The operation point units of distributed constraint solvers

Marius C. Silaghi[1], Robert N. Lass[2], Evan A. Sultanik[2], William C. Regli[2], Toshihiro Matsui[3] and Makoto Yokoo[4]

[1]Florida Institude of Technology, [2]Drexel University, [3]Nagoya Institute of Technology, [4]Kyushu University

Abstract. We propose a way to define *the logic computation cost of operations* to be used in evaluations of scalability and efficiency for simulated distributed constraint reasoning (DCR) algorithms. We also report experiments showing that the cost associated with a constraint check, even within the same algorithm, depends on the problem size. The DCR research has seen heated debate regarding the *correct* way to evaluate efficiency of simulated algorithms. DCR has to accommodate two established practices coming from very different fields: distributed computing and constraint reasoning. The efficiency of distributed algorithms is typically evaluated in terms of the network load and overall computation time, while many (synchronous) algorithms are evaluated in terms of the number of rounds that they require. Constraint reasoning evaluates efficiency in terms of *constraint checks* and *visited search-tree nodes*.
We argue that an algorithm has to be evaluated from the point of view of specific *operating points*, namely of possible or targeted application scenarios. We then show how to report efficiency for a given operating point based on simulation, in particular we show how to tune the distribution used to generate the message latency costs as function of the logic computation unit. New real and simulated experiments show that the cost of a constraint check varies with the size of the problem. We also show how to select logic units for nogood-based algorithms, such that the unit is constant with the size of the problem,

1 Introduction

This article addresses the evaluation of distributed constraint reasoning algorithms. One of the major achievements of computer science consists of the development of the complexity theory for evaluating and comparing the scalability and efficiency of algorithms [7]. Complexity theory proposes to evaluate an algorithm in terms of the number of times it performs *the most inner loop* (aka *the most expensive operation*). This number is seen as a function of the size of the problem. While such metrics do not reveal how much actual time is required for a certain instance, they allow for interpolating how the technique *scales* with larger problems. The assumption that computation speed doubles each few years makes a polynomial factor in the cost irrelevant from a long perspective [7, 3].

Identifying the most inner loop operation is not always as trivial as for centralized sorting and graph traversal. Constraint reasoning researchers have long used either the *constraint check* or the *visited search-tree node* as the most basic operation in classical algorithms. In algorithms whose structure does not present a *most inner loop*, a scalable efficiency evaluation is usually based on the operation that seems to be the most often used and that is relatively expensive. In general, a *basic operation* can prove irrelevant for a competing algorithm who uses extensively another operation. For CSP algorithms, the constraint check is almost ubiquitous, and is typically part of the most inner loop. Here we analyze the distributed constraint reasoning algorithm, ADOPT, and the compliance of the selected computational units with standard evaluation assumptions.

Evaluating distributed computing The main reasons for which distributed constraint reasoning evaluation differs from CSP evaluation are:

- The event-driven design of distributed solvers makes it difficult to detect the inner loops, and these loops often consist only of validating incoming messages and local data rather than in constraint checks.
- The relative ratio between cost (latency) of messages varies by 4-6 orders of magnitude between multi-processors and remote Internet connections.
- While the cost of a local computation can be expected to reduce over years, the cost (latency) of a message between two points is not expected to decrease significantly (in contrast with the other computation costs), since the limits of the current technology are already dictated by the time it takes light to traverse that distance over optical cable.
  Indeed, the minimal time it can theoretically take a message to travel between two diametrically opposed points on the Earth is:

$$\frac{\pi * R_{Earth}}{speed_{light}} = \frac{3.14 * 6.378 * 10^6 m}{3 * 10^8 m/s} \approx 67ms.$$

  Since the optical cables do not travel on a perfect circle around the Earth, it is reasonable to not expect significant improvements beyond the current some 150ms latency for such distances.
- Improvements in the future can only increase bandwidth, which at best may result in removing congestion and obtaining constant latency, at its minimal value computed above.

For a realistic understanding of the behavior of distributed algorithms, some experiments are performed using agents placed on different computers on Internet, typically on a LAN [25, 8, 19, 12] (and we report such experiments in this paper). However, results obtained with LANs may not be valid for any other network topology, or for remote agents on Internet. Also, such results cannot be replicated and verified by other researchers, and therefore results using deterministic network simulators are also commonly requested.

In the following we provide the formal definition for the Distributed Constraint Optimization (DCOP) problem. Then we introduce a simple framework

for unifying various versions of *logic time* systems. We show that the new framework models well the different efficiency metrics and methodologies used so far to evaluate DCOP algorithms based on logic time. These previous methodologies are presented in the unifying framework. We then introduce our new procedure for evaluating scalability and efficiency, and show how it improves on compliance with standard assumptions and evaluation goals.

