Exploration of MPI-backed Parallelization for Tableau-based Description Logic Reasoning

M. Hossain and W. MacCaull
Department of Mathematics, Statistics and Computer Science
St. Francis Xavier University, Antigonish, NS, Canada

Abstract—Description logic (DL) reasoning systems do not scale to the requirements of the rapidly growing amount of data. Although a lot of optimization techniques have been developed over the last decades, reasoning performance is still a bottleneck for users. Moreover, most modern reasoners consist of programs that run on a single machine. When the ontology is very large and complex, the computational resources of a single machine are not enough. Therefore, in order to achieve the vision of the semantic web, developing highly scalable and efficient ontology reasoner is crucial. In this work, we have investigated the potential of improving performance of a tableau-based reasoner via parallelization. In order to achieve practical scalability via parallelization, we have developed a parallel model based on the universal manager-worker model to check the consistency of a knowledge base. We have also implemented this model in a distributed memory environment using a Java message passing library for the DL $\text{ALC}$.

Keywords: parallel reasoning; description logic; high performance computing; message passing interface.

1. Introduction

With the explosive growth of data in the semantic web (SW), large and complex knowledge bases (KBs) are emerging day by day. At present, reasoning over such KBs has become one of the most challenging problems in SW applications. Although a good number of optimization techniques have been developed in the past decades, DL reasoning systems do not scale efficiently to deal with the rapid growth of data. Most optimization techniques have been investigated for sequential reasoners, and scalability (i.e., the ability to use additional computational resources to process larger KBs) of a sequential reasoner is limited by the physical resources of a single machine. Furthermore, most state-of-the-art reasoning systems are based on tableau algorithms and the high computational complexity (e.g., KB satisfiability for $\text{SHIQ}$ is ExpTime-complete) of tableau algorithms makes the process even more difficult. Therefore, in order to support the vision of the SW, developing a highly scalable and efficient ontology reasoner is crucial.

In the last decade, a few attempts have been made to parallelize DL reasoning, but most target a single machine (e.g., thread-level parallelism on multi-core systems) [1], [2], [3], [4]. The performance gain that can be achieved by this approach is limited by the number of available cores. Typically, the number of cores in such a machine is not higher than eight [5]. Moreover, for large or distributed ontologies, thread-based strategies are not suitable because they target a single machine. So, in order to reduce the processing time via parallelization, reasoning engines need to distribute their workload into different computational units. A distributed approach is potentially more scalable than a single machine approach because it can be scaled in two dimensions, namely enhancing the hardware performance of each node and increasing the number of nodes in the system.

The core function of a tableau based reasoner is checking the consistency of a KB, i.e., determining whether a given KB has a model. Therefore, we focused on parallelizing the consistency checking procedure. In this work, we developed a parallel consistency checking algorithm based on a well-known programming paradigm, the Message Passing Interface (MPI) [6]. We showed that independent tableau branches can be processed concurrently on independent processes. Our algorithm is based on a universal model namely manager-worker model. We implemented this parallel consistency checking algorithm for the DL $\text{ALC}$ using MPJ Express [7], an open source Java message passing library, and discussed our initial results by executing in both multi-core processors (shared-memory) and computer clusters (distributed memory). As far as we are aware, this is the first attempt to parallelize consistency checking in a distributed memory environment using MPI. This work is also significant to the high performance computing (HPC) community because it attempts to close the gap between Java and MPI.

The rest of this paper is organized as follows: Section 2 outlines a few related works, Section 3 describes the syntax and semantics, and the tableau algorithm for the DL $\text{ALC}$, Section 4 presents the parallel model to check the consistency of a KB, Section 5 describes the implementation and evaluation of this model for the DL $\text{ALC}$ by means of MPI, and finally, Section 6 summarizes the paper with future work.

2. Related Work

The DL community has already made some notable efforts on adopting the HPC paradigm in DL reasoning. A few of
them are related to our work; these are reported below.

