Massively Parallel Reasoning under the Well-Founded Semantics using X10

Ilias Tachmazidis[1], Long Cheng[2,3], Spyros Kotoulas[3], Grigoris Antoniou[1], Tomas E Ward[2]

[1] University of Huddersfield, UK
[2] National University of Ireland Maynooth, Ireland
[3] IBM Research, Ireland
{ilias.tachmazidis, g.antoniou}@hud.ac.uk, {lcheng, tward}@eeng.nuim.ie, spyros.kotoulas@ie.ibm.com

***Abstract*—Academia and industry are investigating novel approaches for processing vast amounts of data coming from enterprises, the Web, social media and sensor readings in an area that has come to be known as Big Data. Logic programming has traditionally focused on complex knowledge structures/programs. The question arises whether and how it can be applied in the context of Big Data. In this paper, we study how the well-founded semantics can be computed over huge amounts of data using mass parallelization. Specifically, we propose and evaluate a parallel approach based on the X10 programming language. Our experiments demonstrate that our approach has the ability to process up to 1 billion facts within minutes.**

***Keywords*-Well-Founded Semantics; X10; Big Data; Mass Parallelization;**

I. Introduction

The quantity of available data generated by social media, sensor networks, scientific databases and government authorities is increasing in an unprecedented pace. *Big data* comes with new challenges and opportunities for diverse scientific, technological and business developments. A significant aspect of Big Data is its heterogeneity and the desire to combine with other information, including databases and web data, in order to increase utility.

The question arises whether research on areas such as knowledge representation, rule systems, logic programming and the Semantic Web can connect to the Big Data wave. On the one hand, there is a clear application scope, e.g. deriving higher-level knowledge, assisting decision support and data cleaning. On the other hand, there are significant challenges arising from the area's traditional focus on rich knowledge structures instead of large amounts of data and the computational cost of many methods central to the area.

As established by the LarKC project [1], the most prominent approach for enabling reasoning with big data is parallelization by distributing computation among nodes. There are mainly two proposed approaches in the literature, namely rule partitioning and data partitioning [2].

Rule partitioning is based on the idea that the computation of each rule is assigned to a node in the cluster, forming a pipeline of operations. In this case, the workload for each rule (and node) depends on the structure and the size of the given rule set, which could possibly prevent balanced work distribution and high scalability. An experimental evaluation reported in [2] shows sub-linear but monotonic speedups. However, the used rule sets were small, and thus, only a small number of processors could be utilized.

In data partitioning, data is partitioned among nodes, allowing more balanced distribution of the computation. The partitioning must be performed carefully, since an uneven data distribution can have detrimental effects on performance. In [3], the authors pointed out the fact that Semantic Web data follow a highly uneven distribution and, to balance computation costs, they proposed the *divide-conquer-swap* strategy [4] in which data is constantly re-partitioned. Inference is performed independently on each partition and the system guarantees eventual completeness of the process.

WebPIE [5] implements forward reasoning under RDFS and OWL *ter Horst* semantics over the MapReduce framework scaling up to 100 billion triples. In [6], authors deal with parallel materialization of RDFS closure using message passing interface (MPI). Their implementation is reported to scale up to 128 cores with input of hundreds of millions of triples. In [7] authors present an approach for calculating the RDFS closure on billions of triples completely in-memory using the Cray XMT shared-memory supercomputer. The system is shown to be able to scale up to 512 processors while handling 20 billion triples completely in-memory.

$\mathfrak{Deslog}$ [8] is a parallel tableau-based description logic reasoner for $\mathcal{ALC}$. Several optimization techniques were incorporated into the shared-memory parallel reasoner, thus leading to good scalability properties for TBox classification. *UUPR* (*Ulm University Parallel Reasoner*) [9] parallelized a sequential algorithm for $\mathcal{SHN}$ ABoxes, parallelizing the tableau procedure itself, by utilizing concurrent computation of the nondeterministic choices. Various optimizations were added to the shared-memory parallel implementation, speeding up the reasoning process.

Several works based on nonmonotonic reasoning, having the ability of dealing with incomplete information or inconsistencies, have recently been reported. More specifically, [10] presents an approach for defeasible logic with unary predicates scaling up to billions of facts and an extension for predicates of arbitrary arity, under the assumption of stratification, scaling up to millions of facts [11]. The work in [12] deals with the computation of stratified semantics of logic programming that can be applied to billions of facts.

In this paper, we propose a parallel approach for the well-founded semantics computation using the X10 programming framework [13]. In particular, we implement highly efficient parallel join and anti-join operations, and calculate the well-founded model for the given program. The calculation of a negative rule as a sequence of join and anti-join operations was initially described in [12].

Our current paper differs from previous works with the following contribution: (a) our parallel in-memory implementation is more efficient compared to an implementation based on the MapReduce framework as we do not require storing intermediate results in secondary storage, and (b) we allow recursion through negation, meaning that the well-founded model can contain undefined atoms, while avoiding the Herbrand base materialization which is prohibiting for the case of Big Data due to the excessive amount of generated data. Experimental evaluation presents the advantages of using X10, while showing that our approach can process 1 billion facts within minutes.