2 Framework

Distributed Constraint Optimization (DCOP) is a formalism that can model naturally distributed problems. These are problems where agents try to find assignments to a set of variables that are subject to constraints. Several applications are addressed in the literature, such as multi-agent scheduling problems, oil distribution problems, auctions, or distributed control of red lights in a city [14, 22, 17].

Definition 1 (DCOP). *A distributed constraint optimization problem (DCOP), is defined by a set A of agents $A_1, A_2, ..., A_n$, a set X of variables, $x_1, x_2, ..., x_n$, and a set of functions (aka constraints) $f_1, f_2, ...f_i, ..., f_m$, $f_i : X_i \rightarrow \mathbb{R}_+$, $X_i \subseteq X$, where only some agent A_j knows f_i.*

The problem is to find $argmin_x \sum_{i=1}^{m} f_i(x_{|X_i})$. We assume that x_i can only take values from a domain $D_i = \{1, ..., d\}$.

The DCOPs where the functions f_i are defined as $f_i : X_i \rightarrow \{0, \infty\}$, are called Distributed Constraint Satisfaction Problems (DisCSPs). Algorithms for the general DCOP framework can address any DisCSP and specialized algorithms for DisCSPs can often be extended to DCOPs.

3 Evaluation for MIMD

Some of the early works on distributed constraint reasoning were driven by the need to speed up computations on multiprocessors, in particular (multiple instruction multiple data) MIMD architectures [25, 2, 9], sometimes even with a centralized command [2]. However, their authors pointed out that those techniques can be applied straightforwardly for applications where agents are distributed on Internet.

Among the earliest experimental research on DCR we mention [25] by Zhang and Mackworth in 1991. The metric proposed by them is based on Lamport's logic clocks described in the Definition 6.1 and in the Algorithm 18 in [25].

Logic clocks and logic time An event e_1 at agent A_1 is said to *causally precede* an event e_2 at agent A_2 if, had all agents attached all events that they knew to each existing message, A_2 would know about e_1 at the moment when e_2 takes place. Leslie Lamport proposes in [11] a way, called *logic clocks* to construct a tag, called *logic time* (LT), for each event and concurrent message in a distributed

% R_L is the number series generator from which message latencies are extracted using function $next()$
% $E = \{e_1, ..., e_k\}$ is a vector of k local events
% $T = \{t_1, ..., t_k\}$ is a vector of (logic) costs for events E
when *event* e_j *happens* **do**
  $LT_i = LT_i + t_j$;
when *message* m *is sent* **do**
  $LT(m) = LT_i + next(R_L))$;
when *message* m *is received* **do**
  $LT_i = max(LT_i, LT(m))$;

Algorithm 1: Lamport's logic time maintenance for participant P_i. Use of parameters $LT\langle R_L, E, T\rangle$ unifies previous versions for usage with DisCSPs found in (Zhang& Mackworth 1991; Yokoo, Durfee, Ishida& Kuwabara 1992; Silaghi, Haroud& Faltings 2000; Meisels, Kaplansky, Razgon& Zivan 2002; Silaghi& Faltings 2004; Chechetka& Sycara 2006).

computation such that whenever an event e_1 causally precedes e_2 then the logic time of e_1 should be smaller than the logic time of e_2. If $LT(e)$ denotes the logic time of an event e, then we can write $LT(e_1) < LT(e_2)$. Otherwise, the logic time does not reflect the real time and some messages with smaller logic time may actually occur after concurrent messages with bigger logic time. Each process P_i maintains its own logic clock with logic time (LT_i) initially set to zero. Whenever P_i sends a message m, it attaches to m a tag, denoted $LT(m)$, set to the value of LT_i at that moment. The process P_i increments LT_i by the logic duration, t_e, of each local event (computation) e. Assume P_i receives a new message m_k from a process P_j. P_i has to make sure that the logic time LT_i of its future local events is higher than the LT_j of the past events at P_j. This is done by setting $LT_i = max(LT_i, LT(m_k) + L)$, where L is a logic time (duration) assigned to each message passing. We give in Algorithm 1 the procedures proposed in [11], tailored to unify the different metrics used for DCOPs. Certain authors use random values for the logic time of a message [6] and therefore we allow this in our framework by specifying a number series generator (NSG) R_L from which each message logic time (logic latency) is extracted with a function $next()$. A *logic time system* we will use here is therefore parametrized as $LT\langle R_L, E, T\rangle$ where E is a vector of types of local events and T a vector of costs, one for each type of event. For measurements assuming a constant latency of messages set to a value L, the R_L parameter used consists of that particular number, $\mathbf{L}$, (written in bold face).

