Distributed Singleton Consistency

Ilias Sakellariou and Ioannis Vlahavas
Department of Informatics, Aristotle University of Thessaloniki,
54124 Thessaloniki Greece
{iliass, vlahavas}@csd.auth.gr

April 14, 2004

Abstract

Distributed constraint satisfaction has drawn much attention in the past years, with a number of algorithms proposed to tackle the problem. Research in the area has followed two directions: distributed search techniques and distributed filtering techniques. This paper presents a new distributed filtering algorithm, named Distributed Singleton Arc Consistency (*Dis-SAC*), which is based on the singleton consistency algorithm. *Dis-SAC* is a parallel coarse grain filtering algorithm aiming at improving the performance of singleton consistency by distributing the work to be done to a number of agents. The current paper presents the basic idea behind the algorithm and two versions of it that employ different communication policies along with experimental results obtained on a set of random binary CSP problems.

Keywords: Distributed Constraint Satisfaction, Singleton Consistency

1 Introduction

Many problems appearing in diverse areas like Artificial Intelligence, Operational Research, design, etc, can be modeled as constraint satisfaction problems (CSP). 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. CSP's are NP-hard problems, thus finding a solution using simple backtracking is usually affected by thrashing, i.e. repeatedly failing search for the same reason.

An approach to avoid thrashing is reducing the search space by filtering out inconsistent values, i.e. those that could never participate in a valid solution. This pre-processing transforms the original problem to an equivalent one, that has the same set of solutions but a smaller search space, allowing hard problems to be solved in reasonable time. The latter is achieved by some local consistency algorithm that ensures that each value in the domain of a problem variable that violates a constraint is removed. The simplest form of local consistency is

node consistency that removes values from variable domains that violate unary constraints. The most common and most studied form is *arc consistency* that performs removals by examining binary constraints on the problem variables. Research in the field has resulted to quite a few local consistency algorithms, such as the AC3 (Mackworth 1977), AC4 (Mohr and Henderson 1986), AC5 (Van Hentenryck, Deville, and Teng 1992), AC6 (Bessière 1994), NIC (Freuder and Elfe 1996), LAC (Schiex, Regin, Gaspin, and Verfaille 1996) algorithms, to name a few that have been proposed in the literature in the past decade. The above enforce different levels of consistency at a different computational cost. A review of various consistency algorithms can be found at (Debruyne and Bessière 2001).

However the application of filtering techniques introduces overhead in searching for a solution, which in small problems can be significant, sometimes even comparable to the cost of a brute-force solution. On the other hand in hard problems, the benefits of reducing the search space are considerable, making their application a choice of wisdom.

Singleton Consistency (Debruyne and Bessière 1997) is a class of filtering methods that removes more inconsistent values than most local consistency techniques. Even though it is identified as one of the most promising approaches, its time complexity is high posing an obstacle to its application.

One way to overcome this disadvantage is to reduce the overall execution time through distribution over a network of agents. This approach is further supported by the availability of a large number of machines connected through some network and could provide an elegant solution to solving hard problems.

This paper describes the Distributed Singleton Arc Consistency algorithm (*Dis-SAC*), a parallel coarse grained variant of the singleton arc consistency algorithm (*SAC*) and investigates its benefits. Two versions of the algorithm are presented that differ only in the policy employed to communicate inconsistent values. Both versions aim at improving the time efficiency of the sequential version by execution in a distributed memory multiprocessor environment.

The rest of the paper is organised as follows. Section 2 presents research in the area of distributed CSP's. Section 3 describes in brief the singleton arc consistency algorithm. Section 4 presents the two versions of the *Dis-SAC* algorithm in detail. Section 5 presents experimental results on a network of computers. Finally, section 6 concludes the paper.

2 Distributed Constraint Satisfaction

A constraint satisfaction problem P consists of:

- a set of variables X $x_1, x_2, ..., x_n$
- a set of domains D each associated with a variable $D_1, D_2, ..., D_n$, i.e. $x_i \in D_i$
- a set of constraints C that impose restrictions on the values that the variables can take. Each constraint $C_k(x_{k1}, x_{k2}, \ldots, x_{km}), m <= n$ is a

predicate on the Cartesian product $D_{k1} \times D_{k2} \times \ldots \times D_{km}$ that is true on a subset of the product, indicating that an assignment that belongs to the subset is valid under the constraint.

In a distributed setting the constraint problem variables and/or constraints are distributed among a number of communicating agents. Agents cooperate in order to find the solution to the overall problem.

2.1 Related Work

Distributed constraint satisfaction has long been the topic of active research and a large number of approaches that address the problem have been reported in the literature. Currently the algorithms proposed in the literature can be categorised into two classes: those that perform filtering in a distributed manner and those that are concerned with distributed search.

The first class of approaches involves distributed/parallel versions of sequential arc consistency algorithms. Earlier work in the field involved two massively parallel versions of the AC-4 algorithm proposed by P. Cooper and M. Swain (Cooper and Swain 1992).

The *Dis-AC4* algorithm proposed in (Nguyen and Deville 1998), is a coarse-grain distributed version of the AC4 algorithm. According to it, 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 are either produced locally by the agent or by another agent and are received by a message passing mechanism. The algorithm terminates upon detection of silence in the network.