Considering the expressiveness of our method, logic programming (under Well-Founded Semantics) has orthogonal functionality compared to description logic based languages. While logic programs without negation would roughly fall under OWL 2 RL, negation (with recursion) adds new functionality not present in description logics, allowing for reasoning with incomplete and conflicting information.

The rest of the paper is organized as follows. Section II introduces briefly the X10 programming language, the well-founded semantics and the alternating fixpoint procedure. Parallel join and anti-join operations for the well-founded semantics are described in Section III. Section IV describes a parallel implementation and the correctness of the approach, while an experimental evaluation is reported in Section V. We conclude and discuss future directions in Section VI.

II. Preliminaries

In this section, we describe the basic notions of the X10 programming language, the Well-Founded Semantics and the Alternating Fixpoint Procedure.

A. X10

X10 [13] is a new multi-paradigm programming language developed by IBM. It supports the asynchronous partitioned global address space (APGAS) model and is specifically designed to increase programmer productivity, while being amenable to programming shared memory and distributed memory supercomputers. It uses the concepts of `place` and `activity` as the kernel notions to exploit parallelism in the available hardware. A place is a logical abstraction of the underlying heterogeneous processing element in the hardware, such as cores in a multi-core architecture, GPUs, or an entire physical machine. Activities are light-weight threads that run on places. X10 schedules activities on places to best utilize the available parallelism. The number of places is constant through the life-time of an X10 program and is initialized at program startup. Activities on the other hand can be forked at program execution time. Forking an activity can be blocking, wherein the parent returns after the forked activity completes execution, or non-blocking, wherein the parent returns instantaneously, after forking an activity. Furthermore, these activities can be forked locally or on a remote place.

X10 provides a data structure called distributed array (`DistArray`) for programming parallel algorithms. One or more elements in the `DistArray` can be mapped to a single place using the concept of points [13], and such elements can be kept in or operated on memory through the life of the code. The following three X10 primitives are critical in understanding the pseudocode given in the following sections:

- `at(p) S`: this construct executes statement `S` at a specific place `p`. The current activity is blocked until `S` finishes executing on `p`.
- `async S`: a child activity is forked by this construct. The current activity returns immediately (non-blocking) after forking `S`.
- `finish S`: this construct is used to block the current activity and wait for all activities forked by `S` to terminate.

There are a number of advantages using the X10 language, and in turn the APGAS model, to implement our algorithm: (1) flexible and efficient scheduling. APGAS, like PGAS, separates tasks from the underlying concurrency model, thereby allowing one to implement an efficient scheduling strategy irrespective of the number of tasks forked using `async`; (2) APGAS, being derived from both MPI and OpenMP programming models, extracts parallelism at both the distributed and single machine hierarchies; (3) the abstract programming model supports the development of succinct code which is easier to debug and maintain.

B. Well-Founded Semantics

In this section, we provide the definition of the *well-founded semantics* (WFS) as they were defined in [14].

Definition 2.1: [14] A *general logic program* is a finite set of *general rules*, which may have both positive and negative subgoals. A general rule is written with its *head*, or conclusion on the left, and its subgoal (body), if any, to the right of the symbol "←", which may be read "if". For example, "p(X) ← a(X), **not** b(X).", is a rule in which *p(X)* is the head, *a(X)* is a *positive subgoal*, and *b(X)* is a *negative subgoal*. This rule may be read as "*p(X)* if *a(X)* and not *b(X)*". A *Horn rule* is one with no negative subgoals, and a *Horn logic program* is one with only Horn rules.

We use the following conventions: A logical variable starts with a capital letter while a constant or a predicate starts with a lowercase letter. Functions are not allowed. A predicate of arbitrary arity will be referred to as a *literal*. Constants, variables and literals are *terms*. A *ground term* is a term with no variables. The *Herbrand universe* is the set of constants in a given program. The *Herbrand base* is the set of ground terms that are produced by the substitution of variables with constants in the Herbrand universe. In this paper, we refer to Horn rules also as *definite* rules. Likewise, Horn programs are referred to as *definite* programs. In addition, each rule is required to be safe, that is, each variable in a rule must occur (also) in a positive subgoal.

Definition 2.2: [14] Let a program **P**, its associated Herbrand base H and a partial interpretation I be given. We say $A \subseteq H$ is an *unfounded set* (of **P**) with respect to I if each atom $p \in A$ satisfies the following condition: For each instantiated rule R of **P** whose head is p, (at least) one of the following holds:

(1) Some (positive or negative) subgoal q of the body is false in I,

(2) Some positive subgoal of the body occurs in A.

A literal that makes (1) or (2) above true is called a *witness of unusability* for rule R (with respect to I).

Theorem 2.1: [14] The data complexity of the well-founded semantics for function-free programs is polynomial time.

C. Alternating Fixpoint Procedure

In this section, we provide the definition of the alternating fixpoint procedure as it was defined in [15].

Definition 2.3: [15] For a set S of literals we define the following sets:

pos(S) := {A ∈ S | A is a positive literal },
neg(S) := {A | **not** A ∈ S}.

