Parallel Hybrid Best-First Search for Cost Function Networks

Abdelkader Beldjilali, David Allouche, Simon de Givry
INRA, MIA Toulouse, UR-875
31320 Castanet-Tolosan, France

Introduction

Cost Function Networks (CFNs), also known as Weighted Constraint Satisfaction Problems (Meseguer, Rossi, and Schiex 2006) is a mathematical model which has been derived from Constraint Satisfaction Problems by replacing constraints with cost functions. In a CFN, we are given a set of variables with an associated finite domain and a set of local cost functions. Each cost function involves some variables and associates a non-negative integer cost to each of the possible combinations of values they may take. The usual problem considered is to assign all variables in a way that minimizes the sum of all costs. This problem is NP-hard, and exact methods usually rely on Branch and Bound (B&B) algorithms exploring a binary search tree with propagation at each node in order to improve the problem lower bound and prune domain values with a forbidden cost (Cooper et al. 2010). Several B&B search methods have been developed in the CFN solver `toulbar2`[1].

In this work we describe a first parallel version of Hybrid Best-First Search (HBFS) (Allouche et al. 2015) and give a preliminary empirical evaluation on combinatorial optimization problems from uncapacitated warehouse location, computational protein design, and genetics. This last section includes solving time and speed-up comparisons between our approach within `toulbar2` and other parallel approaches available in IBM Ilog `cplex` and `daoopt` (Otten and Dechter 2017) (i.e., respectively a Mixed Integer Programming and an AND/OR search Graphical Model solver).

Parallel HBFS

The sequential version of HBFS (Allouche et al. 2015) is a B&B method for CFNs that combines Best-First Search (BFS) and Depth-First Search (DFS). Like BFS, HBFS provides an anytime global lower bound on the optimum, while also providing anytime upper bounds, like DFS. Hence, it provides feedback on the progress of search and solution quality in the form of an optimality gap. Besides, it exhibits highly dynamic behavior that allows it to perform on par with methods like Limited Discrepancy Search (LDS) and frequent restarting in terms of quickly finding good solutions. As in BFS, HBFS maintains a frontier of open search nodes. It expands each open node using DFS with a limit on its number of backtracks. Each bounded DFS returns a new list of open nodes to be inserted in the BFS frontier.

The parallel version of HBFS is based on the Master-Worker parallel paradigm (Ralphs et al. 2018) where the *Master* is in charge of the open node frontier and dispatches the current best (with minimum lower bound) open node plus the current best solution found so far to the next available *Worker*. The Worker performs a bounded DFS starting from the received node and returns to the Master the resulting list of open nodes (see Fig. 1, with a DFS limit of 3 backtracks). Each open node is associated to a corresponding lower bound and a vector of search decisions. The Worker also returns the best solution found during its limited search if any. Only the Master has a global view of the whole search and reports optimality gaps until the proof of optimality is reached (when the current best frontier lower bound, including active Worker starting nodes, is equal or greater than the cost of the best solution found so far).

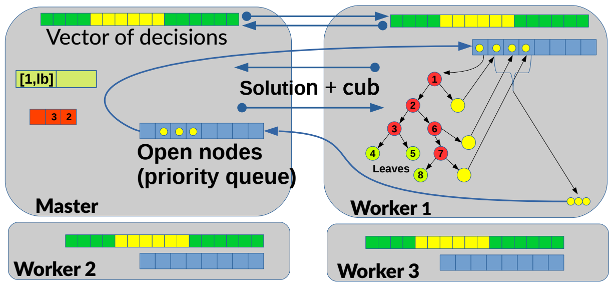

Figure 1: Parallel HBFS using the Master-Worker paradigm.

