Simple Distributed Filtering on a CLP Platform

Ilias Sakellariou and Ioannis Vlahavas

Department of Informatics, Aristotle University of Thessaloniki,
54124 Thessaloniki, Greece.
{iliass, vlahavas}@csd.auth.gr

Abstract. The area of distributed constraint satisfaction has drawn significant attention in the past decade. The approaches proposed in the area can be classified in two large categories: distributed search techniques and distributed filtering techniques. The work described in this paper concerns the CLP implementation of the *Dis-SAC* algorithm, a novel distributed filtering technique that is based on the singleton consistency algorithm. The advantages of the algorithm include a high pruning efficiency and a remarkable simplicity. The latter allows an unproblematic implementation of the algorithm in any constraint programming platform that supports network communication, without the need of tampering with the (low level) consistency algorithm employed. The present paper briefly describes *Dis-SAC* along with its implementation in the Cspcons distributed CLP platform and presents experimental results on a number of structured constraint problems. The motivation behind this work is twofold: to support our argument concerning the simple implementation of the algorithm and to further investigate the benefits of its application to constraint satisfaction problems.

keywords: Constraint Satisfaction, Distributed AI, Logic Programming

1 Introduction

The high interest attracted to the area of constraint satisfaction problems (CSP) springs from the fact that a wide range of problems from diverse areas such as artificial intelligence, operation research, design, etc. can be formulated as constraint satisfaction problems. Informally, a CSP problem consists of finding an assignment of values to variables, each ranging over a finite domain, such that this assignment is consistent with a set of constraints imposed on the variables. The search space of any non-trivial CSP problem is quite large posing an obstacle to their solution.

Probably the most successful approach to overcome the above problem is transforming the original CSP to an equivalent one, by filtering out inconsistent values from the variable domains. This idea lead to the introduction of local consistency algorithms such as AC3 [1], AC4 [2], AC6 [3], NIC [4] algorithms, to name a few proposed in the past decade. Local consistency algorithms enforce different levels of consistency (value pruning efficiency) at a different computational cost; a review of consistency algorithms can be found in [5].

Although filtering out values decreases the cost of search, the application of consistency algorithms can introduce significant delays to the overall solution process, thus, in some cases, cancel out any benefits obtained. A classic example is Singleton Consistency [6], a class of filtering methods that demonstrates a high pruning efficiency but at a high computational cost. An approach to overcome this problem is to execute the filtering algorithm in a distributed setting.

The *Dis-SAC* algorithm [7] is a distributed version of the singleton consistency, that follows this approach. The motivation behind the introduction of distributed singleton consistency (*Dis-SAC*) is twofold: provide a framework under which the execution time of applying a strong consistency method is reduced and also devise a simple distributed consistency algorithm that can be implemented in any existing constraint programming system without major modifications. The former allows the efficient application of singleton consistency to a number of hard problems, while the latter encourages its adoption to a wide range of constraint programming platforms, without requiring sophisticated modifications to the underlying platform. This paper extends our previous work presented in [7] by presenting the implementation of the *Dis-SAC* algorithm in a CLP platform and provide further experiments, to support further these arguments.

The rest of the paper is organized as follows. Section 2 presents related work of the field. Section 3 briefly describes the *Dis-SAC* algorithm. Section 4 states the requirements for implementing the algorithm and briefly describes its implementation on the Cspcons CLP platform. Section 5 presents experimental results on two structured CSP benchmarks the Golomb rulers and Quasigroup Completion with Holes problem. Finally, section 6 concludes the paper.

2 Related Work

Reducing the execution time of local consistency techniques via distribution of the work to a number of co-operating processing units, usually called agents, has long been the topic of research in the CP community. Approaches that have been reported in the literature consist mainly of distributed versions of sequential consistency algorithms. For example, earlier work in the field involved two massively parallel versions of the AC-4 algorithm [8], as well as three parallel and distributed algorithms for computing consistency by formulating a CSP as a dual network [9].