In (Hamadi 2002) the author proposes an alternative distributed arc consistency algorithm, called *DisAC-9* with minimal message passing which is based on the variation of the AC-6 (Bessière 1994) consistency algorithm. As in the previous algorithm agents are assigned a set of variables and start by computing inconsistent labels of that set. However not all inconsistent labels detected are broadcasted, but only a selected set of these that induce deletions to the domains of the receiving agents; thus the number of messages broacasted is minimised.

Y. Zhang and A. Mackworth (Zhang and Mackworth 1992) present three parallel and distributed algorithms for computing consistency by formulating a CSP as a *dual network*, in which constraints correspond to nodes and variables to arcs. These algorithms were tested on a transputer based machine.

More recently, a set of distributed constraint satisfaction algorithms based on the notion of chaotic iteration (Apt 1997; Apt 1999; Apt and Monfroy 2001), have been proposed (Monfroy 2001; Monfroy and Réty 1999). 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 domains from function application. The work described in (Monfroy 2001) describes a generic distributed chaotic iteration algorithm and its modelling in the Manifold language, an implementation of the Idealized Worker Idealized Manager (IWIM) (Arbab 1996), a communication model that enables the construction of complex dynamic coordination protocols. In this work each *drf* is mapped to a worker process and each variable in the problem to a coordination variable and detection is easily implemented using the facilities of the Manifold language.

The second class of algorithms performs distributed search using a set of agents and a message passing scheme. For example in (Yokoo, Durfee, Ishida, and Kuwabara 1998; Yokoo and Hirayama 2000) authors propose an asynchronous backtracking algorithm (ABT) and its modification, the asynchronous weak-commitment search (AWC), that efficiently solve distributed constraint satisfaction problems. Under the ABT algorithm, variables are assigned to agents that exchange messages concerning value assignment (*ok?* messages) or inconsistencies (*nogood* messages) until a solution is reached. An ordering is imposed on agents that defines the direction of message exchange. The asynchronous weak-commitment search (AWC) introduces to the above algorithm two new features: dynamic agent ordering and the min-conflicts heuristic.

In the Distributed Backtracking algorithm (DIBT), introduced in (Hamadi, Bessière, and Quinqueton 1998), a different approach is followed. Agents compute their position in a total ordering of the network such that each has a set of parent and children agents. Parent agents broadcast through messages values to their children, that try to determine a consistent value for their variables. If they fail to do so, backtracking is initiated to parent agents. The Distributed Forward Checking algorithm (DIFC) (Meseguer and Jimenez 2000), follows the DIBT approach to DCSP, but is more focused on privacy aspects of distributed constraint satisfaction. The algorithm is based on the idea that agents communicate the effects that their value assignment has on other agents' domains and not their own values. Finally, the Distributed Dynamic Backtracking algorithm (DisDB) (Bessière, Maestre, and Meseguer 2001) combines ideas from ABT and DIBT, namely nogood communication of the former and the no new relation introduction between the agents of the latter.

2.2 Our Approach

The *Dis-SAC* algorithm described in this paper is closely related to the *DisAC-4* (Nguyen and Deville 1998) and *DisAC-9* (Hamadi 2002), 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 proposed algorithm is very similar to the *DisAC-4*, with the difference that a singleton arc consistency algorithm (*SAC*) (Debruyne and Bessière 1997) 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.

A major advantage of the *Dis-SAC* algorithm is simplicity. The distributed version of the singleton consistency algorithm has very little modifications with respect to the sequential one, however the speed up obtained is significant. To our knowledge no other work has yet investigated the benefits of distributively executing a singleton consistency algorithm.

3 Singleton Consistency

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 (Debruyne and Bessière 1997). 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. The algorithm is shown in figure 1.

```
SingletonAC(P) begin
    P ← AC(P);
    repeat
        change ← false;
        for X_i ∈ X do
            for d_i ∈ D_i do
                if AC(P|_{D_i={d_i}}) is inconsistent then
                    D_i ← D_i \ {d_i};
                    change ← true;
                    propagateChanges(d_i);
                endif
            endfor
        endfor
    until change = false;
end
```

Figure 1: Singleton Arc Consistency Algorithm

The algorithm iterates through all domain values each time restricting a domain to a value and enforcing some AC algorithm to detect domain wipe-outs. In such a case the value in question is removed from the domain of the variable and the effects of its deletion are propagated. As stated in (Debruyne and Bessière 1997) any consistency algorithm can be employed to detect domain wipe-outs when achieving arc consistency, although a lazy approach, such as the one described in (Schiex, Regin, Gaspin, and Verfaille 1996) (LAC7), is sufficient. The worst time complexity of the algorithm depends on the complexity of the arc consistency algorithm used in each step.

Singleton consistency enforces stronger consistency than other local techniques, i.e. removes a larger set of values however the cost of applying it, even as a single preprocessing step is high. A detailed theoretical and empirical study of the benefits of singleton consistency can be found in (Prosser, Stergiou, and Walsh 2000). 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.

4 Distributed Singleton Arc Consistency

The motivation behind this work is to reduce the execution time of SAC, by dividing the work that has to be performed to a number of agents. SAC seems to be ideal for such distributed execution, since each step of enforcing consistency on the reduced variable domain can be done independently.

