Model Streaming for Distributed Multi-Context Systems*

Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{dao,eiter,fink,tkren}@kr.tuwien.ac.at

Abstract. Multi-Context Systems (MCS) are instances of a nonmonotonic formalism for interlinking heterogeneous knowledge bases in a way such that the information flow is in equilibrium. Recently, algorithms for evaluating distributed MCS have been proposed which compute global system models, called equilibria, by local computation and model exchange. Unfortunately, they suffer from a bottleneck that stems from the way models are exchanged, which limits the applicability to situations with small information interfaces. To push MCS to more realistic and practical scenarios, we present a novel algorithm that computes at most $k \geq 1$ models of an MCS using asynchronous communication. Models are wrapped into packages, and contexts in an MCS continuously stream packages to generate at most k models at the root of the system. We have implemented this algorithm in a new solver for distributed MCS, and show promising experimental results.

1 Introduction

During the last decade, there has been an increasing interest, fueled by the rise of the world wide web, in interlinking knowledge bases to obtain comprehensive systems that access, align and integrate distributed information in a modular architecture. Some prominent examples of such formalisms are MWeb [1] and distributed ontologies in different flavors [12], which are based on a uniform format of the knowledge bases.

Nonmonotonic multi-context systems (MCS) [5], instead, are a formalism to interlink heterogeneous knowledge bases, which generalizes the seminal multi-context work of Giunchiglia, Serafini et al. [10, 13]. Knowledge bases, called *contexts*, are associated with belief sets at a high level of abstraction in a logic, which allows to model a range of common knowledge formats and semantics. The information flow between contexts is governed by so called *bridge rules*, which may – like logics in contexts – be nonmonotonic. The semantics of an MCS is given by models (also called *equilibria*) that consist of belief sets, one for each context where the information flow is in equilibrium.

For MCS where the contexts are distributed and accessible by semantic interfaces while the knowledge base is not disclosed (a predominant setting), algorithms to compute the equilibria by local computation and model exchange were given in [6] and [2]. They

* This research has been supported by the Austrian Science Fund (FWF) project P20841 and by the Vienna Science and Technology Fund (WWTF) project ICT 08-020.

incorporate advanced decomposition techniques, but still, these algorithms suffer from a bottleneck that stems from the way in which models are exchanged and that limits the applicability to situations with small information interfaces.

For example, suppose context C_1 accesses information from several other contexts $C_2, \ldots, C_m$, called its neighbors. Consider a simple setting where the information flow is acyclic, meaning that none of the neighbors (directly or indirectly) accesses information from C_1. Furthermore, assume that $n_2, \ldots, n_m$ are the numbers of partial equilibria that exist at the neighbors, respectively. Intuitively, a partial equilibrium at a context is an equilibrium of the subsystem induced by information access. By the current approach for distributed evaluation, all the partial equilibria are returned to the parent context C_1.

Before any local model computation can take place at the parent, it needs to join, i.e., properly combine the partial equilibria obtained from its neighbors. This may result in $n_2 \times n_3 \times \cdots \times n_m$ partial models to be constructed (each one providing a different input for local model computation) which may not only require considerable computation time but also exhaust memory resources. If so, then memory is exhausted before local model computation at C_1 has even been initiated, i.e., before any (partial) equilibrium is obtained.

Note however that, if instead of the above procedure, each neighbor would transfer back a just a portion of its partial equilibria, then the computation at C_1 can avoid such a memory blow up; in general it is indispensable to trade more computation time, due to recomputations, for less memory if eventually *all* partial equilibria at C_1 shall be computed. This is the idea underlying a new *streaming* evaluation method for distributed MCS. It is particularly useful when a user is interested in obtaining just *some* instead of all answers from the system, but also for other realistic scenarios where the current evaluation algorithm does not manage to output under resource constraints in practice any equilibrium at all.

In this paper we pursue the idea sketched above and turn it into a concrete algorithm for computing partial equilibria of a distributed MCS in a streaming fashion. Its main features are briefly summarized as follows:

- the algorithm is *fully distributed*, i.e., instances of its components run at every context and communicate, thus cooperating at the level of peers;
- upon invocation at a context C_i, the algorithm streams, i.e. computes, $k \geq 1$ partial equilibria at C_i at a time; in particular setting $k = 1$ allows for consistency checking of the MCS (sub-)system.
- issuing follow-up invocations one may compute the next k partial equilibria at context C_1 until no further equilibria exist; i.e., this evaluation scheme is complete.
- local buffers can be used for storing and exchanging local models (partial belief states) at contexts, avoiding the space explosion problem.

