DE GRUYTER OPEN

DOI: 10.1515/aucts-2015-0060

ACTA UNIVERSITATIS CIBINIENSIS – TECHNICAL SERIES
Vol. LXVII 2015

# INTRODUCING A NEW DISTRIBUTED CONSTRAINT TOPOLOGY: INITIAL EXPEIMENTS

***ILIE Sorin***
***Ph.D. Eng. , Faculty Department of Computers and Information Technology,***
***University of Craiova, Romania, ilieviorelsorin@gmail.com***

**Abstract:** This paper introduces a new topology and a framework for distributed constraint optimization approaches to solving complex problems. Initial experiments with this approach show decreasing communication overhead, high scalability and low execution times due to parallelization of tasks. In the experiments section we analyze a dry random choice algorithm run of the framework and measure the overhead time, then we compare the topology performance with a two level arborescent approach to distributed constraint optimization. The results show a significant improvement on execution time and better scalability.

**Key words:** Distributed constraint optimization, distributed computing, complex problems

## 1. Introduction

A class of algorithms that solve complex problems is Distributed Constraint Optimization (DCO) [1]. This type of approach uses a set of agents that collaboratively set values to subsets of solutions minding the cost associated with a given set of constraints.

*The No Commitment Branch and Bound* (NCBB) [2] algorithm petitions the search space as well as the constraints. This results in an arborescent virtual communication structure in which the same level siblings partition the solution space and paternal relations distribute the constraints. The leafs explore the space and pass a local-optimal solution to the parent. Solutions can be accepted or rejected by using heuristic functions. NCBB is of polynomial complexity (P), but consumes an exponential number of messages.

ADOPT [3] uses a semi-arborescent communication topology but works just like NCBB. The difference is that ADOPT does not use heuristic functions but transfers minimum and maximum cost values for solutions through asynchronous messages. As a consequence ADOPT has the same complexity as NCBB.

The algorithm DPOP [4] the communication structure is also semi-arborescent communication structure. However, in this case, parents generate all possible solutions to be tested by the offspring. DPOP has exponential asymptotic complexity but linear communication complexity.

OPTAPO [5] uses an arborescent topology. This algorithm tests for conflicts using mediator agents with limited access to agent constraints. Mediators use *branch and bound* just like NCBB to solve these conflicts. OPTAPO has polynomial computational complexity and exponential message complexity.

The Dischoco [5] framework facilitates DCO, by handling message passing, constraints and variables but the last two have to be distributed by some other code.

The reviewed algorithms have workflows of agents that wait for results from other agents, which implies a lot of idle time. This idle time can be reduced by relaxing the topology and permitting any agent to communicate with any other available agent. For that reason we propose a new and completely distributed architecture without critical agents such are the root agent in the arborescent topologies. The solution implies that agents simply pass partial solution between them appending or rejecting them on the basis of their internal constraints.

## 2. Framework architecture

Our architecture is based on assigning each agent a part of the constraints and a subset of the solution space. The preferred topology is complete graph however any other topologies are supported for backward compatibility with other algorithms.

In this approach each agent creates its own population of partial solutions from its solution space, which are software objects [8]. A partial solution can migrate from one agent to another. When the agents are

managed by different machines the partial solution is sent by way of messages. However, if the agents are managed by the same machine the partial solution is enqueued in a local queue. Solution movement is accomplished in both cases by the SEND-TO() method. An agent has two other methods continuously handling incoming partial-solutions: RECEIVE-SOLUTION() that handles solutions received in the form of messages and LOCAL-MOVE() that handles the partial-solutions in the local queue. These methods are detailed in the following code:

```
RECEIVE-SOLUTION()
1. RECEIVE(partialSolution)
2. ADJUST(partialSolution)
3. SEND-TO(BEST-NEIGHBOR(partialSolution), partialSolution)

SEND-TO(destination, partialSolution)
1. if LOCAL(destination) then
2.   ENQUEUE(partialSolution)
3. else
4.   SEND(destination, partialSolution)

LOCAL-MOVE()
1. DEQUEUE(partialSolution)
2. ADJUST(partialSolution)
3. SEND-TO(BEST-NEIGHBOR(partialSolution), partialSolution)
```

