Programming emerged in the late 1980's as a successful paradigm to tackle complex combinatorial problems in a declarative manner [21]. It is somehow at the crossroads of combinatorial optimization, constraint satisfaction problems (CSP), declarative programming language and SAT problems (boolean constraint solvers and verification tools). Experiments to parallelize constraint problems started in the early days of the Constraint Programming paradigm, by exploiting the search parallelism of the host logic language [22]. Parallel implementation of search algorithms has indeed a long history, especially in the context of Logic Programming [13]. In the field of constraint satisfaction problems (CSP), early work has been done in the context of Distributed Artificial Intelligence and multi-agent systems [38], but these methods, even if interesting from a theoretical point of view, did not lead to efficient algorithms.

In the last decade, with desktop computers turning into parallel machines with 2, 4 or even 8 core CPUs, the temptation to implement efficient parallel constraint solvers has become an increasingly developing research field. Most of the proposed implementations are based on the so-called OR-parallelism, splitting the search space between different processors and relying on the Shared

Memory Multiprocessor architecture as the different processors work on shared data-structures representing a global environment in which the subcomputations take place. Only very few implementations of efficient constraint solvers on such machines have been reported, for instance [34] for a shared-memory architectures with 8 core CPUs. The Comet system [23] has been parallelized for small clusters of PCs, both for its local search solver [28] and its propagation-based constraint solver [29]. Recent experiments have been done up to 12 processors [30], and speedups tend somehow to level after 10 processors. For SAT solvers, several multi-core parallel implementations have also been developed [20,8,35] and similarly for Model Checkers, *e.g.,* the SPIN software [5,24]. More recently [32], a SAT solver has been implemented on a larger PC cluster, using a hierarchical shared memory model and trying to minimize communication between nodes. However performances tend to level after a few tens of processors, *i.e.,* with a speed-up of 16 for 31 processors, 21 for 37 processors and 25 for 61 processors.

In this paper we wanted to address the issue of parallelizing constraint solvers for massively parallel architectures, involving several thousands of CPUs. A design principle implied by this goal is to abandon the classical model of shared data structures which have been developed for shared-memory architectures or tightly controlled master-slave communication in cluster-based architectures and to consider either purely independent parallelism or very limited communication between parallel processes.

Up to now, the only parallel method to solve optimization problems being deployed at large scale is the classical branch and bound, because it does not require much information to be communicated between parallel processes (basically: the current bound, see [17]). It has been recently a method of choice for experimenting the solving of optimization problems using Grid computing, because few data has to be exchanged between nodes [1]. Another implementation, described in [7], uses several hundreds of nodes of the Grid'5000 platform. Good speedups are achieved up to a few hundreds of processors but, interestingly, their conclusion is that the execution time tends to stabilize afterwards.

In [14], the authors proposed to parallelize a constraint solver based on local search using a simple multi-start approach requiring no communication between processes. Experiments done on an IBM BladeCenter with 16 Cell/BE cores show nearly ideal linear speed-ups for a variety of classical CSP benchmarks (magic squares, all-interval series, perfect square packing, etc.). We wanted to investigate if this method could scale up to a larger number of processors, *e.g.,* a few hundreds or a few thousands. We therefore developed a parallel OpenMPI-based implementation from the existing sequential Adaptive Search C-based implementation. This parallel version can run on any system based on OpenMPI, *i.e.,* supercomputer, PC cluster or Grid system. We performed experiments with classical CSP benchmarks from the CSPLIB on two systems:

- the HA8000 machine, an Hitachi supercomputer with a maximum of nearly 16000 cores installed at University of Tokyo,

– the GRID'5000 infrastructure, the French national Grid for the research, which contains 5934 cores deployed on 9 sites distributed in France.

The rest of this paper is organized as follows. Section 2 gives some context and background in parallel local search, while Section 3 presents the Adaptive Search algorithm, a constraint-based local search method based on the CSP formalism. Section 4 details the performance analysis on the parallel hardware. A short conclusion and perspectives end the paper.

2 Local Search and Parallelism

Local Search methods and Metaheuristics [25,19] can be applied to solve CSPs as Constraint Satisfaction can be seen as a branch of Combinatorial Optimization in which the objective function to minimize is the number of violated constraints: a solution is therefore obtained when the function has value zero.