Instance	n	d	Time (sec.)		Speed-up
			`HBFS`	`HBFS-24`	
capmo1	200	100	10.92	**5.14**	2.12
capmo2	200	100	**1.80**	2.04	0.88
capmo3	200	100	6.09	**3.73**	1.63
capmo4	200	100	4.36	**3.21**	1.36
capmo5	200	100	2.69	**2.58**	1.04

Table 1: Warehouse benchmark (Larrosa et al. 2005) with n, number of variables, and d, maximum domain size.

Experimental Results

We implemented parallel HBFS inside `toulbar2` using the MPI library. Experiments were performed on 24-core

[1] `https://github.com/toulbar2/toulbar2`

servers (Intel Xeon E5-2680/87 at 2.50/3GHz and 256 GB) and the GenoToul cluster (64-core nodes of Intel Xeon E5-2683 at 2.10GHz).

Instance	n	d	Time (sec.)		Speed-up
			`HBFS`	`HBFS-24`	
1xaw	107	412	721.43	**568.50**	1.27
3lf9	120	416	**407.28**	407.92	1
5dbl	130	384	**122.84**	171.82	0.71
5e10	133	400	**147.73**	198.23	0.75
5e0z	136	420	**148.26**	193.41	0.77
5eqz	138	434	3,366.11	**1,049.11**	3.21
1dvo	152	389	622.03	**463.29**	1.34
4bxp	170	439	**327.46**	395.16	0.83
1is1	185	431	-	**2,545.82**	-
2gee	188	397	**797.22**	863.64	0.92
5jdd	263	406	-	**2,758.98**	-
3r8q	271	418	1,605.30	**1,294.97**	1.24
1f00	282	430	-	**2,140.40**	-

Table 2: Computational Protein Design (CPD) benchmark (Ouali et al. 2017). A '–' indicates that the corresponding method failed to prove optimality in less than 1 hour.

Instance	n	d	`cplex`	`cplex-10`	`HBFS`	`HBFS-10`	Speed-up
1UBI	13	198	-	-	1,023	**214.02**	4.78
2DHC	14	198	-	-	8.2	**5.83**	1.41
2DRI	37	186	-	-	135.5	**30.00**	4.52
1CDL	40	186	-	-	392.6	**54.95**	7.14
1CM1	42	186	-	6,177	6.6	**6.11**	1.08
1BRS	44	194	-	-	555.3	**107.86**	5.15
1GVP	52	182	-	-	596.1	**185.75**	3.21
1RIS	56	182	-	-	129.7	**36.23**	3.58
3CHY	74	66	-	5,259	88.7	**20.71**	4.28

Table 3: Another CPD benchmark (Allouche et al. 2014). A '–' indicates that the corresponding method failed to prove optimality in less than $9,000$ seconds. Only instances solved in more than 5 sec. by any successful method are reported.

	pedigree19	pedigree31	pedigree44	pedigree51
n	793	1,183	811	1,152
d	5	5	4	5
`cplex`	790	59.30	6.35	36.23
`//10`	191 `(4.14)`	9.00 `(6.59)`	2.48 `(2.56)`	9.43 `(3.84)`
`//30`	**75** `(10.53)`	7.17 `(8.27)`	2.69 `(2.36)`	**5.34** `(6.78)`
`daoopt`	375,110	16,238	95,830	101,788
`//20`	27,281 `(13.75)`	1,055 `(15.39)`	6,739 `(14.22)`	6,406 `(15.89)`
`//100`	7,492 `(50.07)`	201 `(80.79)`	1,799 `(53.27)`	1,578 `(64.50)`
`HBFS`	3,126	4.34	39.72	1,608
`//10`	434.27 `(7.20)`	1.51 `(2.87)`	6.08 `(6.53)`	179.22 `(8.97)`
`//20`	227.02 `(13.77)`	1.39 `(3.12)`	3.18 `(12.49)`	72.30 `(22.24)`
`//100`	119.43 `(26.17)`	**0.97** `(4.47)`	**1.64** `(24.22)`	31.40 `(51.21)`

Table 4: Linkage benchmark (Favier et al. 2011) with different number of cores (speed-up in parentheses).