An experiment may simultaneously use several logic time systems, $LT^1\langle R_L^1, E^1, T^1\rangle, ..., LT^N\langle R_L^K, E^K, T^K\rangle$. Each process P_i maintains a separate logic clock, with times LT_i^u, for each $LT^u\langle R_L^u, E^u, T^u\rangle$. Also, to each message m one will attach a separate tag $LT^u(m)$ for each maintained logic time system $LT^u\langle R_L^u, E^u, T^u\rangle$. This is done in order to simultaneously evaluate a given algo-

rithm and set of problems for several different scenarios (MIMD, LAN, remote Internet).

A common metric used to evaluate simulations of DCR algorithms is given by the *logic time to stability* of a computation. The logic time to stability is given by either:

- the highest logic time of an event occurring before *quiescence* is reached [25];
- the logic time tagging the message that makes the solution known to whoever is supposed to be informed about it [19].

Quiescence of an algorithm execution is the state where no agent performs any computation related to that algorithm and no message generated by the algorithm is traveling between agents.

NB	coordinate axis (Oy)	ordinates axis (Ox)	example usage
LTS1	(logic) time to stability (latency=0)	log – ring size –	[25]
LTS2	speedup – size 800 –	number of processors	[25]
TSL	number of time steps (aka ENCCCs)	message delay (time steps)	[23]
ECL	(equivalent) checks (aka ENCCCs)	checks/message (w. lat. 0)	[20]
NCT	NCCCs (ENCCCs latency=0)	(constraint) tightness	[13]
ECT	ENCCCs (at fix checks/message)	(constraint) tightness	[1]
ST	seconds	constraint tightness	[8]
CT	#checks	constraint tightness	[8]
MT	#messages	constraint tightness	[8]
CBR	checks	constraint tightness	[4]

Table 1. Summary of the systems of coordinates used for comparing efficiency of distributed constraint reasoning.

Uses of logic time for multiprocessors The operation environment targeted by Zhang and Mackworth [25] consists of a network of transputers. The metric employed in [25] with simulations for a constraint networks with ring topology is based on the logic time system $LT\langle \mathbf{1}, \{semijoin\}, \{1\}\rangle$, where the number series generator **1** outputs the value 1 at each call to $next()$. Note that the single local event associated in [25] with a cost is the *semijoin*, which is due to the fact that the algorithms being tested there were not based on constraint checks but on semijoin operators (which consist of composing constraints and then projecting the result on a subset of the involved variables). Graph axes used in [25] depict *logic time to stability* vs *problem size as (log scale) number of variables*, and *logic time* vs. *number of processors (aka agents) at a given size of the DisCSP* distributed to those agents (see Entries LTS1 and LTS2 in Table 1).

A theoretical analysis of the time complexity of a DisCSP solver is presented by Collin, Dechter & Katz in 1991 [2]. Logic time analysis is presented there under the name *parallel time*, targeting MIMD multiprocessors, where each value

change (aka *visited search-tree node* in regular CSP solvers) has cost 1. Note that the obtained metric is $LT\langle \mathbf{0}, \{value\text{-}change\}, \{1\}\rangle$, where message passing is considered instantaneous. A sequential version of the same algorithm is also evaluated in [2] using the logic time $LT\langle \mathbf{0}, \{value\text{-}change, privilege\text{-}passing\}, \{1, 1\}\rangle$. The term coined by Kasif in 1990 [9] for a similar theoretical analysis of the time complexity in parallel computations is *sequential time*.

4 Evaluation for applications targeting Internet

Distributed constraint reasoning algorithms targeting the Internet had to account for the possibly high cost of message passing between agents on remote computers. The latency of message passing in this context is a function on the distance and available connections between the locations. As mentioned above, the theoretical lower bound on this latency can be 67ms, eight orders of magnitude larger than a basic operation on a computer (of the order of 1ns).

Network Simulators While some experiments use agents placed on distinct computers on a LAN, such experiments can somewhat shew the results since:

- agents are geographically closer to each others than in Internet applications, and therefore the latency of messages can be 2-3 orders of magnitude smaller (1-2 ms instead of 100-200ms) [5].
- due to the shared medium used by the typical Ethernet implementation of LANs, the bandwidth is shared and communication between a pair of agents slows down communication between any other pair of agents.