More recently, a coarse-grain distributed version of the AC-4 algorithm, *Dis-AC4*, was proposed in [10]. According to *Dis-AC4* the problem is distributed by dividing the variables to a number of agents (workers), which run the same code but on different data. Each agent initially builds the local data structures required by the AC-4 algorithm and then inconsistencies detected are treated. Inconsistencies produced by the agents are broadcasted using a message passing mechanism. In the same spirit, [11] presents an alternative distributed arc consistency algorithm, the *DisAC-9* with minimal message passing, which is based on the variation of the AC-6 [3] consistency algorithm. In *Dis-AC9* only inconsistencies that induce deletions to the domains of the receiving agents are communicated, thus the number of messages broadcasted is minimized.

Finally, a set of distributed constraint satisfaction algorithms based on the notion of chaotic iteration [12, 13] have been proposed [14, 15]. Informally, chaotic iteration enforces local consistency by using a set of domain reduction functions (drf) that are applied until no further modifications occur in the variable domains. In a distributed setting, each agent manages a subset of the problem (drfs and variables) an uses asynchronous message passing to consume and communicate changes in their local domain. The work described in [14] presents a generic distributed chaotic iteration algorithm and its modeling in the Manifold language.

The *Dis-SAC* is closely related to the *DisAC-4* [10] and *DisAC-9*[11], in the sense that it presents a distributed version of a local consistency technique, aiming at improving the execution time of the corresponding sequential algorithm. The algorithm is similar to the *DisAC-4*, with the difference that singleton consistency (*SAC*)[6] is employed for detecting and treating inconsistencies. Thus the benefits expected from the application of the algorithm are greater, since *SAC* enforces a stronger consistency than any arc consistency algorithm. The same argument holds for the distributed chaotic iteration algorithm; furthermore the rule based approach to constraint programming that is used in the chaotic iteration algorithm could be impractical when considering problems with large finite domains, due to the large number of rules (drfs) that have to be generated and managed.

3 A Simple Distributed Consistency Algorithm

As mentioned above, the *Dis-SAC* algorithm is a distributed version of the singleton arc consistency algorithm. Singleton Consistency is a class of filtering techniques that is based on the fact that for each consistent value d_i of a variable x_i, the subproblem obtained by restricting the domain D_i to d_i is consistent [6]. Thus if the subproblem is found to be inconsistent by the application of some local filtering technique, such as arc consistency, then it is safe to remove the value in question from the domain.

SAC enforces stronger consistency than most other local filtering techniques, but the cost of applying it, even as a single preprocessing step, is high as demonstrated by the detailed study found in [16]. In the same work, authors also demonstrate the benefits of employing a restricted form of *SAC* which goes through the variables only once and thus achieves a lesser level of consistency, but at the same time reducing the overall execution time of the filtering process.

3.1 The *Dis-SAC* Algorithm

Singleton consistency is an ideal candidate for distributed execution. Singleton consistency checks on the problem variables can be assigned to different processing elements; this is the main idea behind *Dis-SAC*. The algorithm consists of a community of co-operating agents, in which each agent "knows" the complete problem but is responsible for a subset of the problem variables (its *responsibility set*) on which it enforces singleton consistency. Domain changes (value removals) produced are broadcasted to all agents in the society. Upon reception of a value removal message, the agent updates its current view of the problem and enforces singleton consistency on its responsibility set again. This loop terminates when no more deletions occur in any variable domain, i.e. the community is in a state where all agents are idle and have processed the same number of messages. Thus *Dis-SAC* is a coarse-grain parallel algorithm in which the maximum number of agents involved is equal to the size of the problem i.e. the number of variables. It should be noted that termination detection relies on the existence of scheduling agent (scheduler) as it is described below.

The algorithm assumes an asynchronous message passing model, in which agents exchange messages via communication channels, with no message loss and a finite delivery delay. In this model the send (`sendMsg()`) operation is non-blocking, the receive operation can be either blocking (`getMsg()`) or non blocking (`getMsgNonBlock()`) and there exists a broadcast (`broadcast()`) operation.