Liebig and Muller [2] reported a parallel SHIQ reasoner named UUPR (Ulm University Parallel Reasoner). UUPR parallelizes the tableau algorithm itself by applying concurrent computation on disjunction and the at most number restriction rules. Although various optimization techniques have been adopted in UUPR, a few significant optimizations, e.g., GCI absorption, dependency directed backtracking, are absent. Moreover, UUPR is implemented as a shared memory program using the boost.Threads library and hence the speed up gain is limited by the physical architecture of a single machine.

Bao et al. [8] presented a distributed tableau algorithm using MapReduce [9]. Although MapReduce has proved to be efficient for large datasets on certain kinds of distributable problems, this technique requires a considerable re-thinking of the tableau algorithms in order to conform to the Map and Reduce steps. Therefore, we did not explore this approach.

Wu and Haarslev [10] developed a parallel tableau-based DL reasoner named Deslog for the DL \( \text{ALC} \). Deslog is a shared-memory parallel reasoner for TBox classification. Several optimization techniques were incorporated into Deslog, thus leading to good efficiency for TBox classification. However, the speed up that can be gained is limited by the available cores in the shared-memory environment.

The recently developed LarKC [11] platform provides a high level of flexibility, performance, and scalability for reasoning over large-scale semantic data sets [12]. This platform addresses the limitations of current semantic reasoning engines and enables reasoning with big data by distributing computation among nodes. For the most part, there are two proposed approaches in the literature: rule partitioning and data partitioning. In [5], Cheptsov described an MPI based approach for implementing parallel semantic web applications and evaluating the performance of random indexing over large text volumes on the LarKC platform.

Faddoul and MacCaull [13] outlined a fork/join parallel framework for handling non-determinism arising from algebraic tableau reasoning. This parallel framework allows the execution of non-deterministic rules on independent cores only (not on independent processes). To work with an algebraic reasoning component, a standard tableau calculus needs to be modified and extended.

An observation is that applying thread based strategies such as multi-threading in multi-cored processor is the easiest and simplest way to achieve the high performance [10]. However, speed up gained via thread-level parallelism is limited by the currently existing computer architecture. On the other hand, the process-based strategies, such as MPI, allow one to execute applications in distributed compute architectures such as compute clusters, grid systems, etc. The MPI is a well-known programming paradigm intended for programming in distributed memory environments. It is a high-level library for sending and receiving messages that is commonly used in HPC applications to abstract the underlying networking details. In this work, we introduce and discuss solutions for the implementation of a tableau algorithm with MPI [14].

3. Preliminaries

DLs are a family of knowledge representation formalisms suitable for representing the terminological knowledge in a wide range of applications. They can be used to represent the knowledge of an application domain in a structured and formally well-defined way. A KB of a typical DL system consists of a TBox and an ABox. The TBox introduces the terminology, i.e., the vocabulary of an application domain, while the ABox contains assertions about named individuals in terms of this vocabulary [15]. A DL system sets up KBs to do reasoning and manipulation of content. In this section we introduce the syntax, semantics, and the tableau algorithm for the DL \( \text{ALC} \).

3.1 Syntax and Semantics of \( \text{ALC} \)

\( \text{ALC} \) language is the smallest but relatively expressive propositionally closed DL. It is constructed from atomic concepts, atomic roles, \( \land \) (conjunction), \( \lor \) (disjunction), \( \neg \) (negation), \( \forall R.C \) (value restriction), \( \exists R.C \) (existential restriction).

Let \( N_C \) and \( N_R \) be non-empty and pair-wise disjoint sets of concept names and role names respectively. Let \( N_I \) be a set of all individual names. Below \( A \) is used to denote an atomic concept \((A \in N_C)\), \( R \) is used to denote an atomic role \(( R \in N_R)\). Concept descriptions in \( \text{ALC} \) are formed according to the syntax rule in (1), given in BNF form; where \( C, D \) are \( \text{ALC} \) concepts and \( \top \) (everything) and \( \bot \) (nothing) are the universal concept and bottom concept, respectively.

\[
C, D \to A \mid \top \mid \bot \mid C \land D \mid C \lor D \mid \neg C \mid \forall R.C \mid \exists R.C \quad (1)
\]

The formal definition of semantics of \( \text{ALC} \) is given by means of an interpretation \( \mathcal{I} \). An interpretation \( \mathcal{I} = (\Delta^\mathcal{I}, \mathcal{I}) \) consists of a non-empty set \( \Delta^\mathcal{I} \), called the domain of \( \mathcal{I} \), and a mapping function \( \mathcal{I} \), called the interpretation function of \( \mathcal{I} \), that maps:

- every individual name \( a \in N_I \) to an element, \( a^\mathcal{I} \), of \( \Delta^\mathcal{I} \) (i.e., \( a^\mathcal{I} \in \Delta^\mathcal{I} \))
- every concept name \( C \in N_C \) to a subset, \( A_C^\mathcal{I} \), of \( \Delta^\mathcal{I} \) (i.e., \( A_C^\mathcal{I} \subseteq \Delta^\mathcal{I} \))
- every role name \( R \in N_R \) to a subset, \( R^\mathcal{I} \), of \( \Delta^\mathcal{I} \times \Delta^\mathcal{I} \) (i.e., \( R^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I} \))

The interpretation function is extended to satisfy \( \text{ALC} \)-concept descriptions as follows:

\[
\begin{align*}
\top^\mathcal{I} &= \Delta^\mathcal{I}; & \bot^\mathcal{I} &= \emptyset; \\
(\neg A)^\mathcal{I} &= \Delta^\mathcal{I} \setminus A_C^\mathcal{I}; \\
(C \land D)^\mathcal{I} &= C^\mathcal{I} \cap D^\mathcal{I}; & (C \lor D)^\mathcal{I} &= C^\mathcal{I} \cup D^\mathcal{I}; \\
(\forall R.C)^\mathcal{I} &= \{ x \in \Delta^\mathcal{I} \mid \forall y \in \Delta^\mathcal{I} : \langle x, y \rangle \in R^\mathcal{I} \land y \in C^\mathcal{I} \}; \\
(\exists R.C)^\mathcal{I} &= \{ x \in \Delta^\mathcal{I} \mid \exists y \in \Delta^\mathcal{I} : \langle x, y \rangle \in R^\mathcal{I} \rightarrow y \in C^\mathcal{I} \}.
\end{align*}
\]

An \( \text{ALC} \) KB is a finite set of axioms formed by concepts, roles and individuals. A concept assertion is an axiom of
the form \( C(a) \) (\( a \) is an instance of \( C \)) and a role assertion is an axiom of the form \( R(a, b) \) (\( a \) is related to \( b \) via \( R \)), where \( a, b \) are individuals and \( C \) and \( R \) are concept and role, respectively. A concept inclusion is an axiom of the form \( C \sqsubseteq D \) means that concept \( D \) is more general than concept \( C \). An ABox is set of role assertions and concept assertions; a TBox is a set of concept inclusions.

An interpretation \( I \) satisfies a concept assertion \( C(a) \), denoted by \( I \models C(a) \), if and only if \( I \models R(a, b) \) if a role assertion \( R(a, b) \) is satisfied by \( I \). An interpretation \( I \) satisfies a concept inclusion \( C \sqsubseteq D \), denoted by \( I \models C \sqsubseteq D \), if \( I \) satisfies an ABox, \( A \), (written, \( I \models A \)) iff \( I \) satisfies every assertion in \( A \). If \( I \) satisfies \( A \), then \( I \) is called a model of \( A \). An interpretation \( I \) satisfies a concept inclusion \( C \sqsubseteq D \), denoted by \( I \models C \sqsubseteq D \), if \( C \subseteq D \) and it satisfies a TBox, \( T \), (written, \( I \models T \)) if \( I \) satisfies every inclusion in \( T \). If \( I \) satisfies \( T \) then \( I \) is called a model of \( T \).

An interpretation \( I \) is a model of a KB, \( K = \{ T, A \} \), denoted by \( I \models K \), iff \( I \) is a model of both \( T \) and \( A \). A concept \( C \) is satisfiable with respect to (w.r.t.) a TBox \( T \) iff there exist a model \( I \) of \( T \) such that \( C \not\sqsubseteq \emptyset \). An ABox \( A \) is consistent w.r.t. a TBox \( T \), if there is an interpretation that is a model of both \( A \) and \( T \); \( A \) is inconsistent otherwise.

### 3.2 Tableau Algorithm

State of the art DL systems typically use tableau algorithms to decide satisfiability (consistency) of a KB. The standard tableau algorithm [16] generally contains the following main elements:

- A completion forest (called tableau) that represents a model of the DL language; such a completion forest typically has the tree model property.
- A set of tableau expansion rules to construct a completion forest.
- A set of blocking rules to guarantee termination.
- A set of clash conditions to detect logical contradictions.

For an ALC KB, \( K = \{ T, A \} \), the algorithm will construct a model for both \( T \) and \( A \) to check the consistency of \( K \). If one such model (i.e., a completion forest) is found, \( K \) is consistent, otherwise \( K \) is inconsistent.

In order to work efficiently, it is necessary to transform a KB into the Negation Normal Form (NNF), i.e., the negation only occurs in front of concept names. A KB is in NNF if all concept descriptions in it are in NNF. Each concept description can be transformed to NNF by pushing negations inwards using the following equivalences:

\[ \neg C \equiv \neg \neg C; \quad \neg (C \sqcap D) \equiv \neg C \sqcup \neg D; \quad \neg (C \sqcup D) \equiv \neg C \sqcap \neg D; \quad \neg \exists R.C \equiv \forall x.\neg C; \quad \neg \forall R.C \equiv \exists x.\neg C. \]

Reasoning w.r.t. a TBox \( T \) can be reduced to reasoning w.r.t. an empty TBox by a process called internalization [15].

Given a \( T \), a concept \( C_T \) is defined as

\[ C_T := \bigwedge_{C \sqsubseteq D, e \in T} (\neg C \sqcup D). \]

Any individual \( x \) in any model of \( T \) will be an instance of \( C_T \) in that model [16].

Consistency checking is one of the main inference problems to which all other inferences can be reduced [15]. The main idea of tableau-based approaches for deciding the consistency of an ABox is as follows: the algorithm starts with the input ABox, \( A \) and applies consistency preserving expansion rules (e.g., the expansion rules for ALC-ABoxes are presented in Table 1) until no more rules are applicable (the tableau is complete) or an obvious contradiction (called a clash) is found. If a complete and clash-free tableau is obtained, \( A \) is consistent; otherwise it is inconsistent. For comprehensive background on tableau calculus, the reader is referred to [15], [17], [18].

#### Table 1: ALC-tableau expansion rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall )-rule</td>
<td>If ( C_1 \sqcap C_2(z) \in A ) and ( { C_1(z), C_2(z) } \notin A ), then ( A' := A \cup { C_1(z), C_2(z) } ).</td>
<td></td>
</tr>
<tr>
<td>( \exists )-rule</td>
<td>If ( \exists R.C(z) \in A ) and there is no ( y ) such that ( { C(y), R(x, y) } \in A ), then ( A' := A \cup { C(z), R(x, y) } ) such that ( z ) is a fresh individual.</td>
<td></td>
</tr>
<tr>
<td>( \sqcap )-rule</td>
<td>If ( \forall R.C(x, y) \in A ) and ( C(y) \notin A ), then ( A' := A \cup { C(y) } ).</td>
<td></td>
</tr>
</tbody>
</table>

### 4. Parallelization

#### 4.1 Parallel Consistency Checking

Tableau algorithms are amenable to parallelization due to the existence of inherently non-deterministic rules (e.g., disjunctions rule, qualified cardinality restriction rule, and choose rule for the DL SHIQ). The application of a non-deterministic rule yields multiple alternatives, which can be treated as different possible ABoxes to continue reasoning with. Since there is no dependency between the alternatives, they can be processed concurrently. For example, if we have an ABox, \( A = \{ a : C \sqsubseteq D \} \), then the disjunction rule generates two ABoxes \( A_1 = A \cup \{ a : C \} \) and \( A_2 = A \cup \{ a : D \} \), which are independent, i.e., the consistency checking of one ABox does not depend on others.