These two issues act in different directions and it is not clear in which actual direction are the results skewed. This makes another argument toward evaluating performance on a simulated network. It is worth noting that early research, such as [25] perform experiments both with simulators and with actual execution on multiprocessors (and we also provide here both simulation and LAN results).

Metrics for Internet One of the first algorithms targeting Internet is the Asynchronous Backtracking solver in [23]. That work experimented with a set of different logic times, $LT^1, ..., LT^{25}$, where LT^i is defined by the parameters

$$LT^i\langle \mathbf{i}, \{constraint\text{-}check\}, \{1\}\rangle, \forall i \in [1, 25] \tag{1}$$

[23] reports the importance of the message latency in deciding which algorithm is good for which task. Note that a curve in the obtained type of graph (see Entry TSL in Table 1) reports several metrics, but for a single problem size/type.

The *time steps* introduced in [23] correspond to the cost of a constraint check. A similar results graph is used in [20] having as axes checks vs checks/message, i.e., the logic time cost for one message latency when the unit is the duration of a constraint check (see Entry ECL in Table 1). This last graph also reports logic time for the latency $L = 0$

$$LT^0\langle \mathbf{0}, \{constraint\text{-}check\}, \{1\}\rangle, \tag{2}$$

which corresponds to simulation of execution with agents placed on the processors of a MIMD with very efficient (instantaneous) message passing (similar to [2], but using the constraint check as the logic unit).

Cycles/SMs After the Yokoo et.al's work in 1992, most DCOP research focused on agents placed on remote computers with problem distribution motivated by privacy [24]. Due to the small ratio between the cost of a constraint check and the cost of one message latency in Internet, the standard evaluation model selected in many subsequent publications completely dropped the accounting of constraint checks. A common assumption adopted for evaluation is that local computations can be made arbitrarily fast (local problems are assumed small and an agent can make his computation on arbitrarily fast supercomputers). Instead, message latency between agents is a cost that cannot be circumvented in environments distributed due to privacy constraints. The metric in [24] is:

$$cycles\ (aka.\ sequential\ messages) = LT\langle \mathbf{1}, \emptyset, \emptyset \rangle$$

The original name for this metric is *cycles*, based on the next theorem (known among some researchers but not written down in this context).

Theorem 1. *In a network system where all messages have the same constant latency L and local computations are instantaneous, all local processing is done synchronously only at time points kL (in all agents).*

Proof. One assumes that all agents start the algorithm simultaneously at time L, being announced by a broadcast message, which reaches all agents at exactly time L (due to the constant time latency). Each agent performs computations only either at the beginning, or as a result of receiving a message.

Since each computation is instantaneous, any message generated by that computation is sent only at the exact time when the message triggering that computation was received. It can be noted that (**induction base**) any message sent as a result of the computation at the start will be received at time $2L$, since it takes messages L logic time units after the start to reach the target.

Induction step: All the messages that leave agents at time kL, will reach their destination at exactly time $(k+1)L$ (due to the constant latency L). Therefore the observation is proven by induction.

As a consequence of this observation, any network simulation respecting these assumptions (that local computations are instantaneous and that message latencies are constant) can be performed employing a loop, where at each cycle each agent handles all the messages sent to it at the previous cycle. As such, $LT\langle \mathbf{1}, \emptyset, \emptyset \rangle$ is given by the total number of cycles of this simulator.

NCCCs and ENCCCs Researchers voiced concerns[1] about the lack of accounting for local computation in SMs. A subsequent re-introduction of logic time in the

[1] At the CP 2001 conference.

form of the metric in Equation 2 is made in [13], proposing to build graphs with axes labeled *NCCCs (non-concurrent constraint checks)* versus problem type (Entry NCT in Table 1). Cost of messages in NCCCs is typically restricted to only 0, reporting solely constraint checks, as in [2].

However, the importance of the latency of messages has been rediscovered recently and logic time cost for message latency is reintroduced in [1] under the name Equivalent *Non-Concurrent Constraint Checks (ENCCCs)*. ENCCCs is a new name for the metric in Equation 1. Current ENCCCs usage in graphs typically differs from earlier usage of the metric by being depicted versus *constraint tightness* or versus *density of constraint-graph* (with a label specifying the value of the logic latency L, i.e. the number of checks/message-latency). Each graph depicts the behavior of several problem types for one message latency, rather than the behavior of one problem type for several message latencies (Entry ECT in Table 1).

Evaluations not related with the logic time Three other important metrics (not based on logic time) for evaluating DCOPs algorithm were introduced in [8] in conjunction with a DisCSP solver.

