The effect of synchronization of agents’ execution in randomly generated networks of constraints.

IONEL MUSCALAGIU, ANCA IORDAN, DIANA MARIA MUSCALAGIU, MANUELA PANOIU
The "Politehnica" University of Timisoara
The Faculty of Engineering of Hunedoara
Revolutiei, 5, Hunedoara
ROMANIA
mionel@fih.upt.ro, anca.iordan@fih.upt.ro, dianamariamuscalagiu@yahoo.com m.panoiu@fih.upt.ro

Abstract: The asynchronous searching techniques are characterized by the fact that each agent instantiates its variables in a concurrent way. Then, it sends the values of its variables to other agents directly connected to it by using messages. This article presents the opportunity for synchronizing the execution of agents in case of asynchronous search techniques. It investigates and compares the behaviors of several asynchronous techniques in two cases: agents process the received messages asynchronously (the real situation) and the synchronous case, when a synchronization of the execution of agents is done. In this paper we examine the effect of synchronization of agents’ execution in random binary problem. Experiments with asynchronous search techniques are standardly conducted on randomly generated networks of constraints. Experimental results illustrate that the synchronization is more effective for several families of asynchronous techniques.

Key–Words: constraints programming, distributed problems, asynchronous searching techniques, synchronization

1 Introduction

Constraint programming is a programming approach used to describe and solve large classes of problems such as searching, combinatorial and planning problems. Lately, the AI community has shown increasing interest in the distributed problems that are solvable through modeling by constraints and agents. The idea of sharing various parts of the problem among agents that act independently and collaborate in order to find a solution by using messages proves itself useful. It has also lead to the formalized problem known as the Distributed Constraint Satisfaction Problem (DCSP) [6].

There exist complete asynchronous searching techniques for solving the DCSP, such as the ABT (Asynchronous Backtracking) and DisDB (Distributed Dynamic Backtracking) [1, 6]. There is also the AWCS (Asynchronous Weak-Commitment Search) [6] algorithm which records all the nogood values. This technique allows the agents to modify a wrong decision by a dynamic change of the agent priority order.

The agents can be processes residing on a single computer or on several computers, distributed within a network. Asynchronous algorithms are characterized by a message passing mechanism among agents when searching for solution. Any practical implementation of these techniques need to manipulate the FIFO channels of messages. It is interesting to examine how these algorithms behave under different synchronization assumptions, as this type of analysis has not been done before. Specifically, it is interesting to investigate the opportunity of synchronizing the agents in case of asynchronous techniques for the random binary problem.

In [5] are proposed two solutions for the synchronization of agents of execution. The first one is based on the existence of a central agent or the access possibility to a common memory zone, solution that allows complete synchronization of the agents. The second one, based on the use of a synchronization message between neighboring agents, allows us to obtain a partial synchronization of the agents.

In [5] the evaluation of the effect of synchronizing agents is done using a particular problem: the problem of coloring a graph in the distributed versions. The behaviors of several asynchronous techniques are investigated in two cases: the agents execute asynchronously the processing of received messages (the real situation from practice) and the synchronous case where the agents’ execution is synchronized. In other words, the agents perform a computing cycle in which they process a message from a message queue in the synchronous case. After that, a synchronization is done waiting for the other agents to finalize the processing of their messages. The first experiments show that the synchronization of the execution of agents re-
duces the costs in finding the solution for several families of asynchronous techniques.

The evaluation of the asynchronous search techniques depends on at least two factors: the types of problems used at the evaluation and the units of measurement used. There are a few types of problems about the evaluation in the DisCSP literature: the distributed problem of the m-coloring of a randomly generated graph and the randomly generated (binary) CSP. This problem is characterized by the 4-tuple (n, m, p1, p2), where: n is the number of variables; m is the uniform domain size; p1 is the portion of the n * (n - 1) / 2 possible constraints in the constraint graph; p2 is the portion of the m*m value pairs in each constraint that are disallowed by the constraint.

There should be mentioned that he randomly generated binary are the most suitable for the evaluation, because they allow different densities for the constraints graph and they have many direct applications in real practice. Therefore, a correct evaluation supposes the selection of a varied class of problems, the more dimensions, the more sets of data chosen randomly, or the choosing of sets of data which allow varied densities for the constraints graph. In this paper extensive evaluation of the asynchronous search techniques are conducted on randomly generated networks of constraints.