Partial solutions also contain metadata attributes like:

- cost of the current partial solution
- agent of origin
- best solution cost (the cost of the currently best solution, this is the synchronization means)
- a list of agents representing the path followed so far

### 1.1 Framework Setup

The framework was designed to work on any computer network including a computer cluster. The main requirements to setup the framework are:

1. locating the executable java archive "*framework.jar*" in a storage space that is accessible to every computing node, along with the problem graph file
2. running the JADE platform on one of the computing nodes
3. running auxiliary containers connected to the platform from step 2 on the other computing nodes
4. running the Bootstrap agent that will automatically detect the available resources and distribute the agents and vertices.

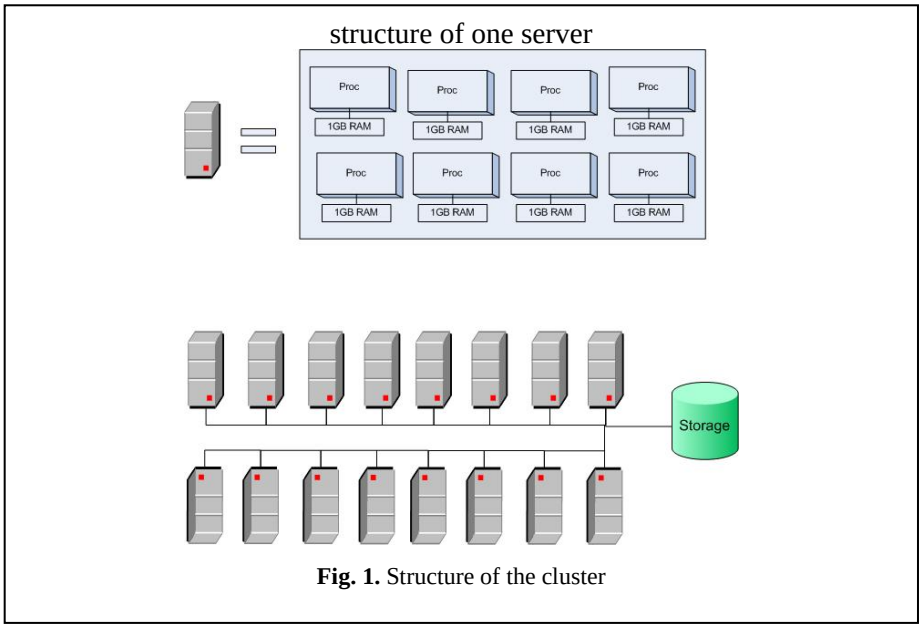


**Fig. 1.** Structure of the cluster

The test framework was deployed and used on an Infragrid cluster consisting of 128 cores, each one equipped with 1GB of RAM, connected by an Infiniband 40 Gbits/s network. This cluster of computing nodes is located in a remote temperature controlled room from the West University of Timisoara HPC[1]. The cores share the same external storage space. This computer cluster is composed of 16 servers with 8 cores per server, each of type Intel Xeon CPU E5504 . Each server is allocated a single IP address. The cluster is running Linux and Condor workload management system [6]. Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users can submit their parallel or serial jobs, which will be placed into a queue and, based upon a policy, resources will be allocated to them. Submission of a job is done by means of a "**.condor*" script that indicates input, output and Condor log files, the executable and the universe. A Condor universe defines an execution environment. To deploy our system we used the Parallel universe [6].
The JADE platform needs to be running in order for additional containers to be created. For the initialization on a Condor computer cluster two jobs must be scheduled: platform execution on a single computing node (i.e. a cluster core) and creation of containers, each on a separate machine.
This stage is particularly challenging since the Java universe supports only serial jobs. We are forced to use the Parallel universe to run a shell script on multiple machines (see script run.condor).

```
run.condor
executable =./run.sh
Universe = parallel
Arguments = platform pr1002 10
output = out.txt
error = err.txt
log = log.txt
machine count= 1
Queue
Arguments = container
output = out$(Node).txt
```