The principle behind the algorithm is rather simple. Agents in the community have a "complete" view of the problem, but enforce singleton consistency to a subset of the domain variables. Changes in the domains that are detected by the consistency algorithm are broadcasted to all agents participating in the society. Each agent that receives a set of deleted values, removes them from its local copy of the store and re-initiates singleton consistency check again. Termination is detected upon quiescence in the network, indicating that there are no more deletions in the domains of the problem's variables. At the end of the application of the filtering algorithm each agent's view of the problem is identical, thus any of them can be used to continue searching for a solution or any of the distributed search algorithms described in section 2.1 can be employed to find the solution distributively.

The communication policy that determines when inconsistent values detected are broadcasted can affect the number of messages transmitted over the network and potentially the efficiency of the algorithm. There are two alternatives concerning this issue; broadcast values either lazily or eagerly. In the *lazy* version of the algorithm communicating inconsistencies is postponed until no further changes can be detected in the subproblem that the agent is responsible for. In the *eager* version inconsistencies are communicated as soon as they are discovered. Each of them yields different benefits; the former minimises the number of messages broadcasted over the network at the risk of increasing the idle time of some agents, while the latter although it avoids such a risk it might introduce delays for large problem instances in low bandwith networks.

The sections that follow present the two versions of the algorithm: section 4.1 presents the message passing model common to both algorithms; section 4.2 describes in detail the lazy-Dis-SAC algorithm while the eager-Dis-SAC algorithm is briefly described in section 4.3 since it presents minor differences with the former. Finally section 4.4 presents how termination is detected among the agents participating in the community, commonly used in both versions.

4.1 Message Passing Model

For the needs of the algorithm we assume an asynchronous message passing model, common in most distributed consistency algorithms. In the model, agents exchange messages via communication channels. The **send** (`sendMsg()`) operation is non-blocking, however the **receive** operation can be either blocking (`getMsg()`) or non blocking (`getMsgNonBlock()`). As usual we also assume that the delay of delivering a message is finite, there is no message loss and that messages are received in the order that they were send. The underlying network provides also a **broadcast** (`broadcast()`) operation that allows the transmission of a message from one agent to all other agents participating in the community.

4.2 The lazy-*Dis-SAC* Algorithm

In the lazy-*Dis-SAC* algorithm computation is divided among a community of agents `lazyDSACAgent(`*1*`)`,`lazyDSACAgent(`*2*`)`,...`lazyDSACAgent(`K`)`, where K is the position of the agent in the community and $1 \leq K \leq N$, where N is the total number of domains/variables of the original problem. Thus lazy-*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. Each agent "knows" the complete problem, i.e. is aware of all variables/domains and constraints of the problem. All agents run the same code, as that is described in figure 2.

Each agent is assigned a subset of the variables of the original problem, called *responsibility set*, which it checks for inconsistencies using a singleton arc consistency algorithm. The responsibility set is returned by the `agentVars()` function (figure 2, line number 1). The latter computes lexicographically the set of variables based on the position of the agent in the community. No special algorithm is employed for dividing the set of variables among the agents and no hierarchy is imposed on agents.

For each variable X_i in the responsibility set of the agent, singleton arc consistency is employed by a call to function `SACStep()` (figure 2, line number 2). The function returns the domain changes, i.e. inconsistent values removed from domain D_i that is associated with the selected variable X_i, which are collected in an aggregated message (figure 2, line number 4). If the domain of any variable in the responsibility set becomes empty, a `stop` message is sent to the scheduler that indicates that the problem is insoluble and that the overall process should terminate immediately (figure 2, line number 3). The described loop continues until no more changes occur in the responsibility set.

After the singleton consistency check, if the aggregated message contains any changes it is broadcasted to all agents in the society. (`broadcast()` function, figure 2, line number 5). This is the implementation of the lazy communication policy of the algorithm: changes are broadcasted if no other inconsistencies can be detected in the subproblem the agent handles.

As a final step, messages broadcasted by other agents are collected (function

```
   lazyDSACAgent(position,P)
   begin
     repeat
       agrMsg ← {} ;
       repeat
         noChanges ← true;
1        for X_i ∈ agentVars(position,P) do
2          domainChanges ← SACStep(D_i,P);
           if D_i = {} then
3            sendMsg(scheduler,stop);
             exit;
           endif
           if domainChanges ≠ {} then
4            agrMsg ← agrMsg ∪ {(i,domainChanges)} ;
             noChanges ← false;
           endif
         endfor
       until noChanges ;
       if agrMsg ≠ {} then
5        broadcast(agrMsg );
6        sendMsg(scheduler,netMsgSend);
         stamp ← stamp + 1;
       endif
7      termination ← collectMessages();
     until termination ;
   end
```

Figure 2: lazy-DSAC Algorithm

`collectMessages()` - figure 2, line number 7). If new messages have arrived indicating value deletions, then the corresponding variable domain is updated and the above described loop is restarted.

4.2.1 Inconsistency Detection

Inconsistencies are detected by the `SACStep()` function, presented in figure 3. The function iterates through all values of the domain of the selected variable. It is in fact a restriction of the singleton consistency algorithm to the domain of a single variable. Inconsistent values are removed from the domain of the variable and are returned as the output of the function.

