Simulation vs Real Execution in DCOP Solving

Francisco Cruz, Patricia Gutierrez, Pedro Meseguer

IIIA - CSIC, Universitat Autònoma de Barcelona, 08193 Bellaterra Spain
{tito,patricia,pedro}@iiia.csic.es

Abstract. This paper presents an experience on solving DCOP instances in a real, distributed scenario with up-to-date technology involving several machines and comparing this process with their solving in a simulator. From the result of this experience, we stress the importance of message communication time, orders of magnitude higher than the elements used in other proposed metrics to assess DCOP algorithms performance. Network latency, captured in message communication time, has an important impact in final performance and it must be necessarily taken into account to accurately approximate the elapsed time of the solving process. The results of this paper are an indicator of this, and we think they may be of interest for the Distributed Constraint Reasoning (DCR) community.

1 Introduction

It is well known that the interest for solving distributed constraint problems started in the 90's. A strong motivation for this kind of problems was to avoid keeping all the problem data in a single computer or in a single agent. Distributed solving assures some kind of privacy, prevents a single point of failure and allows a higher autonomy. This interest crystallized in the pioneer paper of Yokoo and colleagues [15], who introduced the ABT algorithm for solving distributed constraint satisfaction problems (DisCSP for short). Soon the same interest was applied to optimization problems, generating distributed constraint optimization problems (DCOP for short). There are several applications in multi-agent systems for DCOPs, since a wide range of combinatorial problems that are naturally distributed can be modeled in this way [12], [6], [5], [9].

DCOPS are solved by the coordinated action of agents, which communicate through messages, optimizing a global utility function composed by joint utilities of subsets of agents. Decision making algorithms provide mechanisms for agents to explore possible solutions and choose the one that maximize their global utility. Several algoritms has been proposed in this direction, following different strategies in communication, such as ADOPT [10], DPOP [11], AFB [3], BnB-ADOPT [14] and BnB-ADOPT^{+}-MAC/MFDAC [4].

Most –if not all– work done in the DCR community has been done using simulators, running on a single computer but imitating the characteristics of a distributed environment; this allows us to work without requiring a pre-defined infrastructure. Usually simulators make simplified assumptions that, in some cases, "idealize" the distributed environment they try to recreate.

Typically, DCR algorithms have been evaluated along two dimensions: computation and communication. Regarding computation, the number of non-concurrent constraint checks (NCCCs) has been used to assess the longest computation chain that cannot be executed concurrently [8]. Alternatively, other authors use simulated time (following an approach similar to NCCCs but with clock tics instead of constraint checks, assuming the same CPU per agent) [13]. Both ideas are based on the concept of logical clocks [7]. Regarding communication, the number of exchanged messages has been the main evaluation measure, under the assumption that messages do not have very different lengths; otherwise, the total number of bytes exchanged can be used to compare algorithms. The use of logical metrics is convenient in many cases because it allows to compare different algorithms regardless of their specific implementations or the speed of the hardware where they are executed.

In this paper we present a communication protocol for real distributed DCOP solving and describe a simple experiment: the real execution of a DCOP algorithm –BnB-ADOPT$^+$– solving a set of random instances on a real network. Ideally each computer is dedicated to a single agent but overloading a computer with more than one agent is also possible (we loose in distribution though) so we test this as well. Such experiment could be envisioned because we were in full control of the computer network. We encounter a significant number of technical difficulties that we try to clarify in this paper, however final results are robust and with them we try to give a first step to close the gap between simulation and real execution.

The rest of the paper is structured as follows. In Section 2, we review the main evaluation measures performed in the field of DCR. In Section 3 we describe the experiment done, regarding the hardware, the communication software used, the communication protocol designed, the algorithm –BnB-ADOPT$^+$– and the instances tested. In Section 4 we present the results obtained, including a discussion taking into account the main differences observed between simulation and real execution. Finally, in Section 5 we conclude the paper.

2 Related Work

As stated before, usually DCR algorithms are evaluated along two dimensions: communication and computation. In the case of communication, a counter for the number of exchanged messages is typically used. In the case of computation, the most influential metric proposed is the non-concurrent constraint checks (NCCCs) [8]. To calculate NCCCs, every agent has a counter that is incremented every time a constraint is evaluated. This counter is sent in every message. When an agent receives a message, the counter of the agent is updated with the higher value between its own counter and the counter of the received message. When execution ends, the NCCCs metric is calculated as the highest value among all agent counters. This value can be seen as the longest sequent of constraint checks performed non-concurrently.