In this article one family of asynchronous techniques are analyzed: the AWCS. The AWCS family [2, 6] uses a dynamical order for agents. Learning techniques can be applied to this family for building efficient nogoods. This article analyzes two variants of this family improved by applying a nogood learning technique and nogood processor.

NetLogo, is a programmable modeling environment, which can be used for simulating certain natural and social phenomena [7]. In order to make such estimation, these techniques are implemented in NetLogo, a distributed environment, using a special language named NetLogo, [7], [8], [9]. The implementation and evaluation is done using the models proposed in [4], model calling DisCSP-NetLogo. Implementation examples for the the AWCS family can be found on the website [9].

2 The Framework

This section presents theoretical considerations regarding the DCSP modeling and the asynchronous techniques[1], [2], [6].

2.1 The Distribution Constraint Satisfaction Problem

Definition 1 (CSP model) The model based on constraints CSP-Constraint Satisfaction Problem, existing for centralized architectures, consists in:
- n variables X1, X2, ..., Xn, whose values are taken from finite, discrete domains D1, D2, ..., Dn, respectively.
- a set of constraints on their values.

The solution of a CSP supposes finding an association of values to all the variables so that all the constraints should be fulfilled.

Definition 2 (The DCSP model) A problem of satisfying the distributed constraints (DCSP) is a CSP, in which the variables and constraints are distributed among autonomous agents that communicate by transmitting, messages.

This article considers that each agent Ai has allocated a single variable xi.

The two families of techniques (the ABT, and AWCS families) [6] are characterized by the use of many types of messages in the process of agents communication for obtaining the solution. These messages are similar for these families. They are based on asynchronous search principles defined by the ABT technique [6].

2.2 AWCS Technique

The AWCS algorithm [6], is a hybrid algorithm obtained by combining the ABT algorithm with the WCS algorithm, which exists for CSP. It can be considered as being an improved ABT variant, but not necessarily by reducing the nogood values, but by changing the priority order. The AWCS algorithm uses, like ABT, the two types of ok and nogood messages, with the same significance.

When an agent Ai receives an ok? message, it updates its agent view list and tests if a few nogood values are violated. A generic agent Ai can have the following behavior [6]:

- If no higher priority nogood value is violated, it doesn’t do anything.
- If there are a few higher priority nogood values that have inconsistent values and these values could be eliminated by changing the xi value, the agent will change this value and will send the ok? message.
- If a few higher priority values are inconsistent and this inconsistency can not be eliminated, the agent creates a new nogood messages and sends a nogood message to each agent that has variables in nogood. Than the
agent increases the priority of \( x_i \), by changing the \( x_i \) value with another value that minimizes the number of inconsistencies with all the nogood values and sent the ok? message.

The AWCS algorithm can be improved by applying a learning schema. The nogood learning technique induced in [2] is a new method of learning the nogood values. The idea is that for each possible value of the failure variable, a nogood that forbids that value is selected and than a new good is built outside the one obtained by unifying the selected nogoods.

3 The synchronization of the agents’ execution

In [4] are presented two models of implementation and evaluation for the asynchronous techniques in NetLogo. The first model with synchronization is based on NetLogo elements, using the ask command for the execution of the procedures for treating the agents messages. This command performs a synchronization of the commands attached to the agents such as the synchronization of the execution of agents is made automatically. Examples of implementation can be found on the web sites [8],[9].

In reality, the agents run concurrently and asynchronously, each agent treating its messages from the messages queue in the arrival order, without expecting for the finalization of computations from the other agents. The analysis of experimental results shows that the AWCS techniques behave better in case of synchronizing the agents’ execution. Starting from this remark, in [5] are proposed two solutions of synchronization the execution of agents

3.1 The full synchronization of the execution of agents

The first solution proposed is based on the use of a common memory zone to which the agents have access. In that common memory zone the value of a global variable Cagents, accessible to all the agents is stored.

Initially, that variable is initialized with the number of agents. Each agent will mark the status of its execution in the variable Cagents. Practically, each agent \( A_i \) decrements the Cagents variable by 1 when the message processing routine is executed. Also, when the agent finishes to process the messages from its message queue it increments the value of Cagents by 1. At a given moment, that variable can be equal to the number of agents i.e. all agents have finished to process the messages from the message queue. The first solution allows a complete synchronization of all the agents.