4.2.2 Message Collection

The main task of the `collectMessages()` function presented in figure 4 is to receive all messages send asynchronously by other agents in the community and

```
Function SACStep(D_i, P)
for d_i ∈ D_i do
    if AC(P|_{D_i={d_i}}) is inconsistent then
        D_i ← D_i \ {d_i};
        removed ← removed ∪ d_i;
        propagateChanges(d_i);
    endif
endfor
return removed
```

Figure 3: The SACStep(D_i, P) Function

propagate the changes included in them (line number 2). The *msgBody()* function returns the domain changes included in the message. The rest of the code appearing in figure 4 is part of the termination detection algorithm explained in section 4.4.

4.3 The eager-*Dis-SAC* Algorithm

In the eager-*Dis-SAC* algorithm (figure 5) computation is also divided among a community of agents `eagerDSACAgent(1)`, ... `eagerDSACAgent(K)`, where K is the position of the agent in the community and $1 \leq K \leq N$, where N is the total number of domains/variables of the original problem. Thus eager-*Dis-SAC* is also a coarse-grain parallel algorithm.

The two versions share the way inconsistencies are detected (`SACStep()` function), the way messages are collected (`collectMessages()` function) and the termination detection algorithm, as shown in figure 5. Their difference lies in the communication policy used: the eager policy employed in this case involves sending inconsistent values of a variable as soon as they are detected (lines 1 in figure 5). This approach increases the total number of messages broadcasted in the network but reduces the risk of having idle agents waiting for messages. Apart from this difference algorithms behave identically.

4.4 Termination Detection

Although there exist a number of distributed detection termination algorithms, their application to this case was considered unnecessary, due to their high cost. In both versions of the *Dis-SAC* algorithms, a simpler detection schema has been employed, that relies on the existence of a scheduling agent (scheduler) as in the case of *DisAC-4* (Nguyen and Deville 1998). The scheduler is responsible for terminating computations in the society by informing agents via appropriate messages. The scheduler algorithm is shown in figure 6.

Termination occurs in two cases:

- When a domain wipe-out occurs in some agent, which signifies that the

```
   Function collectMessages()
   termination ← false;
   messages ← getMsgNonBlock ();
   if messages = {} then
1  |  sendMsg(scheduler,waiting(stamp));
   |  msg ← getMsg();
   |  if msg = end or msg = stop then
   |  |  termination ← true;
   |  endif
   |  else
   |  |  propagateChanges( msgBody() );
   |  |  stamp ← stamp + 1;
   |  |  termination ← false;
   |  endif
   endif
   else
   |  for msg ∈ messages do
2  |  |  propagateChanges( msgBody() );
   |  |  stamp ← stamp + 1;
   |  endfor
   |  termination ← false;
   endif
   return termination
```

Figure 4: Collect Messages Function

problem has been determined to be insoluble. The term *immediate termination* is introduced to describe such cases.

- When all agents are in a waiting state, as the latter is described below. This signifies that the overall problem is singleton arc consistent. In such a case the scheduler terminates the system by broadcasting an `end` message. The term *normal termination* describes these cases.

In immediate termination the scheduler is notified for the wipe-out by the agent that detected it, through a `stop` message. The scheduler then forwards this message to all the members in the society (figure 6, line number 3).

Normal termination is detected by the scheduler. The scheduling agent monitors message exchange in a passive manner; each agent when broadcasting a message to the society also notifies the scheduler via a `netMsgSend` message (figure 2 line number 6). Thus the scheduler is aware of the total number of messages broadcasted in the society by simply maintaining a counter on such notification messages as shown in figure 6 (line number 1). This is the main difference between termination algorithms of the *Dis-SAC* and the *Dis-AC4*; the latter polls agents if a message is not broadcasted for a specific time interval.

An agent is said to be in a *waiting state* if it has no more messages to consume

```
eagerDSACAgent(position,P)
begin
    repeat
        repeat
            noChanges ← true;
            for X_i ∈ agentVars(position,P) do
                domainChanges ← SACStep(D_i,P);
                if D_i = {} then
                    sendMsg(scheduler,stop);
                    exit;
                endif
                if domainChanges ≠ {} then
1                   broadcast(domainChanges );
                    sendMsg(scheduler,netMsgSend);
                    stamp ← stamp + 1;
                    noChanges ← false;
                endif
            endfor
        until noChanges ;
        termination ← collectMessages();
    until termination ;
end
```

Figure 5: eager-DSAC Algorithm

and no further computations to perform. In such a state the agent is idle, i.e. it executes a blocking receive operation as shown in figure 4 (line number 1). Before it enters a waiting state the agent informs the scheduler of this fact by issuing a `waiting` message, stamped with the total number of messages it has handled, i.e. the sum of messages both sent and received by the agent. Exit from the waiting state occurs upon the reception of any message.

When the scheduler receives `waiting` messages from all agents, stamped correctly according to the message counter it maintains, then it infers that all agents have consumed/broadcasted the same number of messages and are currently idle and thus broadcasts the termination message `end` (figure 6 line number 2). If in the mean time, the scheduler is notified that a new message containing domain changes was broadcasted, then it immediately discards all previous `waiting` messages and increases its message counter appropriately.