Definition 2.4: [15] (*Extended Immediate Consequence Operator*)
Let P be a normal logic program. Let I and J be sets of ground atoms. The set $T_{P,J}(I)$ of *immediate consequences* of I w.r.t. P and J is defined as follows:

$$T_{P,J}(I) := \{A \mid \text{there is } A \leftarrow \mathcal{B} \in \text{ground(P) with } \text{pos}(\mathcal{B}) \subseteq I \text{ and } \text{neg}(\mathcal{B}) \cap J = \emptyset\}.$$

If P is definite, the set J is not needed and we obtain the standard immediate consequence operator T_P by $T_P(I) = T_{P,\emptyset}(I)$.

For an operator T we define $T \uparrow 0 := \emptyset$ and $T \uparrow i := T(T \uparrow i-1)$, for $i > 0$. lfp(T) denotes the least fixpoint of T, i.e. the smallest set S such that $T(S) = S$.

Definition 2.5: [15] (*Alternating Fixpoint Procedure*)
Let P be a normal logic program. Let P^+ denote the subprogram consisting of the definite rules of P. Then the sequence $(K_i,U_i)_{i\geq 0}$ with set K_i of true (known) facts and U_i of possible (unknown) facts is defined by:

$$\begin{array}{lll} & K_0 := \text{lfp}(T_{P^+}) & U_0 := \text{lfp}(T_{P,K_0}) \\ i > 0: & K_i := \text{lfp}(T_{P,U_{i-1}}) & U_i := \text{lfp}(T_{P,K_i}) \end{array}$$

The computation terminates when the sequence becomes stationary, i.e., when a fixpoint is reached in the sense that $(K_i,U_i) = (K_{i+1},U_{i+1})$. This computation schema is called the *Alternating Fixpoint Procedure* (AFP).

We rely on the definition of the *well-founded partial model* W_p^* of P as given in [14].

Theorem 2.2: [15] (Correctness of *AFP*)
Let the sequence $(K_i,U_i)_{i\geq 0}$ be defined as above. Then there is a $j \geq 0$ such that $(K_j,U_j)=(K_{j+1},U_{j+1})$. The well-founded model W_p^* of P can be directly derived from the fixpoint (K_j,U_j), i.e.,

$$W_p^* = \{L \mid L \text{ is a positive ground literal and } L \in K_j \text{ or } L \text{ is a negative ground literal } \textbf{not } A \text{ and } A \in \textit{BASE(P)} - U_j\},$$

where BASE(P) is the Herbrand base of program P.

III. Join and Anti-join for WFS

In this section we provide a description of the $T_{P,J}(I)$ computation, which is modeled as a sequence of join and anti-join operations.

A. Computing $T_{P,J}$(I)

Consider the following program:

$$p(X,Y) \leftarrow a(X,Z),\ b(Z,Y),\ \textbf{not}\ c(X,Z).$$

here *p(X,Y)* is our *final goal*, *a(X,Z)* and *b(Z,Y)* are positive subgoals, while **not** *c(X,Z)* is a negative subgoal. In order to compute our final goal *p(X,Y)* we need to ensure that {*a(X,Z)*, *b(Z,Y)*} ⊆ I and {*c(X,Z)*} ∩ J = ∅, namely both *a(X,Z)* and *b(Z,Y)* are in I while none of *c(X,Z)* is found in J.

As positive subgoals depend on I we can group them into a *positive goal*. A positive goal consists of a new predicate (say *ab*) that contains as arguments the union of two sets: (a) all the arguments of the final goal (X,Y) and (b) all the common arguments between positive and negative subgoals (X,Z), namely we need to compute *ab(X,Z,Y)*. The final goal (*p(X,Y)*) consists of all values of the positive goal (*ab(X,Z,Y)*) that do not match the negative subgoal (**not** *c(X,Z)*) on their common arguments (X,Z).

B. Positive goal calculation

Consider the following program:

$$p(X,Y) \leftarrow a(X,Z),\ b(Z,Y),\ \textbf{not}\ c(X,Z).$$

where

$$I = \{a(1,2),\ a(1,3),\ b(2,4),\ b(3,5)\}$$
$$J = \{c(1,2),\ c(2,3)\}$$

A single join, calculating the positive goal *ab(X,Z,Y)*, can be performed as described in Algorithm 1. There, we divided

Algorithm 1 Single join

```
    Hash-Redistribution:    // $n$ is the number of places
 1: finish async at $p \in P$ {
 2: Initialize $A$:array[array[literal]]($n$),
        $B$:array[array[literal]]($n$)
 3: for all $literal \in I$ do
 4:     if $literal.predicate == a$ then
 5:         $des$=hash($literal.Z$)
 6:         $A(des).add(literal)$
 7:     else if $literal.predicate == b$ then
 8:         $des$=hash($literal.Z$)
 9:         $B(des).add(literal)$
10:     end if
11: end for
12: for $i \leftarrow 0..(n-1)$ do
13:     Push $A(i)$ to $r\_A(i)(here.id)$,
            $B(i)$ to $r\_B(i)(here.id)$ at place $i$
14: end for }

    Local Joins:
15: finish async at $p \in P$ {
16: Initialize $T\_b$:hashmap[Z,Y], $AB$:array[literal]
17: for all $literal \in r\_B(here.id)$ do
18:     Add $literal$ in $T\_b$
19: end for
20: for all $literal \in r\_A(here.id)$ do
21:     if $T\_b.contains(literal.Z)$ then
22:         $AB.add(literal.X, literal.Z,$
                $T\_b.get(literal.Z).value)$
23:     end if
24: end for }
```

our joins into two phases, namely *hash-redistribution* and *local joins*. Note that we use only literals from *I* and that we are interested in distinct literals.