While NCCCs encompass properly the computational issues regarding concurrency, it presents some weaknesses regarding the measure of computational effort. Basically, the criticisms regarding constraint checks in centralized are also applicable to NCCCs in distributed, namely:

- As we move from simple extensional constraints, where a constraint check is simply a table lookup, constraint checks may have quite different costs and their aggregation may not give a fair estimation of the computation effort performed by an algorithm.
- Propagating global constraints is a central part of current constraint technology and it is also considered in distributed constraint solving [1]. However, this propagation does not cause constraint checks. Ad-hoc recipes –none entirely satisfactory– have been used to incorporate the propagation cost into the NCCC counter.

As alternative there is the simulated run time metric [13]: an approach similar to NCCCs but with clock tics instead of constraint checks. Each agent has an internal clock. When an agent sends a message, it includes the value of its internal clock. When an agent receives a message, if the message clock value is higher than the agent's internal clock, the agent updates its clock with the message clock value. When the algorithm terminates, its runtime is then defined as the latest time indicated by any agent's clock. However, this metric is dependent of the programming language used, the speed of the CPU, etc., so it may be difficult to compare two algorithms (in fact, using simulated time is comparing the implementations of these algorithms on a particular platform).

When messages are not instantaneous, the issue of measuring distributed performance becomes complex. The importance of latency of messages has motivated in some authors variations in the concept of NCCCs, proposing combinations between NCCCs and communication time, such is the case of LTC[16] or ENCCC [2], trying to use these metrics to approximate elapsed time (the time period since the solving process starts until the last agent terminates).

Some authors have also introduced delays in simulated message communication, evaluating their impact on running algorithms [16]. This approach allows to recreate more realistic scenarios and to better understand algorithms behavior and robustness with respect to simulated network latency.

3 Experiment

In this section we explain our hardware and software infrastructure and the instances we use in our evaluation. Results are discussed in Section 4.

3.1 Hardware

The hardware used for running the experiments is an IBM BladeCenter HS22. This platform provides shared storage and connectivity to six individual machines, the so-called blades. Each blade has two quad-core processors Intel®Xeon®E5504 @ 2GHz and 16GB RAM. The connectivity is done using the Broadcom BCM5709S dual-port Gigabit Ethernet available in every blade through two redundant Nortel Networks L2/3 GbESM switches.

This infrastructure allow us to run experiments on top of 48 processors which are uniformly distributed over a six computer Local Area Network (LAN). This scenario reproduces a typical distributed environment where different PCs are connected using a network switch. Network latencies and communication bandwidths are equivalent to what we find in labs, offices and other work or domestic network environments.

3.2 Communication Middleware

We developed our communication protocol using the Java Messaging Service (JMS). It is a middleware designed to help Java developers to build distributed applications. The JMS API provides a collection of methods for sending and receiving messages in a distributed scenario.

The basic roles in JMS communication are producers and consumers. Messages must be sent by a producer and must be read by a consumer. Communications is achieved through a broker, which is a piece of software able to process (send/receive/store) messages. The simplest scenario involves at least one broker and all the devices participating on the JMS application send and receive messages from and to the broker respectively (we use this approach in our experiments). Another relevant aspect of JMS communication is that it allows blocking and non-blocking message consumptions, implemented by *receive()* and *receiveNoWait()* methods respectively.

We distinguish two types of communication:

- Topics: A JMS Topic is a broadcast-like communication, where a producer can send a message to the topic and there can be one or more consumers, subscribed to the topic, that receive it.
- Queues : A JMS Queue is a FIFO type queue where messages are inserted by a producer and deleted by a consumer.

There are different implementations of JMS -known as JMS providers- which are able to interact between themselves and share a common interface from the programmer point of view. Known JMS providers are Apache ActiveMQ, JBoss Messaging, Websphere MQ, among others. Our algorithm is developed using Apache ActiveMQ, which is a popular and powerful open source JMS provider.

With this communication API we are able to work with a number of desirable features: we can exchange messages between any pair of agents (computers) using queues regardless of their ordering (if there is any) or their constraints; it allows to broadcast messages to a subset of agents or to all agents using topic subscription; message passing assures the very common assumption in DCR algorithms that messages sent between two agents can be delayed by a finite amount of time but are never lost.