In order to work efficiently, all tableau expansion rules are categorized into two main groups: deterministic and non-deterministic. If a deterministic rule is applicable then the original ABox, \( A \), is transformed into a new ABox, \( A_1 \). Then \( A \) is consistent if the expanded ABox, \( A_1 \), is. On the other hand, if a non-deterministic rule is applicable, then the original ABox, \( A \), is transformed to a set of new ABoxes, \( S = \{ A_1, A_2, ..., A_k \} \), instead of a single ABox. Then \( A \)
is consistent if there is some \( i, 1 \leq i \leq k \), such that \( A_i \) is consistent. Typically, consistency checking is performed on the set of ABoxes sequentially, i.e., one after another. Since the ABoxes \( \{A_1, A_2, \ldots, A_k\} \) are independent, it is possible to perform the consistency checking of these ABoxes concurrently.

### 4.2 Manager-Worker Model

We propose an MPI-backed parallel algorithm for checking the consistency of a KB. Our algorithm is based on one of the most universal manager-worker or task parallelism approaches. The manager-worker pattern is a variant of the master-slave pattern where node-0 is the manager (master) and all other nodes are workers (slave nodes). The variation is based on the fact that components of this pattern are proactive rather than reactive [19]. Each processing unit performs the same operations simultaneously and independently of the processing activity of other units. The manager and worker algorithms are presented below in Algorithm 1 and Algorithm 2, respectively.

![Fig. 1: Manager-worker model.](image)

In manager-worker model, the manager maintains a work pool, a set of work items, and upon request the manager sends the next available work item to a worker (Lines 15–17, Algorithm 1). Each worker processes one work item at a time and, when finished, requests the manager for a new work item. This continues until there are no work items left. When all work items are complete, the manager tells the workers to stop (Lines 27–30, Algorithm 1). Figure 1 shows the manager-worker parallel computation model. Here, the manager executes a different algorithm from that of the worker. Though manager and worker execute different algorithms, we combine both manager and worker routines into a single program which is more convenient, efficient and also supported by all implementations of MPI.

Our algorithm follows a self-scheduling approach where the manager maintains a work pool. Each work item is an ABox, which consists of a set of concepts and expansion rules. The consistency checking algorithm for an ABox is provided in Algorithm 3. When a worker receives an ABox (Line 8, Algorithm 2), it applies consistency preserving expansion rules until no more rules are applicable or a

**Algorithm 1** mpiOWL manager

- **Initialize:**
  1. \( ptr \leftarrow 1 \)
  2. \( A \leftarrow \{A\} \)
  3. \( closed.A \leftarrow \emptyset \)
  4. isClosed \( \leftarrow \text{false} \)
  5. isComplete \( \leftarrow \text{false} \)

- **Ensure:** \( isComplete = \text{true} \lor \text{isClosed} = \text{true} \)

- **while** \( \neg \text{isComplete} \lor \neg \text{isClosed} \) **do**
  6. Receive a message from a worker \( W_j \)
  7. if message is a state request then
     8. if \( ptr \leq |A| \) then
        9. Send the state \( \text{CHECK} \) to the worker \( W_j \)
     else
        10. Send the state \( \text{WAIT} \) to the worker \( W_j \)
     end if
  11. else if message is an ABox request then
     12. Send ABox, \( A_{ptr} \), to the worker \( W_j \)
     13. \( D_j \leftarrow A_{ptr} \)
     14. \( ptr \leftarrow ptr + 1 \)
     15. else if message is a newly generated ABox then
        16. \( A \leftarrow A \cup \{\text{message}\} \)
     17. else if message is a CLASH report then
        18. \( closed.A \leftarrow closed.A \cup \{D_j\} \)
     19. else if message is a COMPLETE status then
        20. isComplete \( \leftarrow \text{true} \)
     end if
     21. isClosed \( \leftarrow (|A| = |closed.A|) \)
  22. end while
  23. for all \( W \in W \) do
     24. Receive a message from the worker \( W_j \)
     25. Send the state \( \text{STOP} \) to the worker \( W_j \)
  26. end for
  27. if \( isComplete = \text{true} \) then
     28. \( A \) is Consistent.
  29. else
     30. \( A \) is Inconsistent.
  31. end if
5. Experiments