For nearly two decades Local Search methods have been used in SAT solvers for checking the satisfaction of boolean constraints. Since the pioneering algorithms such as GSAT and WalkSAT in the mid 90's, there has been a trend to incorporate more and more local search and stochastic aspects in SAT solvers, in order to cope with ever larger problems [27]. Recently, algorithms such as the ASAT heuristics or Focused Metropolis Search, which incorporate even more stochastic aspects, seem to be among the most effective methods for solving random 3-SAT problems [3].

Parallel implementation of local search metaheuristics has been studied since the early 90's, when multiprocessor machines started to become widely available, see [37,33]. With the increasing availability of PC clusters in the early 2000's, this domain became active again [11,4]. Apart from domain-decomposition methods and population-based method (such as genetic algorithms), [37] distinguishes between single-walk and multiple-walk methods for Local Search. Single-walk methods consist in using parallelism inside a single search process, *e.g.,* for parallelizing the exploration of the neighborhood (see for instance [36] for such a method making use of GPUs for the parallel phase). Multiple-walk methods (parallel execution of multi-start methods) consist in developing concurrent explorations of the search space, either independently or cooperatively with some communication between concurrent processes. Sophisticated cooperative strategies for multiple-walk methods can be devised by using solution pools [12], but requires shared-memory or emulation of central memory in distributed clusters, impacting thus on performances. A key point is that independent multiple-walk methods are the most easy to implement on parallel computers without shared memory and can lead in theory to linear speed-up if solutions are uniformly distributed in the search space and if the method is able to diversify correctly [37]. Interestingly, [2] showed pragmatically that this is the case for the GRASP local search method on a few classical optimization problems such as quadratic assignment, graph planarization, MAX-SAT, maximum covering but this experiment was done with a limited number of processors (28 max).

3 The Adaptive Search Algorithm

Adaptive Search was proposed by [9,10] as a generic, domain-independent constraint based local search method. This meta-heuristic takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a single global cost function to optimize, such as for instance the number of violated constraints. The algorithm also uses a short-term adaptive memory in the spirit of Tabu Search in order to prevent stagnation in local minima and loops. This method is generic, can be applied to a large class of constraints (*e.g.,* linear and non-linear arithmetic constraints, symbolic constraints, etc.) and naturally copes with over-constrained problems. The input of the method is a problem in CSP format, that is, a set of variables with their (finite) domains of possible values and a set of constraints over these variables. For each constraint, an "error function" needs to be defined; it gives, for each tuple of variable values, an indication of how much the constraint is violated. This idea has also been proposed independently by [16], where it is called "penalty functions", and then reused by the Comet system [23], where it is called "violations". For example, the error function associated with an arithmetic constraint $|X - Y| < c$, for a given constant $c \geq 0$, can be $max(0, |X - Y| - c)$. Adaptive Search relies on iterative repair, based on variable and constraint error information, seeking to reduce the error on the worst variable so far. The basic idea is to compute the error function for each constraint, then combine for each variable the errors of all constraints in which it appears, thereby projecting constraint errors onto the relevant variables. This combination of errors is problem-dependent, see [9] for details and examples, but it is usually a simple sum or a sum of absolute values, although it might also be a weighted sum if constraints are given different priorities. Finally, the variable with the highest error is designated as the "culprit" and its value is modified. In this second step, the well known min-conflict heuristic [31] is used to select the value in the variable domain which is the most promising, that is, the value for which the total error in the next configuration is minimal. In order to prevent being trapped in local minima, the Adaptive Search method also includes a short-term memory mechanism to store configurations to avoid (variables can be marked Tabu and "frozen" for a number of iterations). It also integrates reset transitions to escape stagnation around local minima. A reset consists in assigning fresh random values to some variables (also randomly chosen). A reset is guided by the number of variables being marked Tabu. It is also possible to restart from scratch when the number of iterations becomes too large (this can be viewed as a reset of all variables but it is guided by the number of iterations). The core ideas of adaptive search can be summarized as follows:

– to consider for each constraint a heuristic function that is able to compute an approximated degree of satisfaction of the goals (the current "error" on the constraint);
– to aggregate constraints on each variable and project the error on variables thus trying to repair the "worst" variable with the most promising value;

– to keep a short-term memory of bad configurations to avoid looping (*i.e.*, some sort of "tabu list") together with a reset mechanism.