5 Experimental Results

A prototype of the algorithm was implemented in JAVA. Although the latter is not considered to be the best candidate in terms of efficiency, our choice was based to the fact that the resulting code is portable and therefore it can be

```
   scheduler(Agents)
   begin
     continue ← false;
     messageCounter ← 0;
     repeat
       msg ← getMsg();
       switch msg do
1        case netMsgSend:
           messageCounter ← messageCounter + 1;
           idleAgents ← {};
2        case waiting(stamp):
           if stamp = messageCounter then
             idleAgents ← idleAgents ∪ msgFrom() ;
           endif
           if idleAgents = Agents then
             termination ← true;
             broadcast(end);
           endif
3        case stop:
           termination ← true;
           broadcast(stop);
       endsw
     until termination ;
   end
```

Figure 6: The Scheduler Algorithm. The *from()* function returns the sender of the message.

tested in the future using heterogeneous networks of machines.

The implementation of the singleton arc consistency algorithm is based on the Java Constraint Library developed by the Artificial Intelligence Laboratory (LIA)[1]. Agent communication was implemented using Pathwalker, a Java programming library for distributed applications developed by FUJITSU Labs[2]. In Pathwalker applications communicate by asynchronous message passing and can reside either in a single or multiple hosts.

A network of SUN workstations connected by a local network (Ethernet) was used to conduct the experiments. Each machine participating in the network hosted one agent, as those are described in the algorithms of figures 2 and 5.

For the evaluation of the algorithm a set of randomly generated binary CSP problems of 15 variables that contained 15 values in their initial domains were employed. Problems were generated using the Random Uniform CSP Generator provided by Christian Bessiére [3]. Apart from their size (number of variables), the two important parameters of the generated problems are their *density* and

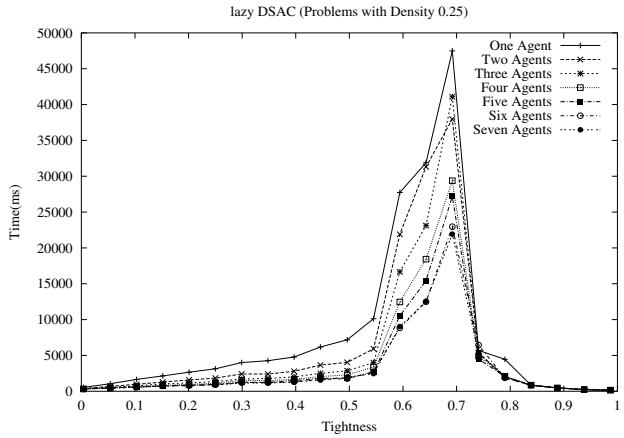

Figure 7: Results on running lazy-DSAC on a network of hosts (density 0.25)

tightness. The former is fraction of the number of binary constraints appearing in the problem over the maximum number of possible constraints. The latter is the fraction of the number of value pairs disallowed by the constraint over the maximum number of disallowed pairs.

The tests included three sets of random problem instances with a density of 0.25, 0.50 and 0.75 respectively. The tightness of the problems varied from 0 to 1. For each tightness value three instances of the problem were generated and each instance was tried 10 times. In the experiments conducted, it can be safely assumed that the single agent version corresponds to the sequential singleton arc consistency algorithm, since the communication overhead introduced is minimal. All comparisons will be based on this assumption.

Figure 7 shows the results obtained by running the lazy-*Dis-SAC* algorithm on a network of seven hosts for problems with density 0.25. As shown the distributed multi-agent version perform better than the single agent for all values of tightness that are before the insolubility point (around tightness 0.7), after which the performance of all versions converges. The behaviour is explained by the fact that after that point domain wipe-outs occur so the algorithm is terminated by any agent immediately.

Almost the same conclusions apply for the results obtained by applying the eager-*Dis-SAC* on the same set of problems. The results are shown in figure 8 and were collected in the same manner as in the previous case. The eager version shows a slightly better behaviour around the insolubility point, justified by the fact that it reduces the idle time of the agents.

The performance of both algorithms slightly changes around the insolubility point. In order to investigate in more detail the behaviour of the algorithm in that area, a new set of experiments was conducted on problems with tightness varying from 0.5 to 0.8 with a finer step. The results are presented in figure 9.

This new set of experiments revealed that for both versions for a specific

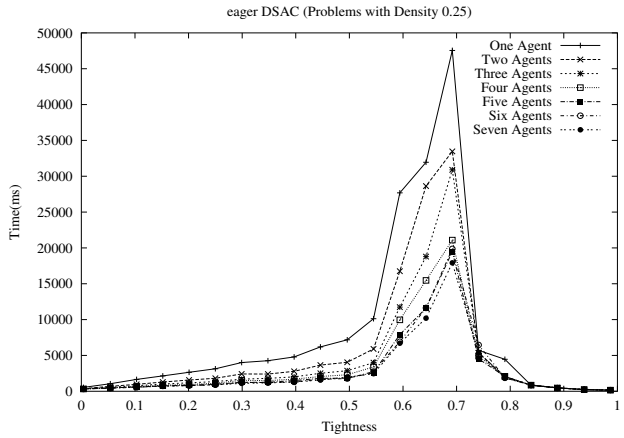

Figure 8: Results of eager-DSAC on a network of hosts (density 0.25)

value of tightness around 0.725 the single agent version performs better than some of the multi-agent versions that involve a small number of agents. The above is justified by the fact that in some cases the single agent version more easily detects insoluble problems than the multi-agent version, since in the latter the offending domain wipe-outs in the subproblem may heavily depend on removed values from a variable belonging to some other agent. In this case, the agent has to finish examining his subproblem first before receiving the deleted values and detecting the wipe-out, thus increasing the total execution time of the algorithm. However this phenomenon is rather limited and it can be safely concluded that overall the performance of the multi-agent version is better than the single agent one.