- the total running time in seconds (Entry ST of Table 1);
- the total number of constraint checks for solving a DisCSP (or DCOP) with a simulator (see Entry CT of Table 1), and
- the total number of exchanged messages (Entry MT of Table 1).

Cycle-based runtime (CBR) gives the ENCCCs on a modified version of the algorithm, which adds synchronizations before sending each message [4].

5 A new methodology

Next we describe a new methodology for evaluating DCOP algorithms that we decided to employ recently [21], but which has not yet been introduced in sufficient detail.

Let us first mention the weaknesses in currently common methodologies, and which we want to fix with our new proposed approach:

- the weakness of the *cycles/sequential-messages* metric is that its assumptions do not apply to DCOP solvers with extensive local processing at each message (such as in the recent DPOP algorithm [18]). DPOP has very few messages and very expensive local computation at each message.
- NCCCs (in the version with message cost zero) do not take into account message latencies, which are an important cost for many typical DCOP algorithms. Moreover, (see the Experiments section) the cost of a constraint check grows linearly with the problem size (for the same algorithm), causing misleading curves.
- ENCCCs require depicting many graphs, one for each checks/latency ratio, and still does not help to know which ratio is relevant to a given application.

This is because the cost that has to be associated with a constraint check depends on many factors, being a function of the algorithm, of the programming language, and (as we report here) even function of the problem size. Plots of different algorithms on the same ENCCCs graph are not comparable since their units often have different meaning and relevance (and may not even be bounded by a polynomial relation).
- time in seconds of experiments on a LAN, besides the fact that its measuring requires important hardware resources, it does not apply to remote Internet applications, or to other hardware, and cannot be replicated.

Our proposal is, given any well defined application scenario, to start by first computing the expected latency/checks ratio, following the next procedure.

Proposed Evaluation Method Congestion can lead to variable latency, varying according to a distribution where a lower bound on latency is given by catalog values [16]. Various such distributions can be designed and used directly in the experiments. For simplicity, the following description assumes a future where bandwidth improvements will remove congestion and therefore where the latency will be constant.

1. Retrieve the typical latency L_s in seconds for messages in the type of network of the targeted application. Such information is found in technical catalogs, encyclopedias, and technical articles. For example, some typical message latencies for remote machines on Internet are found in [16].
2. Compute the total execution time in seconds, t_p, for solving each complete test set of problems at size p using the simulator. Note that this is machine and programming language dependent, and therefore the used machine and programming language have to be specified.
3. Select a *computation unit CU* (e.g., constraint check, CC). Compute the total number of computation units, $\#CU_p$, at each problem size p [8].
4. Compute the cost in seconds that should be associated with a computation unit by computing the ratio $t_p/\#CU_p$.
5. We note that for a given machine and programming language this ratio, $t_p/\#CU_p$, may depend on the problem size p, varying as much as an order of magnitude. For example, our C simulator for ADOPT on the problems in [15] uses between 3 to 28 microseconds per constraint check on a Linux PC at 700MHz. The smaller value was found at problems with 8 agents and 8 variables and the larger one at problems with 40 agents and variables. We discuss later our explanation for this phenomenon. If this happens we recommend the selection of a different CU (as shown later) and return at the Step 3 until the ratio is practically constant for different p.
6. Compute *the Operating Point* (i.e., ratio message-latency/computation-unit) for the given problem size p as $L_p = L_s * \#CU_p/t_p$.
7. Compute the graph in the Operating Point.

As it follows from the aforementioned weaknesses, the main problem with reporting ENCCCs is that we can find out neither where is a particular latency/check ratio relevant, nor which latency/check ratio is relevant for a given

$$LT^i\langle \mathbf{i}, \{constraint\text{-}check, nogood\text{-}inference, nogood\text{-}validity, nogood\text{-}applicability\}, \{1,3,2,2\}\rangle, \forall i > 0 \quad (3)$$

application. Our proposal solves this problem by offering a little bit of additional information besides ENCCCs graphs. To compute the graph based on *Equivalent Non-concurrent Computation Units* (ENCCUs) in the Operation Point (ENCCU-OPs) we identify the following alternatives:

- the ENCCUs/ENCCCs graph with the logic time cost given by the targeted/average value of L_p as interpolated from values for the different problem types p, or
- the ENCCUs graph with the value of the logic message latency L as the i that is closest to the targeted values of L_p, among the different values of i used for the logic times schemes LT^i evaluated in experiments (see Equation 1).

The term *operating point* comes from graphs depicting behavior of transistors. The operating point is the area of these graphs that is of real interest for an application.