An issue that arises and can affect the efficiency of the algorithm is when should inconsistent values be broadcasted, i.e the **communication policy** employed by the algorithm. If the communication of removed values is postponed until no further changes occur in the responsibility set of the agent, the total number of messages is minimized at the risk of increasing the idle time of some agents. Broadcasting removals as soon as they are discovered avoids the idle agent problem, but introduces the risk of network delays due to a high number of messages. Detailed descriptions of versions of the *Dis-SAC* algorithm implementing the above strategies can be found in [7]. This paper investigates an alternative communication policy that lies between these two extremes: interleave a reduced singleton consistency step on the agent's subproblem with a broadcast step of any value deletions that occurred, until no further changes occur in the domain. This strategy provides a more balanced trade off between the number of messages and the agents' idle time. Figure 1 presents the algorithm executed by each agent participating in the society.

In *Dis-SAC*, **termination** can be either *immediate*, signaled by a `stop` message, i.e. some agent detected a domain wipe-out and thus the problem is insoluble (line 1 in Figure 1) or *normal*, signaled by an `end` message, i.e. all agents are idle and have no more messages to process. Normal termination is detected by the scheduler, that monitors message exchange in a passive manner. For every broadcasted message in the society the scheduler receives a notification message (figure 1 line 2) and thus is aware of the total number of messages broadcasted. When an agent enters an idle state (a blocking receive operation as shown in figure 1-line number 3), it informs the scheduler by issuing a `waiting` message, stamped with the total number of messages it has handled (messages both sent and received). Normal termination is inferred when the scheduler receives correctly stamped `waiting` messages from all agents in the society. A more detailed description of the algorithm can be found in [7].

```
DSACAgent(position,P)
begin
    repeat
        repeat
            agrMsg ← {} ; noChanges ← true;
            for $X_i \in$ agentVars(position,P) do
                domainChanges ← SACStep($D_i$,P);
1               if $D_i = \{\}$ then sendMsg(scheduler,stop); exit ;
                if domainChanges $\neq \{\}$ then
                    agrMsg ← agrMsg ∪ domainChanges ;
                    noChanges ← false;

            if agrMsg $\neq \{\}$ then
                broadcast(agrMsg );
2               sendMsg(scheduler,netMsgSend); stamp ← stamp + 1;

        until noChanges ;
        termination ← collectMessages();
    until termination ;
end

Function SACStep($D_i, P$)
for $d_i \in D_i$ do
    if $AC(P|_{D_i=\{d_i\}})$ is inconsistent then
        $D_i \leftarrow D_i \setminus \{d_i\}$; removed ← removed ∪ $(X_i, d_i)$;
        propagateChanges($d_i$);

return removed

Function collectMessages()
termination ← false; messages ← getMsgNonBlock ();
if messages = {} then
3   sendMsg(scheduler,waiting(stamp)); msg ← getMsg();
    if msg = end or msg = stop then
        termination ← true;
    else
        propagateChanges( msgBody() ); stamp ← stamp + 1;
        termination ← false;

else
    for msg ∈ messages do
        propagateChanges( msgBody() ); stamp ← stamp + 1;
    termination ← false;
return termination
```

Fig. 1: The balanced-DSAC Algorithm

4 Implementation of the *Dis-SAC* Algorithm

The design of the *Dis-SAC* algorithm targets to an architecture of loosely coupled distributed memory processors. Thus any number of machines connected through a network is sufficient for implementing the algorithm, without the need for exotic multi-processor hardware. On the other hand, the simplicity of the algorithm allows any programming platform to be used as a vehicle for its implementation, under the condition that it provides support for constraint programming and network communication.

We have chosen for the implementation a CLP language that is targeted toward distributed applications: the Communicating Sequential Prolog with Constraints platform (Cspcons). Cspcons [17] is an extension of the Communicating Sequential Prolog II (Csp-ii) [18] and provides an excellent platform for building any distributed CLP application, since it offers advanced communication facilities and constraint solving all under the logic programming framework.