We have implemented this algorithm yielding a new solver for distributed MCS, and report promising experimental results.

To the best of our knowledge, a similar streaming algorithm has neither been developed for the particular case of computing equilibria of a MCS, nor more generally for computing models of distributed knowledge bases. Thus, our results are not only of interest in the setting of heterogeneous MCS, but they are relevant in general for model

computation and reasoning over distributed (potentially homogeneous) knowledge bases like e.g. distributed SAT instances.

The rest of the paper is organized as follows. Section 2 recalls some background on multi-context systems, while basic details on the current DMCS algorithm(s) for evaluating MCS are summarized in Section 3. The new streaming algorithm is presented in detail in Section 4 including a discussion on parallelization, and Section 5 reports some initial experimental results obtained with a corresponding prototype implementation. Finally, we conclude in Section 6.

2 Multi-Context Systems

We first recall some basic notions of nonmonotonic MCS [5]. For simplicity and in order to get to the essence of the distributed algorithm, we focus here on a subclass of MCS in which knowledge bases consist of propositional clause sets (under different semantics).

A *logic* is a tuple $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$, where

- $\mathbf{KB}_L$ is a set of admissible knowledge bases, each being a finite set of clauses

$$h_1 \vee \cdots \vee h_k \leftarrow b_1 \wedge \cdots \wedge b_n \tag{1}$$

  where all h_i and b_j are literals (i.e., atoms or negated atoms) over a propositional signature Σ_L;
- $\mathbf{BS}_L$ is the set of possible belief sets, where a *belief set* is a satisfiable set of literals; it is *total*, if it is maximal under $\subseteq$; and
- $\mathbf{ACC}_L \colon \mathbf{KB}_L \to 2^{\mathbf{BS}_L}$ assigns each $kb \in \mathbf{KB}_L$ a set of *acceptable belief sets*.

In particular, classical logic results if $\mathbf{ACC}_L(kb)$ are the total belief sets that satisfy kb as usual, and Answer Set Programming (ASP) if all h_i are positive literals and $\mathbf{ACC}_L(kb)$ are the answer sets of kb.

A *multi-context system (MCS)* $M = (C_1, \ldots, C_n)$ consists of *contexts* $C_i = (L_i, kb_i, br_i)$, $1 \leq i \leq n$, where $L_i = (\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i)$ is a logic, $kb_i \in \mathbf{KB}_i$ is a knowledge base, and br_i is a set of L_i*-bridge rules* of the form

$$s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \mathrm{not}\,(c_{j+1} : p_{j+1}), \ldots, \mathrm{not}\,(c_m : p_m) \tag{2}$$

where $1 \leq c_k \leq n$, p_k is an element of some belief set of L_{c_k}, $1 \leq k \leq m$, and $kb \cup \{s\} \in \mathbf{KB}_i$ for each $kb \in \mathbf{KB}_i$.

Informally, bridge rules allow to modify the knowledge base by adding s, depending on the beliefs in other contexts.

Example 1. Let $M = (C_1, \ldots, C_n)$ be an MCS such that for a given integer $m > 0$, we have $n = 2^{m+1} - 1$ contexts, and let $\ell > 0$ be an integer. For $i < 2^m$, a context $C_i = (L_i, kb_i, br_i)$ of M consists of ASP logics L_i,

$$kb_i = \{a_i^1 \vee \cdots \vee a_i^\ell \leftarrow t_i\} \text{ and} \tag{3}$$
$$br_i = \left\{ \begin{array}{llll} t_i \leftarrow (2i : a_{2i}^1) & \cdots & t_i \leftarrow (2i : a_{2i}^\ell) \\ t_i \leftarrow (2i+1 : a_{2i+1}^1) & \cdots & t_i \leftarrow (2i+1 : a_{2i+1}^\ell) \end{array} \right\},$$

and for $i \geq 2^m$, we let C_i have

$$kb_i = \{a_i^1 \vee \cdots \vee a_i^\ell \leftarrow t_i\} \cup \{t_i\} \text{ and } br_i = \emptyset \ . \tag{4}$$

Intuitively, M is a binary tree-shaped MCS with depth m and $\ell + 1$ is the size of the alphabet in each context. Fig. 1a shows such an MCS with $n = 7$ contexts and depth $m = 2$; the internal contexts have knowledge bases and bridge rules as in (3), while the leaf contexts are as in (4). The directed edges show the dependencies of the bridge rules.