As shown in lines 3-11 of Algorithm 1, all literals with predicate a and b at each place (recall that a place is a logical abstraction for a processing element and *here* means current place in X10) are firstly grouped according to the hash values of their join keys respectively, so as to reduce transferred data and re-use computation. The grouped literals are pushed to the corresponding remote places for joining. We use the `finish` operation to guarantee the completion of the data transfer at each place. For a two-node ($n = 2$) system with a hash function based on the modulo of n, we have the following pairs at each place after redistribution:

place 1		*place 2*	
a(1,3)	b(3,5)	a(1,2)	b(2,4)

Once the grouped literals have been transferred to the appropriate remote places, the local joins can commence. Line 15-24 of Algorithm 1 presents the details of this process. A local `HashMap` T_b is built based on the received literals with predicate b, and all received literals with predicate a are looked up over T_b to retrieve the matched literals. Intermediate results are kept in memory for the latter processing. All these processes take place in parallel at each place, and we use the `finish` operation for synchronization. Thus, after execution, we have:

place 1 keeps *ab(1,3,5)* in memory
place 2 keeps *ab(1,2,4)* in memory

For rules with more than one join between positive subgoals, we do multi-way joins. Consider the following program:

$$\text{q(X,Y)} \leftarrow \text{a(X,Z), b(Z,W), c(W,Y), } \textbf{not} \text{ d(X,W).}$$

We can compute the positive goal (*abc(X,W,Y)*) by applying our approach for the single join twice. First, we need to join *a(X,Z)* and *b(Z,W)* on *Z*, producing a temporary literal (say *ab(X,W)*), and then join *ab(X,W)* and *c(W,Y)* on *W* producing the positive goal (*abc(X,W,Y)*). Once *abc(X,W,Y)* is calculated, we proceed with calculating the final goal *q(X,Y)* by retaining all the values of *abc(X,W,Y)* that do not match **not** *d(X,W)* on their common arguments (X,W).

C. Final goal calculation

Considering the program mentioned at the beginning of Section III-B, by calculating the positive goal *ab(X,Z,Y)* we obtain the following knowledge:

ab(1,3,5) at place 1 *ab(1,2,4) at place 2*

In order to calculate the final goal (*p(X,Y)*) we need to perform an anti-join between the positive goal (*ab(X,Z,Y)*) and the negative subgoal (**not** *c(X,Z)*). To perform an anti-join we use only the previously calculated positive goal (*ab(X,Z,Y)*) and literals from *J*.

We perform an anti-join between *ab(X,Z,Y)* and **not** *c(X,Z)* on their common arguments (X,Z), calculating the final goal (*p(X,Y)*), which contains all the results from *ab(X,Z,Y)* that are not found in *c(X,Z)*, as described in Algorithm 2.

The process is similar to the one in Algorithm 1. Since the intermediate results AB have already been redistributed based on Z, we only need to transfer the literals with predicate *c* in J, as shown in lines 3-11 of Algorithm 2. The redistribution results in the following pairs at each place:

place 1	*place 2*
c(2,3)	c(1,2)

During the phase of local anti-joins we output literals of the predicate *ab* only if it is not matched by predicate *c* on their common arguments Z and X at each place, and output *p(X,Y)* correspondingly (lines 18-21). Finally, we get:

place 1 outputs *p(1,5)*
place 2 has no output

Algorithm 2 Anti-join

```
    Hash-Redistribution:
 1: finish async at $p \in P$ {
 2: Initialize $C$:array[array[literal]]($n$)
 3: for all $literal \in J$ do
 4:     if $literal.predicate == c$ then
 5:         $des$=hash($literal.Z$)
 6:         $C(des).add(literal)$
 7:     end if
 8: end for
 9: for $i \leftarrow 0..(n-1)$ do
10:     Push $C(i)$ to $r\_C(i)(here.id)$ at place $i$
11: end for }

    Local Anti-joins:
12: finish async at $p \in P$ {
13: Initialize $T\_c$:hashmap[Z,X], $C$:array[literal]
14: for all $literal \in r\_C(here.id)$ do
15:     Add $literal$ in $T\_c$
16: end for
17: for all $literal \in AB$ do
18:     if $!T\_c.contains(literal.Z)$ then
19:         Output $p(literal.X, literal.Y)$
20:     else if $literal.X \neq T\_c.get(literal.Z).value$ then
21:         Output $p(literal.X, literal.Y)$
22:     end if
23: end for }
```