3.3 Communication Protocol

In this section we describe how problems are distributed and solved along the network, i.e. the communication protocol. Following this protocol, agents are distributed in different computers, the optimal solution is calculated and statistics are collected to compute the final evaluation metrics (total messages exchanged, NCCCs and total elapsed time). The following protocol works for any problem specification regardless of the number of computers available and the problem features (number of variables, domain size, number of constraints). In many cases synchronization points are needed, for example to determine if all agents are ready to start, if all agents have finished their execution, etc. These synchronizations are achieved selecting a master computer which is in charge of performing special checks and using topic subscription and special messages.

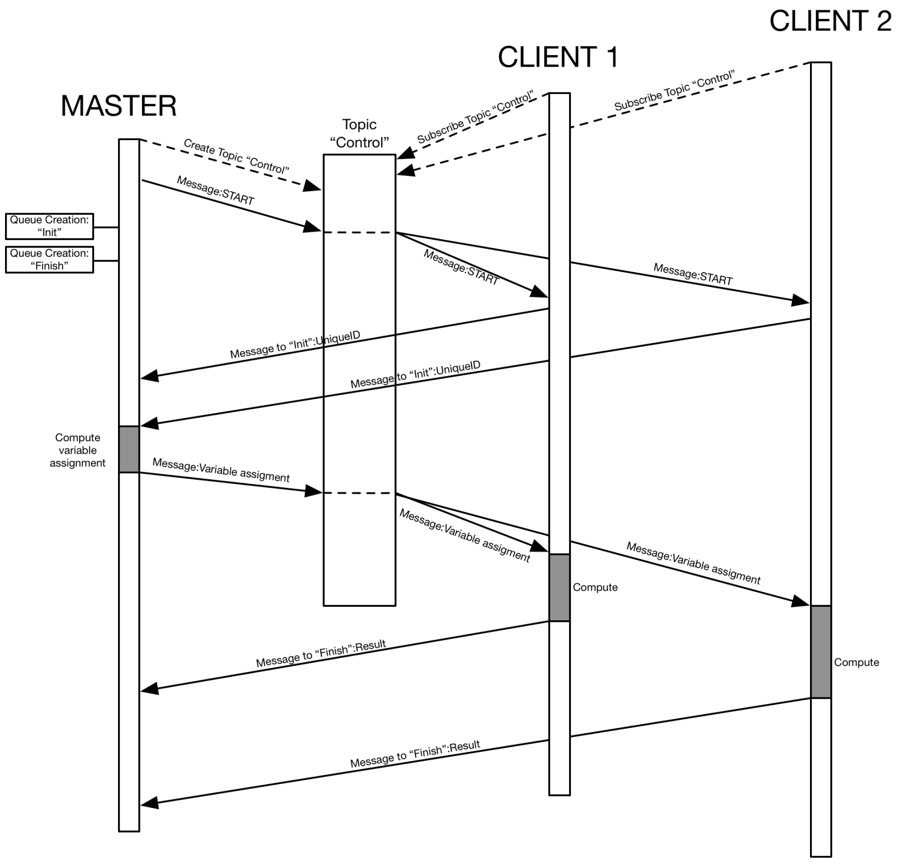
MASTER
CLIENT 1
CLIENT 2
Topic "Control"
Create Topic "Control"
Subscribe Topic "Control"
Subscribe Topic "Control"
Message:START
Queue Creation: "Init"
Queue Creation: "Finish"
Message:START
Message:START
Message to "Init":UniqueID
Message to "Init":UniqueID
Compute variable assignment
Message:Variable assigment
Message:Variable assigment
Message:Variable assigment
Compute
Compute
Message to "Finish":Result
Message to "Finish":Result

Fig. 1. Communication structure between agents using ActiveMQ.

A flow diagram is shown in Figure 1, showing the main phases of communication. A complete pseudocode of the protocol is show in in Figure 2. In words, the master creates a topic to broadcast messages to the rest of computers in the network (from now on, we call them client computers). It also creates an "Init" and "Finish" queue to synchronize execution, and waits for client computers to subscribe to the topic and queues. After this, it sends a "Start" message to all subscribers. Client computers receiving the "Start" message send their connection Id to the master using the "Init" queue. After this, the master knows how many computers are active in the network, and it can send messages to the clients directly, without the need to broadcast. The master assigns the problem variables to the clients and to itself in a round-robin way, so that variables are proportionally assigned among all computers.