3.2 The partial synchronization of the agents

The second solution consists in the synchronizing only the neighboring agents. Each agent will wait for its connected neighbors to finish their computations, which are placed before him in a lexicographical order. That solution allows a partial synchronization of the execution of agents. That second solution of partial synchronization is based on the use of a synchronization message. This message is similar to a token that each agent needs to receive in order to carry on with the execution of it’s computing cycle. For that, each agent uses a second channel of communication for receiving the synchronization messages.

The working protocol supposes for each agent the completion of two stages:

- each agent processes all the messages from it’s main communications channel performing a computing cycle. The moment when the main message channel is empty, it sends a message of the "synchronous" type message to the neighboring agents, that are before him in a lexicographical order.

- after each cycle, the agents check if they have received the synchronization messages from all of it’s neighbors, placed after him in a lexicographical order, and if not it waits until it receives all those messages.

4 Experimental results

The evaluation of the asynchronous search techniques depends on at least two factors: the types of problems used at the evaluation and the metrics of measurement used. There a few types of problems about the evaluation in the DisCSP literature:

- the distributed problem of the m-coloring of a randomly generated graph, characterized by the number of nodes/agents, \( k=3 \) colors and the m-number of connections between the nodes/agents. There are defined two types of problems: sparse problems (having \( m=n \times 2 \) connections) and dense problems (\( m=n \times 2.7 \)).

- The randomly generated (binary) CSPs are characterised by the 4-tuple \( (n, m,p1,p2) \), where: \( n \) is the number of variables; \( m \) is the uniform domain size; \( p1 \) is the portion of the \( n \times (n - 1) / 2 \) possible constraints in the constraint graph; \( p2 \) is the portion of the \( m \times m \) value pairs in each constraint that are disallowed by the constraint. That is, \( p1 \) may be thought of as the density of the constraint graph, and \( p2 \) as the tightness of constraints.
4.1 The randomly generated DisCSP

A randomly generated DisCSP is an example of a homogeneous unstructured problem. These problems have a number of variables with a fixed domain. Variables belonging to constraints are chosen at random. Specifically, we implement and generated in NetLogo both solvable and unsolvable randomly generated DisCSPs. These problems had one variable per agent so all constraints are between variables belonging to different agents (inter-agent constraints). Specifically, a tuple \(< n, d, p1, p2 >\) was used to generate where \(n\) is the number of variables, \(d\) is the domain size of all variables, \(p1\) is the constraint density and \(p2\) is the constraint tightness.

We implement in NetLogo a random instance generators in two steps: S1: We select with repetition \(t = p1 \frac{n(n-1)}{2}\) random constraints. Each random constraint is formed by selecting without repetition 2 of \(n\) variables. S2: For each constraint we uniformly select without repetition \(q = p2 \cdot d^2\) incompatible tuples of values, i.e. each constraint relation contains exactly \(1 - p2 \cdot d^2\) compatible tuples of values.

Implementation examples for the random instance generator can be found on the website [9].

We used binary constraints with the constraint density controlling how many constraints were generated and the constraint tightness determining the proportion of value combinations forbidden by each constraint. For example, a constraint density of 0.2 would generate 20 where \(n\) is the number of variables) and a constraint tightness of 0.4 would prevent 40 of the possible value combinations of variables involved in a constraint from satisfying the constraint. Such uniform random constraints networks of \(n\) variables, \(k\) values in each domain, a constraints density of \(p1\) and tightness \(p2\), are commonly used in experimental evaluations of DisCSP algorithms.