The shell script executes the framework.jar. This jar can be used to start the framework.

```
run.sh
#!/bin/bash
java -jar framework.jar $@
```

In the script we present the actual scripts that are used to launch our framework on the Condor computer cluster. Each machine has its own output and error file defined by its unique identification number $(Node). The Condor log file of the job batch can also be specified in the script. In this example 7 containers are created for the purpose of the experiment.
The framework.jar, the problem input file, the job scheduling script, and the shell script have to be transferred to the shared storage space using File Transfer Protocol (FTP). Afterwards remote access must be utilized to make the shell script executable by running the Linux command "*chmod +x run.sh*". Using an executable shell script to run the *framework.jar* allows us to quickly update the code to the latest version. Specifically, there is no need to change the use rights of every new version of the file framework.jar using the command "*chmod*".
At this time a job scheduling script can be executed in mere seconds, starting experiments at will from a single remote point of access.

## 3. Framework Overhead Experimental Results

We tested the framework overhead using the simple random choice algorithm implementation on $k$= {1,2,3,4,5,6,7,10} computing nodes, using one agent per computing node and executing 10 experimental rounds for every value of $k$. The stop condition is set for each agent to finish moving $M$=10000 partial solutions. Since at this point in our research we are just trying to improve the topology and communication time we will only be using a random choice algorithm to explore solutions without evaluating them.

As a test map we used "pr1002", a 1002 vertex graph from the benchmark library TSPLIB [7]. Therefore in total, during the experiment the framework will execute a total of approximately 10 million

[1] http://hpc.uvt.ro

partial-solution migrations. We used AGENT-INIT() to initialize one empty partial-solution per agent, summing up 1002 partial-solutions in the whole experiment.

We used the HPC Infragrid cluster[2] in this experiment and we obtained the performance measures in **Table 1** where $T_{min}$ is the minimum execution time obtained by experimental round, respectively $T_{max}$ is the maximum execution time while $T_{avg}$ is the average execution time per experiment. We also calculated the maximum deviation $d_{max}$ from $T_{avg}$ normalized to the value of $T_{avg}$ as presented in the following equation.

$$d_{\max} = \frac{(T_{\max} - T_{\min}) / 2}{T_{avg}} \tag{1}$$

**Table 1.** Framework overhead

| $k$ | $T_{min}[s]$ | $T_{max}[s]$ | $T_{avg}[s]$ | $d_{max}$ |
|---|---|---|---|---|
| **1** | 15.5 | 16.1 | 15.8 | 0.017101 |
| **2** | 1095.2 | 1244.4 | 1141.6 | 0.065324 |
| **3** | 1105.2 | 1201.6 | 1153.3 | 0.041763 |
| **4** | 1011.5 | 1091.5 | 1050.8 | 0.038042 |
| **5** | 942.4 | 997.5 | 974.3 | 0.028279 |
| **7** | 872.9 | 924.3 | 888.8 | 0.028899 |
| **10** | 729.1 | 771.7 | 751.1 | 0.028356 |

In **Fig. 2** we present a graphical representation of the values $T_{min}$, $T_{max}$ and $T_{avg}$ from **Table 1**. We can observe a steep climb of overhead when the framework uses more computing nodes compared to a single computing node. This is due to the fact that part of the partial-solution moves become messages between the two agents running on the two available computing nodes. Furthermore we observe a slight decreasing trend in the execution time as the number of computing nodes further increases to $k$=3 and so on.