The semantics of an MCS M is defined in terms of particular *belief states*, which are tuples $S = (S_1, \ldots, S_n)$ of belief sets $S_i \in \mathbf{BS}_i$. Intuitively, S_i should be a belief set of the knowledge base kb_i; however, also the bridge rules br_i must be respected. To this end, kb_i is augmented with the conclusions of all $r \in br_i$ that are applicable in S. Formally, for any r of form (2) let $head(r) = s$ and $B(r) = \{(c_k : p_k) \mid 1 \leq k \leq m\}$. We call r *applicable in* $S = (S_1, \ldots, S_n)$, if $p_i \in S_{c_i}$, for $1 \leq i \leq j$, and $p_k \notin S_{c_k}$, for $j < k \leq m$; for any set of bridge rules R, let $app(R, S)$ denote the set of all $r \in R$ applicable in S. Then S is an *equilibrium* of M, iff for all $1 \leq i \leq n$, $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$.

Example 2. The multi-context system M from Ex. 1 has equilibria $S = (S_1, \ldots, S_n)$ with $S_i = \{a_i^k, t_i\}$, for $1 \leq k \leq \ell$.

Without loss of generality, we assume the signatures Σ_i of the contexts C_i are pairwise disjoint, and let $\Sigma = \bigcup_i \Sigma_i$.

Partial Equilibria. For a context C_k, the equilibria of the sub-MCS it generates are of natural interest, which are partial equilibria of the global MCS [6].

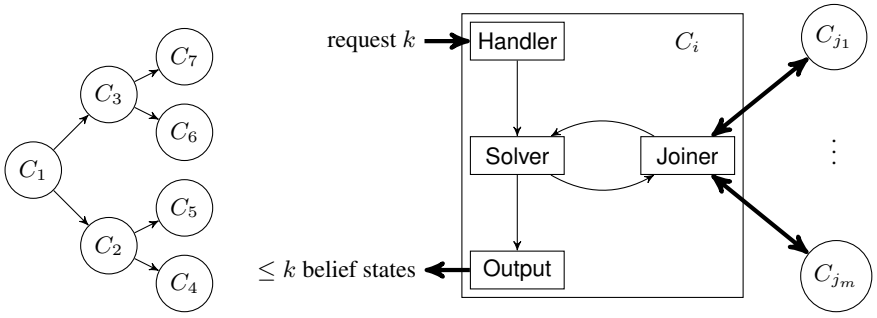
The sub-MCS is generated via recursive belief access. The *import neighborhood of a context* C_k in an MCS $M = (C_1, \ldots, C_n)$ is the set $In(k) = \{c_i \mid (c_i : p_i) \in B(r), r \in br_k\}$, and its *import interface* is $V(k) = \{p \mid (c : p) \in B(r), r \in br_k\}$, Moreover, the *import closure* $IC(k)$ *of* C_k is the smallest set I such that (i) $k \in I$ and (ii) for all $j \in I$, $In(j) \subseteq I$. and its *recursive import interface* is $V^*(k) = V(k) \cup \{p \in V(j) \mid j \in IC(k)\}$.

Based on the import closure, we define partial equilibria. Let $\epsilon \notin \bigcup_{i=1}^n \mathbf{BS}_i$. A *partial belief state* of M is a tuple $S = (S_1, \ldots, S_n)$, such that $S_i \in \mathbf{BS}_i \cup \{\epsilon\}$, for $1 \leq i \leq n$; S is a *partial equilibrium of* M *w.r.t. a context* C_k iff $i \in IC(k)$ implies $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, and if $i \notin IC(k)$, then $S_i = \epsilon$, for all $1 \leq i \leq n$.

Example 3. Continuing with Ex. 1, for $m = 2$ and $\ell = 3$, we get an MCS with seven contexts $M = (C_1, \ldots, C_7)$. A partial equilibrium of M w.r.t. context C_2 is the partial belief state $S = (\epsilon, \{a_2^1, t_2\}, \epsilon, \{a_4^3, t_4\}, \{a_5^2, t_5\}, \epsilon, \epsilon)$.

If one is only interested in consistency, (partial) equilibria of C_k may be projected to $V^*(k)$, hiding all literals outside. In accordance with this only belief sets projected to $V^*(k)$ would need to be considered.