The experiments were conducted on networks with 15 Agents (\(n = 15\)) and 10 values (\(k = 10\)). Three density parameters were used, \(p1 = 0.2\), \(p1 = 0.4\) and \(p1 = 0.5\). In many cases a density of \(p1 = 0.2\) or 0.3 was used to represent sparse constraint networks and a density of \(p1 = 0.4\) or \(p1 = 0.5\) used for medium networks. The value of \(p2\) was varied between 0.3 to 0.5. This creates problems that cover a wide range of difficulty, from easy problem instances to instances that take several CPU minutes to solve. For every pair \((p1,p2)\) in the experiments we present the average over 100 randomly generated instances(for each version we carried out a number of 100 trials, retaining the average of the measured values). Specifically, we tested the random classes: \(< 15; 10; 0.2; 0.3 >, < 15; 10; 0.5; 0.3 >, < 15; 10; 0.5; 0.5 >\)(100 solvable instances) and \(< 20; 10; 0.2; 0.3 >, < 20; 10; 0.2; 0.5 >, < 20; 10; 0.5; 0.3 >, < 20; 10; 0.5; 0.5 >\).

4.2 Evaluation of AWCS family

In order to make such estimation, two families of AWCS techniques are implemented in NetLogo [7], [8], [9]. The implementation and evaluation is done using the two models proposed in [4].

In order to make the evaluation of the asynchronous search techniques, the message flow was counted i.e. the quantity of messages ok and nogood exchanged by the agents, the number of checked constraints i.e. the local effort made by each agent, and the number of concurrent constraints checked (defined in [3], noted with c-cks) necessary for obtaining of the solution.

In the AWCS family there are many variants that are based on building of efficient nogoods (nogood learning) or on storing and using those nogoods in the process of selecting the values (nogood processor). Two families of AWCS techniques are evaluated:

- the basic variant proposed in [6] improved with the nogood learning technique (notated with AWCS-nl).
- the basic variant proposed in [6] improved with the nogood learning technique and the nogood processor technique (notated with AWCS-nlng).

A version in which each agent treats entirely the existing messages in its message queue is implemented for this variant. In fig. 1 shows the new handle-message procedure. Four implementations are done corresponding to the obtained models:

- variants based on synchronization with the aid of the "ask" command: AWCS-nl1, AWCS-nlng1.
- variants based on the asynchronous model from [4]: AWCS-nl2, AWCS-nlng2.
- variant based on the first method of synchronization(complete synchronization)-AWCS-nl3, AWCS-nlng3.
- variant based on the second solution of synchronization -AWCS-nl4, AWCS-nlng4.

Concerning the complete or partial processing of the messages, it can be done by means of the mszize variable or renouncing of the second condition of package limitation (the line labeled with *** in fig. 1). That has a role to decide the number of extracted and processed messages from the message queue. If mszize is equal to the number of elements from the
to handle-message [msize]
set nrm 0
1 while [not empty? message-queue and nrm <= msize] or
1' while [not empty? message-queue] ***
[
  set msg retrieve-message
  if (first msg = "stop")
...
  set nrm nrm + 1
]
If (Nrm != 0)
  [Check-agent-view]
end

Figure 1: The handle-message for the AWCS message queue or if that condition is missing, the procedure presented in fig. 1 allows the processing of all the messages. In this article the performances of the variant that was obtained for msize=\text{length(message-queue)} are investigated and analyzed.

In figure 1 exist two ways of finishing message treatment. The first variant supposes the insertion of line 1 instead of line 1'. In that case, each agent stops in the moment in which either it has no more messages or msize messages were processed. The second variant supposes the insertion of line 1'. That variant will force the agents to stop if there are no more messages in their message queues. Message processing supposes an effort and thus the occurrence of a delay. It is possible to arrive other messages beside those from the initial moment. In the case of the second variant (obtained by inserting the line 1’) if later new messages appear, those are still treated thus surpassing the number msize of messages allowed. But, for the first variant is treated only those messages existent at the beginning in the message queue. In this paper, we implement the both variants which supposes the insertion of line 1’ and line 1.

Results appear in Tables 1 and 2, where we report the number of checked constraints (Constr.) the number of concurrent constraint checks (Tc-ccks) and the total number of messages exchanged (Tmess), averaged over 100 executions. These are the results of the first version of message management, which involves the insertion of line 1 instead of line 1’. In that case, each agent stops in the moment in which either it has msize messages were processed.

As known, the checked constraints quantity evaluates the local effort given by each agent, but the number of concurrent constraint checks allows the evaluation of this effort without considering that the agents work concurrently (informally, the number of concurrent constraint checks approximates the longest sequence of constraint checks not performed concurrently). Analyzing the results from table 1, one can notice remark that synchronization of execution of agents reduces the local effort made by the agents. In case random instances of 15 agents and 10 values per agent In case of instances with 15 agents, the two synchronization solutions have proximate costs. On the other hand, for instances with a larger number of agents, the version with partial synchronization of neighbours is the best.