If there are more variables than computers, computers are overloaded with more than one variable. Observe that we can obtain here three possible scenarios (we consider all three in Section 4):

1. Fully simulated execution in one computer. This is, only one computer is used and it handles all problem variables/agents. In this case we have a classic simulation involving all agents.
2. Fully distributed. This is the case where we have the same number of variables or less than computers. In this case, only one variable/agent is assigned to a computer and we obtain a fully distributed execution.
3. Partially distributed/simulated: In this case, several computers may be overloaded with more than one variable/agent. A real distributed execution occurs between the computers, although a simulation is taking place inside each computer involving its agents.

Once agents have been assigned and informed to all computer with an ”Init” message, execution begins. Depending on the nature of the solving algorithm, we might need some extra synchronization at this point. In this case, we work with an asynchronous algorithm where all agents can start execution at the same time, so no further synchronization is needed. In other case, a possible solution to start agents in a given order is to use special messages as seen above.

The solving algorithm starts communication using a separate queue for each agent. Execution ends in a computer when all its agents terminate. The statistics calculated in each computer are local. To obtain the total number of messages, the highest non-concurrent constraint check value and the highest elapsed time among all agents, we gather the statistic from all computers in a single machine. Therefore when a computer terminates, it sends a ”Finish” message with all its logical metrics to the master. When the master ends its execution and receives a ”Finish” message from all the client computers, global statistics are calculated. After this, the next problem can be executed.

3.4 Algorithm

In this paper we focus on the BnB-ADOPT+ algorithm. In the case of distributed execution, we replace its logical message structures by ActiveMQ queues, and use the send and receive functions from ActiveMQ.

Since we may have more than one agent in a computer, we need to use the *receiveNoWait()* ActiveMQ function (non-blocking message consumption). This function, unlike the *receive()* function, checks the queue and, if it is empty, it continues the execution and does not wait for a message to arrive. An agent needs to check its queue in this way because otherwise, if the queue is empty, it may prevent other agents occupying the same computer to check their queues. Such behaviour may lead to a deadlock in execution, easily solvable by using the *receiveNoWait()* function which iterates through all the agent’s queues in the computer.

3.5 Instances

As described in section 3.1, we run our experiments in a platform with six individual machines sharing hard disk storage (where the problem instances are stored and seen

```
**procedure** executeDistributedProblem($problem$)
  $Statistics.clear();$
  *//Init connections and clean all queues;*
  **if** $master$ **then**
    *//Create a control topic to broadcast messages to client computers, and create the Init and Finish queues*
    $topic = createTopic("Control");$
    $initQueue = createQueue("Init");$
    $finishQueue = createQueue("Finish");$
    *//Wait for client computers to subscribe to the topic and queues*
    $System.wait(somemilliseconds);$
    $numClients = topic.getSubscribers() + 1;$
    *//Send Start message to all the clients subscribed to the Control topic*
    $topic.sendMessage("Start");$
    *//Receives all the clients ids*
    **do**
      $msg = initQueue.receive();$
      $clientsIds.add(msg.getId());$
    **while** (number of received messages less than $numClients$)
    *//Assigns problem variables to computers and informs with a message*
    $msg = ""; index = 0;$
    **for each** $i$ **in** $problem.variables.size()$
      $msg.concat(clientsId[index], "handles", problem.variable[i])$
      **if** $index == numClients$ **then** $index = 0;$
    $topic.sendMessage(msg);$
    *//Get my variables, we assume one agent per variable*
    $myAgents = getMyVariables(msg);$
    $Statistics.initTime = getCurrentMilliseconds();$
    $startAndProcess(myAgents);$
    $Statistics.finishTime = getCurrentMilliseconds();$
    *//receive finish message from every client with their local statistics before starting with a new problem*
    **do**
      $msg = finishQueue.receive();$
      $updateGlobalStatistics(msg.getTime(), msg.getNCCC(), msg.getTotalMessages());$
    **while** (number of received messages less thann $numClients$)
  **if not** $master$ **then**
    $topic = getTopic("Control")$
    *//Wait for Start message*
    **do**
      $msg = topic.receive();$
    **while** (msg is not received)
    *//Send my client Id*
    $initQueue = getQueue("Init");$
    $initQueue.sendMessage(myId);$
    *//Wait to receive my variables from master, we assume one agent per variable*
    **do**
      $msg = topic.receive();$
      $myAgents = getMyVariables(msg);$
    **while** (msg is not received)
    $Statistics.initTime = getCurrentMilliseconds();$
    $startAndProcess(myAgents);$
    $Statistics.finishTime = getCurrentMilliseconds();$
    $finishQueue = getQueue("Finish");$
    $finishQueue.sendMessage(Statistics.getTime(), Statistics.getNCCC(), Statistics.getTotalMessages());$
  *//Close connections and clean all queues;*

**procedure** startAndProcess($myAgents$)
    **do**
      **for each** agent **in** $myAgents$;
        $hasFinished = agent.ProcessQueue();$
    **while** (agents have not finished execution)
```