For combining partial belief states $S = (S_1, \ldots, S_n)$ and $T = (T_1, \ldots, T_n)$, their *join* $S \bowtie T$ is given by the partial belief state $(U_1, \ldots, U_n)$, where (i) $U_i = S_i$, if $T_i = \epsilon \vee S_i = T_i$, and (ii) $U_i = T_i$, if $T_i \neq \epsilon \wedge S_i = \epsilon$, for all $1 \leq i \leq n$. Here $S \bowtie T$ is void, if some S_i, T_i are from $\mathbf{BS}_i$ but different.



(a) Binary tree MCS (b) DMCS-STREAMING Architecture

Fig. 1

3 DMCS Algorithms

There are two algorithms for computing (partial) equilibria of Multi-Context systems: (i) DMCS, a basic algorithm that needs explicit cycle-breaking during evaluation [6]; and (ii) DMCSOPT, an algorithm that operates on a pre-processed query plan [2].

Both DMCS and DMCSOPT aim at computing *all* partial equilibria starting from a particular context in an MCS. They run independently at each context in an MCS, and allow to project belief states to a relevant portion of the signature of the import closure. Upon request, they compute local models based on partial equilibria from neighboring contexts. The difference between the algorithms is that DMCS has no knowledge about the dependencies in an MCS and needs to detect cycles during evaluation using a history of visited contexts, whereas DMCSOPT assumes that there exists an acyclic query plan that has been created prior to evaluation-time using a decomposition technique based on biconnected components. Additionally, DMCSOPT includes information in the query plan to project partial equilibria to a minimum during evaluation. On the other hand, DMCS needs to get this information as input; this projection information in form of a set of relevant interface variables cannot be changed while the overall computation proceeds along the dependencies of the input MCS.

The algorithms use lsolve, an algorithm that completes combined partial equilibria from neighboring context with local belief sets. Formally, given a partial equilibrium S, lsolve(S) imports truth values of bridge atoms from neighboring contexts, solves the local theory, and returns all models. In the next section, we show a new algorithm that removes the restriction of DMCS and DMCSOPT to compute *all* partial equilibria to computing up to k partial equilibria, and if at least k partial equilibria exist, this algorithm will compute k of them.

4 Streaming Algorithm for DMCS

Given an MCS M, a starting (root) context C_r, and an integer k, we aim at finding at most k partial equilibria of M w.r.t. C_r in a *distributed* and *streaming* way. While the distributed aspect has been investigated in the DMCS and DMCSOPT algorithms,

adding the streaming aspect is not easy as one needs to take care of the communication between contexts in a nontrivial way. In this section, we present our new algorithm DMCS-STREAMING which allows for gradual streaming of partial equilibria between contexts. Section 4.1 describes a basic version of the algorithm, which concentrates on transferring packages of k equilibria with one return message. The system design is extendable to a parallel version, whose idea is discussed in Section 4.2.

4.1 Basic Streaming Procedure

In DMCSOPT, the reply to a request of a parent contains *all* partial equilibria from one context. This means that communication between contexts is synchronous—one request gets exactly one answer. While this is the easiest way to send solutions, it is very ineffective with larger MCS instances, as a small increase in the size of the alphabet may force the creation of many (partial) equilibria, which in turn may exceed memory limitations. The goal of this work is to develop an algorithm which allows for asynchronous communication for belief state exchange, i.e., one request for a bounded number of k (partial) equilibria may result in *at most* k solutions. This way we can restrict memory needs and evaluate multi-context systems that could not be handled by previous algorithms.

The basic idea is as follows: each pair of neighboring contexts can communicate in multiple rounds, and each request has the effect to receive at most k partial equilibria. Each communication window of k partial equilibria ranges from the k_1-th partial equilibria to the k_2-th ($= k_1 + k - 1$). A parent context C_i requests from a child context C_j a pair (k_1, k_2), and then receives at a future time point a package of at most k partial equilibria. Receiving ϵ indicates that C_j has fewer than k_1 models.

Important subroutines of the new algorithm DMCS-STREAMING take care of receiving the requests from parents, receiving and joining answers from neighbors, local solving and returning results to parents. They are reflected in four components: Handler, Solver, Output, and Joiner (only active in non-leaf contexts); see Fig. 1b for an architectural overview.

All components except Handler communicate using message queues: Joiner has j queues to store partial equilibria from j neighbors, Solver has one queue to hold joined equilibria from Joiner, and Output has a queue to carry results from Solver. As our purpose is to bound space usage, each queue has a limit on the number of entries. When a queue is full (resp., empty), the enqueuing writer (resp., dequeuing reader) is automatically blocked. Furthermore, getting an element also removes it from the queue, which makes room for other partial equilibria to be stored in the queue later. This property frees us from synchronization technicalities.