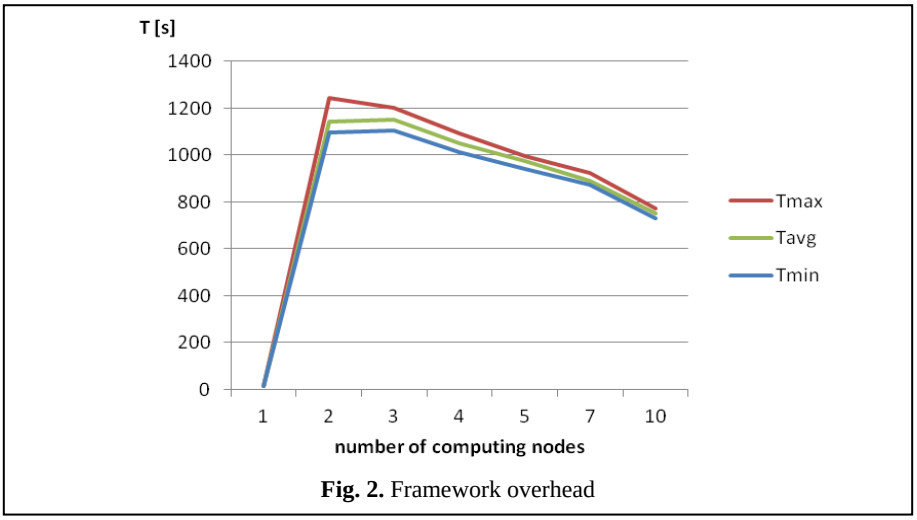


**Fig. 2.** Framework overhead

In **Fig. 3** we present a graphical representation of the deviation from $T_{avg}$ of the execution time of experimental rounds. We can observe how this value is very low when using a single computing node and greater when using more computing nodes. This is due to the random nature of choosing the next agent to send the partial-solution to. The random choice generates a variable number of messages and local moves. We can, however, observe a decreasing pattern in $d_{max}$ when the number of computing nodes increases. This is due to the fact that the probability of choosing a agent that is managed by a different machine rises. In turn, this causes a stabilization of execution time. For example, when using 2 computing

[2] http://hpc.uvt.ro

nodes a partial-solution will move to another agent with the probability of 50%, while when using 10 computing nodes the probability of sending a partial-solution to another agent is 90%.

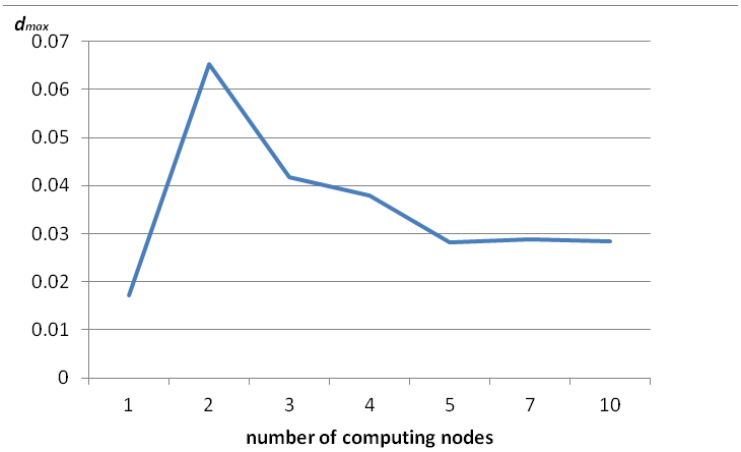


**Fig. 3.** Maximum deviation from $T_{avg}$

These results show how the overhead of the test framework evolves in relation to the number of computing nodes used. Note that although the number of messages rises the execution time still decreases thanks to the distributed nature of the framework.

In **Table 2** we present a side by side comparison of our approach and a two level arborescent topology (also known as the master-slave topology) trying to solve TSP map pr1002 using a simple random solution generation and cost evaluation. Note that in this case we are using evaluation of solutions since we are trying to compare actual working models. Execution stops at 10 000 partial-solution evaluations. **Table 2** shows that our framework solutions offer significantly better execution times and better scalability than a master slave approach.