Adaptive Search is a simple algorithm but it turns out to be quite efficient in practice. The following table compares its performances with the Comet 2.1.1 system on a few benchmarks from CSPLib [18], included in the distribution of Comet. Timings are in seconds and taken for both solvers on a PC with a Core2 Duo E7300 processor at 2.66 GHz, and are the average of 100 executions for AS and of 50 executions for Comet. Of course, it should be noticed that Comet is a complete and very versatile system while Adaptive Search is just a C-based library, but one can see that Adaptive Search is about two orders of magnitude faster than Comet. Also note that [26] compares a new metaheuristics named Dialectic Search with the older (2001) version of Adaptive Search [9], showing that both methods have similar results. However when using the timings from [10], the newer (2003) version of Adaptive Search is about 15 to 40 times faster than Dialectic Search on the same reference machine.

Table 1. Execution times and speedups of Adaptive Search vs Comet

Benchmark	Comet	Adaptive Search	Speedup
Queens n=10000	24.5	0.52	47
Queens n=20000	96.2	2.16	44.5
Queens n=50000	599	13.88	43.2
Magic Square 30x30	56.5	0.34	166
Magic Square 40x40	199	0.53	375
Magic Square 50x50	609	1.18	516

We can thus state the overall Adaptive Search algorithm as follows:

Input:
A problem given in CSP format:
- a set of variables $V = \{V_1, V_2, ..., V_n\}$ with associated domains
- a set of constraints $C = \{C_1, C_2, ..., C_k\}$ with associated error functions
- a combination function to project constraint errors on variables
- a (positive) cost function to minimize
And some tuning parameters:
- T: Tabu tenure (number of iterations a variable is frozen)
- RL: reset limit (number of frozen variables to trigger reset)
- RP: reset percentage (percentage of variables to reset)
- Max_I: maximal number of iterations before restart
- Max_R: maximal number of restarts

Output:
A solution (configuration where all constraints are satisfied) if the CSP is satisfied or a quasi-solution of minimal cost otherwise.

Algorithm

```
Restart = 0
Repeat
    Restart = Restart + 1 ; Iteration = 0 ; Tabu_Nb = 0
    compute a random assignment A of variables in V
    Opt_Sol = A ; Opt_Cost = cost(A)
    Repeat
            Iteration = Iteration +1
            compute errors of all constraints in C and combine errors on each var.
                    (by considering only the constraints in which a variable appears)
            select the variable X (not marked Tabu) with highest error
            evaluate costs of possible moves from X
            if      no improvement move exists
            then mark X as Tabu until Iteration + T
                    Tabu_Nb = Tabu_Nb + 1
                    if      Tabu_Nb ≥ RL
                    then randomly reset RP variables in V
                            (and unmark those which are Tabu)
            else    select the best move and change the value of X
                    accordingly to produce next configuration A'
                    if      cost(A') < Opt_Cost
                    then Opt_Sol = A = A' ; Opt_Cost = cost(A')
    until a solution is found or Iteration ≥ Max_I
until a solution is found or Restart ≥ Max_R
output (Opt_Sol, Opt_Cost)
```

4 Parallel Performance Analysis

We used the implementation of the Adaptive Search method consisting of a C-based framework library available as freeware at the URL: `http://contraintes.inria.fr/∼ diaz/adaptive/`

The parallelization of the Adaptive Search method was done with OpenMPI, an implementation of the MPI standard [15]. The idea of the parallelization is straightforward, and based on the idea of multi-start and independent multiple-walks: fork a sequential Adaptive Search method on every available cores. But on the opposite of the classical fork-join paradigm, parallel Adaptive Search shall terminate as soon as a solution is found, not wait until all the processes have finished (since some searches initialized with "bad" initial configurations can take some time). Thus, some non-blocking tests are involved every c iterations to check if there is a message indicating that some other processes has found a solution; in which case it terminates the execution properly. Note however that several processes can find a solution "at the same time", *i.e.,* during the same c-block of iterations. Thus, those processes send their statistics (among which the execution time) to the process 0 which will then determine which of them is actually the fastest.