Algorithms 2 and 3 show how the two main components Solver and Joiner work. They use the following primitives:

- lsolve(S): works as lsolve in DMCSOPT, but in addition may return only one answer at a time and be able to tell whether there are models left.
- *get_first*(ℓ_1, ℓ_2, k): send to each neighbor from c_{ℓ_1} to c_{ℓ_2} a query for the first k partial equilibria, i.e., $k_1 = 1$ and $k_2 = k$; if all neighbors in this range return some models then store them in the respective queues and return $true$; otherwise, return $false$.

Algorithm 1: Handler(k_1, k_2: package range) at C_i

Output.$k_1 := k_1$, Output.$k_2 := k_2$, Solver.$k_2 := k_2$, Joiner.$k := k_2 - k1 + 1$
call Solver

Algorithm 2: Solver() at C_i

Data: Input queue: q, maximal number of models: k_2
$count := 0$
while $count < k_2$ **do**
(a) **if** C_i *is a leaf* **then** $S := \emptyset$
(b) **else** call Joiner and pop S from q
if $S = \epsilon$ **then** $count := k_2$
(c) **while** $count < k_2$ **do**
pick the next model $S^\star$ from Isolve(S)
if $S^\star \neq \epsilon$ **then** push $S^\star$ to Output.q and $count := count + 1$
else $count := k_2$
refresh() and push ϵ to Output.q

- *get_next*(ℓ, k): pose a query asking for the next k equilibria from neighbor C_{c_ℓ}; if C_{c_ℓ} sends back some models, then store them in the queue q_ℓ and return $true$; otherwise, return $false$. Note that this subroutine needs to keep track of which range has been already asked for to which neighbor by maintaining a set of counters. When a counter wrt. a neighbor C_{c_ℓ} is set to value t, then the latest request to C_{c_ℓ} asks for the t'th package of k models, i.e., models in the range given by $k_1 = (t-1) \times k + 1$ and $k_2 = t \times k$. For simplicity, we do not go into further details.
- *refresh*(): reset all counters and flags of Joiner to their starting states, e.g., $first\_join$ to $true$, all counters to 0.

The process at each context C_i is triggered when a message from a parent arrives at the Handler. Then Handler notifies Solver to compute up to k_2 model, and Output to collect the ones in the range from k_1 to k_2 and return them to the parent. Furthermore, it sets the package size at Joiner to $k = k_2 - k_1 + 1$ in case C_i needs to query further neighbors (cf. Algorithm 1).

When receiving a notification from Handler, Solver first prepares the input for its local solver. If C_i is a leaf context then the input S simply is the empty set assigned in Step (a); otherwise, Solver has to trigger Joiner (Step (b)) for input from neighbors. With input fed from neighbors, the subroutine Isolve is used in Step (c) to compute at most k_2 results and send them to the output queue.

The Joiner, only activated for intermediate contexts as discussed, gathers partial equilibria from the neighbors in a fixed ordering and stores the joined, consistent input to a local buffer. It communicates just one input at a time to Solver upon request. The fixed joining order is guaranteed by always asking the first package of k models from all neighbors at the beginning in Step (d). In subsequent rounds, we just query the first neighbor that can return further models (Step (e)). When all neighbors run out of models in Step (f), the joining process reaches its end and sends ϵ to Solver.

Algorithm 3: Joiner() at C_i

Data: Queue $q_1, \ldots,$ queue q_j for $In(i) = \{c_1, \ldots, c_j\}$, buffer for partial equilibria: buf, flag $first\_join$

while $true$ **do**
  if buf *is not empty* **then** pop S from buf, push S to Solver.q and **return**
  if $first\_join$ **then**
(d) **if** $get\_first(1, j, k) = false$ **then** push ϵ to Solver.q and **return**
    else $first\_join := false$
  else
    $\ell := 1$
(e) **while** $get\_next(\ell, k) = false$ *and* $\ell \leq j$ **do** $\ell := \ell + 1$
    if $1 < \ell \leq j$ **then** $get\_first(1, \ell - 1, k)$
(f) **else if** $\ell > j$ **then** push ϵ to Solver.q and **return**
  for $S_1 \in q_1, \ldots, S_j \in q_j$ **do** add $S_1 \bowtie \cdots \bowtie S_j$ to buf