The more the difficulty of the problems and the density of the constraint graph grow (p2=0.5 or p1=0.5), the more the costs of the synchronization solutions decrease. But, as the difficult of the problems increases (n=20 agents, p2=0.5), the asynchronous variants AWCS-nl2 required much greater efforts compared to the variants with synchronization. Comparing the two synchronous variants, is remarkable the version with partial synchronization of neighbours (table 2).

Table 1: The results for AWCS-nl versions (The randomly generated DisCSP - n=15 agents)

<table>
<thead>
<tr>
<th>n = 15 agents</th>
<th>p1=0.2</th>
<th>p1=0.5</th>
<th>p2=0.3</th>
<th>p2=0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWCS-nl1</td>
<td>TMess: 57.13</td>
<td>78.38</td>
<td>181.94</td>
<td>1320.62</td>
</tr>
<tr>
<td></td>
<td>Constr: 148.06</td>
<td>270.34</td>
<td>643.76</td>
<td>6448.14</td>
</tr>
<tr>
<td></td>
<td>Tc-ccks: 68.25</td>
<td>163.28</td>
<td>397.18</td>
<td>4941.52</td>
</tr>
<tr>
<td>AWCS-nl2</td>
<td>TMess: 58.84</td>
<td>383.14</td>
<td>245.00</td>
<td>3844.36</td>
</tr>
<tr>
<td></td>
<td>Constr: 172.61</td>
<td>383.14</td>
<td>1187.72</td>
<td>19434.41</td>
</tr>
<tr>
<td></td>
<td>Tc-ccks: 52.65</td>
<td>167.43</td>
<td>408.33</td>
<td>5440.09</td>
</tr>
<tr>
<td>AWCS-nl3</td>
<td>TMess: 59.35</td>
<td>91.18</td>
<td>245.17</td>
<td>4085.82</td>
</tr>
<tr>
<td></td>
<td>Constr: 173.52</td>
<td>366.39</td>
<td>1102.83</td>
<td>19423.43</td>
</tr>
<tr>
<td></td>
<td>Tc-ccks: 54.91</td>
<td>167.61</td>
<td>413.00</td>
<td>6137.24</td>
</tr>
<tr>
<td>AWCS-nl4</td>
<td>TMess: 59.14</td>
<td>91.04</td>
<td>240.89</td>
<td>2726.05</td>
</tr>
<tr>
<td></td>
<td>Constr: 171.04</td>
<td>368.54</td>
<td>1066.89</td>
<td>13201.52</td>
</tr>
<tr>
<td></td>
<td>Tc-ccks: 54.54</td>
<td>165.96</td>
<td>402.22</td>
<td>4359.52</td>
</tr>
</tbody>
</table>

Table 2: The results for AWCS-nl versions (n=20)

<table>
<thead>
<tr>
<th>n = 20 agents</th>
<th>p1=0.2</th>
<th>p1=0.5</th>
<th>p2=0.3</th>
<th>p2=0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWCS-nl1</td>
<td>TMess: 100.26</td>
<td>180.35</td>
<td>661.59</td>
<td>104981</td>
</tr>
<tr>
<td></td>
<td>Constr: 259.00</td>
<td>667.77</td>
<td>2695.89</td>
<td>642629</td>
</tr>
<tr>
<td></td>
<td>Tc-ccks: 100.00</td>
<td>340.08</td>
<td>1843.38</td>
<td>450727</td>
</tr>
<tr>
<td>AWCS-nl2</td>
<td>TMess: 104.23</td>
<td>222.05</td>
<td>4725.65</td>
<td>228429</td>
</tr>
<tr>
<td></td>
<td>Constr: 304.58</td>
<td>966.42</td>
<td>1253.31</td>
<td>46030</td>
</tr>
<tr>
<td></td>
<td>Tc-ccks: 60.06</td>
<td>283.57</td>
<td>1472.80</td>
<td>34600</td>
</tr>
<tr>
<td>AWCS-nl3</td>
<td>TMess: 103.77</td>
<td>220.65</td>
<td>1085.28</td>
<td>155253</td>
</tr>
<tr>
<td></td>
<td>Constr: 293.63</td>
<td>917.00</td>
<td>4682.45</td>
<td>155253</td>
</tr>
<tr>
<td></td>
<td>Tc-ccks: 60.17</td>
<td>293.83</td>
<td>1472.80</td>
<td>34600</td>
</tr>
<tr>
<td>AWCS-nl4</td>
<td>TMess: 101.61</td>
<td>221.17</td>
<td>1055.97</td>
<td>13837</td>
</tr>
<tr>
<td></td>
<td>Constr: 281.21</td>
<td>915.90</td>
<td>4523.29</td>
<td>70710</td>
</tr>
<tr>
<td></td>
<td>Tc-ccks: 57.39</td>
<td>283.13</td>
<td>1452.14</td>
<td>20725</td>
</tr>
</tbody>
</table>