We report in Table 1 and Table 2 solving times to find and prove optimality on Warehouse and CPD benchmarks for the sequential and 24-core parallel versions of HBFS. Parallel HBFS solved three more CPD instances within the 1-hour time-out. The maximum speed-up was 2.12 (resp. 3.21) for Warehouse (resp. CPD), which is rather limited compared to the number of cores used. Experiments on difficult instances of another CPD benchmark using only 10 cores resulted in better speed-ups (up to 7.14 on 1CDL, see Table 3). Moreover our CFN approach was much faster than `cplex`. In Table 4, we compared `cplex`, `daoopt`, and HBFS on the Linkage benchmark. We report `daoopt` time from (Otten and Dechter 2017), obtained on a cluster of dual 2.67 GHz Intel Xeon X5650 6-core CPUs and 24 GB of RAM. Here, the sequential version of HBFS is dominated by `cplex` on three instances among four. But, the parallel version of HBFS got better relative speed-ups than `cplex` when the number of cores increases. We found `daoopt` got very good speed-ups on these instances but still was far from `cplex` in terms of CPU times.

Conclusions

Parallel HBFS is a first parallel approach for HBFS. It provides interesting results on several instances, outperforming in some cases state of the art solvers like `cplex` and `daoopt`. Even if the scalability of our approach must be subject of deeper investigation, due to the minimal size of the information shared between the Master and the Workers, the approach is very likely compliant with a larger number of cores. We found that the speed-up was very instance dependent, and must be also investigated.

As future work, we will take into account the structure of CFNs by parallelizing Backtrack with Tree Decomposition (BTD-HBFS) (Allouche et al. 2015). The resulting parallel method could replace LDS inside a parallel large neighborhood search strategy (Ouali et al. 2017) offering better anytime lower and upper bounds.

Acknowledgments This work has been partially funded by the french Agence nationale de la Recherche, reference ANR-16-C40-0028. We are grateful to the genotoul bioinformatics platform Toulouse Midi-Pyrenees (Bioinfo Genotoul) for providing computing and storage resources.

References

Allouche, D.; André, I.; Barbe, S.; Davies, J.; de Givry, S.; Katsirelos, G.; O'Sullivan, B.; Prestwich, S.; Schiex, T.; and Traoré, S. 2014. Computational protein design as an optimization problem. *Artificial Intelligence* 212:59–79.

Allouche, D.; de Givry, S.; Katsirelos, G.; Schiex, T.; and Zytnicki, M. 2015. Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP. In *Proc. of CP*, 12–28.

Cooper, M.; de Givry, S.; Sanchez, M.; Schiex, T.; Zytnicki, M.; and Werner, T. 2010. Soft arc consistency revisited. *AI* 174:449–478.

Favier, A.; Givry, S.; Legarra, A.; and Schiex, T. 2011. Pairwise decomposition for combinatorial optim. in graphical models. In *Proc. of IJCAI*, 2126–2132.

Larrosa, J.; de Givry, S.; Heras, F.; and Zytnicki, M. 2005. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *Proc. of IJCAI*, 84–89.

Meseguer, P.; Rossi, F.; and Schiex, T. 2006. Soft constraints processing. In *Handbook of Constraint Programming*. Elsevier. chapter 9, 279–326.

Otten, L., and Dechter, R. 2017. And/or branch-and-bound on a computational grid. *JAIR* 59:351–435.

Ouali, A.; Allouche, D.; de Givry, S.; Loudni, S.; Lebbah, Y.; Eckhardt, F.; and Loukil, L. 2017. Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization. In *Proc. of UAI-17*, 550–559.

Ralphs, T.; Shinano, Y.; Berthold, T.; and Koch, T. 2018. *Handbook of Parallel Constraint Reasoning*. Springer. chapter Parallel Solvers for Mixed Integer Linear Optimization, 283–336.