Algorithm 4: Output() at C_i

Data: Input queue: q, starting model: k_1, end model: k_2

$buf := \emptyset$ and $count := 0$

while $count < k_1$ **do**
  pick an S from Output.q
  if $S = \epsilon$ **then** $count := k_2 + 1$
  else $count := count + 1$

while $count < k_2 + 1$ **do**
  wait for an S from Output.q
  if $S = \epsilon$ **then** $count := k_2 + 1$
  else $count := count + 1$ and add S to buf

if buf *is empty* **then** send ϵ to parent **else** send content of buf to parent

Note that while proceeding as above guarantees that no models are missed, it can in general lead to multiple considerations of combinations (inputs to Solver). Using a fixed size cache might mitigate these effects of recomputation, but since limitless buffering again quickly exceeds memory limits, recomputation is an unavoidable part of trading computation time for less memory.

The Output component simply reads from its queue until it receives ϵ (cf. Algorithm 4). Upon reading, it throws away the first $k_1 - 1$ models and only keeps the ones from k_1 onwards. Eventually, if fewer than k_1 models have been returned by Solver, then Output will return ϵ to the parent.

Example 4. Consider an instance of the MCS in Example 1 with $m = 1, \ell = 5$, i.e., $M = (C_1, C_2, C_3)$. Querying C_1 with a package size of $k = 1$, first causes the query to be forwarded to C_2 in terms of a pair $k_1 = k_2 = 1$. As a leaf context, C_2 invokes the local solver and eventually gets five different models. However, it just returns one partial equilibrium back to C_1, e.g., $(\epsilon, \{a_2^1\}, \epsilon)$. Note that t_2 is projected away since it does not appear among the atoms of C_2 accessed in bridge rules of C_1. The same happens at C_3 and we assume that it returns $(\epsilon, \epsilon, \{a_3^2\})$ to C_1. At the root context C_1, the two

single partial equilibria from its neighbors are consistently combined into $(\epsilon, \{a_2^1\}, \{a_3^2\})$. Taking this as an input to the local solving process, C_1 can eventually compute 5 answers, but in fact just returns one of them to the user, e.g., $S = (\{a_1^1, t_1\}, \{a_2^1\}, \{a_3^2\})$.

The following proposition shows the correctness of our algorithm.

Proposition 1. *Let $M = (C_1, \ldots, C_n)$ be an MCS, $i \in \{1, \ldots, n\}$ and let $k \geq 1$ be an integer. On input $(1, k)$ to C_i.*Handler*, C_i.*Output *returns up to k different partial equilibria with respect to C_i, and in fact k if at least k such partial equilibria exist.*

4.2 Parallelized Streaming

As the reader may have anticipated, the strategy of ignoring up to k_1 models and then collecting the next k is not likely to be the most effective. The reason is that each context uses only one Solver, which in general has to serve more than one parent, i.e., requests for different ranges of models of size k. When a new parent context requests models, we have to refresh the state of Solver and Joiner and redo from scratch. This is unavoidable, unless a context satisfies the specific property that only one parent can call it.

Another possibility to circumvent this problem is parallelization. The idea is to serve each parent with a set of the Handler, Joiner, Solver and Output components. In this respect, the basic interaction between each unit is still as shown in Fig. 1b, with the notable difference that each component now runs in an *individual thread*. The significant change is that Solver does not control Joiner but rather waits at its queue to get new input for the local solving process. The Joiner independently queries the neighbors, combines partial equilibria from neighbors, and puts the results into the Solver queue.

The effect is that we do not waste recomputation time for unused models. However, in practice, unlimited parallelization also faces a similar problem of exhausting resources as observed in DMCSOPT. While DMCSOPT runs out of memory with instances whose local theories are large, unlimited parallel instances of the streaming algorithm can exceed the number of threads/processes that the operating system can support, e.g., in topologies that allow contexts to reach other context using alternative paths such as the diamond topology. In such situations, the number of threads generated is exponential in the number of pairs of connected contexts, which prohibits scaling to large system sizes.

A compromise between the two extreme approaches is to have a *bounded parallel* algorithm. The underlying idea is to create a fixed-size pool of multiple threads and components, and when incoming requests cannot be served with the available resources, the algorithm continues with the basic streaming procedure, i.e., to share computational resources (the components in the system) between different parents at the cost of recomputation and unused models. This approach is targeted for future work.