The advantage of our method is that it can be performed using only a simulator, its results are reproducible, and can be applied to difficult to evaluate experimentally settings, such as remote Internet connections.

EML As an extension of SMs, one can also draw graphs representing the Equivalent Message Latencies in the Operating Point (EML-OP) from the ENCCC-OP graph, where each ordinate is divided by the latency/computation-unit ratio L of the graph. The axis of ordinates shows the number of (equivalent) message delays. This graph has the advantage that the the ordinate has an easy to understand meaning, namely the message latency in the targeted destination, which is readily available. EMLs can also be plotted against abscissae showing different ratio latency/checks, to illustrate better how algorithms behave in areas neighboring the operating point.

Yet an additional metric can be obtained measuring the *logic simulated seconds*, where each event is measured in the number of (micro)seconds it lasts (in average) as observed during experimentation. This has the advantage over actual seconds that they can be replicated and verified by other researchers.

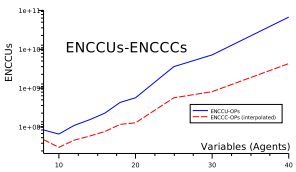

Fig. 1. ADOPT performance: operating point ENCCUs.

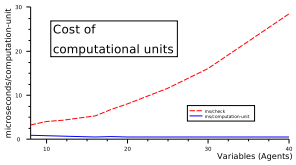

Fig. 2. The running time associated with a computation-unit for two metrics: checks and CUs.

Accounting for nogood validation Certain DCOP algorithms are not based on checking constraints repeatedly, but rather they compile information about constraints into new entities called *nogoods*. Afterward, these techniques work by performing inferences on such nogoods. Nogoods are a kind of constraints themselves. In such algorithms it makes sense to attribute costs to the different important operations on nogoods such as *nogood inference*, *nogood validity check*, and *nogood applicability check*. The new method (computation-unit) for computing logic times at various message latencies is the Equation 3, where the coefficients of different nogood handling operations are selected based on a perceived complexity for those operations. The nogood inference operation is typically the most complex of these operations as it accesses two nogoods to create a third one (suggesting a logical cost of 3). Nogood-validity and nogood-applicability both typically involved the analysis of a nogood and of other data, local assignments and remote assignments, to be compared with the nogood (hence a logical cost of 2). These costs do not typically have an exact value since the sizes of nogoods vary within the same problem. A constraint check for binary constraints is cheaper than the verification of an average-sized nogood, and is given the logical cost of 1.

Our experiments reported here confirm that computation units selected according to Equation 3 are closer to constant, with slightly higher cost per computation unit at small problem size (Figure 2). The slightly higher cost at small problem size is likely due to the overhead of creating and initializing data structures at the beginning of the execution, and which is evened out at problems larger than 10 variables. It may be fixed in the future by adding an event accounting for the creation of such data structures.

Why cost of checks varies with the problem size An interesting question raised by our experimental results is: Why do experiments reported here show that the cost associated with a constraint check varies with the size of the problem?

The cost associated with a constraint check (as measured above) consist of an aggregation of the costs of all other operations executed by DCOP algorithm in preparation of the constraint check and in processing the results of the constraint check. Typically there are several data structures to maintain and certain information to validate, and these data structures may be larger with large problem sizes than with small problem sizes. The variation may also come

p (agents)	8	10	12	14	16	18	20	25	30	40
t_p (total seconds)	0.1404	0.1528	0.3012	0.5516	1.0068	2.5708	4.1176	47.7112	174.06	3767.38
$\#CC_p$ total checks	43887.8	38279.3	70279.4	116080	191501	381415	516835	$4.1*10^6$	$10.9*10^6$	$132*10^6$
$\frac{microseconds(t_p)}{check(\#CC_p)}$	**3.199**	**3.992**	**4.286**	**4.752**	**5.257**	**6.74**	**7.967**	**11.47**	**15.98**	**28.4**
$L_p = (\frac{checks}{latency(200ms)})$	62518.3	50103.8	46666.3	42088.5	38041.6	29672.9	25103.7	17437.3	12519	7041.1
(10^6) ENCCC L=10^4	7,94	6,32	10,5	14,8	21,6	41,8	54,1	343	694	6594
(10^6) ENCCC L=10^5	79	63	105	148	216	417	541	3429	6939	65880
simulated time (s)	142	113	188	266	388	751	974	6175	12500	118759
$\frac{microseconds(t_p)}{comp-unit(\#CU_p)}$	**0.81**	**0.743**	**0.636**	**0.54**	**0.487**	**0.493**	**0.475**	**0.476**	**0.439**	**0.442**

Table 2. Sample re-evaluation of ADOPT with our method. Columns represent problem size.

from approximations in the way in which the cost of a constraint check is evaluated in comparison to operations for handling other data structure (such as nogoods [24]).