5.1 MPI Libraries in Java

MPI is a standardized and portable message-passing system that follows a process-oriented parallel computing paradigm. It is based on a Single Program Multiple Data (SPMD) execution model. Although there is no official MPI binding for Java, there exist several projects (e.g., mpiJava, JavaMPI, MPJ) that provide required functions with different degrees of success and compatibility [7]. Most of these projects are prototype implementations, without any maintenance. Currently, the most successful ones in terms of uptake by the HPC community are mpiJava and MPJ Express, which we now present.

**mpiJava**: The mpiJava project in late 1997 by Carpenter et al. [20]. The implementation of mpiJava is through Java Native Interface (JNI) wrappers to native MPI software. It takes an existing native MPI implementation and provides Java wrappers through the JNI which is a mechanism that allows the application programmer to call native subroutines and libraries (written in other languages such as C, C++) from Java and vice versa. In mpiJava, every MPI process or node corresponds to a single JVM, running on the same host computer. The mpiJava is implemented on top of a native MPI and most native implementations of MPI are not thread-safe. Therefore, it is not possible to perform MPI operations concurrently for more than one thread in a single JVM.

**MPJ Express**: MPJ Express² is a message passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers [7]. Although MPJ Express is designed for distributed memory environments like networks of computers or clusters, it can execute parallel programs efficiently in a multi-core processor with a shared memory environment. The MPJ Express software can be configured in two ways: (1) multi-core configuration and (2) cluster configuration. The multi-core configuration is used by the developers who want to execute their parallel Java applications on multicore processors. The cluster configuration is used to execute

---


parallel Java applications on distributed memory platforms including clusters and network of computers.

Both mpiJava and MPJ Express provide message passing functionality to the Java programmers. But mpiJava can incur a noticeable overhead for large messages and also presents some portability and instability issues. There is no recent update in mpiJava and it only supports some native MPI implementations. On the other hand, MPJ Express is maintained regularly and most recent version was released on April 18, 2015. We therefore choose to use MPJ Express to implement our parallel consistency checking task.

5.2 Parallel Implementation for the DL ALC

There is an important property in ALC tableau algorithm which makes the algorithm pleasingly parallel. The whole knowledge necessary for node expansion or for clash detection is contained in a given node. Therefore, there is no information exchange between nodes belonging to different branches. As a result, branches of the tableau can be constructed independently of one another. The parallelization strategy described in this paper takes advantage of this property. The standard tableau expansion rules for the DL ALC are presented in Table 1. Note that only the $\sqcap$-rule is non-deterministic in the ALC tableau algorithm.

In this work, the manager-worker model is implemented based on Pellet by means of MPI as a distributed memory program using MPJ Express library. Pellet [21] is a tableau-based OWL-DL reasoner developed by the Mind Swap group. It is an open source reasoner and developed in Java. In MPI programming, all processes execute the same program executable and each process is identified by means of a special process identifier, called rank, which is unique within a group of processes involved in the execution. The rank allows every process to identify what part of the data to be processed. In order for two processes to communicate, we use MPI blocking send and receive operations. The blocking operations do not return until the communication is finished.

There are many technical challenges in implementing this dynamic manager-worker algorithm in Java by means of MPI. One of the major challenges is passing an object (e.g., an ABox) from one process to another. As the object is not a primitive data type, to pass an object using MPI, all classes of that object must implement the Serializable interface. Since we are working on legacy code, it is not feasible for every class to implement the Serializable interface. Moreover, standard Java serialization is inefficient both in terms of speed and size. To deal with these problems, we converted an object to byte vectors using Kryo\(^3\), a fast and efficient serialization framework for Java, and sent these byte vectors using the same method as primitive byte buffers. At the receiving end, the object is reconstructed using these byte buffers.