5 Experimental Results

We present initial experimental results for a SAT-solver based prototype implementation of our streaming algorithm DMCS-STREAMING written in C++.[1] The host system was

[1] Available at `http://www.kr.tuwien.ac.at/research/systems/dmcs/`

topology / parameter	#	DMCSOPT	DMCS-STREAMING		
			$k = 0$	$k = 10$	$k = 100$
T_1 / $(10, 10, 5, 5)$	18	1.21	0.24	0.39	6.80
T_2 / $(10, 10, 5, 5)$	308	7.46	0.34	0.27	0.65
T_3 / $(50, 10, 5, 5)$	32	8.98	2.28	3.22	139.83
T_4 / $(50, 10, 5, 5)$	24	5.92	2.27	1.80	156.18
T_5 / $(100, 10, 5, 5)$	16	24.87	9.10	5.43	—
T_6 / $(100, 10, 5, 5)$	4	13.95	6.94	86.26	—
T_7 / $(10, 40, 20, 20)$	—	—	—	0.15	0.76
T_8 / $(10, 40, 20, 20)$	—	—	—	0.14	0.68
T_9 / $(50, 40, 20, 20)$	—	—	—	1.66	8.45
T_{10} / $(50, 40, 20, 20)$	—	—	—	1.64	8.04
T_{11} / $(100, 40, 20, 20)$	—	—	—	5.04	27.84
T_{12} / $(100, 40, 20, 20)$	—	—	—	5.00	26.30
R_1 / $(10, 10, 5, 5)$	12	2.17	0.99	2.44	—
R_2 / $(10, 10, 5, 5)$	16	3.11	3.31	0.17	143.83
R_3 / $(50, 10, 5, 5)$	21	15.49	13.43	—	—
R_4 / $(50, 10, 5, 5)$	12	10.30	6.43	—	—
R_5 / $(10, 40, 20, 20)$	—	—	—	0.86	6.27
R_6 / $(10, 40, 20, 20)$	—	—	—	0.51	5.66
R_7 / $(50, 40, 20, 20)$	—	—	—	3.06	29.81
R_8 / $(50, 40, 20, 20)$	—	—	—	4.29	43.94

Table 1: Runtime in secs, timeout 180 secs (—)

using two 12-core AMD Opteron 2.30GHz processors with 128GB RAM running Ubuntu Linux 10.10. We compare the basic version of the algorithm DMCS-STREAMING with DMCSOPT.

We used *clasp* 2.0.0 [9] and *relsat* 2.02 [3] as back-end SAT solvers. Specifically, all generated instantiations of multi-context systems have contexts with ASP logics. We use the translation defined in [6] to create SAT instances at contexts C_k, *clasp* to compute all models in case of DMCSOPT, and *relsat*[2] to enumerate models in case of DMCS-STREAMING.

For initial experimentation, we created random MCS instances with fixed topologies that should resemble the context dependencies of realistic scenarios. We have generated instances with binary tree (T) and ring (R) topologies. Binary trees grow balanced, i.e., every level is complete except for the last level, which grows from the left-most context.

A parameter setting (n, s, b, r) specifies (i) the number n of contexts, (ii) the local alphabet size $|\Sigma_i| = s$ (each C_i has a random ASP program on s atoms with 2^k answer sets, $0 \leq k \leq s/2$), (iii) the maximum interface size b (number of atoms exported), and (iv) the maximum number r of bridge rules per context, each having ≤ 2 body literals.

[2] The use of *relsat* is for technical reasons, and since *clasp* and *relsat* use different enumeration algorithms, the subsequent results are to be considered preliminary.

Table 1 shows some experimental results for parameter settings $(n, 10, 5, 5)$ and $(n, 40, 20, 20)$ with system size n ranging from 10 to 100. For each setting, running times on two instances are reported. Each row X_i ($X \in \{T, R\}$) displays pure computation time (no output) for rings (R) and binary trees (T), where the $\#$ columns show the number of projected partial equilibria computed at C_1 (initiated by sending the request to C_1 for the respective algorithms with the optimized query plan mentioned in [2]). With respect to DMCS-STREAMING, we run the algorithm with three request package sizes, namely $k = \{0, 10, 100\}$. The package size $k = 0$ amounts to an unbounded package, which means that the DMCS-STREAMING model exchange strategy is equivalent to DMCSOPT. For $k > 0$, we reported the running time until the first k unique models are returned, or all answers in case the total number of models is smaller than k.

We have observed several improvements. The new implementation appears to be faster than DMCSOPT when computing all partial equilibria ($k = 0$). This can be explained by the fact that in DMCSOPT we call the *clasp* binary and parse models using I/O streams, while DMCS-STREAMING tightly integrates *relsat* into the system, hence saving a significant amount of time used just for parsing models.

Getting the first k answers also runs faster in general. When we increase the size of the local theories, DMCSOPT and DMCS-STREAMING with $k = 0$ are stuck. However, DMCS-STREAMING can, with a reasonable small package size k, still return up to k answers in an acceptable time. When the package size increases, it usually takes longer or even timeouts. This can be explained by recomputation of models when requesting the next package of k models. We also observed this behavior in the ring topology with parameter setting $(50, 10, 5, 5)$, where DMCS-STREAMING timed out with $k \in \{10, 100\}$.

For the same reason, one would expect that asking for the next packages of k unique models might take more than linear amount of time compare to the time required to get the first package.

Comparing the two topologies, observe that rings are cyclic and thus the algorithm makes guesses for the values of the bridge atoms at the cycle-breaking context, and eventually checks consistency of the guess with the models computed locally at the same context. Hence, the system size that our algorithm can evaluate ring topology is smaller than that for the tree topology, which is acyclic.

6 Conclusion

Our work on computing equilibria for distributed multi-context systems is clearly related to work on solving constraint satisfaction problems (CSP) and SAT solving in a distributed setting; Yokoo et al. [14] survey some algorithms for distributed CSP solving, which are usually developed for a setting where each node (agent) holds exactly one variable, the constraints are binary, communication is done via messages, and every node holds constraints in which it is involved. This is also adopted by later works [15, 8] but can be generalized [14]. The predominant solution method are backtracking algorithms. In [11], a suite of algorithms was presented for solving distributed SAT (DisSAT), based on a random assignment and improvement flips to reduce conflicts. However, the algorithms are geared towards finding a single model, and an extension to streaming multiple

(or all) models is not straightforward; for other works on distributed CSP and SAT, this is similar. A closer comparison, in which the nature of bridge rules and local solvers as in our setting is considered, remains to be done.

Very recently, a model streaming algorithm for HEX-programs (which generalize answer set programs by external information access) has been proposed [7]. It bares some similarities to the one in this paper, but is rather different. There, monolithic programs are syntactically decomposed into modules (akin to contexts in MCS) and models are computed in a modular fashion. However, the algorithm is not fully distributed and allows exponential space use in components. Furthermore, it has a straightforward strategy to combine partial models from lower components to produce input for the upper component.

Currently, we are working on a conflict driven version of model streaming, i.e., in which clauses (nogoods) are learned from conflicts and exploited to reduce the search space, and on an enhancement by parallelization. We expect that this and optimizations tailored for the local solver and context setting (e.g., aspects of privacy) will lead to further improvements.

References

1. Analyti, A., Antoniou, G., Damasio, C.V.: Mweb: A principled framework for modular web rule bases and its semantics. ACM Trans. Comput. Logic 12(2), 17:1–17:46 (2011)
2. Bairakdar, S., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Decomposition of Distributed Nonmonotonic Multi-Context Systems. In: JELIA'10. LNAI, Springer (2010)
3. Bayardo, R.J., Pehoushek, J.D.: Counting models using connected components. In: AAAI'00. pp. 157–162. AAAI Press (2000)
4. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T.: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
5. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI'07. pp. 385–390. AAAI Press (2007)
6. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed Nonmonotonic Multi-Context Systems. In: KR'10. pp. 60–70. AAAI Press (2010)
7. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Schüller, P.: Pushing Efficient Evaluation of HEX Programs by Modular Decomposition. In: LPNMR'11. pp. 93–106. Springer (2011)
8. Gao, J., Sun, J., Zhang, Y.: An improved concurrent search algorithm for distributed csps. In: Australian Conference on Artificial Intelligence. pp. 181–190. Springer (2007)
9. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam Answer Set Solving Collection. AI Commun. 24(2), 107–124 (2011)
10. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics or: How we can do without modal logics. Artif. Intell. 65(1), 29–70 (1994)
11. Hirayama, K., Yokoo, M.: The distributed breakout algorithms. Artif. Intell. 161(1–2), 89–115 (2005)
12. Homola, M.: Semantic Investigations in Distributed Ontologies. Ph.D. thesis, Comenius University, Bratislava, Slovakia (2010)
13. Roelofsen, F., Serafini, L.: Minimal and absent information in contexts. In: IJCAI'05 (2005)
14. Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: A review. Autonomous Agents and Multi-Agent Systems 3(2), 185–207 (2000)
15. Zivan, R., Meisels, A.: Concurrent search for distributed CSPs. Artif. Intell. 170(4-5), 440–461 (2006)