Three testbeds were used to perform our experiments:

- **HA8000**, the Hitachi HA8000 supercomputer of the University of Tokyo with a total number of 15232 cores. This machine is composed of 952 nodes, each of which is composed of 4 AMD Opteron 8356 (Quad core, 2.3 GHz) with 32 GB of memory. Nodes are interconnected with a Myrinet-10G network with a full bisection connection, attaining 5 GB/sec in both directions. HA8000 can theoretically achieve a performance of 147 Tflops, but we only accessed to a subset of its nodes as users can only have a maximum of 64 nodes (1,024 cores) in normal service.
- GRID'5000 [6], the French national Grid for the research, which contains 5934 cores deployed on 9 sites distributed in France. We used two subsets of the computing resources of the Sophia-Antipolis node: **Suno**, composed of 45 Dell PowerEdge R410 with 8 cores each, thus a total of 360 cores, and **Helios**, composed of 56 Sun Fire X4100 with 4 cores each, thus a total of 224 cores.

We use a series of classical benchmarks from CSPLib [18] consisting of:

- all-interval: the All Interval Series problem (prob007 in CSPLib),
- perfect-square: the Perfect Square placement problem (prob009 in CSPLib),
- magic-square: the Magic Square problem (prob019 in CSPLib).

Although these benchmarks are academic, they are abstractions of real-world problems and could involve very large combinatorial search spaces, *e.g.,* the 400x400 magic square problem requires 160000 variables whose domains range over 160000 values and the time to find a solution on a single processor by local search is *nearly 2 hours* on average. Classical propagation-based constraint solvers cannot solve this problem for instances higher than 10x10. Also note that we are tackling constraint *satisfaction* problems as optimization problems, that is, we want to minimize the global error (representing the violation of constraints) to value zero, therefore finding a solution means that we actually reach the bound (zero) of the objective function to minimize.

Table 2. Speedups on HA8000, Suno and Helios

Platform	Problem	Time on	Speedup on k cores				
		1 core	16	32	64	128	256
HA8000	MS 400	6282	10.6	20.6	31.7	41.3	54.1
	Perfect 5	42.7	15.0	29.5	44.6	49.1	57.0
	A-I 700	638	8.19	14.8	17.8	23.4	27.7
Suno	MS 400	5362	8.4	22.8	32.6	41.3	52.8
	Perfect 5	106	15.1	23	46.1	70.7	106
	A-I 700	662	10.1	15.8	19.9	23.9	28.3
Helios	MS 400	6565	13.2	20.6	31	44	-
	Perfect 5	139.7	15.8	24.5	46.6	77.2	-
	A-I 700	865.8	9.1	14.9	23.5	27.3	-

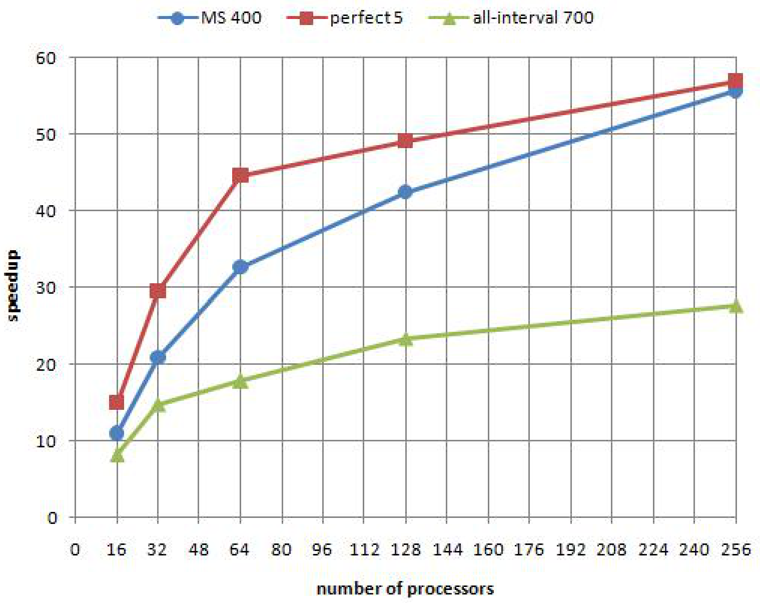

Fig. 1. Speedups on HA8000

Table 2 presents the execution times and speedups for executions up to 256 cores on HA8000 and on the GRID'5000 platform. The same code has been ported and executed, timings are given in seconds and are the *average of 50 runs*, except for MS 400 on HA8000 where it is the average of 20 runs.

We can see that the speedups are more or less equivalent on both platforms. Only in the case of `perfect-square` are the results significantly different between the two platforms, for 128 and 256 cores. In those cases GRID'5000 has much better speedups than on HA8000. Maybe this is because execution time is getting too small (less than one second) and therefore some other mechanisms interfere. The stabilization point is not yet obtained for 256 cores, even if speedups do not increase as fast as the number of cores, *i.e.,* are getting further away from linear speedup. This is visually depicted on Fig. 1 and Fig. 2. As the speedups on the two GRID'5000 platforms (Helios and Suno nodes) are nearly identical, we only depicted the speedups with Suno, as we can experiment up to 256 cores on this platform.

4.1 Discussion

As we can see in the results obtained, the parallelization of the method gives good benefits on both the HA8000 and the GRID'5000 platforms, achieving speedups of about 30 with 64 cores, 40 with 128 cores and more than 50 with 256 cores. Of course speedups depend on the benchmarks and the bigger the benchmark, the better the speedup.

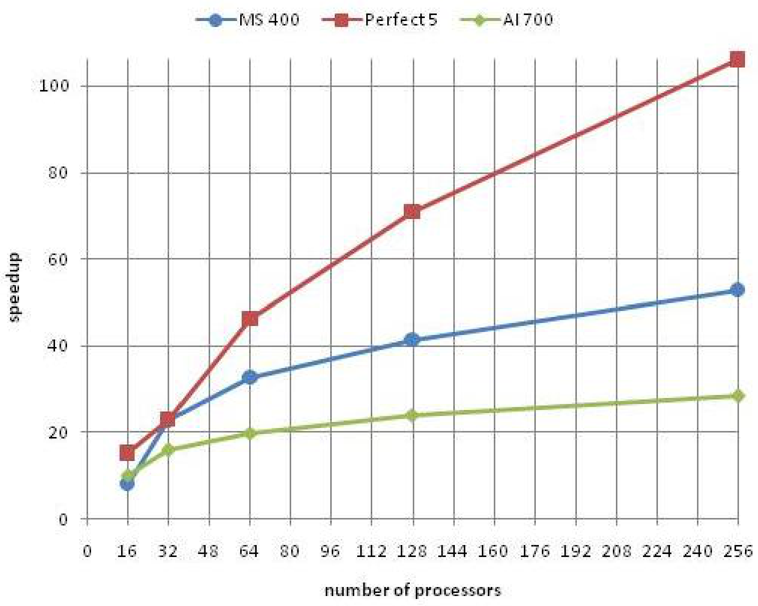

Fig. 2. Speedups on Grid5000 (Suno)

To see the impact of the problem size on performances, let us detail a single benchmark, `magic square`, on three instances of increasing difficulty. Table 3 details the performances on HA8000 for the following instances: 100x100, 120x120 and 200x200. The three plots on Fig. 3 show a similar shape, but the bigger the benchmark, the better the parallel speedup, and for those smaller benchmarks the speedup curve start to flatten after 64 processors.

As these experiments show that every speedup curves tend to flatten at some point, it suggests that there is maybe an intrinsically sequential aspect in local search methods and that the improvement given by the multi-start aspect might reach some limit when increasing the number of parallel processors. This might be theoretically explained by the fact that, as we use structured problem instances and not random instances, solutions may be not uniformly distributed in the search space.

Table 3. Performances for `magic square` on HA8000

# cores	MS 100		MS 120		MS200	
	time	speed	time	speed	time	speed
1	18.2	1.0	53.4	1.0	338	1.0
8	2.16	8.41	5.84	9.14	42.3	8.0
16	1.69	10.8	3.99	13.4	22.4	15.1
32	1.43	12.7	3.03	17.7	14.8	22.9
64	1.20	15.1	2.26	23.6	12.2	27.8
128	1.16	15.5	2.24	23.9	12.1	28.0