In the case of the message flow, one can notice almost equal efforts for obtaining the solution. The message flow increased for the synchronous variants together with the increase of dimension for the solved problems. Comparing the two synchronous variants, one can notice almost equal efforts for obtaining the solution, such as the practical solution proposed in this
The techniques from the AWCS family, based on a dynamic order for the agents, require lower costs for obtaining the solution in case of synchronization of the agents’ execution. The experiments show a decrease of local computing effort and of message flow compared to the asynchronous variants in randomly generated networks of constraints.

The analysis of two large families of asynchronous search techniques shows that there are different behaviors for the agents, which can influence in a good or in a bad manner the costs for obtaining the solution. The analysis of the experimental values in randomly generated networks of constraints shows that the synchronization can reduce the costs of obtaining the solutions. The relative performance of the two synchronization solutions was not influenced by the density of the constraints graph or the difficulty of the problem.

Table 3: The results for AWCS-nlng versions (n=20)

<table>
<thead>
<tr>
<th>n = 20 agents</th>
<th>p1 = 0.2</th>
<th>p1 = 0.5</th>
<th>p2 = 0.5</th>
<th>p2 = 0.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWCS-nlng1</td>
<td>TMess=188.65, Constr=707.04, Tc-ckks=357.32</td>
<td>TMess=941.18, Constr=599.05, Tc-ckks=38524.25</td>
<td>TMess=297.81, Constr=19187.09, Tc-ckks=287.81</td>
<td></td>
</tr>
<tr>
<td>AWCS-nlng2</td>
<td>TMess=228.83, Constr=1005.00, Tc-ckks=307.13</td>
<td>TMess=17108.22, Constr=83662.00, Tc-ckks=16945.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWCS-nlng3</td>
<td>TMess=226.72, Constr=941.07, Tc-ckks=305.17</td>
<td>TMess=228.83, Constr=9187.09, Tc-ckks=224.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWCS-nlng4</td>
<td>TMess=217.91, Constr=911.80, Tc-ckks=287.81</td>
<td>TMess=12382.00, Constr=63897.00, Tc-ckks=51987.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: The results for AWCS-nl versions (The second version of message management)

<table>
<thead>
<tr>
<th>n = 20 agents</th>
<th>p1 = 0.2</th>
<th>p1 = 0.5</th>
<th>p2 = 0.5</th>
<th>p2 = 0.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWCS-nl1</td>
<td>TMess=190.58, Constr=703.92, Tc-ckks=359.21</td>
<td>TMess=5581.31, Constr=28762.63, Tc-ckks=20468.53</td>
<td>TMess=287.81, Constr=19187.09, Tc-ckks=287.81</td>
<td></td>
</tr>
<tr>
<td>AWCS-nl2</td>
<td>TMess=202.33, Constr=803.15, Tc-ckks=335.51</td>
<td>TMess=11019.00, Constr=57683.14, Tc-ckks=28246.62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWCS-nl3</td>
<td>TMess=194.85, Constr=742.02, Tc-ckks=323.26</td>
<td>TMess=6144.96, Constr=32875.30, Tc-ckks=18968.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWCS-nl4</td>
<td>TMess=203.97, Constr=769.51, Tc-ckks=339.14</td>
<td>TMess=7612.78, Constr=40555.26, Tc-ckks=24377.56</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 Conclusions

References:


