Distributed Deductive Closure Computation

Maximilian Wenzel

Institute of Artificial Intelligence, University of Ulm, Germany,
maximilian.wenzel@uni-ulm.de

Abstract. Since the year 2010, the single-thread performance in processors does not increase as much as in the decades before[1], e.g., due to heat dissipation problems in the increasingly smaller components. Therefore, programs are now primarily sped up by executing the appropriate subproblems in parallel using multiple cores or processors. One of the common approaches to parallelize an application is the MapReduce programming model. In this case, a control node distributes data to a set of *Mapper* workers that create work packages. These work packages are in turn propagated to the *Reducer* workers that process them in some predefined manner. If the reducers, however, produce intermediate results that are in turn relevant to other reducers, this approach might be suboptimal since these results have to be directed over the control node and the mappers again.
Such intermediate results exist for example in the parallelized deductive closure computation. In this particular problem, we have given a set of axioms, i.e., true statements, and rules that can be used to derive conclusions, i.e., new axioms, from the initial ones. For a given set of rules and a set of initial axioms M, the deductive closure is then computed in the following manner: All possible conclusions for the axioms in M and the rules are derived. Subsequently, these conclusions are added back to M. This procedure is repeated until M does not change anymore and, eventually, M represents the deductive closure of the initial set of axioms. For the parallel computation of the deductive closure with multiple workers, the MapReduce approach might not be best suited since newly derived conclusions on a given worker often have to be distributed to other workers to ensure the correct outcome.
In this project, we examined, therefore, an alternative approach, where individual worker nodes communicate with one another in a peer-to-peer fashion. In order to determine whether all worker nodes found a fixpoint, we developed a convergence protocol which uses counters for the total number of sent and received axioms in the procedure. In our benchmark on a single processor with 8 cores, we compared our distributed implementation against other baseline approaches in Java. In the echo benchmark and transitive closure benchmark for directed graphs, we achieved maximum speedups of 6.2 and 3.3, respectively, against the single-threaded variant. The evaluation might indicate promising results if our approach is executed on a High Performance Computing (HPC) cluster for related reasoning problems.

[1] https://github.com/karlrupp/microprocessor-trend-data

1 Introduction

Algorithms and applications, in general, are parallelized in order to speed up the execution of all required tasks that constitute the given problem. One of the established approaches to parallelize programs is the *MapReduce* model [6, 19]: The MapReduce programming model consists of two phases as its name implies, namely, the *Map* phase and the *Reduce* phase. At the beginning, a given control node distributes data to the *mapper* worker nodes, which generate from the data key-value pairs in parallel: A *value* corresponds to some given date and the *key* determines which *reducer* worker node of the next phase is responsible for the given value. Thus, the resulting key-value pairs are propagated from the mappers to the appropriate reducers of the next phase, which process the key-value pairs in some predefined way. Eventually, the results of all reducers are collected on the control node again. All these steps combined represent a single iteration in the MapReduce programming model.
The MapReduce approach is most suitable if the problem can be solved in a single iteration. For instance, a typical problem that can be efficiently solved using the MapReduce model is the counting of word occurrences in a set of documents. In this case, each mapper gets assigned by the control node a subset of documents and generates appropriate key-value pairs, where the keys correspond to the words and the values contain the number of occurrences of a given word. Afterwards, these word counts are propagated to the reducers, which are responsible for a predefined range of words, for instance, in case of two reducers r_1, r_2, r_1 receives all words that begin with the letters *a*-*m* and r_2 gets the remaining words. The reducers now aggregate all key-value pairs with the same key and sum up the corresponding counts. Finally, the merged results are collected by the control node again.
The described problem requires only a single MapReduce iteration. If, however, in other problems intermediate results from a given reducer might be relevant to another reducer, these results can only be exchanged after they have been collected by the control node and a new iteration has been initiated. Thus, multiple iterations might be required to compute the correct outcome. A problem, which often requires multiple iterations using the MapReduce approach, is the parallelized computation of the deductive closure. In this problem, we have given a set of initial axioms, i.e., true statements, and rules that can be used to derive new axioms, i.e., conclusions, from the given axioms. To compute the deductive closure computation for a given set of axioms M, we start applying all rules to the axioms in M to derive all possible conclusions. Afterwards, these conclusions are added again back to the set M. This procedure is repeated until the set M does not change anymore and, if that is the case, M represents the deductive closure of our initial set of axioms. In case of the parallelized computation of

the deductive closure, each worker is usually responsible for different axioms or rules. Therefore, newly derived conclusions from a given worker might be relevant to other workers and must be distributed to them in order to ensure that all possible conclusions are indeed derived. For the exchange of conclusions, the MapReduce model is however suboptimal since usually the *reducer* workers apply the rules to the received axioms and the corresponding conclusions from a given reducer can only be distributed to the other reducers in the following iteration [19].
Therefore in this project, we examined an alternative approach, where all involved workers communicate in a peer-to-peer fashion. Newly generated conclusions can thus be directly exchanged between the workers over the network and the procedure does not rely on multiple iterations as it is the case in the MapReduce model. Our goal in this project was to implement such a distributed approach and find out whether it can compete with other related variants.
In Section 2, we start off with an introduction to our terminology and the fundamentals, which are essential for the further understanding. Afterwards in Section 3, we discuss the architecture of our system for the distributed deductive closure computation, i.e., the design of the control node, the worker nodes, and the deployed networking component. Furthermore, we introduce our convergence protocol that determines if all worker nodes converged in order to terminate the procedure. In Section 4, we present the results of our echo benchmark and transitive closure benchmark, where our distributed implementation is compared to other baseline approaches. Subsequently in Section 5, we consider other related parallelized procedures for computing the deductive closure for different problems and Section 6 eventually concludes our paper.

2 Preliminaries

In the following, we present and define all concepts which are essential for the further understanding. We begin with the fundamentals for the deductive closure computation procedure [8, 12, 15] and subsequently introduce possible parallelization approaches in this regard [17].

2.1 Terminology

Let L be the the alphabet of first-order logic which comprises the set of logical and non-logical symbols. The logical symbols comprise variables, logical operators, quantifiers, and the equality symbol, whereas the non-logical symbols consist of the constant symbols, function symbols, and predicates. Furthermore, let $\mathbb{D}$ be the domain of discourse, i.e., the set of objects over which certain variables may range. The logical symbols are then defined as follows:

- **Variables**: represent place holders for objects from the domain under consideration
- **Logical operators**: negation $\neg$, conjunction $\wedge$, disjunction $\vee$, implies $\rightarrow$, and equivalence $\iff$

- **Quantifiers**: existential quantifier $\exists$, universal quantifier $\forall$
- **Equality symbol**: indicates an equality relation using the symbol $=$

On the other hand, the set of non-logical symbols are defined in the following way:

- **Constant symbols**: Represent fixed objects from the domain $\mathbb{D}$.
- **Function symbols**: Stand for n-ary functions which take n objects as argument and return one object as value.
- **Predicates**: Predicates describe fixed, n-ary relations between objects from the given domain.

Definition 1 (Terms). *Each first-order variable and constant is a term. Let t_i, $i \in \{1, \ldots, n\}$ be arbitrary terms and F an n-ary function, then the resulting value of $F(t_1, \ldots, t_n)$ is also a term.*

Definition 2 (Formulae).

(F0) If t_1 and t_2 are terms, then $t_1 = t_2$ is a formula.
(F1) If $t_1, \ldots, t_n$ are terms and P is an n-ary predicate, then $P(t_1, \ldots, t_n)$ is a formula.
(F2) If ϕ is a formula, $\neg\phi$ is a formula.
(F3) If ϕ, ψ are formulae, then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, and $\phi \iff \psi$ are formulae.

The formulae of $(F0)$ and $(F1)$ are also denoted as *atomic formulae*. Furthermore, atomic formulae which are either negated or not represent *literals*.
Let ϕ, ψ be formulae and x a variable. We say x is in the *range* of a logical quantifier if ϕ is of the form $\exists x\psi$ or $\forall x\psi$. All occurrences of the variable x are *bound* by the innermost quantifier in whose range they occur. If a variable x does not occur in the range of a quantifier, we say x is *free*.
If t is an arbitrary term, $\phi\theta$ with $\theta = \{\, x/t \,\}$ represents a *substitution* where all free instances of x are replaced by the term t in the formula ϕ. The substitution is *admissible iff* no occurrence of x in the original formula ϕ is in the range of a quantifier that binds a variable occurring in the substituted term t.
Given a formula $\phi = \forall x\psi$, we call a *universal instantiation* the substitution of all free occurrences of the variable x in ψ by the term t to derive the new formula $\psi\theta$, where $\theta = \{\, x/t \,\}$. In our particular use case, we allow in the substitution θ only *ground terms*, i.e., terms which do not contain any variables.

Definition 3 (Rules and Axioms). *Let $x_1, \ldots, x_n$ be variables and p, $q_1, \ldots, q_m$ literals which may contain these variables. We write the Horn clause $\forall x_1 \ldots \forall x_n (p \vee \neg q_1 \vee \neg q_2 \vee \cdots \vee \neg q_m)$ as $p :\!- q_1, q_2, \ldots, q_m$ and label it as rule. We call p the head of the rule and $q_1, \ldots, q_m$ the body of the rule. Given a set of literals M, a rule can be read as an inverted implication: If for the literals $q_1, \ldots, q_m$ a universal instantiation exists such that these literals are contained in M, all premises of the rule are satisfied and we deduce p as a conclusion. All literals that are used in a rule or deduced from it are called axioms. If an axiom p can be deduced from a set of axioms M with a set of rules R, we write $M \vdash_R p$.*

Given an axiom $\alpha \in M$, and a rule ℓ, we *apply* the rule ℓ to α under the set M in the following way: Initially, we attempt to find a universal instantiation for the rule body such that at least one of the axioms q_i with $i \in \{ 1, \ldots, m \}$ corresponds to α. Subsequently, for all remaining free variables, we use all possible universal instantiations such that $\forall i \in \{ 1, \ldots, m \} : q_i \in M$ in order to deduce all possible conclusions p with the rule ℓ.

Definition 4 (Deductive Closure). *Let R be a set of rules and let A, M be sets of axioms. The set M is deductively closed if all conclusions p that can be derived from M with the rules R are also contained in M, i.e., $M \vdash_R p \implies p \in M$. The set M is called the* deductive closure *of A if it is the smallest superset of A that is deductively closed.*

2.2 Deductive Approaches

A *logic program* consists of a set of rules R and a set of initial axioms A. In principle, there are two different deductive approaches for a given logic program, namely, the *top-down / backward chaining / SLD-resolution* and the *bottom-up / forward chaining / fixpoint computation / deductive closure computation / materialization* approach. In our particular case, we consider only the forward chaining method which aims to derive the deductive closure of the initial axioms A. In the following sections, we take a closer look at the semi-naive deductive closure computation algorithm [3, 11, 16].

Semi-naive Deductive Closure Computation In case of the semi-naive deductive closure computation, we have given a set of rules R, a set of initial axioms A, an empty to-do queue, and an empty set of axioms M. The procedure is presented in Algorithm 1. At the beginning, all initial axioms are added to the to-do queue. The semi-naive algorithm then takes an axiom α from the queue and checks if it is contained in M. If $\alpha \in M$, we discard α and take the next axiom from the queue. Otherwise, we apply all rules to α under the set M and, subsequently, add the derived conclusions back to the queue. This procedure is repeated until the queue is empty, i.e., when all given axioms and derived conclusions are contained in M. Eventually, M represents the deductive closure of our initial set of axioms A.
A concrete example for the computation of the deductive closure using the semi-naive algorithm is presented in Figure 1. In case of a directed graph, where individual vertices are represented by integers, we can use the predicate $edge(n_1, n_2)$ in order to indicate that a directed edge exists from vertex n_1 to n_2. On the other hand, the predicate $path(n_1, n_2)$ states that n_2 is reachable from n_1 over a sequence of edges in the directed graph. Given an initial set of edges, we can, therefore, use the following rules to compute all transitive paths in the graph:

$$path(x, y) \coloneq edge(x, y). \tag{1}$$

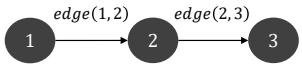
$$path(x, z) \coloneq path(x, y), edge(y, z). \tag{2}$$

Algorithm 1 Semi-naive deductive closure computation procedure

```
Input  : A set of rules R, a set of axioms A
Output: The deductive closure of the initial axioms A and the rules R
M, ToDo ← ∅
foreach a ∈ A do ToDo.add(a)
 while ToDo ≠ ∅ do
    α ← ToDo.takeNext()
    if α ∉ M then
       M.add(α)
       foreach ℓ ∈ R do
          conclusions ← apply rule ℓ to axiom α under M
          ToDo.addAll(conclusions)
       end
    end
end
return M
```

In our example from Figure 1, we start with our initialized to-do queue [$edge(1,2)$, $edge(2,3)$], which contains all edges from the given graph. In the first step, we take the axiom $edge(1,2)$ out of the queue and apply all rules to it since the axiom is not contained in M yet. From the first rule, we derive the conclusion $path(1,2)$, which is in turn added back to the to-do queue. Analogously, we take $edge(2,3)$ from the queue, add it to M and derive the conclusion $path(2,3)$. Now, from the next axiom $path(1,2)$, we can use the second rule and the axioms in M in order to derive $path(1,3)$. For all remaining axioms in the queue, we cannot derive any new axioms and the procedure eventually converges. The set M now represents the deductive closure of our initial axioms.