A Cspcons application consists of several processes that run in parallel and communicate through message passing over channels; the latter can also be established over TCP/IP between processes residing on different hosts. Processes are independent, i.e. each has its own Prolog execution environment and constraint store.

Each *Dis-SAC* agent is modeled as a Cspcons application and consists of two processes: a process that executes the main code of the agent (as that is shown in Figure 1) and a real-time process that responds to network events generated by the Cspcons network mechanism, to respond to abnormal situations. In order to communicate agents have to setup connections between them (initialization phase), each of which is associated with two channels: an incoming channel for the agent that is in the receiving end of the connection and an outgoing channel for the one that is on the sending end. Thus, at the end of the initialization phase all agents have two sets of channels, and can be used in the Cspcons predicates `send(Channels,Msg)` and `receive(Channels,Msg)` to exchange messages.

The network Cspcons `send/2` operation is asynchronous thus fits the requirements of the algorithm. The `receive/2` operation however is synchronous i.e. it is a blocking receive, however providing an asynchronous receive operation in Cspcons is trivial. It should be noted that both these predicates work on lists of channels: for instance the `send(Channels,Msg)` predicate is in fact a broadcast to the list of channels of its first argument.

Implementing the reduced arc consistency check is trivial in a CLP language: the code shown below, shows a simple implementation that takes advantage of the consistency algorithm employed by the language. Minor modifications could return the list of deleted values from the variable domains.

```
rsac([]).
rsac([VAR|REST]):-domain(VAR,DOMAIN),try(VAR,DOMAIN),rsac(REST).

try(_,[]).
try(VAR,[Val|RDomain]):-not(VAR=Val),!, VAR #\= Val,try(VAR,RDomain).
try(VAR,[_Val|RDomain]):-try(VAR,RDomain).
```

Thus the unparalleled simplicity of the *Dis-SAC* algorithm that lies both to its simple communication requirements and the ease with which the singleton consistency algorithm can be implemented in any CP system, allows it unproblematic implementation on a variety of platforms. To further stress this fact, it should be noted that the complete Prolog code of the *Dis-SAC* agent was no more than 600 lines. The scheduler code is much shorter at about 300 lines.

5 Experimental Results

Although the initial experimental results reported in [7] demonstrate that the *Dis-SAC* algorithm performs well on random binary constraint problems, further investigation of the performance of the algorithm on structured problems was required to prove the benefits of its application. Thus we have selected to test the algorithm on two well known benchmarks: the Golomb rules and Quasigroup Completion with Holes problem.

The experiments were conducted on a set of SUN workstations running Solaris connected by a relatively low bandwidth (10Mbps) Ethernet network. Wall time was measured in all experiments in order to provide a fair comparison with the singleton consistency algorithm. The speedup obtained by running distributed version of the algorithm is computed as the faction of the execution time of the sequential version over the distributed one.

Measuring the impact of the application of singleton arc consistency on finding a complete solution to the selected problems is not the issue of the current work. This has been thoroughly investigated in other research works such the one found in [16]. Our aim was to prove that the *Dis-SAC* algorithm reduces the execution time of the sequential *SAC* algorithm on enforce singleton consistency on a problem instance, therefore all the experiments involve running the algorithm as a preprocessing step. However it is obvious that employing *Dis-SAC* as a step in a search process will yield similar benefits.

Golomb Rulers A Golomb ruler of m marks is a set of m integers $x_1, x_2, ..., x_m$, such that $x_i < x_j, \forall i, j : 1 \leq i < j \leq m$, $x_1 = 0$ and the $m(m-1)$ differences $x_j - x_i, i, j : 1 \leq i < j \leq m$ are distinct. The length of such a ruler is the value of the maximum integer of this set, i.e. x_m. The problem consists of finding the optimal Golomb ruler of m marks, i.e. the one with the minimum length. [1]

The experiments include testing the *Dis-SAC* algorithm on Golomb ruler problems ranging from 9 to 15 marks. The number of agents in the society varies from 2 to 7. The speedup obtained is presented in figure 2.