Fig. 2. Pseudocode of the communication protocol.

by all the computers in the network) and network connectivity (network connectivity refers to the density of the problem and defines the ratio of existing constraints). We generate 20 instances with 6 variables, domain size 10 and network connectivity 0.5. Constrained variables are selected randomly until the specified network connectivity is reached. Costs are selected randomly from the set 0-200 following a uniform distribution. Problem generation assures connected problems, so all agents are part of the same constraint graph.

In order to explore all three scenarios described in Section 3.3, we use in our experiments:

- 6 computers for a fully distributed execution.
- 2 computers (overloading 3 agents per computer) for a partially distributed execution.
- 1 computer with the classic simulator for a fully simulated execution.

4 Experimental Results

Experimental evaluation appears in Figure 3. Results are given in three dimensions: the number of exchanged messages, the number of NCCCs and the total elapsed time with respect to each problem instance.

First thing to notice in the results is the contrast between the elapsed time graph and the messages and NCCCs graph. While messages and NCCCs show a pretty similar alignment in all three scenarios for each instance, in the elapse time graph we see clearly that simulated and distributed solving show differences in several orders of magnitude.

Surprisingly, even though in the distributed resolution we have six times more computational capacity, the time required to solve the instances distributively is significatly higher than in the simulated case, where we only have a single computer. This is of course the effect of network latency: since we are dealing with message exchange in a real network, communication time is higher than in the simulated case, where sending a message is just a logical operation in the computer CPU. A slowing effect in distributed execution was expected, however we must say that we were surprised by how much communication impacted final results.

In order to better understand this effect, we tried to reduce communication by introducing a reading delay of 200 and 500 milliseconds in each agent. This is, each agent was idle at each iteration during a random period of time (maximum 200 or 500 milliseconds), just before reading the input queue. Introducing this reading delay has the following effect. If agents check their queue constantly, they are likely to change value more often with each new piece of information they receive from the coordination of neighboring agents. For each value change, they inform their neighbors generating more messages (this is the case for BnB-ADOPT$^+$ algorithm, for other solving algorithms this might not be the case and others forms of communication reduction might be tested). If we introduce a small delay before checking the message queue, we encourage the probability that more than one message gets in the agent queue and is processed in each iteration, generating less coordination demands. Results of this second experiment are presented in Figure 4. We can see that messages and NCCCs are little affected by

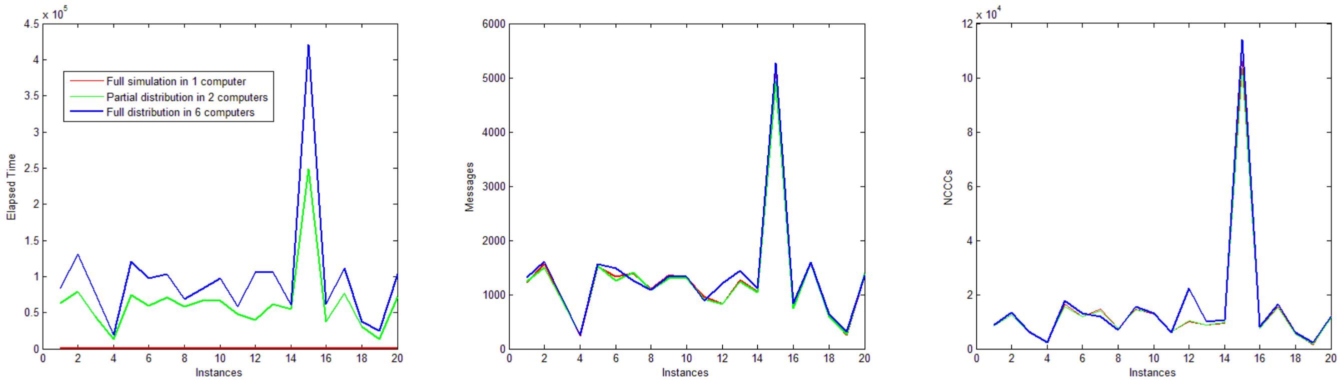

Fig. 3. Results of simulated and distributed execution without delay.

this delay and that the elapsed time increases as it is expected, obtaining therefore the same kind of behavior as in our first experiment.