Results concerning problems of density 0.50 and 0.75 are presented in figure 10. As shown by the diagrams, the results obtained by this set of expeeriments do not yield any new conclusions on the performance of the algorithm.

5.1 Multiprocessor vs Multihost Experiments

The aim of the experiments presented in this section is twofold: evaluate how the presence of a relatively slow underlying network connection affects its performance and test the algorithm in larger problem instances. In this direction two sets of experiments were carried out: a single host experiment on a four processor machine (Sun E450) and a multihost experiment, involving a number of hosts connected by a local area network. On the single host set each agent is a separate process and communication is still handled by the Pathwalker library; the benefit is that the delay in message passing operations introduced by the underlying Ethernet network simply does not exist, permitting to draw conclusions on the affect of network delays on the performance.

For the current evaluation of the algorithm a set of randomly generated

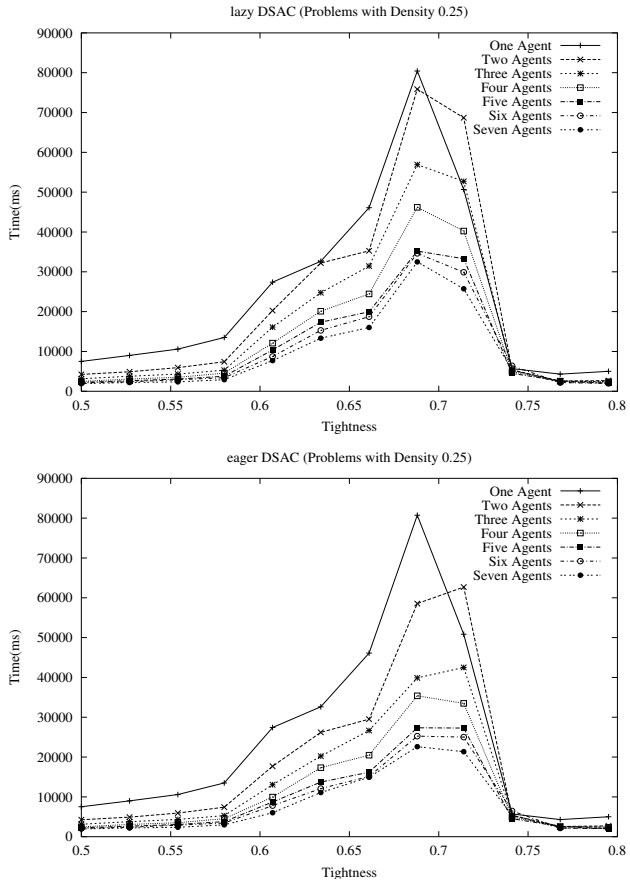

Figure 9: Lazy and Eager DSAC performance on a network of hosts (density 0.25) for tightness values 0.5 to 0.8, with a finer step.

binary CSP problems of 30 variables was employed each having 20 values in its initial domain. All the problems were fully connected i.e. the problems' density was set to 1. The tightness of the problems varied from 0 to 1. For each tightness value three instances of the problem were generated and each instance was tried 10 times. It should be noted that the eager-*Dis-SAC* version was used since it invlolves broadcasting a larger number of messages and thus is more likely to suffer from slow network connections.

Figure 11 shows the results obtained by this set of experiments. As shown in the figure the behaviour of the algorithm remains the same in both cases indicating that the underlying network does not affect greatly the performance of the algorithm.