Algorithm 3 WFS fixpoint

```
    WFS_fixpoint($P$):                         ▷ input: program $P$
 1: $K_0$ = lfp($P$+, $\emptyset$);           ▷ output: set of literals $K_i$, $U_i$
 2: $U_0$ = lfp($P$, $K_0$);
 3: $i$ = 0;
 4: repeat
 5:     $i$++;                                 ▷ next "inference step"
 6:     $K_i$ = lfp($P$, $U_{i-1}$);
 7:     $U_i$ = lfp($P$, $K_i$);
 8: until $K_{i-1} == K_i$ and $U_{i-1} == U_i$
 9: return $K_i$, $U_i$;
```

Algorithm 4 Least fixpoint of $T_{P,J}(I)$

```
    lfp($P$, $J$):              ▷ input: program $P$, set of literals $J$
        ▷ output: set of literals $I$ (least fixpoint of $T_{P,J}(\emptyset)$)
 1: $I$ = $\emptyset$;
 2: $new$ = $\emptyset$;
 3: repeat
 4:     $I$ = $I \cup new$;
 5:     $new$ = T($P$, $I$, $J$);
 6:     $new$ = $new$ - $I$;
 7: until $new$ == $\emptyset$
 8: return $I$;
```

IV. Computing the Well-Founded Semantics

In this section, we provide the algorithm for the calculation of the well-founded semantics and provide a proof sketch in order to justify the correctness of our approach.

A. Algorithm description

An implementation of the well-founded semantics fixpoint is depicted in Algorithm 3. The algorithm takes as input a program *P* and calculates the sets of literals K_i and U_i until fixpoint is reached, namely (K_{i-1},U_{i-1}) = (K_i,U_i) for $i \geq 1$. Each set of literals (K_i and U_i) is calculated by the least fixpoint of $T_{P,J}(I)$ depicted in Algorithm 4.

The least fixpoint of $T_{P,J}(I)$ (*lfp*) takes as input two arguments, a program *P* and a set of literals *J*. Practically, we calculate the least fixpoint of $T_{P,J}(I)$ where *P* and *J* are given as input while *I* is initially set as empty (I=$\emptyset$). We also use a temporary set of inferred literals (*new*) in order to eliminate duplicates (*new* = *new* − *I*) prior to adding newly inferred literals to the set *I* ($I = I \cup$ *new*). Thus, we start by having I=$\emptyset$ and stop when no new knowledge can be inferred (*new* == $\emptyset$). The function *T(P, I, J)* is the computation of $T_{P,J}(I)$ as described Section III.

Let us now consider the calculation of the WFS fixpoint. Initially, we calculate K_0 over the positive part of the program *P* ($P+$) and set J= $\emptyset$. Then, we proceed with the calculation of U_0 given the already available set K_0 (J=K_0). Subsequently, we increase the counter *i* and calculate the least fixpoint of K_i (resp. U_i) given U_{i-1} (resp. K_i) until fixpoint is reached. WFS fixpoint is reached when K_{i-1} == K_i and U_{i-1} == U_i, and finally the sets of literals K_i and U_i are returned.

According to Theorem 2.2, having reached WFS fixpoint at step i, we can determine which literals are true, undefined and false as follows: (a) **true** literals, denoted by K_i, (b) **undefined** literals, denoted by $U_i - K_i$ and (c) **false** literals, denoted by BASE(P) − U_i.

For the WFS fixpoint algorithm we need to store the sets of literals K_i and U_i only for the current and the previous "inference step", namely if $i = k$, for $k \geq 1$, then we only need to store K_{i-1} and U_{i-1} in our knowledge base while calculating K_i and U_i. Since a fixpoint was not reached for $i = k - 1$, any K_j and U_j for $j < k - 1$ becomes irrelevant as it will not be used for the remainder of the computation. Thus, at any time of calculation (step i, for $i \geq 1$) we need to store up to four sets of literals (K_{i-1}, U_{i-1}, K_i, U_i).

B. Approach correctness

We need to ensure that our approach is in line with the definition of the alternating fixpoint procedure (see Definition 2.5). Thus, we provide the following proof sketch.

Proof sketch. First, we need to ensure that we calculate $T_{P,J}(I)$ according to Definition 2.4. Consider a given program P and given sets of literals I and J.

According to Section III-B, the positive part (*pos($\mathcal{B}$)*) of each rule of the instantiated program P ($A \leftarrow \mathcal{B} \in$ *ground(P)*) is calculated using literals from I (*pos($\mathcal{B}$)* $\subseteq I$), which agrees with Definition 2.4. The auxiliary predicates used in positive goal computation do not affect the final result as these predicates are not part of the given program. In addition, the fact that the positive goal contains the smallest set of arguments containing arguments of both final goal and negative subgoals reassures correctness, since no information is lost, and minimizes communication costs among nodes, as the overhead coming from redundant arguments is eliminated.