In the following subsections we discuss results in detail and provide our insight of the observed metrics.

4.1 Number of Messages

The number of messages remains fairly similar between simulation and full/partial distributed execution. Small differences are more remarked in certain instances (instances 5 to 8, instances 12 and 13) where fully distributed execution is more message consuming and where, by introducing reading delays, we were able to soften this effect. This indicates that, for some algorithms a *too eager* response in a real scenario may be counterproductive in terms of the total number of messages [1]. This appreciation has to be carefully balanced though, since sending a message is a logical operation that is an step forward in exploring the search space and delaying execution is normally not a good idea (even if our logical metric decreases) as we can observe in the elapsed time graph.

From this results we conclude that the number of exchanged messages can be seen as a robust measure, somehow correlated with the size of the search space explored, and relatively stable with respect to the degree of distribution of agents, communication time and reading delay. On the other hand, as we know, it is not able to reflect how these features impact in real execution time. [2]

[1] This behavior may be explained in BnB-ADOPT$^+$ as follows: at each iteration it reads all messages until exhausting the input queue; a too eager reading means getting less messages per iteration, which causes in reaction sending a higher number of messages (in this algorithm agents send messages after reading their queue if, as result of incoming messages, they have changed their value or new information about the current solution can be sent to other agents).

[2] BnB-ADOPT$^+$ messages are either constant or linear (at most the number of problem variables) size.

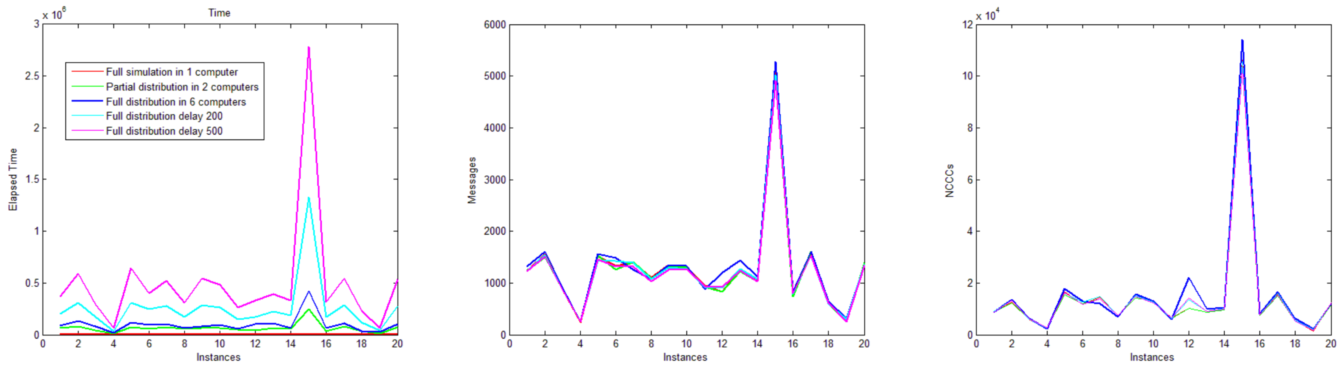

Fig. 4. Results of simulated and distributed executions with reading delays.

4.2 NCCCs

For a more accurate idea of computation effort, NCCCs are generally used. In our experiments the number of NCCCs remains fairly similar between simulation and real execution. Small differences are more remarked in some instances (instances 7 and 12 and 13) but generally we obtain a similar trend.