**Table 2.** Side by side comparison

| | | ***Our approach*** | ***2 level arborescent model*** |
|---|---|---|---|
| ***k*** | **Partial solutions per agent** | $T_{avg}[s]$ | $T_{avg}[s]$ |
| **1** | 1002 | 22718,23 | 22558,11 |
| **5** | 200 | 4844,64 | 17903,89 |
| **8** | 125 | 2915,77 | 11525,83 |
| **10** | 100 | 2312,82 | 12082,34 |
| **15** | 66 | 1448,88 | 16419,84 |
| **20** | 50 | 1536,75 | 21855,6 |

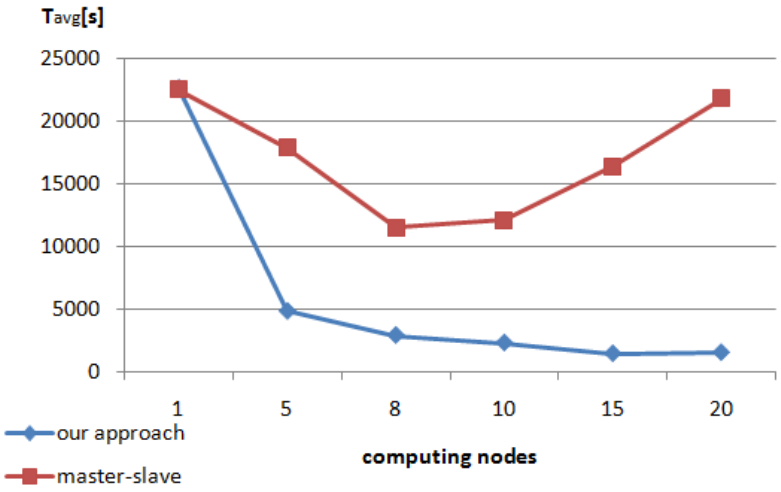


**Fig. 4.** Our approach compared to the arborescent approach on map pr1002

**Fig. 4** depicts a graphical representation of the contents of **Table 2** resulting in a visual comparison of the execution time $T_{avg}$ obtained by our architecture compared with $T_{avg}$ of the arborescent model when increasing the number of computing nodes.

## 4. Conclusions and future work

In this paper we proposed a new topology for DCO that shows promising results in initial experiments: high scalability and better execution time in the case of random solution search. Future work includes a more in-depth comparison with existing approaches and further analysis of the frameworks scalability.

## 5. Acknowledgement

This work was supported by the strategic grant POSDRU/159/1.5/2/133255. Project ID 133225(2014), co-financed by the European Social Fund within the Sectorial Operational Program Human Resources Development 2007- 2013.

## 6. References

1. Yokoo, Makoto (2012), ”Distributed constraint satisfaction: foundations of cooperation in multi-agent systems”, Springer, ISBN 978-3-642-59546-2
2. Yeoh, W., Felner, A., Koenig, S.An asynchronous Branch-and-Bound DCOP algorithm (2010) Journal of Artificial Intelligence Research, 38, pp. 85-133
3. Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, Makoto Yokoo (2005), „Adopt: asynchronous distributed constraint optimization with quality guarantees”, Artificial Intelligence, Volume 161, Issues 1–2, January 2005, Pages 149-180, ISSN 0004-3702, http://dx.doi.org/10.1016/j.artint.2004.09.003.
4. Vinyals, M., Rodriguez-Aguilar, J.A., Cerquides, J. Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law (2011) Autonomous Agents and Multi-Agent Systems, 22 (3), pp. 439-464.
5. Mailler, Roger; Lesser, Victor (2004), "Solving Distributed Constraint Optimization Problems Using Cooperative Mediation", Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2004, IEEE Computer Society, pp. 438–445
6. D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience, Concurrency and Computation: Practice and Experience 17 (2-4), pages 323 - 356, 2005
7. G. Reinelt. Tsplib - a traveling salesman library. ORSA Journal on Computing 3, pages 376 - 384, 1991.
8. C Thomas Wu. An Introduction to Object-Oriented Programming with Java, ISBN-10: 0071283684, ISBN-13: 978-0071283687, McGraw-Hill Higher Education, 2009.