Since the positive goal is equivalent to the computation of the positive part of a rule, we may proceed with the negative part. According to Definition 2.4, a final goal of a rule is computed from a set of positive subgoals that belong to I (A | *there is* $A \leftarrow \mathcal{B} \in$ *ground(P) with pos($\mathcal{B}$)* $\subseteq I$), namely the positive goal of the rule, and a set of negative subgoals that do not belong to J (*neg($\mathcal{B}$)* $\cap J = \emptyset$). Thus, negative subgoals that could possibly match the positive goal on their common arguments should not be found in J. This is modeled by the anti-join as described in Section III-C.

Our next step is to investigate the equivalence of the least fixpoint calculation. According to the definition of the least fixpoint, provided in Section II-C, for a given program P and a set of literals J we start with $I = \emptyset$ and gradually calculate applicable rules until no new literals are inferred, namely $T_{P,J}(I) = I$. This is directly modeled by the least fixpoint (see Algorithm 4), since the computation starts with an empty set ($I = \emptyset$) and $T_{P,J}(I)$ is applied until no new knowledge is derived.

We have demonstrated that the calculation of the least fixpoint is in line with the calculation of *lfp($T_{P,J}(I)$)* of the alternating fixpoint procedure (see Section II-C). The correctness of Algorithm 3 is ensured by the carefully assigned sets of literals I and J, given a program P, for each *lfp($T_{P,J}(I)$)* following the Definition 2.5.

V. Evaluation

In this section, we present the results of our experimental evaluation on a commodity cluster. We conduct a quantitative evaluation of our implementation and also compare them slightly with the MapReduce implementation.

A. Methodology

The evaluation of our approach is based on the OpenRuleBench [16] benchmark. In [16], the authors propose a set of benchmarks for analyzing the performance and scalability of different rule engines. As our approach is based on large scale nonmonotonic reasoning, we follow the proposed methodology in [16] while adjusting several parameters. In [16], *loading* and *inference* time are separated, focusing on inference time. In this paper, we additionally report *writing* and *total* time.

We evaluate our approach considering *default negation* by applying the *win-not-win* test and merge *large (anti-)join tests* with *datalog recursion* and *default negation*, creating a new test called *transitive closure with negation*. Other metrics in [16], such as *indexing*, optimizations and cost-based analysis were performed manually.

B. Platform

We have implemented our experiments using the X10 programming language. We have performed experiments on a cluster with 16 IBM System x iDataPlex nodes, using a Gigabit Ethernet interconnect. Each node was equipped with dual Intel Xeon Westmere 6-core processors (resulting in a total of 12 cores per physical node), 128GB RAM and a single 1TB SATA hard drive. The operating system is Linux kernel version 2.6.32-220 and the software stack consists of X10 version 2.3 compiling to C++ and gcc version 4.4.6.

C. Evaluation tests

The *win-not-win* test [16] consists of a single rule, where *move* is the base relation:

$$\text{win(X)} \leftarrow \text{move(X,Y), not win(Y).}$$

We test the following data distributions: (a) the base facts form a cycle: {move(1,2), ..., move(i, i+1), ..., move(n-1,n), move(n,1)}, (b) the data is tree-structured: {move(i, 2*i), move(i, 2*i+1) | 1 $\leq$ i $\leq$ n}. We used four cyclic datasets and four tree-structured datasets with 125M, 250M, 500M and 1000M facts.

The *transitive closure with negation* test consists of the following rule set, where *b* is the base relation:

tc(X,Y) $\leftarrow$ par(X,Y).
tc(X,Y) $\leftarrow$ par(X,Z), tc(Z,Y).
par(X,Y) $\leftarrow$ b(X,Y), **not** q(X,Y).
par(X,Y) $\leftarrow$ b(X,Y), b(Y,Z), **not** q(Y,Z).
q(X,Y) $\leftarrow$ b(Z,X), b(X,Y), **not** q(Z,X).

We test the following data distribution: the base facts are chain-structured: {b(i, i+k) | 1 $\leq$ i $\leq$ n, k $<$ n}. We used four chain-structured datasets for a constant number of joins in the initially formed chain ($\lceil n/k \rceil - 1$) with n = 62.5M, 125M, 250M and 500M, and k = 12.5M, 25M, 50M and 100M respectively, and four chain-structured datasets for increasing number of joins in the initially formed chain ($\lceil n/k \rceil - 1$) with n = 125M, and k = 41.7M, 25M, 13.9M and 7.36M.

D. Results

Our experimental results are summarized in Figures 1, 2, 3 and 4. Each figure reports the runtime in seconds for *loading*, *inference* and *writing* (while *total* time corresponds to the time of each stacked column) for a given dataset and number of nodes. Runtimes are grouped for each dataset, while the number of nodes is shown explicitly for each stacked column.

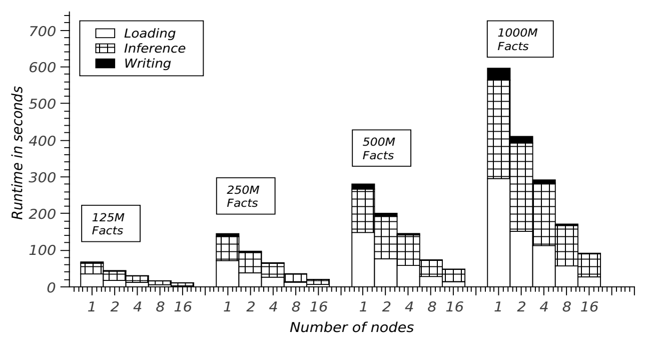

Figure 1. *Win-not-win* test for cyclic datasets. Runtime in seconds for various dataset sizes and numbers of nodes.

