Parallel Constraint-Based Local Search on the HA8000 supercomputer (Abstract)

Yves Caniou
JFLI, CNRS / NII
Japan
Yves.Caniou@ens-lyon.fr

Philippe Codognet
JFLI, CNRS / UPMC / University of Tokyo
Japan
codognet@jfli.itc.u-tokyo.ac.jp

Daniel Diaz
Université Paris-I Sorbonne
France
Daniel.Diaz@univ-paris1.fr

Salvador Abreu
Universidade de Evora and CENTRIA FCT/UNL
Portugal
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.

1. INTRODUCTION

In this paper we want to address the issue of parallelizing constraint solvers for massively parallel architectures, involving several thousands of CPUs. In [3], 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 parallel constraint solving could scale up to a larger number of processors, e.g. a few hundreds or a few thousands. Most previous experiments in parallel constraint solving have been done with just few processors [8, 5, 6]. They showed good speedups, but does it scale up? Experiments with the Comet system show that speedups tend to stabilize after 10 processors [7].

2. THE ADAPTIVE SEARCH ALGORITHM

Adaptive Search was proposed by [1, 2] as a generic, domain-independent constraint-based local search method. It is a simple algorithm but it turns out to be quite efficient in practice 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 an 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 will give, for each tuple of variable values, an indication of how much the constraint is violated. For instance, 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. Finally, the variable with the highest error will be designated as the "culprit" and its value will be modified. In this second step, the well known min-conflict heuristic 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. As in any local search method, it is also possible to restart from scratch when the number of iterations reaches a given limit.

3. 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:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'11 March 21-25, 2011, TaiChung, Taiwan.
Copyright 2011 ACM 978-1-4503-0113-8/11/03 ...$10.00.

http://cri-dist.univ-paris1.fr/diaz/adaptive/
Parallelization was done by using OpenMPI, an implementation of the MPI standard. The main idea is straightforward and based on the idea of multi-start and independent multiple-walks: just fork a sequential Adaptive Search engine on every available core, starting from a different (random) configuration.

We performed experiments on the Hitachi HA8000 supercomputer of the University of Tokyo. 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. It therefore contains a total number of 15232 cores. These 952 nodes are divided in five sub-clusters of respectively 16, 36, 256, 128 and 512 nodes. 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.

The benchmarks used were taken from CSPLib [4]: the *All-Interval Series* problem (prob007 in CSPLib), the *Perfect-Square* placement problem (prob009 in CSPLib), and the *Magic Square* problem (prob019 in CSPLib). Although these benchmarks are academic, they are abstractions of real-world problems and involve very large combinatorial search spaces, e.g. the 200x200 magic square problem requires 40000 variables whose domains range over 40000 values. Classical propagation-based constraint solvers cannot solve this problem for instances higher than 10x10.

The execution times in the table below are in seconds and represent the average of 50 runs; they only account for the actual solving time.

# cores	MS 120		perfect 5		A-I 450	
	time	speed	time	speed	time	speed
1	53.4	1.0	36.2	1.0	177	1.0
8	5.84	9.14	3.86	9.39	18.6	9.53
16	3.99	13.4	2.49	15.5	11.7	15.1
32	3.03	17.7	1.45	25.0	6.29	28.1
64	2.26	23.6	1.19	30.3	5.58	31.7
128	2.24	23.9	1.01	35.9	5.09	34.8

Table 1: execution times and speedups on HA8000

We can see that speedups tend to stabilize after 64 processors and that only a marginal gain (about 10%) is then achieved when doubling the execution power, that is, using 128 processors. When using 256 processors an even smaller gain is achieved.

If one concentrates on a single benchmark, e.g. *Magic Square*, and consider various problem instances, better speedups can of course be achieved with larger instances, but they all follow the same pattern. Figure 1 details the performances for the instances 100x100, 120x120 and 200x200, together with that of *Perfect Square* (instance 5) and *All-Interval* (n=450).

4. 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.

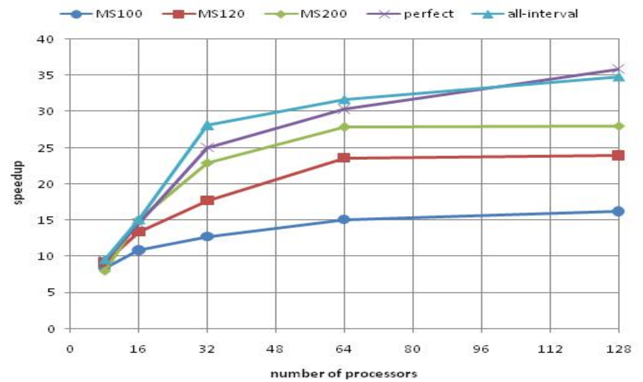

Figure 1: speedups on HA8000

As we can see in the obtained results, the parallelization of the method gives benefits, but the speedup tends to stabilize after 64 processors. Of course speedups depend on benchmarks and the bigger the benchmark, the better the speedups; also benchmarks are big enough to let all processors busy during the whole execution time. But these experiments point out that there is maybe an intrinsically sequential aspect in local search methods and that the improvement given by the multi-start approach might reach some limit.

We are currently working on a more complex algorithm, with communication between parallel processes in order to reach better performances.

5. REFERENCES

[1] P. Codognet and D. Diaz. Yet another local search method for constraint solving. In *proceedings of SAGA'01*, pages 73–90. Springer Verlag, 2001.

[2] P. Codognet and D. Diaz. An efficient library for solving CSP with local search. In T. Ibaraki, editor, *MIC'03, 5th International Conference on Metaheuristics*, 2003.

[3] D. Diaz, S. Abreu, and P. Codognet. Parallel constraint-based local search on the cell/be multicore architecture. In *proceedings of Intelligent Distributed Computing IV*. Springer Verlag, 2010.

[4] I. P. Gent and T. Walsh. Csplib: A benchmark library for constraints. In *proceedings of CP'99*, pages 480–481. Springer Verlag, 1999.

[5] L. Michel, A. See, and P. V. Hentenryck. Distributed constraint-based local search. In F. Benhamou, editor, *proceedings of CP'06*, pages 344–358. Springer Verlag, 2006.

[6] L. Michel, A. See, and P. Van Hentenryck. Parallelizing constraint programs transparently. In C. Bessiere, editor, *proceedings of CP'07*, pages 514–528. Springer Verlag, 2007.

[7] L. Michel, A. See, and P. Van Hentenryck. Parallel and distribited local search in comet. *Computers and Operations Research*, 36:2357–2375, 2009.

[8] L. Perron. Search procedures and parallelism in constraint programming. In *proceedings of CP'99*, pages 346–360. Springer Verlag, 1999.