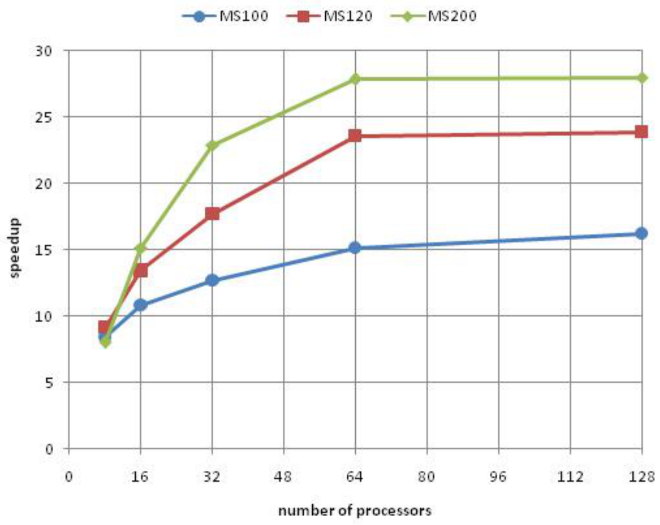

Fig. 3. Speedups for 3 instances of `magic square` on HA8000

5 Conclusion and Future Work

We presented a parallel implementation of a constraint-based local search algorithm, the "Adaptive Search" method in a multiple independent-walk manner. Each process is an independent search engine and there is no communication between the simultaneous computations except for completion. Performance evaluation on a variety of classical CSPs benchmarks and on two different parallel architectures (a supercomputer and a Grid platform) shows that the method is achieving speedups of about 30 with 64 cores, 40 with 128 cores and more than 50 with 256 cores. Of course speedups depend on the benchmarks and the bigger the benchmark, the better the speedup.

In order to take full advantage of the execution power at hand (*i.e.,* hundreds or thousands of processors), we have to seek a new way to further increase the benefit of parallelization. We are currently working on a more complex algorithm, with communication between parallel processes in order to reach better performances. The basic idea is as follows: Every c iteration a process will send the value of its current best total configuration cost to other processes. Every c iteration each process also checks messages from other processes and if it received a message with a cost lower than its own cost, which means that it is further away from a solution, then it can decide to stop its current computation and make a random restart. This will be done following a given probability p. Therefore the two key parameters are c, the number of iterations between messages and p, the probability to make a restart. We are currently experimenting this algorithm with various values for the benchmarks described in this paper.

References

1. Aida, K., Osumi, T.: A case study in running a parallel branch and bound application on the grid. In: SAINT 205: Proceedings of the the 2005 Symposium on Applications and the Internet, pp. 164–173. IEEE Computer Society, Washington, DC, USA (2005)
2. Aiex, R.M., Resende, M.G.C., Ribeiro, C.C.: Probability distribution of solution time in grasp: An experimental investigation. Journal of Heuristics 8(3), 343–373 (2002)
3. Alava, M., Ardelius, J., Aurell, E., Kaski, P., Orponen, P., Krishnamurthy, S., Seitz, S.: Circumspect descent prevails in solving random constraint satisfaction problems. PNAS 105(40), 15253–15257 (2007)
4. Alba, E.: Special issue on new advances on parallel meta-heuristics for complex problems. Journal of Heuristics 10(3), 239–380 (2004)
5. Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core LTL model-checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 187–203. Springer, Heidelberg (2007)
6. Bolze, R., et al.: Grid 5000: A large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. 20(4), 481–494 (2006)
7. Caromel, D., di Costanzo, A., Baduel, L., Matsuoka, S.: Grid'BnB: a parallel branch and bound framework for grids. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 566–579. Springer, Heidelberg (2007)
8. Chu, G., Stuckey, P.: A parallelization of MiniSAT 2.0. In: Proceedings of SAT race (2008)
9. Codognet, P., Diaz, D.: Yet another local search method for constraint solving. In: Steinhöfel, K. (ed.) SAGA 2001. LNCS, vol. 2264, pp. 73–90. Springer, Heidelberg (2001)
10. Codognet, P., Diaz, D.: An efficient library for solving CSP with local search. In: Ibaraki, T. (ed.) MIC 2003, 5th International Conference on Metaheuristics (2003)
11. Crainic, T., Toulouse, M.: Special issue on parallel meta-heuristics. Journal of Heuristics 8(3), 247–388 (2002)
12. Crainic, T.G., Gendreau, M., Hansen, P., Mladenovic, N.: Cooperative parallel variable neighborhood search for the -median. Journal of Heuristics 10(3), 293–314 (2004)
13. de Kergommeaux, J.C., Codognet, P.: Parallel logic programming systems. ACM Computing Surveys 26(3), 295–336 (1994)
14. Diaz, D., Abreu, S., Codognet, P.: Parallel constraint-based local search on the cell/BE multicore architecture. In: Essaaidi, M., Malgeri, M., Badica, C. (eds.) Intelligent Distributed Computing IV. Studies in Computational Intelligence, vol. 315, pp. 265–274. Springer, Heidelberg (2010)
15. Gabriel, E., et al.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)
16. Galinier, P., Hao, J.-K.: A general approach for constraint solving by local search. In: 2nd Workshop CP-AI-OR 2000, Paderborn, Germany (2000)
17. Gendron, B., Crainic, T.: Parallel branch-and-bound algorithms: Survey and synthesis. Operations Research 42(6), 1042–1066 (1994)
18. Gent, I.P., Walsh, T.: CSPlib: A benchmark library for constraints. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 480–481. Springer, Heidelberg (1999)