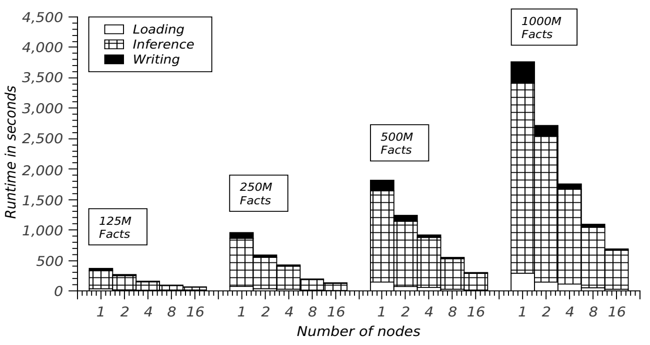

Figure 2. *Win-not-win* test for tree-structured datasets. Runtime in seconds for various dataset sizes and numbers of nodes.

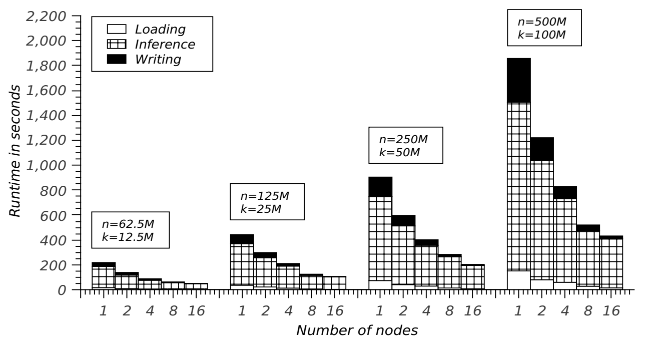

Figure 3. *Transitive closure with negation* test for chain-structured datasets. Runtime in seconds for constant $\lceil n/k \rceil - 1$.

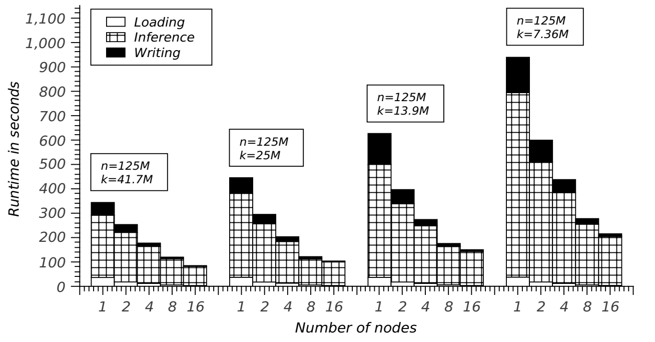

Figure 4. *Transitive closure with negation* test for chain-structured datasets. Runtime in seconds for increasing $\lceil n/k \rceil - 1$.

Figures 1 and 2 present the runtimes of our system for the *win-not-win* test over cyclic and tree-structured datasets respectively, applied for up to 1 billion facts and 16 nodes. Our system scales linearly in both cases with respect to dataset size and number of nodes. For cyclic distribution, the number of "inference steps" remains constant, following the same process for increasing inputs. However, for tree-structured datasets, the number of "inference steps" is gradually increasing, causing several notwithstanding fluctuations.

Figures 3 and 4 illustrates the scalability properties of our system for the *transitive closure with negation* test over chain-structured datasets for *constant* and *increasing* number of joins in the initially formed chain respectively, when run for up to 16 nodes. As expected, our approach scales linearly for constant $\lceil n/k \rceil - 1$, for dataset size (n) and number of facts per level (k). Here, the number of "inference steps" remains constant for increasing datasets, following the same inference sequence.

Note that the linear scalability for increasing $\lceil n/k \rceil - 1$, along with relatively low runtimes, show the superiority of our in-memory implementation when compared to one based on MapReduce. More specifically, the number of applied rules increases polynomially with respect to the length of chain, leading to polynomial runtimes for MapReduce based implementations, due to initialization overheads. However, the required number of in-memory read and written literals throughout the computation scales linearly for increasing chain lengths, allowing our implementation to be independent of the number of applied rules, thus leading to linear runtimes.

VI. Conclusion

In this paper we proposed a parallel and highly efficient approach for the computation of the well-founded semantics over large amounts of data based on the X10 programming language. We ran experiments for various rule sets and data sizes, showing that our approach is performant and can be applied to billions of facts.

In future work, we plan to study more complex knowledge representation methods including Answer-Set programming [17], RDF/S ontology evolution [18] and repair [19]. We believe that these complex forms of reasoning will benefit from the high degree of flexibility provided by X10, leading to a fine-grained resolution of the arising challenges. In the meantime, we will also employ more efficient and robust joins [20], [21], [22], [23] in our implementation so as to achieve even higher performance in the presence of different workloads.