Directed Graph:



Rules:

$path(x, y) :- edge(x, y).$
$path(x, z) :- path(x, y), edge(y, z).$

Initialized To-Do: [edge(1, 2), edge(2, 3)]

Iteration	Current	To-Do	Set of Axioms *M*	Conclusions
1	edge(1, 2)	[edge(2, 3)]	{ edge(1, 2) }	edge(1, 2) → **path(1, 2)**
2	edge(2, 3)	[path(1, 2)]	{edge(1, 2), edge(2, 3) }	edge(2, 3) → **path(2, 3)**
3	path(1, 2)	[path(2, 3)]	{edge(1, 2), edge(2, 3), path(1, 2) }	path(1, 2), edge(2, 3)→ **path(1, 3)**
4	path(2, 3)	[path(1, 3)]	{edge(1, 2), edge(2, 3), path(1, 2), path(2, 3) }	
5	path(1, 3)	[ ]	{edge(1, 2), edge(2, 3), path(1, 2), path(2, 3), path(1, 3) }	

Fig. 1: Example for the computation of the transitive closure of a given directed graph by deploying the semi-naive deductive closure computation algorithm

2.3 Parallelized Deductive Closure Computation

In the following sections, we first describe the possible architectures, which can be deployed in the parallelized deductive closure computation. Afterwards, we describe two partitioning strategies, namely, data partitioning and rule partitioning [17]. The partitioning strategies are used in order to subdivide the deductive closure computation procedure into subproblems, which are subsequently executed in parallel. Finally, we explain how the parallelized procedures relate to shared and non-shared memory environments and describe the associated challenges.

Parallelization Architectures In the parallelized deductive closure computation, we distinguish between multiple *worker nodes*, which derive new conclusions in parallel, and a *control node*, which initiates and coordinates the appropriate worker nodes. There exist two concrete architectures in order to parallelize the deductive closure computation procedure, namely, *farm parallelism* and *communicating objects parallelism* [4]. In both cases, the *control node* initially distributes the axioms and rules to the *worker nodes*. Afterwards, the worker nodes start deriving new conclusions from the rules and given axioms. Meanwhile, they may produce conclusions that might be relevant to other workers. In case of *farm parallelism*, the worker nodes do not communicate with one another and the communication is, therefore, always directed over the control node, which may lead to a communication bottleneck. An alternative is thus the concept of

object communicating parallelism, where all workers are initialized at the beginning by the control node but afterwards communicate directly with one another in order to exchange newly derived conclusions. In this project, we lay our focus on the *object communicating parallelism* architecture.

Data Partitioning In data parallelism, a single task is parallelized by partitioning the original data space into subspaces and executing the same task by distinct workers in parallel on these different subspaces [5]. The individual results by the workers are eventually aggregated on the control node in order to obtain the final outcome. In case of our deductive closure computation algorithm, the data space corresponds to all initial axioms and all conclusions, which may be deduced with the given rules. For instance, when we compute the transitive closure of a directed graph with the node IDs $N = \{ 1, 2, 3 \}$, the overall data space corresponds to $\{ edge(x, y) \mid x,y \in N \} \cup \{ path(x, y) \mid x,y \in N \}$.
On the other hand, the single task, which is executed by each worker, is an adjusted variant of our semi-naive procedure as depicted in Algorithm 2. In comparison to the usual semi-naive procedure, the workers do not compute the deductive closure for their respective subspace of axioms since the newly derived conclusions are only added back to the to-do if they are contained in their assigned subspace. If a new conclusion is relevant to other workers, it is added to the appropriate to-do queue, i.e., in practice, it is either added directly to the queue or sent over the network to the respective worker. Note that the to-do queue can now also contain a termination symbol that is used by the control node in order to terminate the procedure.

Workload Distribution in Data Partitioning In order to ensure that all possible conclusions are indeed derived, the basic principle of data partitioning is that any two axioms which might join in a rule application process must be present on the same partition [17]. For instance in our running example from Figure 1, the variable y is present in both premises of the second rule and, therefore, the term y represents a join term. Now, we divide the set of all possible constants from N for our join term y into subsets N_{p_i} in order to partition the workload between the workers. In our example, the graph contains the node IDs $\{1, 2, 3\}$ and we assume to have two workers available. One option is, therefore, to assign to Worker 1 all even node IDs N_{p_1} and to Worker 2 all odd IDs N_{p_2}. Each worker is thus responsible for a subspace of axioms, i.e., all axioms $\{ path(x, y) \mid y \in N_{p_i} \} \cup \{ edge(x, y) \mid x \in N_{p_i} \}$ with $i \in \{1, 2\}$. In case of more than two workers, a common partitioning method is the *hash based partitioning approach* which has been deployed, e.g., by Dean and Ghemawat in a MapReduce implementation [6]. The approach is based on hash functions, which are generally used to obtain a fixed length output of n bits from an input of arbitrary length, such as a string. In case of hash based partitioning, we consider the value of the appropriate join term, e.g., t_1 in case of $edge(t_1, t_2)$, and use a hash function h to compute $(h(t_1) \mod n) + 1$, where n represents the total number of desired partitions. The result then corresponds to the ID of the responsible worker for the given

Algorithm 2 Procedure for a single worker in case of the parallelized semi-naive deductive closure computation

Input: A set of rules R and a to-do queue that is filled by the same worker, by the control node or other workers with axioms or a termination symbol. If the to-do queue is empty and an element is requested from it, the operation blocks until an element is added to the queue. Furthermore, a function w which takes an axiom and returns the set of responsible workers for the respective axiom.

Output: A set of axioms M that contains all axioms that have been added to the to-do at least once.

```
M ← ∅
while true do
    α ← ToDo.takeNext()
    if α is termination symbol then return M
    if α ∉ M then
        M.add(α)
        foreach ℓ ∈ R do
            conclusions ← apply rule ℓ to axiom α under M
            foreach c ∈ conclusions do
                foreach responsibleWorker ∈ w(c) do
                    responsibleWorker.ToDo.add(c)
                end
            end
        end
    end
end
```

axiom. The disadvantage of the hash based partitioning approach is that it does not consider the associated workload of an appropriate join term, which may lead to an unbalanced workload distribution between the workers. Therefore, more sophisticated partitioning algorithms for the occurring join terms are often deployed to obtain a more balanced workload distribution. However in this case, all appropriate join term partitions N_{p_i} have to be distributed at the beginning by the control node to all workers since each worker eventually has to determine for a newly derived conclusion all responsible workers.

Data Partitioning Example In Figure 2, we apply the data partitioning method to our running example in order to compute the transitive closure in parallel with two workers. Since only two workers are deployed, Worker 1 gets all odd node IDs and Worker 2 all even join node IDs assigned. As the procedure starts, the control node distributes the initial axioms $edge(1,2), edge(2,3)$ to the responsible workers by considering the relevant join terms of the axioms. Since $edge(1,2)$ has the odd join node 1, it is added to the to-do queue of Worker 1. On the other hand, because $edge(2,3)$ has the even join node 2, it is assigned to Worker 2. Afterwards, the workers deduce now in parallel the conclusions $path(1,2)$ and $path(2,3)$ with the first rule. Since $path(2,3)$ has the odd join term 3, it is sent from Worker 2 to Worker 1 but Worker 1 cannot deduce any new conclusions with

the axiom. Correspondingly, since $path(1,2)$ has the even join term 2, Worker 1 distributes it to Worker 2, which subsequently uses the second rule with the given axioms $path(1,2), edge(2,3)$ to derive $path(1,3)$. The join term 3 of $path(1,3)$ is odd and, thus, the axiom is sent to Worker 1, which however cannot derive any new conclusions. The deductive closure computation ends here since all workers converged. However up to this point, the control node has no means to determine if all workers terminated – this problem is addressed later.

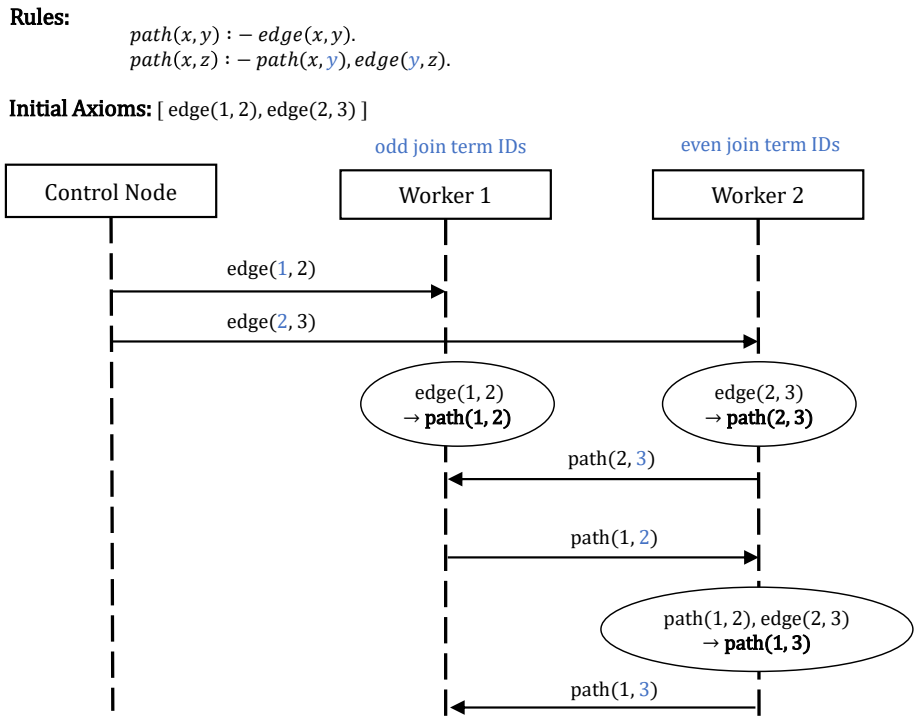


Fig. 2: Example for computing the transitive closure of our running example by using the data partitioning approach. The control node initially distributes all axioms to the responsible workers by considering the appropriate join terms. Subsequently, the workers start deriving all conclusions for the given axioms and exchange the new axioms with one another to ensure that all possible conclusions are derived.

Rule Partitioning Another option for parallelizing the deductive closure computation is to partition the rules between the workers [17], i.e., each worker executes the adjusted procedure from Algorithm 2 with different subsets of rules. In this particular case, it has to be ensured that a worker node receives all axioms that can be used in a rule body of the assigned set of rules. Therefore, a rule

dependency graph is initially generated, where each rule is represented by a vertex. An edge from a given rule R_1 to another rule R_2 indicates that a conclusion of R_1 may be used in the rule body of R_2. Subsequently, the vertices, i.e., rules, of the dependency graph are partitioned between the workers.

In Figure 3, the rule partitioning strategy has been deployed to compute the deductive closure of our running example with two workers. Since the *path* conclusion of R_1 might be used in the rule body of R_2, an edge from R_1 to R_2 exists in the rule dependency graph. In order to divide the workload, Worker 1 gets assigned R_1 and Worker 2 the rule R_2. At the beginning, the control node distributes each given axiom to a worker if the axiom can be used as premise in one of the assigned rules. Since the initial *edge* axioms can be used in both rule bodies as premise, the control node distributes the axioms to Workers 1 and 2. Now, as the depiction shows, Worker 1 sends the derived conclusions $path(1, 2), path(2, 3)$ according to the rule dependency graph to Worker 2. Worker 2 finally deduces $path(1, 3)$ and both workers converge.

An advantage of rule partitioning over data partitioning is that the rule set can be distributed faster to the workers than sets of join term partitions since the rule set is usually small. Note that this is not the case for the hash based partitioning approach since we only require a hash function and the IDs of all involved workers. However, the small number of rules is also a disadvantage because, e.g., in case of the transitive closure computation for directed graphs, only two rules are available for the partitioning process, which corresponds to the maximum number of partitions.

An optimization for the rule partitioning procedure with rule dependency graphs is to cut as little edges as possible when deriving the subsets of rules for the appropriate workers. If a directed edge from a rule R_1 to another rule R_2 is cut, since the rules are assigned to different workers, it means that all conclusions from R_1 must be communicated to the worker that is responsible for R_2. Another optimization for the partitioning process is, thus, to assign weights to the edges based on the number of conclusions that they may produce. These weights can then be used in the partitioning process to cut the edges with the smallest weights in order to reduce the number of communicated axioms. This approach requires however an appropriate heuristic that depends on the considered rules and initial axioms.

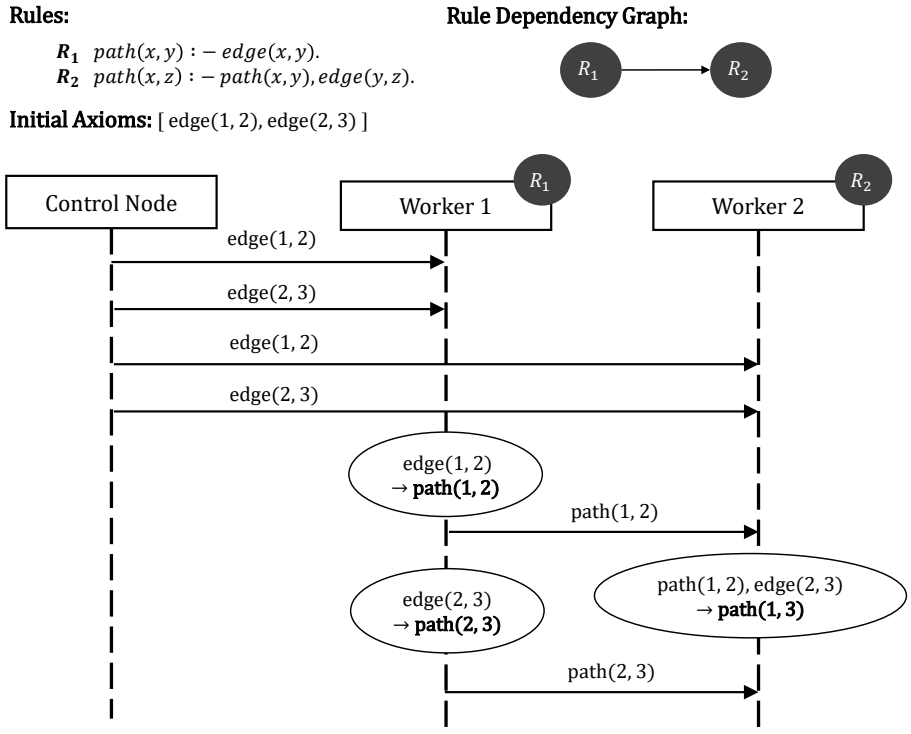


Fig. 3: Example for computing the transitive closure of our running example by using the rule partitioning approach. Initially a rule dependency graph is derived from the given rules, where a directed edge between two rules R_1 and R_2 indicates that a conclusion of R_1 might be used as premise in R_2. Each worker subsequently gets assigned a set of vertices, i.e., rules, from the rule dependency graph and receives from the control node and other worker nodes all axioms that might be used as premises in the assigned rules.

2.4 Selection Criteria for Partitioning Methods

The selection of a suitable partitioning method for parallelizing the deductive closure computation depends on the given rules and axioms but also on the deployed hardware. Therefore, before choosing the rule partitioning approach, data partitioning approach, or a combination of both variants, the following goals should be kept in mind:

- **Balanced partitioning**: The workload should be for each worker approximately the same in order to avoid that some workers finish their task sooner than others.
- **Reduce duplicate computations**: A good distribution of the workload minimizes duplicate conclusions which can be an additional overhead in comparison to the sequential execution.

– **Minimize communication between workers**: The communication between workers can be costly depending on the underlying communication medium (RAM, Ethernet etc.). Thus, the data or rules should be partitioned in such a way that the generated conclusions on a given worker preferably have little dependencies with other workers.

Applications can be parallelized on shared and non-shared memory systems, i.e., on single processors and on a cluster of multiple processors, respectively. The communication for parallel applications running on a single processor with multiple cores is usually faster since the lower cache levels $L_2, \ldots, L_n$ and RAM are shared among the appropriate cores. On the other hand, non-shared memory systems communicate over Ethernet, Fiber Distributed Data Interconnect (FDDI), or InfiniBand and, generally, have a higher communication overhead [5]. Especially in case of non-shared memory systems, the workload must thus be partitioned in such a way that as little communication as possible takes place between the workers.

3 Distributed Closure Computation

In this section, we present our approach to compute the deductive closure in parallel by multiple workers that communicate over the network. Initially, we describe the purpose of the control node and the worker nodes in our distributed, parallelized deductive closure computation procedure and present our convergence protocol, which is used in order to determine whether all worker nodes converged. Afterwards, we provide an in-depth view to the implemented networking component that we utilize to send and receive messages.

3.1 Control Node and Worker Nodes

In our parallelized approach, we use the object parallelism architecture as described in Section 2.3 and apply the data partitioning approach from Section 2.3 to derive the corresponding workers. Thus, we distinguish between a single control node and multiple worker nodes, which have the following responsibilities:

– **Control Node**: The control node initiates the parallel computation, verifies that all workers have converged, and finally collects all computed sets of axioms from the workers.
– **Worker Nodes**: Each worker node is responsible for a subspace of the axioms and computes all appropriate conclusions for the given subspace using the adjusted semi-naive deductive closure computation procedure from Algorithm 2.

At the beginning of the procedure, the worker nodes have no rules and no data partition assigned and, therefore, wait for the control node to connect. After the control node initiates the connection to a worker, the control node sends an initialization message, which contains the assigned rules (in our case all rules), a

unique worker ID $w_i \in \mathbb{N}$, and the IP addresses and ports of the other involved workers. All worker IDs must be consecutive integers since we deploy the hash based partitioning approach from Section 2.3, i.e., for a given axiom, we consider the respective join term t and compute $(h(t) \mod n) + 1$, where n corresponds to the total number of workers. If the worker IDs are not consecutive, multiple workers get assigned the same axioms.
After all workers have been initialized, the control node starts distributing all initial axioms to the appropriate workers. At the same time, the worker nodes initiate a connection to one another, i.e., every worker with a given ID w_i connects to another worker with an ID w_j if $w_i < w_j$. When a given worker established all connections to all other workers, the adjusted semi-naive worker procedure starts. In order to determine whether all worker nodes found a fixpoint, we deploy a specific convergence protocol, which we introduce in Section 3.2. The individual control node states and worker states that are traversed in the procedure are additionally presented in Figures 4 and 5.

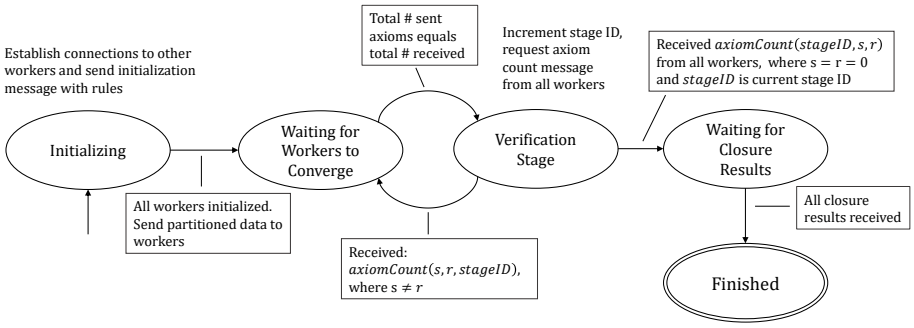


Fig. 4: Control node states in the distributed deductive closure computation procedure

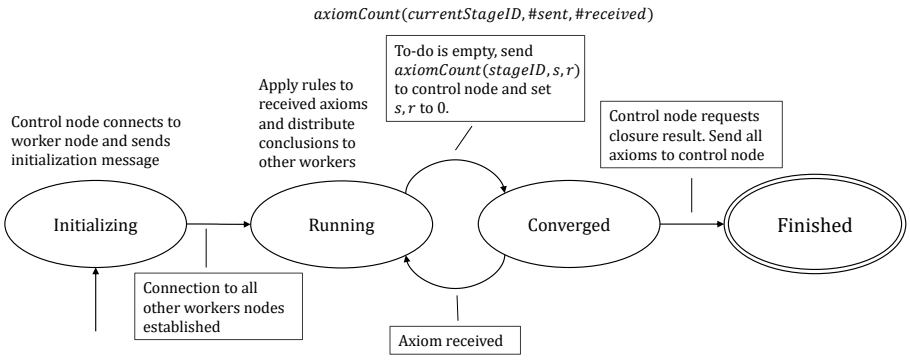


Fig. 5: Worker node states in the distributed deductive closure computation procedure

3.2 Deductive Closure Computation Convergence Protocol

To determine whether all worker nodes have converged in the parallelized deductive closure computation procedure is a non-trivial problem. Figure 6 shows an example of a simple convergence protocol applied to the data partitioning example from Figure 2. If the to-do queue of a given worker becomes empty, the worker sends an appropriate notification to the control node. After all workers notified the control node that their to-do became empty, the control node assumes that all workers converged and terminates the procedure.
In the given example, this approach leads to a premature termination of the procedure: After the workers derived the conclusions $path(1, 2)$ and $path(2, 3)$ and have distributed them to one another, they send a notification to the control node that their to-do queue became empty. The control node in turn receives the notifications and assumes that both workers have converged. However, since Worker 2 receives the newly derived conclusion $path(1, 2)$ after the control node has been already notified, the procedure terminates prematurely because the conclusion $path(1, 3)$ has not been derived yet.

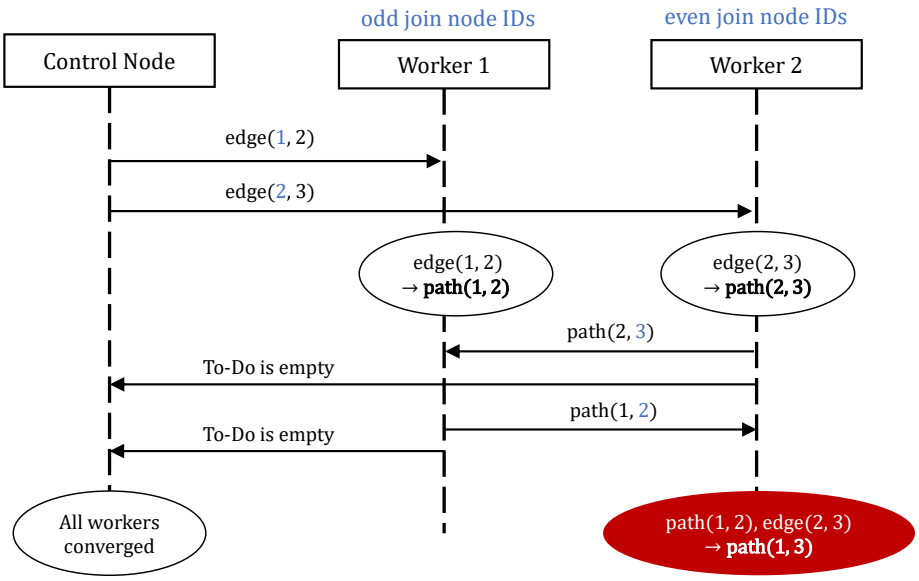


Fig. 6: Example of a simple convergence protocol applied to the scenario of Figure 2, which determines if a worker has converged based on the information whether its to-do queue is empty. The procedure terminates prematurely because the conclusion $path(1, 2)$ arrives at Worker 2 after Worker 2 has already notified the control node that its to-do queue became empty.

For this reason, the following two conditions must apply in order to safely terminate the procedure:

- **All sent axioms have been received**. Even though all to-do queues are empty, a transmitted conclusion may still be on its way to a worker.
- **All received axioms have been also processed**. Processing an axiom means that all rules have been applied to it and the derived conclusions have been distributed to the responsible workers.

Therefore, we developed a special convergence protocol that ensures that all sent messages have been indeed received and processed. It is schematically depicted in Figures 4 and 5. The protocol operates on the principle that the total number of sent axioms S by the control node and workers must be equal to the total number of received axioms R by the workers. After the to-do queue of a worker becomes empty, the worker reports to the control node the number of sent and and received axioms. The control node, correspondingly, increments the counters S and R. If $S = R$, the control node enters a verification stage, where the appropriate counters are again requested from each worker node. If $S = R$ still holds, the control node knows that all workers converged because all sent messages have been indeed received and processed by the workers. We will now consider the concrete protocol procedure in more detail.

Variables and Protocol Messages For our convergence protocol, the control node maintains the following variables:

- S: The total number of sent axioms in the procedure. The counter S is incremented the first time when the control node distributes the initial axioms to the workers. Afterwards, the counter gets updated when the workers report their number of sent axioms to the control node.
- R: The total number of received axioms in the procedure. Since the workers report the number of received messages only after the to-do queue became empty, it also means that the received axioms have been already processed.
- I: The verification stage ID which is initially set to 1. This ID is utilized in order to associate the reported number of sent and received axioms of the workers with a specific point in time.

On the other hand, the worker nodes require the following variables:

- s: The number of sent axioms of the worker, i.e., the number of times a conclusion has been sent to another worker. The counter is set to 0 again when it has been reported to the control node.
- r: The number of received axioms from the control node or other workers. As it is the case for the counter s, it gets reset to 0 after it has been reported to the control node.
- i: The verification stage ID which is initially set to 1 and updated by the control node with appropriate *axiomCountRequest* messages which we will consider next.

The convergence protocol now utilizes two distinct messages:

- $axiomCount(i, s, r)$: The counters s and r represent the number of sent and received axioms, respectively. On the other hand, the variable i represents the current stage ID that is held by the worker. The message is sent by a worker node to the control node after the worker to-do became empty and, as already mentioned, the variables s and r are set to 0 after the message has been transmitted.
- $axiomCountRequest(i_{new})$: This message is sent by the control node to all workers in order to request an *axiomCount* message from all workers. The parameter i_{new} corresponds to the new stage ID to which the workers update their variable i before sending the *axiomCount* message. The new stage ID ensures that the corresponding *axiomCount* responses are not older than the current *axiomCountRequest* message.

After the control node starts the procedure, it distributes all initial axioms to the workers - S gets incremented appropriately. When a given worker receives an axiom, the individual worker increments its counter r. On the other hand, when a conclusion is deduced that is sent to n responsible workers, s gets incremented by n. Eventually, when the to-do queue of a worker becomes empty, the message $axiomCount(i, s, r)$ is sent to the control node and the counters s and r are set to 0 afterwards.

When the control node receives a message $axiomCount(i, s, r)$ from a worker, the control node updates S to $S + s$ and R to $R + r$. Afterwards, it is examined whether $S = R$. If this is the case, the procedure has not necessarily converged: The counters S and R sum up the total number of sent and received axioms, however, it does not mean that the counters S and R summarize at each point in time the s and r counters of all workers. Therefore, a verification stage is required which requests *axiomCount* messages from all workers that are not older than the request.
When the verification stage is initiated, the verification stage ID I is incremented and the control node sends an $axiomCountRequest(I)$ message to all workers. When the workers receive the request, they update their stage ID counter i to the value of I and send an *axiomCount* message to the control node when their to-do becomes empty.
In the verification stage, the control node collects all messages $axiomCount(i, s, r)$ from the worker nodes and examines whether $s = r = 0$ and $i = I$. If this is the case for all responses, it is known that all sent axioms have been indeed received and processed and, therefore, the procedure has converged. If $i = I$ but $s \neq r$, the verification stage ends and it is waited until $S = R$ to enter the verification stage again. Otherwise, if $i < I$, it means that the *axiomCount* message belongs to a previous verification stage. If $s = r$, the counters S and R are updated but the verification stage continues because S and R remain equal. However, if $s \neq r$, the counters S and R are updated and the verification stage is aborted.
We assume that all protocol messages arrive reliably and in the order in which they have been sent. In Section 3.2, we will consider why this condition is necessary with the aid of a concrete example. After the control node distributes the initial axioms to k distinct workers, the control node receives at least k *axiomCount* messages since the to-do queue will eventually be empty after all axioms have been processed. If $S \neq R$, it means that some distributed axiom has not been received yet and, therefore, at least one *axiomCount* message is to be expected by the control node. If $S = R$, the control node enters the verification stage and, subsequently, sends an *axiomCountRequest* to all n workers, which results in at least n more *axiomCount* messages with the new stage ID. For this reason, the protocol terminates when $S = R$ and if $s = r = 0$ in all messages $axiomCount(i, s, r)$ from the initiated verification stage, which indicates that all sent axioms have been indeed received and processed.

Convergence Protocol Example Figure 7 shows the situation, where we use our convergence protocol in order to solve the problematic case from our previous example in Figure 6. At the beginning of our procedure, the control node distributes the known axioms to Worker 1 and 2, and updates the counter S. Afterwards, Worker 1 and 2 derive the conclusions $path(1, 2)$ and $path(2, 3)$ in parallel. Worker 2 now distributes $path(2, 3)$ to Worker 1 and subsequently sends the message $axiomCount(1, 1, 1)$ to the control node since its to-do queue became empty. Because $S = 3$ and $R = 1$, the control node knows that not all sent axioms have been received yet and the procedure continues. Worker 1

now distributes the conclusion $path(1, 2)$ to Worker 2 but it cannot derive any new conclusions which is why it sends the message $axiomCount(1, 1, 2)$ to the control node. The counters S and R still do not match. When Worker 2 receives the axiom $path(1, 2)$ from Worker 1, it derives the conclusion $path(1, 3)$, which is in turn distributed to Worker 1. Again, Worker 1 cannot produce any new conclusions and sends, therefore, the message $axiomCount(1, 0, 1)$ to the control node. Since $S = R = 4$, the verification stage is now initiated, i.e., the stage ID is updated to 2 and an *axiomCount* message is requested from all workers. Note that Worker 2 still did not send its *axiomCount* message from the previous stage with ID 1. The control node now receives the message $axiomCount(1, 1, 1)$ from the previous stage but since still $S = R$, the verification stage continues. Finally, both messages $axiomCount(2, 0, 0)$ with the current verification stage ID 2 arrive at the control node and, because $s = r = 0$, it is verified that all sent messages have been indeed received and processed.

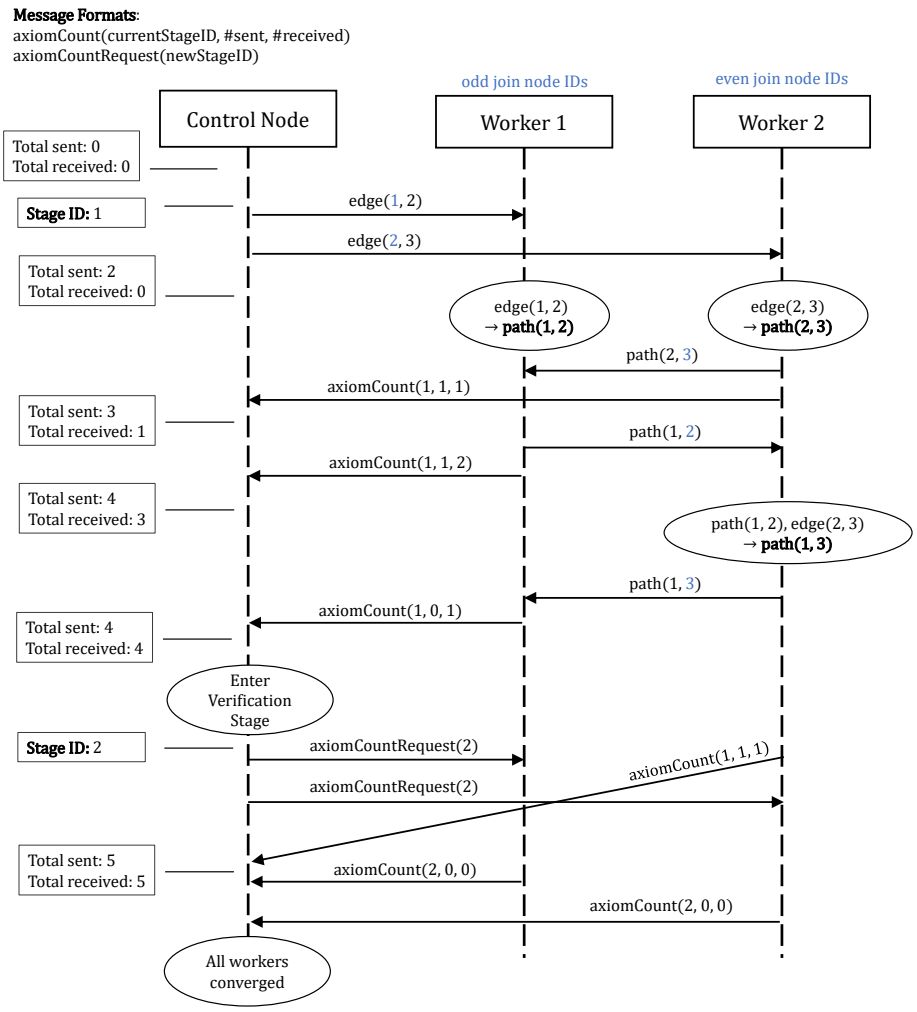


Fig. 7: Example of our convergence protocol which solves the problematic case from Figure 6 by ensuring that all sent messages have been indeed received and processed.

Reliable, In-order Delivery of All Protocol Messages Note that all *axiomCount* messages must arrive in the same order as they have been sent. Otherwise, the procedure may terminate prematurely as depicted in Figure 8, which represents an extended version of the example from Figure 7. Suppose Worker 2 in our previous example has to distribute the conclusion $path(1, 3)$ to a new worker, namely, Worker 3, which has the following unique rule assigned:

$$specialPath(1, 3) :\!- path(1, 3).$$

After Worker 2 derives the conclusion $path(1,3)$, the axiom is sent first to Worker 1 and after some delay to Worker 3 since it occurs as premise in the given, special rule. Worker 2 now sends the message $axiomCount(1,2,1)$ to the control node because its to-do became empty but we assume that the message arrives delayed. Before Worker 3 receives the axiom $path(1,3)$ from Worker 2, the control node has already initiated the verification stage and Worker 3 responded with the message $axiomCount(2,0,0)$. Since Worker 1 and Worker 2 afterwards also send the message $axiomCount(2,0,0)$ to the control node, the control node assumes that all workers converged. However, as can be seen, Worker 3 has not processed yet the axiom $path(1,3)$ and, therefore, the procedure ends prematurely. If the message $axiomCount(1,2,1)$ from Worker 2 arrived before the message $axiomCount(2,0,0)$, this situation would not occur. Admittedly, the example reflects a rare case but it shows why a reliable, in-order delivery of all *axiomCount* messages is required.

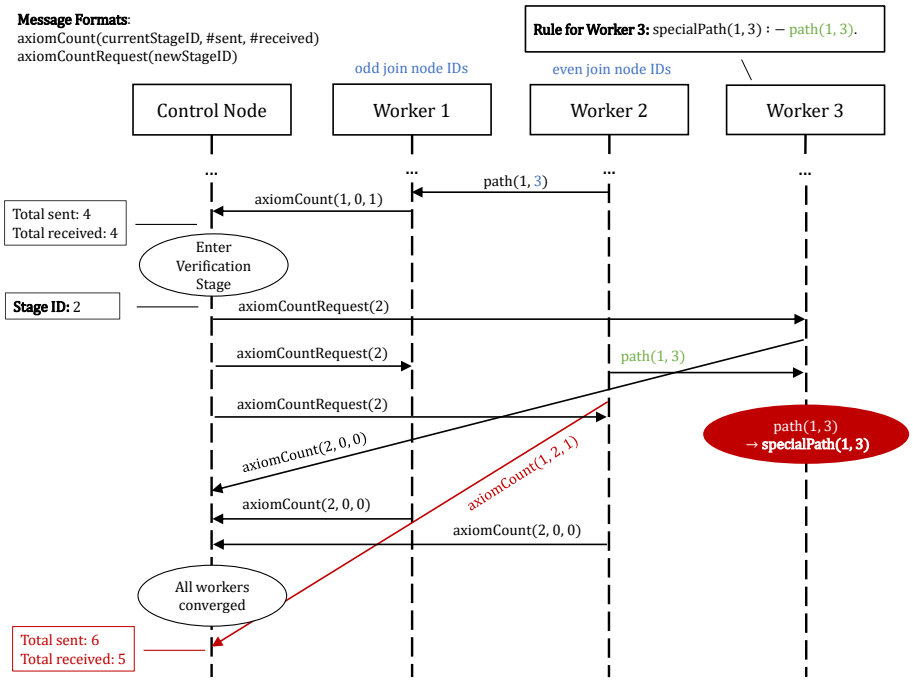


Fig. 8: Extended version of the example in Figure 7. It shows a situation, where the procedure terminates prematurely because the condition is not fulfilled that all *axiomCount* messages of a given worker arrive at the control node in the same order as they have been sent.

3.3 Non-blocking Networking Component

In case of usual synchronous I/O operations, such as reading or writing to a file, a thread of a given process performs the operation and subsequently waits until the I/O request to the operating system (OS) finishes, i.e., the thread is meanwhile blocked and cannot execute other tasks. The alternative is the usage of asynchronous I/O operations, which we utilized in the implementation of our networking component: In this case, the application uses a thread pool T, which contains a set of threads. If a task is submitted to the thread pool, a thread $t \in T$ executes the task and gets returned to the thread pool T as soon as it finishes the execution. If an I/O operation is to be performed by a thread $t \in T$, the I/O operation is delegated to a separate background thread t_B that is responsible for the execution of all I/O operations. When t_B finishes the I/O operation, it submits a predefined task to our thread pool T, in which the result of the I/O operation can be eventually processed. Thus, the thread $t \in T$, which initiated the I/O operation, can immediately continue its execution without blocking after the operation has been delegated to t_B.

Socket I/O Operations A socket essentially represents a communication endpoint that is defined by an IP address and a given port. For the incoming and outgoing messages, sockets use *buffers*. A buffer essentially represents a reserved memory segment that is used to exchange messages between a producer and a consumer, i.e., a message generating entity and a message processing entity. The *outbound* buffer stores all bytes that will be sent over the network. Therefore, our application represents the producer and the network connection the consumer. On the other hand, the *inbound* buffer of a socket contains all bytes that are received over the network. Thus, the network represents the producer and our application the corresponding consumer.
In order to write a message from the application to the socket outbound buffer, the message needs to be serialized to the buffer, i.e., the message is converted into a stream of bytes, which is subsequently written to the respective buffer. On the other hand, if a message is read from a socket, the appropriate inbound buffer contains a stream of bytes, which has to be converted into corresponding messages again.

Socket Writer After an asynchronous *send operation* gets called on the socket with a filled outbound buffer, the background I/O thread reads from the outbound buffer and writes the data to the underlying connection. Meanwhile, the buffer cannot be used for any further write operations by our application since the buffer is not thread-safe. In these instances, a byte buffer pool is often used, which contains buffers that are currently not involved in background I/O operations and, therefore, can be utilized by the application [13]. If the background send operation completes, the appropriate buffer gets returned to the buffer pool. In our application, only a single thread is involved to write messages to a given outbound buffer. Therefore, we deployed for each *socket writer* two buffers such that one buffer can be used by the application and the other buffer may

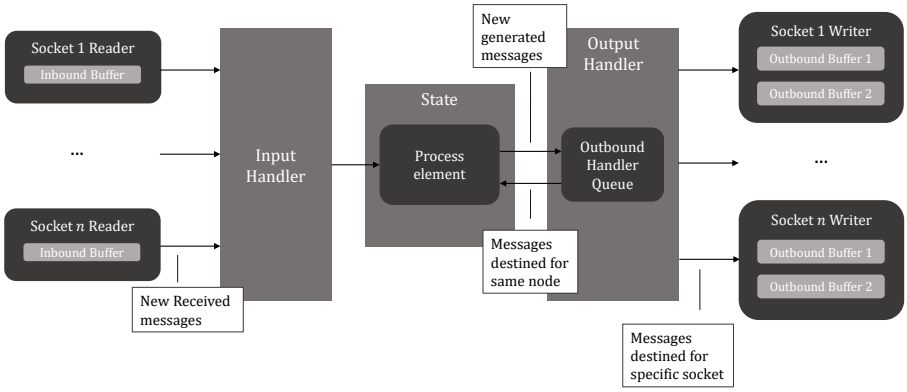


Fig. 9: Depiction of our networking pipeline to process input from n sockets and handle the appropriate output. The output is either processed by the same node again or propagated to one of the responsible n sockets.

be involved in a background write operation. We denote the buffer that is accessed by our application always by b_A and the buffer that may be involved in an asynchronous send operation by b_B. The first time our application writes to the buffer b_A, the buffers b_A and b_B are switched and an asynchronous write operation is initiated on b_B. The buffer b_A can still be accessed by our application. When the asynchronous write operation completes, the application thread examines whether all bytes could be written from the buffer b_B to the underlying connection. If that is the case and the other buffer b_A is non-empty, the buffers are switched and an asynchronous send operation is again initiated. If, however, b_B is not empty yet, an asynchronous send operation is again initiated for b_B. Eventually, when both buffers b_A and b_B are empty, no asynchronous send operations are initiated anymore.

Socket Reader In case of asynchronous *read operations*, the background I/O thread reads data from the network connection and writes it to the appropriate inbound buffer of the socket. Meanwhile, the involved inbound buffer cannot be accessed by our application. But since our application only deserializes messages from the inbound buffer after a background read operation completes, the buffer is never accessed by the application thread and the background I/O thread at the same time and we, therefore, deploy in comparison to the *socket writer* only a single buffer.

Input and Output Handler Tasks Given our networking component, the pipeline of the control and worker nodes is now built as follows: The node receives messages from n sockets, which are subsequently propagated to the appropriate worker or control node method that provides a state-dependent processing. When a received message gets processed, new messages may be generated, which are

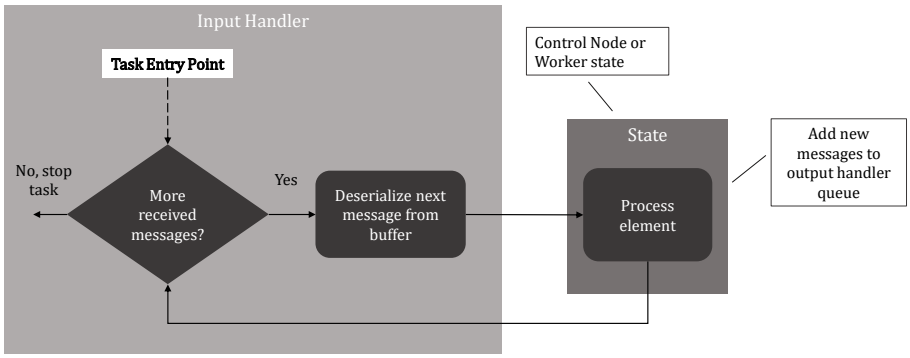


Fig. 10: Flow chart for the input handler task, which processes newly received messages from the network. The task gets submitted to the application thread pool when an asynchronous read operation completes.

either destined for the same node or other nodes. If outbound messages are destined for a specific socket, they are serialized to the appropriate outbound buffer and, otherwise, they are processed again on the same node. This high-level description of our pipeline is additionally depicted in Figure 9. As elaborated in the previous sections, the asynchronous, non-blocking networking component is based upon a thread pool. As an abstraction from the actual networking component, we use two tasks which are repeatedly submitted to the thread pool:

- **Input handler task – the task to process new received messages**: A flow chart for the appropriate input handler task is depicted in Figure 10. As soon as a socket connection has been established, an asynchronous read operation is initiated. When this read operation completes, the buffer is filled, at least partially, with bytes and the input handler task gets submitted to the thread pool. Messages are now deserialized and processed as long as there are complete messages contained in the buffer. Therefore, it has to be examined whether the buffer contains enough bytes in order to deserialize a new message. If this is not the case, an asynchronous read operation is initiated again and the input handler task stops. If enough bytes are available, the message is deserialized and subsequently processed by the appropriate method of the control node or worker state.
- **Output handler task – the task to process new generated messages**: The procedure of the output handler task is presented in Figure 11. Newly generated messages from the appropriate method of the control node and worker node state are added to the queue of our networking I/O handler. Thus, if a new message is generated and added to the queue, the output handler task gets submitted to the thread pool if not already done. When the output handler task gets eventually executed by the thread pool of our application, it takes the next element from the queue. If the element is des-

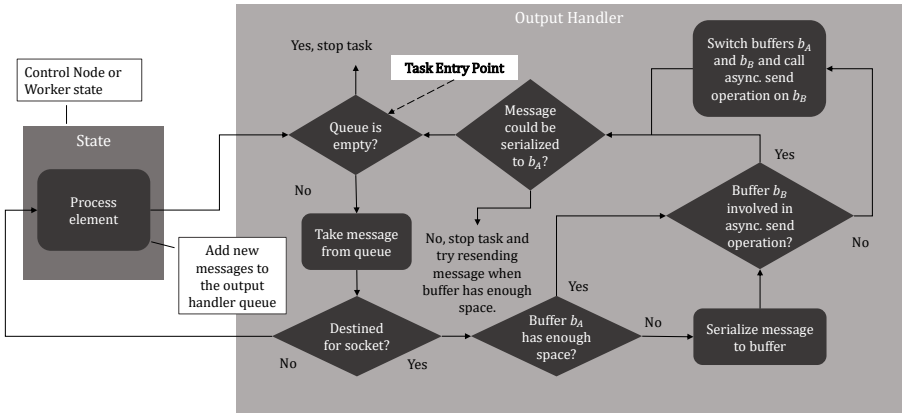


Fig. 11: Flow chart for the output handler task, which processes messages that have been generated by the method of an appropriate worker or control node state. The output handler task gets submitted to the thread pool if the task has not been submitted yet and one of the following two conditions apply: Either a new message gets added to the queue, or a message could not be serialized to an outbound buffer and the buffer has now space again. The messages from the queue are processed until the queue is empty or a message could not be serialized to a buffer. We denote the buffer that is accessed by our application by b_A and the buffer that may be involved in an asynchronous send operation by b_B.

tined for the same node, it gets processed by the method of the appropriate state. On the other hand, if the message is destined for a socket, we examine whether our buffer b_A of the given socket has enough space for the message. If that is the case, the message is serialized to b_A. Now, we check if the buffer b_B is currently involved in an asynchronous send operation. If not, we switch both buffers b_A, b_B and subsequently initiate an asynchronous send operation on b_B. Eventually, if our current message could be serialized to b_A, we process the next message from the queue. Otherwise, the output handler task is stopped and we attempt to send the current message again when the task is executed the next time. Thus, the output handler task is submitted to the thread pool if the asynchronous send operation completes and the buffer b_A has space again.

Networking Component Java Implementation Our distributed approach has been implemented in Java and uses the *NIO.2 asynchronous channel API* [1, 13], which provides methods for asynchronous I/O operations. In NIO.2, each asynchronous socket channel belongs to a given channel group, where each group shares a pool of threads. Listening on a port for new client connections, connecting to remote servers, reading, and writing from an underlying socket thus can be performed asynchronously by the appropriate thread pool. In order

to examine the scalability of our distributed approach concerning the number of deployed workers, we always used a thread pool with a single thread. This single thread, consequently, executes for a given worker or control node all appropriate input handler and output handler tasks that have been submitted to the pool. For the serialization of all messages, which are transmitted over the network, we deployed the Kryo[2] serialization framework.

4 Evaluation

In the following, we present the evaluation of our distributed deductive closure computation approach and other related variants. We executed two distinct benchmarks in order to determine the performance and scalability in relation to the number of deployed workers. All approaches have been implemented in Java and the experiments were conducted on an Intel(R) Xeon(R) CPU E5-2440 0 @ 2.40GHz with 8 cores, 134GB RAM. The implementation of the experiments and of all deployed approaches is available on GitHub[3].

4.1 Experimental Setup

A single experiment corresponds to the computation of the deductive closure for a given set of rules and initial axioms. The sending of all results from the worker nodes back to the control node is not included in the time measurements. An experiment consists of 5 rounds, i.e., 2 warm-up rounds and 3 actual experiment rounds. The average evaluation time of the actual experiment rounds is eventually used in our plots.
The approaches which we deployed in our evaluation are the following:

- **Single-threaded**: Represents an implementation of the semi-naive deductive closure computation algorithm which is executed with a single thread.
- **Multi-threaded**: Represents a parallelized implementation of the semi-naive deductive closure computation algorithm, which is executed in a single process. As it is the case in our distributed approach, we use a control node and multiple worker nodes, which get executed in separate threads. The conclusions are distributed by adding them directly to the respective worker to-do queue. In order to determine whether all worker nodes found a fixpoint, the convergence protocol from Section 3.2 has been deployed.
- **Distributed, multi-threaded**: Implementation of our distributed deductive closure computation procedure, where the control node and each worker node is executed in a separate thread. All messages are sent over the local network.
- **Distributed, in separate JVMs**: Distributed approach, where the control node and each worker node is executed in a separate JVM.

[2] `https://github.com/EsotericSoftware/kryo`
[3] `https://github.com/MaximilianWenzel/DeductiveClosureComputation/`

– **Distributed, in separate docker containers**: Distributed approach, where the control node and each worker node is executed in a separate docker container. The communication between the control node and the worker nodes takes place in a virtual docker network.

All approaches use queues in order to store all newly generated messages or conclusions. Although only the multi-threaded approach requires a synchronization of the queues, we nevertheless used the same unbounded Java *LinkedBlockingQueue* implementation for a better comparison between the approaches.

4.2 Echo Closure Computation

The echo closure experiment represents a simple benchmark, where workers exchange derived conclusions in an echo fashion. Let $s : \mathbb{N}_0 \to \mathbb{N}$, $s(x) = x + 1$ be the successor function and $p : \mathbb{N} \to \mathbb{N}_0$, $p(x) = x - 1$ be the predecessor function. Given an initial axiom $A(1)$, a worker w_1 derives the conclusion $B(s(1)) = B(2)$. Afterwards, the axiom $B(2)$ is distributed to another worker w_2 which is responsible for the ID 2. Worker w_2 in turn derives from the axiom $B(2)$ the conclusion $A(p(2)) = A(1)$, which again represents the initial axiom that is returned to w_1. Thus, each initial axiom $A(c)$ with $c \in \mathbb{N}$ initiates a simple echo message sequence. The rules are hence defined as follows:

$$B(s(x)) :\!- A(x). \tag{3}$$

$$A(p(x)) :\!- B(x). \tag{4}$$

Since the process of applying a rule to a given axiom $A(c)$ corresponds to a simple conversion that does not require expensive computations or lookup operations, the echo benchmark compares how fast the worker and control nodes of the different approaches can exchange messages. In order to ensure that each axiom $A(c)$ and $B(c)$ is sent to another worker, we use the hash based partitioning approach from Section 2.3, i.e., an axiom $A(c)$ or $B(c)$ is sent to the worker that has the ID $(c \mod n) + 1$, where n represents the total number of workers. If t distinct, initial echo axioms $A(c)$ are given, t conclusions $B(s(c))$ are derived and, in turn, t conclusions $A(p(c))$ from $B(c)$ are generated. The total number of derived axioms for t initial axioms is hence $2t$. When an experiment includes n initial echo axioms, the set of given axioms corresponds to $\{\, A(x) \mid x \in \{\, 1, \ldots, n \,\} \,\}$.

Results Discussion In Figure 12, the results of the echo benchmark with the appropriate average evaluation times are presented, whereas Figure 13 shows the associated speedup in relation to the single-threaded approach. As can be seen, the leading approach is the distributed variant with 8 deployed workers. For a smaller number of initial axioms, i.e., for 500K and 1M axioms, the distributed approaches do not attain their highest speedup which is likely caused by the proportionally higher initialization effort for all workers. In case of 5M and 10M

initial echo axioms, however, the distributed approaches show a high scalability in that the evaluation time decreases as the number of workers grows.
The multi-threaded approach attains its maximum performance only with 5 workers instead of 8. The control node of the multi-threaded approach initially distributes the axioms to all workers, i.e., the axioms are added to the appropriate queue. When a worker receives the first axiom, it starts deriving new conclusions and distributes them to the other workers. Therefore, a given queue may be accessed by multiple threads at once and, hence, additional overhead for the synchronization of the threads arises. This problem does not occur in case of the distributed approaches because the communication between workers takes place over the network.
The distributed approaches perform with a single worker also better than the single-threaded approach. This is because the single-threaded approach adds all initial axioms to the queue and only afterwards the axioms can be processed. In case of the distributed approaches with a single worker, we have a control node and a worker node. Thus, meanwhile the control node distributes the initial axioms to the worker, the worker can already begin deriving new conclusions after receiving the first axiom.
The distributed approaches achieve with the docker container variant for 10M initial echo axioms a maximum speedup of about 6.2 compared to the single-threaded approach. The distributed, multi-threaded approach performs slightly worse than the other two distributed approaches probably since the default Java garbage collector has been deployed. When the default garbage collector executes its *mark-and-sweep* algorithm, it freezes all threads of the given process and, therefore, also all workers.
In case of the experiment with 10M initial echo axioms, a linear dependency between the number of workers and the speedup can be observed for the distributed docker approach, which indicates a high scalability of the approach for this particular problem. If the number of deployed worker doubles for the distributed docker approach, we thus expect a doubling of the resulting speedup.

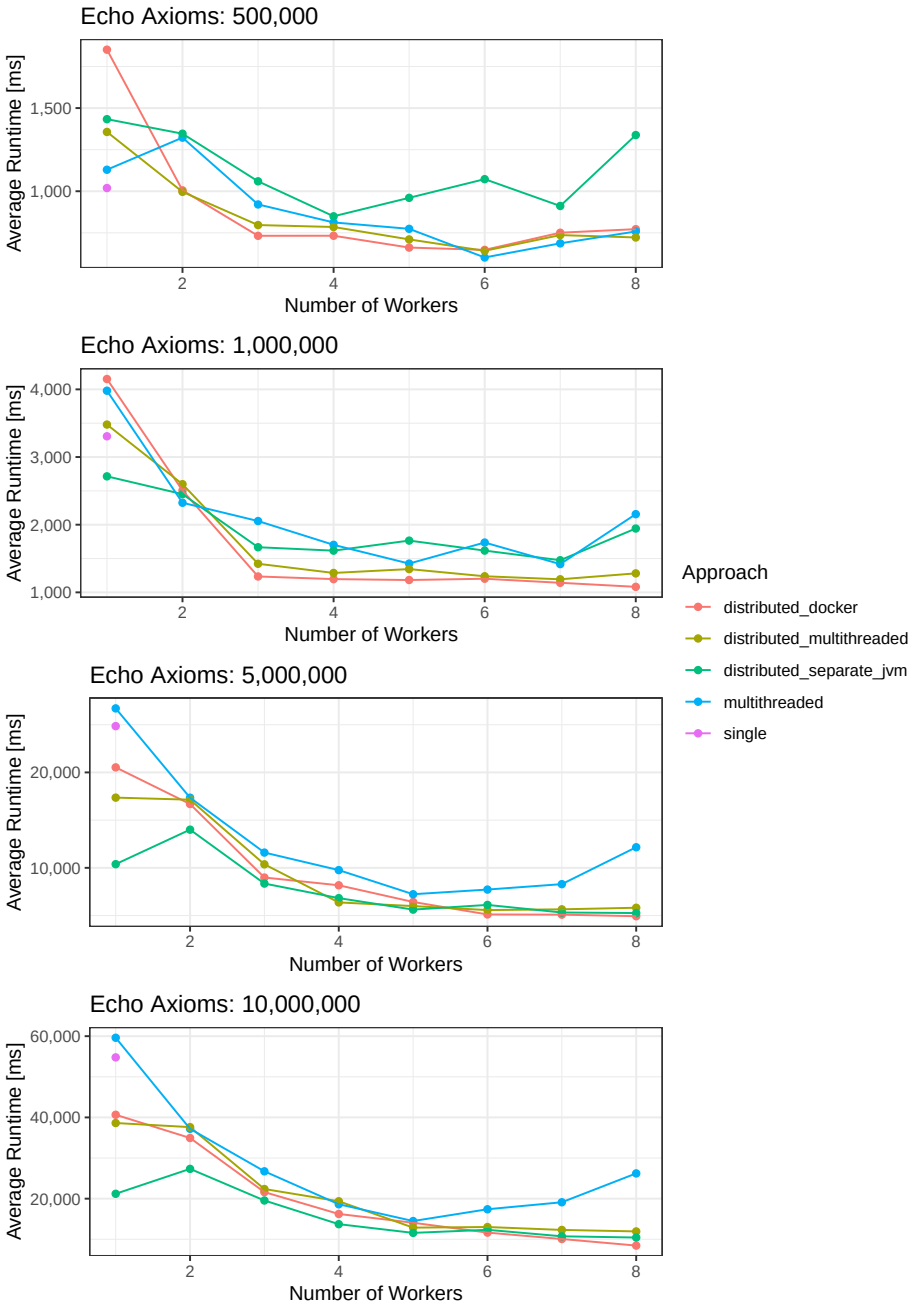


Fig. 12: Results of the echo benchmark

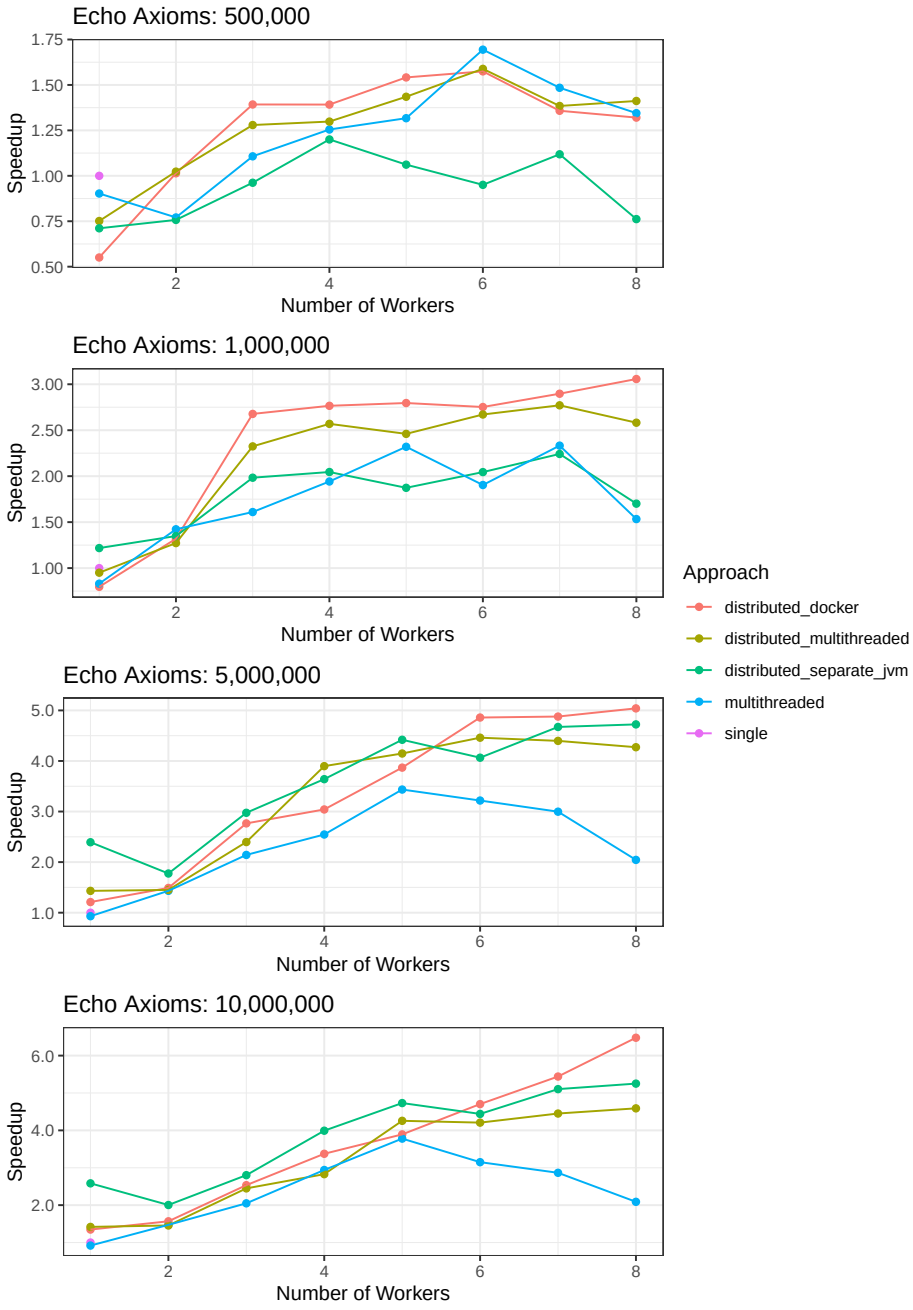


Fig. 13: Speedup of the individual deductive closure computation approaches in the echo benchmark in relation to the single-threaded variant

4.3 Transitive Closure Computation for Binary Trees

In the transitive closure benchmark, we compute for a given directed graph all transitive paths, i.e., for each node in the graph, we compute which other nodes are reachable over a sequence of edges. The deployed rules, which we also used in our running example, are presented in the following:

$$path(x, y) \text{ :- } edge(x, y). \tag{5}$$
$$path(x, z) \text{ :- } path(x, y), edge(y, z). \tag{6}$$

As graphs, we used binary trees of different depth $d \in \mathbb{N}$. The root node of the binary tree corresponds to level 1 and has two child nodes. All nodes on the levels $1 < i < d$ have a parent node on level $i - 1$ and in turn two child nodes on level $i + 1$. The leaf nodes on level d have again one parent node however no child nodes. Thus, if a binary tree has a depth d, we get altogether $2^d - 1$ nodes, where each level $i \in \{1, \ldots, d\}$ consists of 2^{i-1} nodes.
If a node n_1 on a given level i has a child n_2 on level $i+1$, an $edge(n_1, n_2)$ exists as initial axiom. A binary tree of depth d has $2^d - 2$ child relations which, therefore, also corresponds to the number of initial axioms. The number of additional paths with length $\ell > 1$ in the transitive closure can be derived in the following way: For the set of nodes from level $i \in \{\, 1, \ldots, d-2 \,\}$ exists exactly one path with $\ell > 1$ for each node on the levels $j \in \{\, i+2, \ldots, d \,\}$. Hence, we get

$$\begin{aligned}
&\sum_{i=1}^{d-2} \sum_{j=i+2}^{d} 2^{j-1} = \sum_{i=1}^{d-2} \sum_{j=i+1}^{d-1} 2^j \\
&= \sum_{i=1}^{d-2} (\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{i} 2^j) \\
&= \sum_{i=1}^{d-2} ((2^d - 1) - (2^{i+1} - 1)) = \sum_{i=1}^{d-2} (2^d - 2^{i+1}) \\
&= (d-2)2^d - \sum_{i=1}^{d-2} 2^{i+1} = (d-2)2^d - \sum_{i=2}^{d-1} 2^i \\
&= (d-2)2^d - (2^d - 1 - 2^1 - 2^0) = (d-2)2^d - (2^d - 4) \\
&= (d-3)2^d + 4
\end{aligned}$$

Thus, $2^d - 2$ and $(d-3)2^d + 4$ corresponds to the number of given edges and derived paths with $\ell > 1$, respectively, and the sum is the total number of $path(x,y)$ axioms which are derived in the procedure. As it is the case in the echo benchmark, we use the hash based partitioning approach to distribute the workload, which we already introduced in Section 2.3 for the given rules.
The workload for an axiom $edge(n_1, n_2)$, where n_2 is a leaf node, is higher in comparison if n_1 is the root node since leaf nodes are involved in much more rule applications. Therefore, we assign all node IDs to the appropriate binary tree

nodes randomly beforehand, which ensures a balanced workload distribution if the depth d is large enough.
As can be seen in (6) for deriving $path(x,z)$, we need to access efficiently for a given node y all outgoing connected nodes z and incoming connected nodes x. Therefore, we use an index which stores this information in a hash map. The index gets updated when an axiom is added locally to the set of already considered axioms on a given worker.
In comparison to the echo benchmark, the deductive closure for a given set of initial axioms is comparably high. For instance, in case of a binary tree with $d = 20$, we have 1,048,574 initial *edge* axioms and altogether 18,874,370 derived *path* conclusions, which is approximately 18 times larger than the initial axioms. In case of the echo benchmark the number of derived conclusions is always only twice as high than the number of initial axioms. This means in turn, that comparatively more time is spent applying the rules to the axioms. Another difference is the number of data dependencies between the workers: In case of the echo benchmark, a given worker communicates at most with two other workers due to the successor/predecessor function and our hash based partitioning approach. In comparison, when we compute the transitive closure for a given graph, a given worker might communicate with all other workers.

Results Discussion Figures 14 and 15 show the experiment results and the speedup plots of the binary tree benchmark. The leading approaches are the distributed implementations, where each worker gets executed in a separate JVM or docker container. For a smaller number of workers, the JVM approach performs even slightly better than the docker container workers.
As already mentioned, in case of computing the transitive closure for graphs, usually all workers communicate with one another because more data dependencies exist between the workers in comparison to the echo benchmark. This leads in case of the multi-threaded approach to more synchronization effort of the appropriate workers. As a consequence, the multi-threaded approach performs worse than the distributed docker and JVM variant. Furthermore, in case of the benchmark for a binary tree of depth 20, the multi-threaded approach has its best performance with only 4 workers instead of the full 8.
The distributed, multi-threaded approach performs worse than the other two distributed implementations probably again because a single garbage collector is shared between all workers. In case of a binary tree of depth 19, the distributed docker variant attains a maximum speedup of about 3.3 against the single-threaded approach. However, in comparison to the echo benchmark, there is no linear dependency between the number of deployed workers and the speedup.

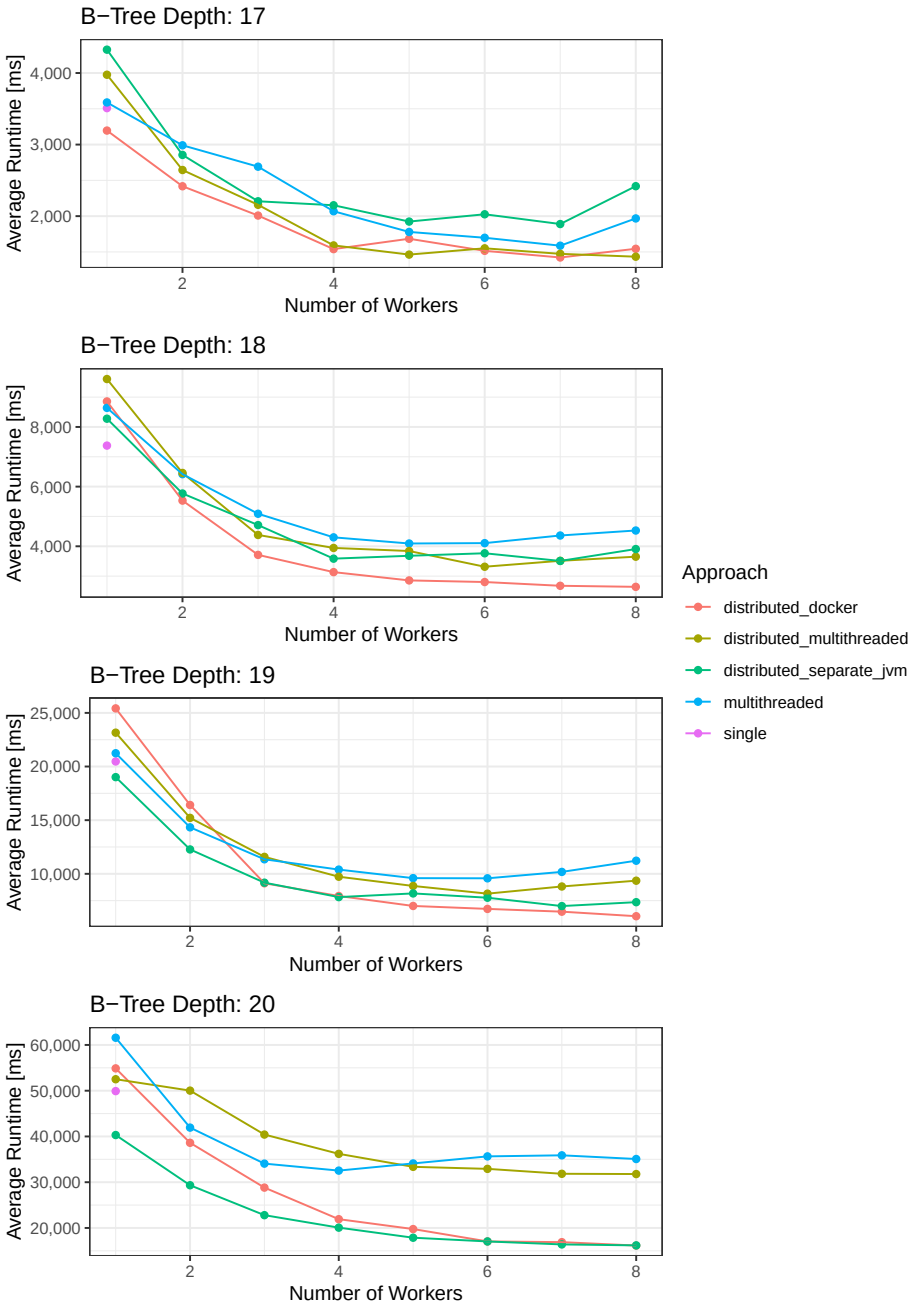


Fig. 14: Results of the transitive closure benchmark on binary trees

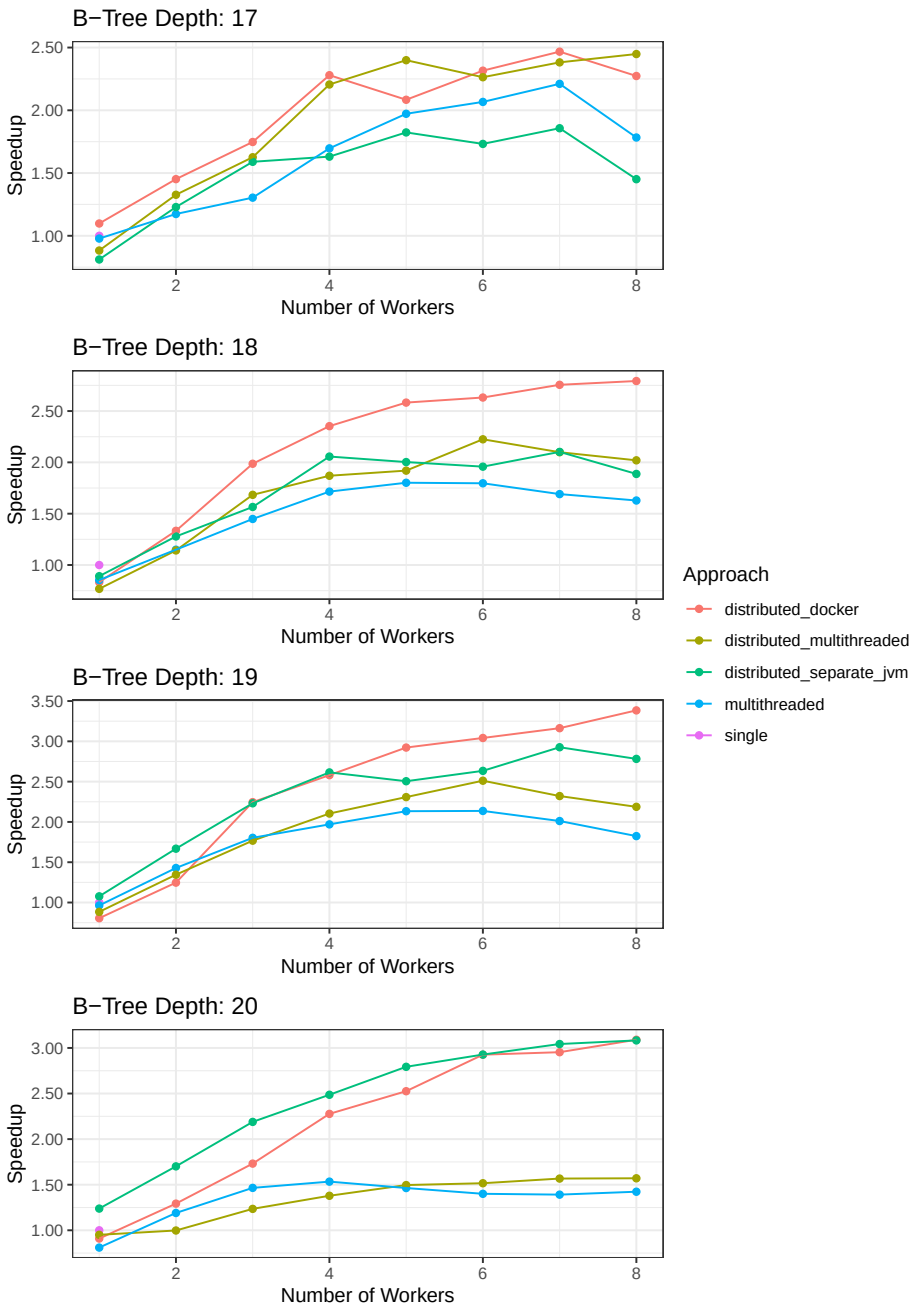


Fig. 15: Speedup of the individual deductive closure computation approaches in the transitive closure benchmark on binary trees in relation to the single-threaded variant

Network Message Overhead Figure 16 shows the distributed JVM implementation computing the transitive closure for a binary tree of depth 20 with two different approaches concerning the distribution of newly derived conclusions: The first approach sends all messages over the network, whereas the other approach adds messages destined to the same worker directly to the to-do. As can be seen, the network transmission of all conclusions does not lead to an additional overhead, which in turn suggests that the overall communication costs remain constant as the number of deployed workers increases. In case of a single worker, there is a difference between both approaches which is due to the networking pipeline: Newly generated messages that are destined for a specific socket are first added to the outbound queue and subsequently sent over the network which represents additional effort in case of only a single worker.

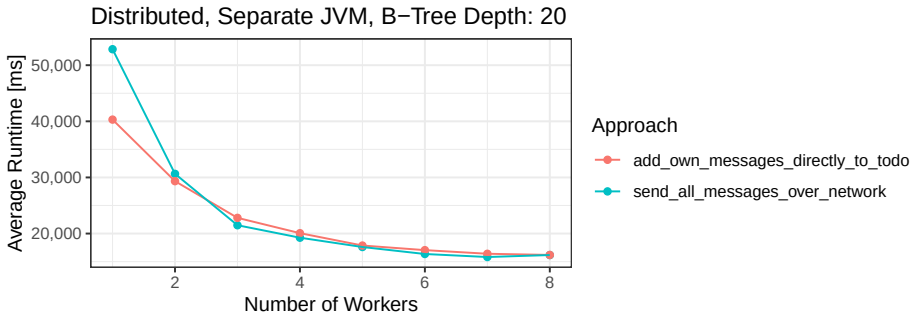


Fig. 16: Comparison between the case where all messages are sent over the network against the case where messages, which are destined to the same node, are directly added to the respective to-do queue.

Fine-Grained Time Measurements In Figure 17, more detailed time measurements of our deductive closure computation procedure are presented for a binary tree of depth 19. Since the appropriate times had to be recorded very frequently, e.g., every time all rules are applied to a given axiom, the absolute time measurements may be larger than in the previous experiment. As can be seen, the average time of each worker to apply all rules is for all approaches with an equal amount of workers approximately the same. The average required time to distribute all axioms to other workers increases in case of the multi-threaded approach as the number of workers grows, which supports our assumption that the data dependencies lead to additional overhead because of the required synchronization effort – this is not the case for the distributed approaches. The bottom plot shows the required time to send all results from the workers back to the control node after the deductive closure computation found a fixpoint. As already mentioned, this additional time is not recorded in our experimental results since we assume that the results may be directly queried from the appro-

priate nodes. However, the plot demonstrates that the aggregation of all closure results is associated with further costs.

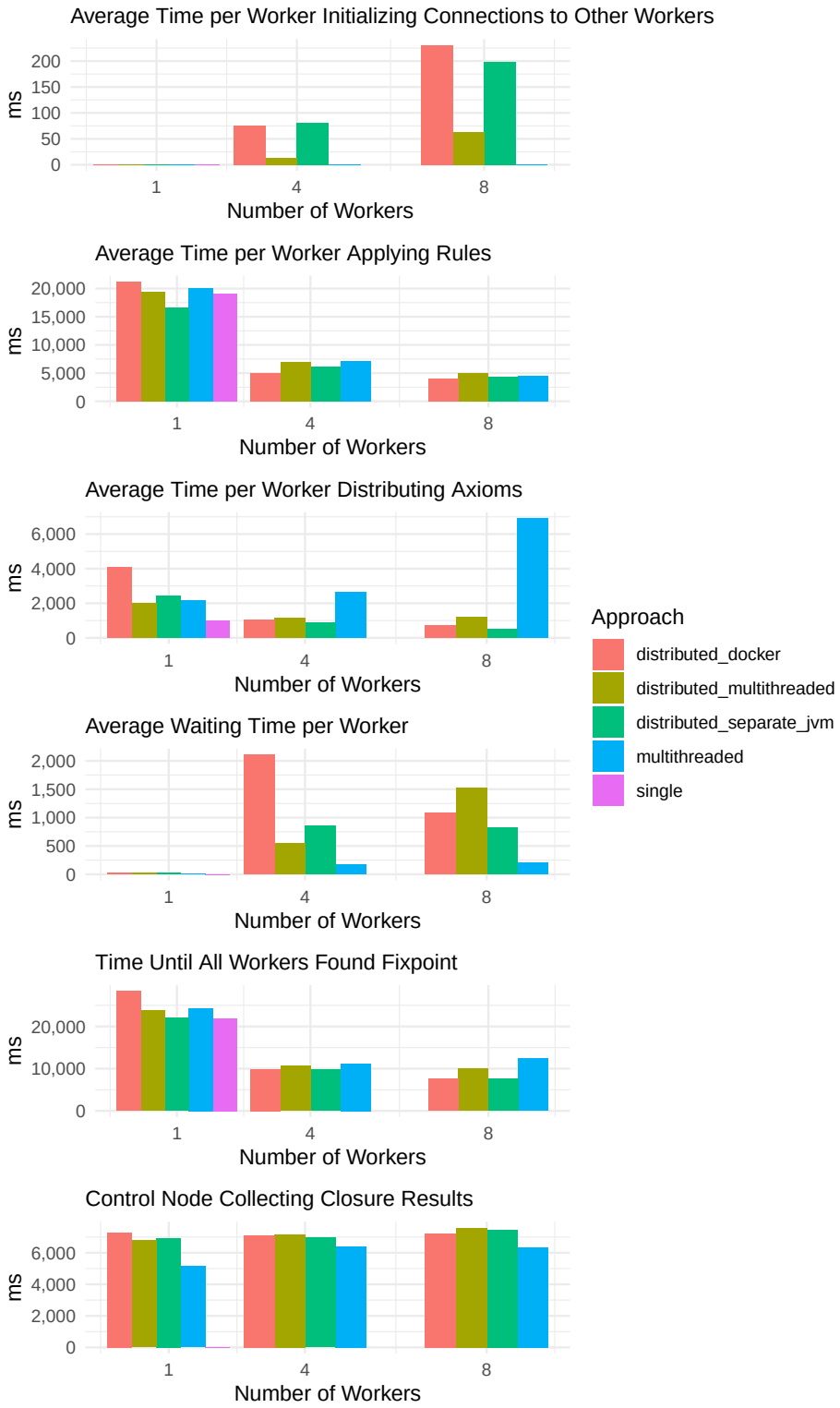


Fig. 17: More detailed time measurements for the transitive closure benchmark on a binary tree of depth 19. The measurements (*"Time Until All Workers Found Fixpoint"*) may be higher than in the results from Figure 14 since the measuring itself leads to additional overhead.