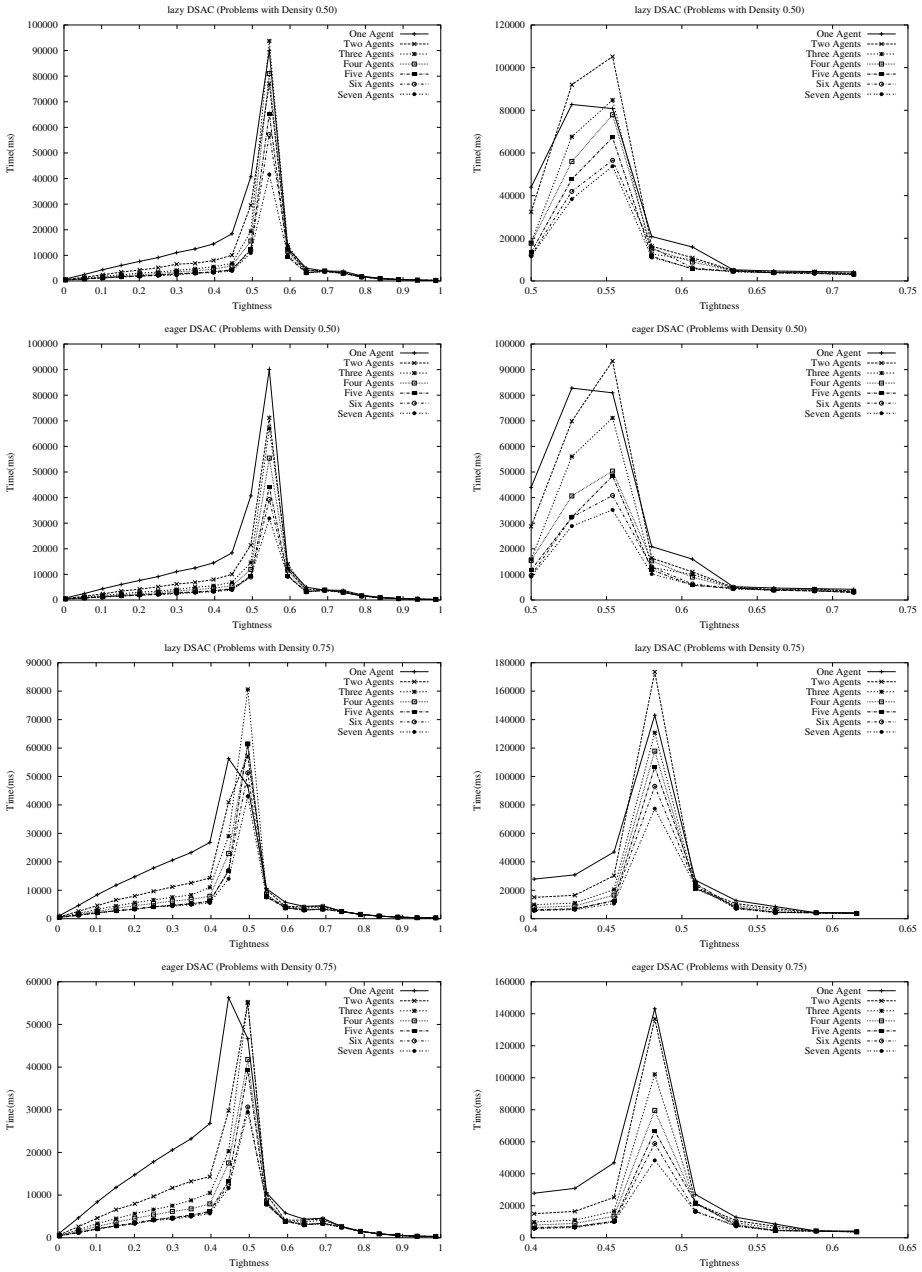

Figure 10: General and detailed results on running DSAC on a network of hosts. Experiments concern problems with density 0.50 and 0.75 for both lazy and eager versions of the algorithm.

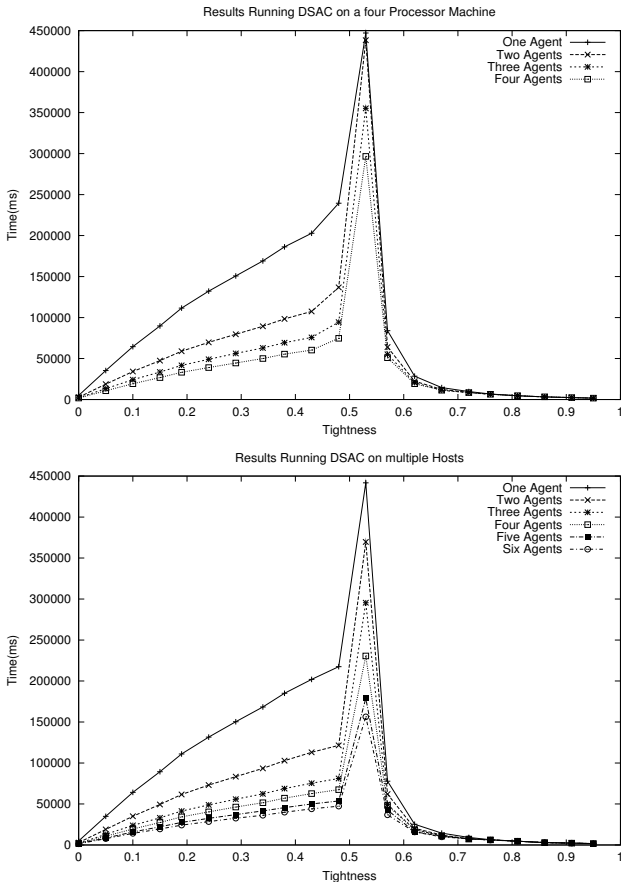

Figure 11: Running DSAC on a single host and multiple hosts

6 Conclusions

Distributed singleton consistency defines a class of distributed algorithms in the same sense that singleton consistency does, since any local consistency algorithm can be used by the agents to detect inconsistent labels. Even more, since each agent is independent it can apply to its set of variables a different consistency algorithm according to the nature of its subproblem.

As in the case of *Dis-AC4*, a strong point of the algorithm is that it can be implemented in any platform since it does not require exotic shared memory hardware. A number of hosts connected over a network is sufficient.

A point worth mentioning is that since each agent has a complete view of the problem, finding the final solution can still be achieved even if some agents abnormally terminate during execution. Thus a dynamic fault tolerant version could be introduced, that when it detects some abnormal conditions for some agent in the society could reassign it responsibility set to some other agent.