19. Gonzalez, T. (ed.): Handbook of Approximation Algorithms and Metaheuristics. Chapman and Hall / CRC, Boca Raton (2007)
20. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 6, 245–262 (2009)
21. Hentenryck, P.V.: Constraint Satisfaction in Logic Programming. The MIT Press, Cambridge (1989)
22. Hentenryck, P.V.: Parallel constraint satisfaction in logic programming: Preliminary results of CHIP within PEPSys. In: International Conference on Logic Programming, pp. 165–180. MIT Press, Cambridge (1989)
23. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. The MIT Press, Cambridge (2005)
24. Holzmann, G.J., Bosnacki, D.: The design of a multicore extension of the spin model checker. IEEE Transactions on Software Engineering 33(10), 659–674 (2007)
25. Ibaraki, T., Nonobe, K., Yagiura, M. (eds.): Metaheuristics: Progress as Real Problem Solvers. Springer, Heidelberg (2005)
26. Kadioglu, S., Sellmann, M.: Dialectic search. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 486–500. Springer, Heidelberg (2009)
27. Kautz, H.A., Sabharwal, A., Selman, B.: Incomplete algorithms. In: Biere, A., Heule, M., van Maaren, H., Walsch, T. (eds.) Handbook of Satisability. IOS Press, Amsterdam (2008)
28. Michel, L., See, A., Van Hentenryck, P.: Distributed constraint-based local search. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 344–358. Springer, Heidelberg (2006)
29. Michel, L., See, A., Van Hentenryck, P.: Parallelizing constraint programs transparently. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 514–528. Springer, Heidelberg (2007)
30. Michel, L., See, A., Van Hentenryck, P.: Parallel and distribited local search in comet. Computers and Operations Research 36, 2357–2375 (2009)
31. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. Artificial Intelligence 58(1-3), 161–205 (1992)
32. Ohmura, K., Ueda, K.: c-SAT: A parallel SAT solver for clusters. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 524–537. Springer, Heidelberg (2009)
33. Pardalos, P.M., Pitsoulis, L.S., Mavridou, T.D., Resende, M.G.C.: Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing, tabu search and GRASP. In: Ferreira, A., Rolim, J.D.P. (eds.) IRREGULAR 1995. LNCS, vol. 980, pp. 317–331. Springer, Heidelberg (1995)
34. Perron, L.: Search procedures and parallelism in constraint programming. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 346–360. Springer, Heidelberg (1999)
35. Schubert, T., Lewis, M.D.T., Becker, B.: Pamiraxt: Parallel sat solving with threads and message passing. Journal on Satisfiability, Boolean Modeling and Computation 6, 203–222 (2009)
36. Van Luong, T., Melab, N., Talbi, E.-G.: Local search algorithms on graphics processing units. In: Cowling, P., Merz, P. (eds.) EvoCOP 2010. LNCS, vol. 6022, pp. 264–275. Springer, Heidelberg (2010)
37. Verhoeven, M., Aarts, E.: Parallel local search. Journal of Heuristics 1(1), 43–65 (1995)
38. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms. IEEE Transactions on Knowledge and Data Engineering 10(5), 673–685 (1998)