In certain situations, algorithms change their relative behavior in situations that are close to the operating point. Then precise measurements are important, and it makes sense to try to tune the logic time associated with each operation, in order to reduce the variation of the meaning of a unit of logic time with the problem size. One can approach this problem by trying many different combinations, or trying a hill climbing approach that tunes successively each of the parameters. One has to run complete sets of experiments for each of these possible costs (which is computationally expensive). A valuable future research direction consists in finding an efficient way of tuning these parameters.

However, a currently simpler alternative is to report efficiency in *simulated seconds* [21, 12], where each significant event is given a logic cost equal with the average time in microseconds as obtained from experiments.

6 Experiments

We will describe here how we conduct experiments with ADOPT [15], as an example of how our evaluation method can be applied to other algorithms. The illustration is based on a sample of Teamcore random graph coloring problems with 10 different sizes, ranging between 8 agents and 40 agents, with graph density 30%. The results are averaged over 25 problems of each size [15]. The targeted application scenario consists of remote computers on Internet.

Following the steps of our method we report the following:

1. The catalog message latency for our scenario is 200ms, varying between 150ms and 250ms (see [16, 10]).
2. Simulated ADOPT with randomized latencies is implemented in C++ and runs on a the 700MHz node of a Beowulf (Linux Red Hat). The total time in seconds is given in the second row of Table 2.
3. The total number of constraint checks $\#CC_p$ for each problem size is given in the third row of Table 2.

4. The cost in (micro)seconds associated with each constraint check is computed as $t_p/\#CC_p$. It is given in the fourth row of Table 2.
5. The ratio is not constant. We repeat the analysis with the CU in Equation 3.
6. The message-latency/constraint-check ratio (L_p) is computed by dividing the average latency found at Step 1 (200ms) by the items in the 4^{th} row. The results are given in the 5^th row of Table 2.
7. The operating point is defined by the fourth and fifth rows. The last step consists of reporting the results for this operating point (here we will use a Table rather than a graph, to make the processing more visible). We performed the experiments using several logic time systems, the available ones that are the closest to the obtained operating point are $L = 100,000$ and $L = 10,000$. It is now possible to re-run the experiments with all the L_p values found in our table. Here we will just report the results of the closest L, which is 10,000 for most problem sizes (one also can use $L = 100000$ for problems with 8 and 10 agents), see the 6^{th} and 7^{th} rows of Table 2. One can also interpolate the time between the predictions based on $L = 10,000$ and $L = 100,000$, function of the predicted L_p at each problem size.
   Next, for example, one can also report the simulated time (in simulated seconds) by multiplying each logic time (in ENCCU-OPs) with the corresponding cost per logic unit (here reported in the third row). We interpolate (linearly) the time between the predictions based on $L = 10,000$ and $L = 100,000$, function of the predicted L_p at each problem size. We report the *simulated time* in the 8^th row of Table 2. This simulated time represents the average actual time (in seconds) that a problem of the corresponding size is expected to need in our *operating point*.

The last row in Table 2 shows the cost of computational units (CU) at different problem sizes when their computation is based on Equation 3. As mentioned earlier, we can observe that this computational unit respects better standard assumptions. At the chosen operation-point L, the choice of the computation unit does not have a strong impact on the ENCCCs measurement which is overwhelmingly influenced bu the number of sequential messages. The equivalent non-concurrent CUs (ENCCU) measure, corresponding to ENCCCs in the previous method, yields almost the same numbers at these L values. The impact of the computational units due to local computation starts to be visible in our ADOPT implementation only at $L < 1000$. This highlights the importance of correctly selecting the operation point.

It is remarkable that that the cost associated with constraints checks varies with the problem size even for the same implementation of the same algorithm. We therefore felt the need to verify this observation on a different implementation, and in particular on a LAN solver. We therefore run a set of experiments using DCOPolis [12]. Here the agents are distributed on five HP-TC4200 tablet PCs with 1.73Ghz Intel Pentium M processors and 512M of RAM connected via Ethernet to a Netgear FS108 switch, isolated from the Internet and running Ubuntu Linux (see Figure 3). These experiments show similar large variability of the checks.