Although the preliminary results from applying the algorithm to random problems demonstrate significant speedup, further investigation is required to determine its true potential and benefits. This investigation should most definitely include experimental results on small world examples, i.e. structured problems, for it has been stated in the work described in (Prosser, Stergiou, and Walsh 2000) that results obtained from the application of SAC on random and small world problems were contradictory.

Our future plans include a number of more or less obvious extensions to the algorithm. For example, instead of assigning lexicographically the responsibility set of each agent, one could define these sets based on the characteristics of the constraint graph of the problem. For instance an idea would be assign all the variables in the cycle-cutset of the graph to one agent in order to minimise the number of message exchanged.

Aknowledgements

The work described in this paper has been partially supported by SUN Microsystems, grant number: EDUD-7832-010326-GR.

Notes

[1] Java Constraint Library v 2.1, http://liawww.epfl.ch/JCL LIA, Swiss Federal Institute of Technology, Lausanne (EPFL)

[2] PathWalker, Fujitsu Laboratories Ltd, http://www.labs.fujitsu.com/free/paw/index.html

[3] Random Uniform CSP Generator, http://www.lirmm.fr/ bessiere/generator.html

References

Apt, K. (1997). From Chaotic Iteration to Constraint Propagation. In *Proceedings of 24th International Colloquium on Automata, Languages and Programming, ICALP'97*, Volume 1256, pp. 36–55. Springer-Verlag.

Apt, K. R. (1999). The essence of constraint propagation. *Theoretical Computer Science 221*(1–2), 179–210.

Apt, K. R. and E. Monfroy (2001). Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming (TPLP) 1*(6), 713–750.

Arbab, F. (1996). The IWIM Model for Coordination of Concurrent Activities. In P. Ciancarini and C. Hankin (Eds.), *Proc. 1st Int. Conf. on Coordination Models and Languages*, Volume 1061, Cesena, Italy, pp. 34–56. Springer-Verlag, Berlin.

Bessière, C. (1994). Arc-consistency and arc-consistency again. *Artificial Intelligence 65*(1), 179–190.

Bessière, C., A. Maestre, and P. Meseguer (2001). Distributed dynamic backtracking. In M. Silaghi (Ed.), *Proceedings of the IJCAI'01 Workshop on Distributed Constraint Reasoning, Seattle WA*, pp. 9–16.

Cooper, P. R. and M. J. Swain (1992). Arc consistency: Parallelism and domain dependence. *Artificial Intelligence 58*(1–3), 207–235.

Debruyne, R. and C. Bessière (1997). Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the 15th International Joint Conference on Artificial intelligence IJCAI (1)*, pp. 412–417.

Debruyne, R. and C. Bessière (2001, may). Domain filtering consistencies. *Journal of Artificial Intelligence Research 14*, 205–230.

Freuder, E. C. and C. D. Elfe (1996, August 4–8). 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, pp. 202–208. AAAI Press / MIT Press.

Hamadi, Y. (2002, July). Optimal distributed arc-consistency. *Constraints 7*(3), 367–385.

Hamadi, Y., C. Bessière, and J. Quinqueton (1998, August 23–28). Backtracking in Distributed Constraint Networks. In H. Prade (Ed.), *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, Chichester, pp. 219–223. John Wiley & Sons.

Mackworth, A. K. (1977). Consistency in Networks of Relations. *Artificial Intelligence 8*(1), 99–118.

Meseguer, P. and M. A. Jimenez (2000). Distributed forward checking. In *Proceedings of the CP 2000 International Workshop on Distributed Constraint Satisfaction, Singapore*.

Mohr, R. and T. C. Henderson (1986). Arc and path consistency revisited. *Artificial Intelligence 28*(2), 225–233.

Monfroy, E. (2001). Control-driven constraint propagation. *Applied Artificial Intelligence 15*(1), 79–103.

Monfroy, E. and J.-H. Réty (1999, March). 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*, pp. 19–24.

Nguyen, T. and Y. Deville (1998, January). A distributed arc-consistency algorithm. *Science of Computer Programming 30*(1–2), 227–250. Concurrent constraint programming (Venice, 1995).

Prosser, P., K. Stergiou, and T. Walsh (2000). Singleton Consistencies. In R. Dechter (Ed.), *Principles and Practice of Constraint Programming - CP 2000 6th International Conference, CP 2000, Singapore, September 2000. Proceedings*, Lecture Notes in Artificial Intelligence 1894, pp. 353–368. Springer Verlag.

Schiex, T., J.-C. Regin, C. Gaspin, and G. Verfaille (1996). Lazy arc consistency. In *Proceedings of 13th National Conference on Artificial Intelligence AAAI96*, pp. 216–221. AAAI Press / The MIT Press.

Van Hentenryck, P., Y. Deville, and C.-M. Teng (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57*(2–3), 291–321.

Yokoo, M., E. H. Durfee, T. Ishida, and K. Kuwabara (1998). The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. on Knowledge and Data Engineering 10*(5), 673–685.

Yokoo, M. and K. Hirayama (2000, June). Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents and Multi-Agent Systems 3*(2), 185–207.

Zhang, Y. and A. K. Mackworth (1992, November). Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments. Technical Report TR-92-30, Department of Computer Science, University of British Columbia.