As shown in the figure in almost all cases the distributed version reduces the execution time of the sequential algorithm. The speedup obtained is linear to the number of the agents involved, reaching a value around three when the number of agents is seven. There are however three cases that require some attention: the

[1] Golomb rulers are considered a most challenging CP benchmark and are included in the CSPLib(http://4c.ucc.ie/ tw/csplib/) as prob006.

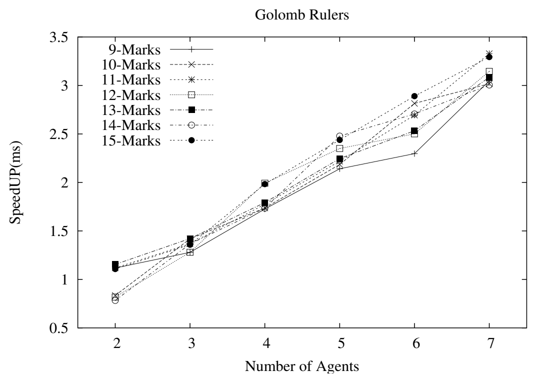

Fig. 2: Speedup of the balanced-DSAC algorithm on Golomb Rulers

distributed algorithm for a community of two agents and for Golomb rulers of 10,12 and 14 marks performs slightly worse than the sequential one. A possible reason behind this behavior is that the problem has such a structure that value removals follow a "butterfly" pattern, i.e. an inconsistent value removed by agent A triggers a value removal on agent B, which in turn is responsible for removing a value in agent A, etc. Such situations can significantly degrade the performance of the algorithm, since they impose a sequential order on the checks.

Quasigroup with Holes Problem A Latin square of size N is a table $N \times N$ filled with N elements such that each element appears once in each row and column. Such a Latin square defines the multiplication table of the binary operation of a Quasigroup of order N. The Quasigroup Completion Problem (QCP) or Latin Square Completion problem[19] consists of determining whether a partial Latin square can be completed to a full one. The Quasigroup Completion with Holes (QWH) problem [20] is a variation of QCP in which the initial partial Latin square is obtained from a complete one by "punching" a number of holes in it, i.e. a uniformly distributed removal of a fraction of the table entries. Thus the main difference between the QCP and QWH is that the problem instances of the latter are always satisfiable. Both problems have been widely used as CSP benchmarks to evaluate the performance of a number of algorithms.

Using the QWH problem generator [2] described in [20] we have generated problem instances for Latin squares of different sizes ranging from 30 to 40 and different number of "holes" varying from 20% to 70% of the total table entries. However from these large set of problems we have excluded those for which the singleton consistency algorithm does not produce any value removals; it is obvious that in these instances *Dis-SAC* has a linear speedup compared to the sequential version. Figure 3 presents the average speedup for large problems in-

[2] The authors would like to thank Dr. Carla Gomes (Dept of Informatics, Cornell University) for providing the code for the QWH problem generator.

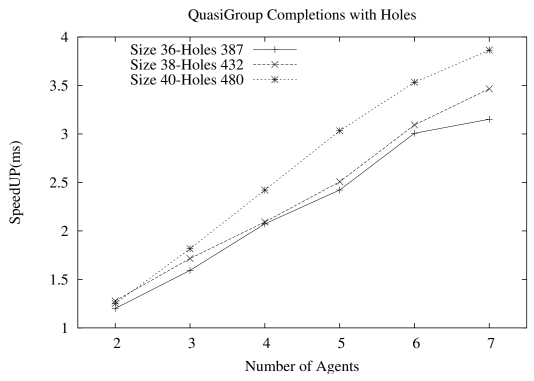

Fig. 3: Speedup of the balanced-DSAC algorithm for the QWH problem

stances of the QWH problem, which is shown to be linear. Results for other problem instances are not presented due to space limitations, but a similar behavior of the algorithm is observed.

6 Conclusions

The *Dis-SAC* algorithm enforces the same strong consistency as the *SAC* algorithm but at a lower computational cost, as demonstrated by the experiments. Its unparalleled simplicity allows easy implementation on any CP platform without the need of changing the underlying consistency algorithm. Another strong point of the *Dis-SAC* algorithm is the absence of any requirements for exotic shared memory hardware: any network of machines is sufficient for its implementation.

Distributed singleton consistency defines a class of distributed filtering algorithms in the same sense that the singleton consistency does: any local consistency algorithm can be employed to detect inconsistent labels in the agents' subproblems.

Although the currently obtained results indicate that the algorithm has a good speed up and a good scale up in most problems tested, our aim is to test the algorithm in a real-world application in order to complete the evaluation of its performance. Our future plans also include further investigation of a number of issues, as for example the impact of not having responsibility sets lexicographically assigned to agents, but instead use some information about the problem's structure to do the assignment.

References

1. Mackworth, A.K.: Consistency in Networks of Relations. Artificial Intelligence **8** (1977) 99–118
2. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence **28** (1986) 225–233

3. Bessière, C.: Arc-consistency and arc-consistency again. Artificial Intelligence **65** (1994) 179–190
4. Freuder, E.C., Elfe, C.D.: Neighborhood inverse consistency preprocessing. In: Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference, Menlo Park, AAAI Press / MIT Press (1996) 202–208
5. Debruyne, R., Bessière, C.: Domain filtering consistencies. Journal of Artificial Intelligence Research **14** (2001) 205–230
6. Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: Proceedings of the 15th International Joint Conference on Artificial intelligence IJCAI (1). (1997) 412–417
7. Sakellariou, I., Vlahavas, I.: Distributed singleton consistency. Technical Report TR-LPIS-149-03, Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki (2003)
8. Cooper, P.R., Swain, M.J.: Arc consistency: Parallelism and domain dependence. Artificial Intelligence **58** (1992) 207–235
9. Zhang, Y., Mackworth, A.K.: Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments. Technical Report TR-92-30, Department of Computer Science, University of British Columbia (1992)
10. Nguyen, T., Deville, Y.: A distributed arc-consistency algorithm. Science of Computer Programming **30** (1998) 227–250 Concurrent constraint programming (Venice, 1995).
11. Hamadi, Y.: Optimal distributed arc-consistency. Constraints **7** (2002) 367–385
12. Apt, K.R.: The essence of constraint propagation. Theoretical Computer Science **221** (1999) 179–210
13. Apt, K.R., Monfroy, E.: Constraint programming viewed as rule-based programming. Theory and Practice of Logic Programming (TPLP) **1** (2001) 713–750
14. Monfroy, E.: Control-driven constraint propagation. Applied Artificial Intelligence **15** (2001) 79–103
15. Monfroy, E., Réty, J.H.: Chaotic iteration for distributed constraint propagation. In: Proceedings of The 14th ACM Symposium on Applied Computing, SAC'99, Artificial Intelligence and Computational Logic Track, San Antonio, Texas, USA. (1999) 19–24
16. Prosser, P., Stergiou, K., Walsh, T.: Singleton Consistencies. In Dechter, R., ed.: Principles and Practice of Constraint Programming - CP 2000 6th International Conference, CP 2000, Singapore, September 2000. Proceedings. Lecture Notes in Artificial Intelligence 1894, Springer Verlag (2000) 353–368
17. Vlahavas, I.P., Sakellariou, I., Futo, I., Pasztor, Z., Szeredi, J.: CSPCONS: A Communicating Sequential Prolog with constraints. In: Methods and Applications of Artificial Intelligence, Procs of the 2nd Hellenic Conference on AI, SETN 2002. Volume 2308 of Lecture Notes in Computer Science., Springer (2002) 72–84
18. I. Futo: A Distributed Network Prolog System. In: Proceedings of the 20th International Conference on Information Technology Interfaces, ITI 99. (1998) 613–618
19. Gomes, C.P., Selman, B.: Problem structure in the presence of perturbations. In: Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97), Menlo Park, AAAI Press (1997) 221–226
20. Achlioptas, D., Gomes, C., Kautz, H., Selman, B.: Generating satisfiable problem instances. In: Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00), Menlo Park, CA, AAAI Press (2000) 256–261