Acknowledgments

The second author was supported by Irish Research Council and IBM Research Ireland.

References

[1] D. Fensel, F. van Harmelen, B. Andersson, P. Brennan, H. Cunningham, E. D. Valle, F. Fischer, Z. Huang, A. Kiryakov, T. K. il Lee, L. Schooler, V. Tresp, S. Wesner, M. Witbrock, and N. Zhong, "Towards LarKC: A Platform for Web-Scale Reasoning," in *ICSC*, 2008, pp. 524–529.

[2] R. Soma and V. K. Prasanna, "Parallel Inferencing for OWL Knowledge Bases," in *ICPP*. IEEE Computer Society, 2008, pp. 75–82.

[3] S. Kotoulas, E. Oren, and F. van Harmelen, "Mind the data skew: distributed inferencing by speeddating in elastic regions," in *WWW*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 2010, pp. 531–540.

[4] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, and F. van Harmelen, "Marvin: Distributed reasoning over large-scale Semantic Web data," *J. Web Sem.*, vol. 7, no. 4, pp. 305–316, 2009.

[5] J. Urbani, S. Kotoulas, J. Massen, F. van Harmelen, and H. Bal, "WebPIE: A Web-scale parallel inference engine using MapReduce," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 10, no. 0, 2012.

[6] J. Weaver and J. A. Hendler, "Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples," in *ISWC*, 2009, pp. 682–697.

[7] E. L. Goodman, E. Jimenez, D. Mizell, S. Al-Saffar, B. Adolf, and D. J. Haglin, "High-Performance Computing Applied to Semantic Databases," in *ESWC (2)*, ser. Lecture Notes in Computer Science, G. Antoniou, M. Grobelnik, E. P. B. Simperl, B. Parsia, D. Plexousakis, P. D. Leenheer, and J. Z. Pan, Eds., vol. 6644. Springer, 2011, pp. 31–45.

[8] K. Wu and V. Haarslev, "A Parallel Reasoner for the Description Logic ALC," in *Description Logics*, 2012.

[9] T. Liebig and F. Müller, "Parallelizing Tableaux-Based Description Logic Reasoning," in *OTM Workshops (2)*, 2007, pp. 1135–1144.

[10] I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas, "Towards Parallel Nonmonotonic Reasoning with Billions of Facts," in *KR*, G. Brewka, T. Eiter, and S. A. McIlraith, Eds. AAAI Press, 2012.

[11] I. Tachmazidis, G. Antoniou, G. Flouris, S. Kotoulas, and L. McCluskey, "Large-scale Parallel Stratified Defeasible Reasoning," in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, Eds., vol. 242. IOS Press, 2012, pp. 738–743.

[12] I. Tachmazidis and G. Antoniou, "Computing the Stratified Semantics of Logic Programs over Big Data through Mass Parallelization," in *RuleML*, ser. Lecture Notes in Computer Science, L. Morgenstern, P. S. Stefaneas, F. Lévy, A. Wyner, and A. Paschke, Eds., vol. 8035. Springer, 2013, pp. 188–202.

[13] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*, R. E. Johnson and R. P. Gabriel, Eds. ACM, 2005, pp. 519–538.

[14] A. V. Gelder, K. A. Ross, and J. S. Schlipf, "The Well-Founded Semantics for General Logic Programs," *J. ACM*, vol. 38, no. 3, pp. 620–650, 1991.

[15] S. Brass, J. Dix, B. Freitag, and U. Zukowski, "Transformation-Based Bottom-Up Computation of the Well-Founded Model," *Theory and Practice of Logic Programming*, vol. 1, no. 5, pp. 497–538, 2001.

[16] S. Liang, P. Fodor, H. Wan, and M. Kifer, "OpenRuleBench: an analysis of the performance of rule engines," in *WWW*. New York, NY, USA: ACM, 2009, pp. 601–610.

[17] M. Gelfond, "Chapter 7 Answer Sets," in *Handbook of Knowledge Representation*, ser. Foundations of Artificial Intelligence, V. L. F. van Harmelen and B. Porter, Eds. Elsevier, 2008, vol. 3, pp. 285–316.

[18] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides, "A Formal Approach for RDF/S Ontology Evolution," in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, Eds., vol. 178. IOS Press, 2008, pp. 70–74.

[19] Y. Roussakis, G. Flouris, and V. Christophides, "Declarative Repairing Policies for Curated KBs," in *HDMS*, 2011.

[20] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "QbDJ: A Novel Framework for Handling Skew in Parallel Join Processing on Distributed Memory," in *HPCC*, 2013, pp. 1519–1527.

[21] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Efficiently handling skew in outer joins on distributed systems," in *CCGrid*, 2014, pp. 295–304.

[22] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and Efficient Large-Large Table Outer Joins on Distributed Infrastructures," in *Euro-Par*, 2014, pp. 258–269.

[23] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and Skew-resistant Parallel Joins in Shared-nothing Systems," in *CIKM*, 2014.