5.3 Evaluation

In the manager-worker model described in the previous section, the manager is dedicated to distributing work items to workers and does not itself do any computation. Consequently, if there are $p$ processes, only $p-1$ processes are available to process the computation tasks, i.e., to perform the consistency checking. Therefore, maximum parallel efficiency can be obtained in this scheme is \((p-1)/p \times 100\%\).

In order to remove this limitation, it is possible to ask the manager to participate in the computation as do other workers. In that case, all $p$ processes will participate in the computation, but the manager may be less likely to be available to respond instantly to the worker’s requests. If there is a very large number of workers, the processing of requests for the work item on the manager may become a bottleneck. If $t_{chk}$ is the average time required to perform consistency checking on a given ABox on a worker, and $t_{req}$ is the time required to process a request for a work item on the manager, then the manager can process $t_{chk}/t_{req}$ requests without keeping any worker waiting. So, the maximum number of workers that the manager can support efficiently in this scheme is $t_{chk}/t_{req}$. The time required to perform consistency checking on a given ABox can vary from one ABox to another. However, $t_{chk}$ is sufficiently large compared to $t_{req}$. So we can expect that this scheme will perform efficiently for a large number of processes. The efficiency of this scheme also depends on the existence of non-determinism in a KB. Typically, the number of disjunctions exist in a KB is very large compared to $p$. For example, Thesaurus ontology contains 83,644 subsumption axioms and 10,242 equivalent axioms. Since each equivalence axiom can be replaced by two subsumption axioms, there are approximately 105,000 disjunctions in total. The possible advantage of parallelism increases for more expressive fragments, where there are more non-deterministic rules.

It is noted that no optimization techniques have been addressed in this manager-worker model. As it takes much time to develop a new reasoner, we implemented our manager-worker model on top of Pellet. We conducted our experiments on the ACENET\(^4\) cluster both in shared memory (using multi-core configuration of MPJ Express) and distributed memory (using cluster configuration of MPJ Express) environments. As is observed in the Table 2, our preliminary results are not encouraging. The main reason for performance degradation is the absence of optimizations. Pellet implements most state-of-the-art optimization techniques whereas there are no optimizations in our model. Dependency-directed backtracking, also known as backjumping, is the most significant among them. Backjumping allows an algorithm to detect the source of a clash and prune the search space to avoid facing the same clash again. When we distribute the computation into different computa-

---

\(^3\)https://github.com/EsotericSoftware/kryo

\(^4\)http://www.ace-net.ca/
Tables and figures

Table 2: Consistency test result for the Transportation ontology.

<table>
<thead>
<tr>
<th>Cluster configuration</th>
<th>Multi-core configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No of Worker</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

6. Conclusions and Future Work

With the progress of semantic web technologies, knowledge bases are becoming larger and more complex. Reasoning with large and complex ontologies is one of the biggest challenges for DL reasoners. In this work, the potential for improving the scalability of a DL reasoner via parallelization was investigated. A parallel model was developed for handling non-determinism arising from tableau-based reasoning. This parallel model allows the execution of non-deterministic rules on independent processes. The parallel model was implemented to check the consistency of a KB in a distributed memory environment using MPI. Even though the model was implemented for the DL ALCOIQ, the provided algorithm is applicable for the whole DL family including SROIQ. For the parallel implementation, MPJ Express, a Java MPI library, was used. In this work, the process-based programming model is applied to Java, which brings the parallel computation paradigm, i.e., MPI, closer to Java.

Most DL reasoners implement tableau algorithms with a set of optimization techniques. State-of-the-art optimization techniques are keys to the performance of a modern tableau-based reasoner. So, a parallel model should address the main optimizations to achieve the high performance. Currently, we are in the process of implementing this parallel model with a set of optimizations, namely dependency-directed backtracking, semantic branching, etc., in order to get better performance. We also plan to implement this model for an expressive DL, e.g., SHIQ, in the near future.

Acknowledgments: The second author wishes to thank the Natural Sciences and Engineering Research Council of Canada for financial support.

References