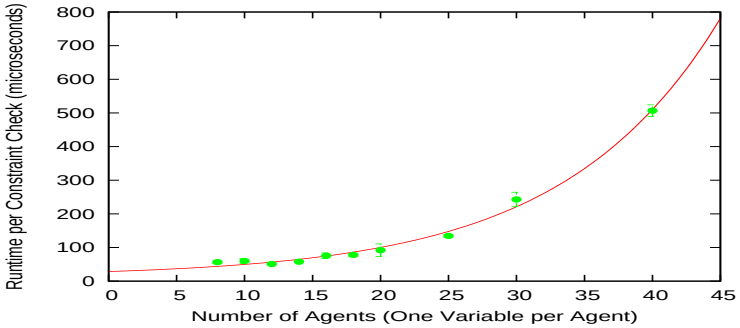

Fig. 3. Results on a LAN with DCOPolis

7 Conclusion

We started introducing a framework unifying the representation of different logic clocks-based metrics used for efficiency evaluation of DCOPs. We identify contradictions between basic assumptions and common evaluation methodologies in the case of ADOPT. We propose a new methodology to analyze DCOPs, extending the one known as Equivalent Non-Concurrent Constraint Checks (ENCCCs). Our extension shows how to select a computation unit that is constant across problem sizes. We also show how to identify the ENCCCs graph that fits a given application scenario (named *operation point*). The obtained metric counts the equivalent non-concurrent computational units in the operation point (ENCCUOPs) and its construction requires the evaluation of several other metrics, such as the total number of constraint-checks (or computation-units) and the total time to run the simulator as a centralized solver. A different computation unit may be appropriate for each family of algorithms. Our method to select computation units that correctly show the efficiency and scalability trends apply easily to other (even centralized) algorithms using nogoods.

We discuss remarkable experimental results showing that cost associated with constraint checks can vary by orders of magnitude with the size of the problem even for the same implementation of the same algorithm, and skewing efficiency graphs. Further we present results on a real network with DCOPolis, confirming our finding. We discuss the possible explanations, their implications, and how the issue can be handled (including open research directions). The impact of the computational units due to local computation starts to be visible in our ADOPT implementation only at $L < 1000$. This highlights the importance of correctly selecting the operation point.

References

1. A. Chechetka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *AAMAS*, 2006.
2. Z. Collin, R. Dechter, and S. Katz. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*, 2000.

3. Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. Mc Graw Hill, 2003.
4. J. Davin and P. J. Modi. Impact of problem centralization in distributed COPs. In *DCR*, 2005.
5. Motion Engineering. Synqnet. Technical report, Motion Engineering Inc, 2003. `www.motioneng.com/pdf/SynqNet_Tech_Whitepaper.pdf`.
6. C. Fernàndez, R. Béjar, B. Krishnamachari, and C. Gomes. Communication and computation in distributed CSP algorithms. In *CP*, pages 664–679, 2002.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H.Freeman&Co, 1979.
8. Y. Hamadi and C. Bessière. Backtracking in distributed constraint networks. In *ECAI'98*, pages 219–223, 1998.
9. S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3):275–286, October 1990.
10. Joseph Kopena, Gurav Nail, Maxim Peysakhov, Evan Sultanik, William Regli, and Moshe Kam. Service-based computing for agents on disruption and delay prone networks. In *AAMAS*, pages 1341–1342, 2004.
11. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
12. Robert N. Lass, Evan A. Sultanik, Pragnesh Jay Modi, and William C. Regli. Evaluation of cbr on live networks. In *DCR Workshop at CP*, 2007.
13. A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *DCR*, 2002.
14. P.J. Modi and M. Veloso. Bumping strategies for the multiagent agreement problem. In *AAMAS*, 2005.
15. Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *AIJ*, 161, 2005.
16. John Neystadt and Nadav Har'El. Israeli internet guide (iguide). http://www.iguide.co.il/isp-sum.htm, 1997.
17. A. Petcu, B. Faltings, and D. C. Parkes. M-dpop: Faithful distributed implementation of efficient social choice problems. *submitted to JAIR*, 2007.
18. Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, 2005.
19. M.-C. Silaghi and B. Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161(1-2):25–53, 2004.
20. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. of AAAI2000*, pages 917–922, Austin, August 2000.
21. M.-C. Silaghi and M. Yokoo. Dynamic dfs tree in adopt-ing. In *Twenty-Second AAAI Conference on Artificial Intelligence (AAAI)*, Vancouver Canada, 2007.
22. Toby Walsh. Traffic light scheduling: a challenging distributed constraint optimization problem. In *DCR*, India, January 2007.
23. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS*, pages 614–621, June 1992.
24. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE TKDE*, 10(5):673–685, 1998.
25. Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, 1991.