NCCCs assess the longest sequence of non-concurrent constraint checks performed in distributed constraint solving. Naturally, the execution flow in our simulated and distributed executions is not exactly the same. Small differences in messages reception, caused by time synchronizations leading to different message orderings, might cause variations in the agents coordination flow. However, we observe that within these variations, optimal solutions are founded with approximately the same effort in each execution (for example instance 4 is shown to be easy while instance 15 is shown to be particularly hard in all executions, etc). The NCCCs metric has proved to reflect robustly a measure of this effort, although it is not able to differentiate between a full distribution of agents and a partial distribution, which can be, as we have seen, orders of magnitude slower in elapsed time. This fact is relevant specially for partially distributed algorithms or for settings where one agents handles more than one variable, whether the cost of network communication is measured in the same way as the cost of local communication or it is not considered at all.

4.3 Elapsed Time

Elapsed time is the time measured since DCOP resolution starts until the latest agent terminates. In simulation, generally, either message communication is assumed practically instantaneous (fast network), or it costs a certain amount of NCCCs (slow networks).

Experiments revealed that communication time was quite large. A modern computer with a clock speed of a few GHz and a reasonable communication network, sending a message [3] is at least 7 orders of magnitude more costly than performing a machine lan-

[3] Typical times of sending a message with current technology are: from 10^5 nanoseconds (local area networks without routers) to 10^7 nanoseconds (communication networks with routers).

guage operation, and from 3 to 5 orders of magnitude more costly than a table lookup in main memory (without using any cache). This is the basic operation for a constraint check (assuming constraints as tables). Differences are so large that, to assess elapsed time, communication time has to be necessarily taken into account, either adding a certain number to NCCCs (depending if communication is fast or slow), or the corresponding times if we are measuring simulated time.

In simulation, if only NCCCs are used to approximate elapsed time, communication effort is not fairly reflected and its usage to approximate elapsed time can be misleading. The presented results show that communication time is significantly much larger that we generally see in the literature, with a strong impact in the final execution time even when disposing of a high computational capacity.

5 Conclusions

This experiment stresses the importance of communication time when DCOPs are solved in a real scenario. To assess elapsed time, communication time has to be explicitly taken into account, and it is orders of magnitude higher than the figures that have been used in the literature to assess it in terms of equivalent NCCCs. It is also orders of magnitude higher than any reasonable computational effort done by an algorithm during one iteration in a modern computer.

As future works, it remains to extend this experiment with a larger set of computers, bigger size instances and other solving DCOP algorithms. Even though we are aware of the limitations of these results and the fact that we have used a particular hardware and a specific communication software, we believe that this work is an step forward trying to close the gap between simulation and real execution. Its results may orientate DCOP research, specially regarding those aspects based on empirical evaluations.

References

1. C. Bessiere, I. Brito, P. Gutierrez, and P. Meseguer. Global constraints in distributed constraint satisfaction and optimization. *The Computer Journal*, 2014. In press.
2. A. Chechetka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)l*, pages 1427–1429, 2006.
3. A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward bounding for distributed COPs. *Journal of Artificial Intelligence Research*, 34:61–88, 2009.
4. P. Gutierrez, J. Lee, K. Lei, T. Mak, and P. Meseguer. Maintaining soft arc consistency in BnB-ADOPT+ during search. In *International Conference on Principles and Practice of Constraint Programming (CP 2013)*, LNCS 8124, pages 365–380, 2013.
5. M. Jain, M. Taylor, M. Tambe, and M. Yokoo. DCOPs meet the realworld: Exploring unknown reward matrices with applications to mobile sensor networks. In *International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 181–186, 2009.
6. R. Junges and A. L. C. Bazzan. Evaluating the performance of DCOP algorithms in a real world dynamic problem. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 599–606, 2008.

7. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978.
8. A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Distributed Constraint Reasoning (DCR) workshop in AAMAS 2002*, pages 86–93, 2002.
9. S. Miller, S. D. Ramchurn, and A. Rogers. Optimal decentralised dispatch of embedded generation in the smart grid. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, pages 281–288, 2012.
10. P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.
11. A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 266–271, 2005.
12. A. Petcu and B. Faltings. Distributed constraint optimization applications in power networks. *International Journal of Innovations in Energy Systems and Power*, 3, 2008.
13. E. Sultanik, R. Lass, and W. Regli. Dcopolis: A framework for simulating and deploying distributed constraint optimization algorithms. In *Distributed Constraint Reasoning (DCR) workshop in CP 2007*, 2007.
14. W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.
15. M. Yokoo, E. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.
16. R. Zivan and A. Meisels. Message delay and DisCSP search algorithms. *Ann Math Artif Intell*, 46:415–439, 2006.