5 Related Work

In recent years, a variety of approaches have been developed in order to parallelize the computation of the deductive closure for a given set of rules and axioms. In the following, we present approaches that are based on the MapReduce programming model [2, 7, 19] and implementations which attempt to provide an alternative to the traditional two-stage MapReduce model [9, 11, 14, 17, 21], e.g., in that the worker nodes communicate in a peer-to-peer fashion.
First, we present instances of the parallelized deductive closure computation for OWL/RDFS datasets. Afterwards, we consider the special case of the transitive closure computation for directed graphs.

5.1 Deductive Closure Computation for OWL/RDFS

RDF represents a data model for directed graphs that store meta data for web resources. An RDF graph is represented by triple statements, where each statement describes a relation between two nodes of the given graph. The languages OWL and RDFS subsequently can be utilized in order to define additional knowledge on RDF classes (groups of nodes) and relations, which may add implicit statements to the given RDF dataset. In order to explicitly derive these triples, the inference rules of OWL/RDFS are used to compute the deductive closure for the given set of axioms, i.e., triples.

Embarrassingly Parallel RDFS Closure Computation Weaver et al. [21] implemented the parallel computation of the RDFS closure in such a way that individual worker nodes do not need to communicate with one another. For this reason, each worker receives the schema of the dataset and, furthermore, a partition of the original assertional triples. In order to partition the RDFS dataset, the authors used the N-Triples format because each line in the file corresponds to a single triple and, thus, the dataset can be partitioned simply by the appropriate line numbers. The workers subsequently read in parallel from the given RDFS dataset and write their conclusions independent from one another to separate files. The approach has been implemented in C and, for the coordination of all workers, an implementation of the Message Parsing Interface (MPI)[4] standard has been used. Each worker node deploys the Redland in-memory RDF store[5] to execute the required queries on the respective RDFS dataset partition for computing the deductive closure. The parallelized procedure has been evaluated on a cluster with 128 cores and on RDFS datasets with up to 346 million triples. The results show that the implementation scales linearly with the number of workers, i.e., if the number of workers doubles, the required time to produce conclusion halves. This high degree of scalability is expected since the worker nodes do not need to exchange conclusions with one other and the partitioning overhead of the RDFS dataset is negligible.

[4] `https://www.mpi-forum.org/`
[5] `https://librdf.org/`

WebPIE – Parallel Inference Engine Based on Hadoop MapReduce WebPIE [19] is a parallel inference engine for RDFS and OWL, which uses the data partitioning approach and is based on the Apache Hadoop[6] framework for the MapReduce programming model. As described in our introduction, the MapReduce programming model consists of two phases as its name implies, namely, the *Map* phase and the *Reduce* phase. At the beginning, a control node distributes the data to the *mapper* worker nodes, which generate from the individual statements key-value pairs: A *value* corresponds in the given case to an RDF triple and a *key* to a responsible *reducer* worker of the next phase. Subsequently, the resulting key-value pairs are propagated to the appropriate reducers, which process the key-value pairs, i.e., in case of RDFS or OWL, they compute all conclusions for the received triples and their assigned rules. Afterwards, the results of all reducers are collected again and one MapReduce iteration finishes. Since in a single iteration possibly not all conclusions could be derived, this procedure has to be repeated until no new conclusions have been generated in a given iteration. The WebPIE engine has been evaluated on a cluster with 64 nodes over large real-world and synthetic datasets with up to 100 billion triples. In order to increase the performance, the approach relies on several optimizations, e.g., to keep the schema on all workers in-memory or a specific ordering of the RDFS rules. Furthermore, to reduce the communication overhead of the verbose resource IRIs, the approach generates a dictionary for all deployed terms beforehand by using a data compression procedure [18] that is also based on the MapReduce approach. In case of 1100 million triples with 64 deployed worker nodes, 45 minutes are required alone to perform this compression. The required time to compress the dataset is oftentimes twice as long as the computation time for the deductive closure.

Spark Framework as Alternative to Hadoop MapReduce An alternative approach, which is not based on the Hadoop MapReduce, has been proposed by Jagvaral and Park [9], which uses the Spark framework: In case of MapReduce Hadoop, intermediate results from a MapReduce iteration must be written into a distributed file system which increases the communication costs. The Spark framework proposes, therefore, an alternative by deploying so-called resilient distributed datasets (RDDs), which provide a more efficient reuse of data and direct in-memory storage. In their evaluation, an about 80% higher throughput of conclusions has been attained compared to WebPIE.

MARVIN – SpeedDate Strategy for Dynamic Workload Distribution In case of data partitioning, the number of conclusions, which result from a given triple, may not always be known beforehand. The distributed reasoning engine MARVIN (MAssive RDF Versatile Inference Network) [14] therefore introduces a *SpeedDate* strategy which combines data clustering with random exchange in order to ensure load balancing between the deployed workers. MARVIN is

[6] `https://hadoop.apache.org/`

based on a peer-to-peer architecture, i.e., individual workers communicate directly with one another. For this reason, their Java implementation uses the *Ibis grid programming environment* [20] for all communication related aspects. For the computation of the deductive closure on a given worker, the in-memory triple store RDF4J (formerly Sesame)[7] has been deployed. Individual worker nodes then operate in a divide-conquer-swap manner: The workers receive a partition, compute the deductive closure of it, and subsequently distribute the conclusions to other worker nodes according to the SpeedDate strategy. In their experiments on a cluster with 271 nodes, they computed the RDFS deductive closure on the SwetoDBLP dataset in 3.4 minutes with 64 utilized nodes and, thus, attained a maximum speedup of 12.94 in comparison to the approach with only a single node. In contrast to the previously discussed implementations for RDFS reasoning, the schema is not copied to all worker nodes since they assumed that, in general, this optimization might not be possible for larger, unknown schemas. A linear dependency between the deployed number of workers and the speedup shows that the approach has a high scalability.

ELK – Reasoner for OWL EL Ontologies ELK [10, 11] is a parallel reasoner for OWL EL ontologies that efficiently computes the deductive closure on real-world datasets such as SNOMED CT, which contains about 300,000 initial axioms. The implementation runs in a single process and each worker, in turn, runs in a separate thread. The workers use separate to-do queues for the deductive closure computation but share a common ontology data structure with one another. The data is partitioned by the occurring EL concepts and newly derived conclusions by a worker are directly distributed to the responsible workers again. In order to attain a high performance, several optimizations have been applied, such as the indexing of axioms. In the evaluation, ELK computed the deductive closure of the SNOMED CT dataset in about $4.85s$ using 8 worker threads on a single processor and attained a speedup of 3.84 compared to the single-threaded approach. The next best baseline approach Snorocket[8] required approximately $25.8s$.

5.2 Computation of the Transitive Closure for Directed Graphs

Gribkoff [7] used the Hadoop MapReduce framework in order to implement a parallelized procedure to compute the transitive closure of synthetic binary trees, acyclic, cyclic graphs and real-world datasets. Since in case of an approach, which is based on the MapReduce programming model, multiple iterations of the MapReduce stages are required, different transitive closure algorithms have been implemented, namely, the standard semi-naive and smart approach. The semi-naive approach corresponds to the rules that we also used in our implementation. On the other hand, the smart approach is a specialized procedure which derives new paths by joining paths whose length is a power of two with other

[7] `https://rdf4j.org/`
[8] `https://github.com/aehrc/snorocket`

paths of strictly lesser length. Thus, the deductive closure can be computed in a logarithmic number of MapReduce rounds. In the evaluation on a 4 node cluster, the smart approach, therefore, outperformed the semi-naive approach: In case of a binary tree graph of depth 16, the semi-naive approach required $561s$ whereas the smart variant had an evaluation time of $294s$. This is, however, significantly higher than the evaluation time of our distributed approach since we required even for a binary of depth 20 with 4 workers only about $20s$.
Alves et al. [2] introduce a parallel algorithm for computing the transitive closure of arbitrary directed graphs, which requires similar to the MapReduce model multiple iterations. However, in contrast to the pure MapReduce approach, the workers exchange newly derived paths also in a peer-to-peer fashion. Initially, the worker nodes get a partition of vertices from the graph and, thus, a subset of the original Boolean adjacency matrix assigned. Subsequently, the first iteration starts: Each worker computes the transitive closure for the given adjacency matrix and distributes newly derived paths directly to the other responsible workers. After all workers have finished, the control node examines whether new transitive paths could be derived. If no new paths have been derived, the procedure terminates. Otherwise, a new iteration starts since the new paths might not have been processed yet in the previous iteration. The approach has been implemented in ANSI C, uses an implementation of the MPI standard to coordinate the workers, and has been evaluated on a 64 node cluster. In the evaluation, Alves et al. used random directed graphs, where for a probability of 20% an edge exists between two given vertices. When all 64 nodes from the cluster were deployed, the parallelized approach performed up to 30 times faster than the implementation on a single processing node. In case of the largest input size with a 6144×6144 adjacency matrix, a linear dependency between the number of deployed workers and the speedup could be attained. However, the computation of the transitive closure for these type of graphs is not *embarrassingly parallel*, insofar as n deployed workers cannot execute the problem in $\frac{1}{n}$ of the time.

6 Conclusion

In this project, we investigated whether the computation of the deductive closure for a given set of axioms and rules can be efficiently parallelized using an alternative approach to the standard MapReduce programming model. For this purpose, we used an architecture where individual worker nodes directly distribute newly derived conclusions to one another. In order to determine whether all worker nodes converged, we developed a convergence protocol which counts the number of sent and received axioms of each worker to ensure that all distributed axioms have been indeed received and processed. In our evaluation on a single processor with 8 cores, we attained a maximum speedup of 6.2 in the echo benchmark and 3.3 in the transitive closure experiment on binary trees against the single-threaded approach.
The benchmarks in our evaluation used axioms that were comparably small in size since they essentially consisted only of integer IDs. This would not be

the case, for instance, in RDFS reasoning since the terms in the axioms are represented by resource IRIs. The generation of a dictionary with a *string-to-ID* mapping might, therefore, be often unavoidable to keep the size of the network messages as small as possible and, thus, to reduce the resulting communication overhead. As we have seen in case of the WebPIE parallel inference engine for RDFS and OWL, the compression requires, however, usually more time than the actual computation of the deductive closure [18, 19].

For a better comparison between the different approaches in our evaluation, we used for the to-do queue and outbound handler queue of the networking component the same queue implementation. The performance of the deployed, unbounded *LinkedBlockingQueue* is however not optimal since each *add*-operation generates a new node object for the linked list. By using a queue that operates on an underlying array, an *add*-operation can be executed about 2-3 times as fast. Our single-threaded and distributed approach can use such an alternative queue implementation and, from our first impression, the resulting performance might be higher. Altogether, the results of our evaluation are even without this optimization promising and might indicate that the distributed procedure can be successfully deployed on an HPC cluster with more than 8 processors for similar problems.

Bibliography

[1] Java I/O, NIO, and NIO.2. `https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html`. Accessed: 2022-02-09.

[2] Carlos Eduardo Rodrigues Alves, EN Cáceres, AA de Castro, Siang Wun Song, and Jayme Luiz Szwarcfiter. Parallel Transitive Closure Algorithm. *Journal of the Brazilian Computer Society*, 19(2):161–166, 2013.

[3] Susan Anderson-Freed. Principles of Database and Knowledge-Base Systems: Volume I.

[4] Frank W Bergmann and J Joachim Quantz. Parallelizing Description Logics. In *Annual Conference on Artificial Intelligence*, pages 137–148. Springer, 1995.

[5] Why Parallel Computing and I Foster. Designing and Building Parallel Programs, 1995.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] Eric Gribkoff. Distributed Algorithms for the Transitive Closure. 2013.

[8] Lorenz J Halbeisen. First-Order Logic in a Nutshell. In *Combinatorial Set Theory*, pages 11–29. Springer, 2017.

[9] Batselem Jagvaral and Young-Tack Park. Distributed Scalable RDFS Reasoning. In *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, pages 31–34. IEEE, 2015.

[10] Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. ELK Reasoner: Architecture and Evaluation. In *ORE*, 2012.

[11] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. The Incredible ELK. *Journal of Automated Reasoning*, 53(1):1–61, 2014.

[12] Maria Keet. *An Introduction to Ontology Engineering*, volume 1. Maria Keet Cape Town, 2018.

[13] Anghel Leonard. *Pro Java 7 NIO.2*. Apress, 2012.

[14] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: Distributed Reasoning Over Large-Scale Semantic Web Data. *Journal of Web Semantics*, 7(4):305–316, 2009.

[15] Kotagiri Ramamohanarao and James Harland. An Introduction to Deductive Database Languages and Systems. *VLDB J.*, 3(2):107–122, 1994.

[16] Christian Schlepphorst. *Semi-naive Evaluation of F-logic Programs*. Univ., Inst. für Informatik, 1997.

[17] Ramakrishna Soma and Viktor K Prasanna. Parallel Inferencing for OWL Knowledge Bases. In *2008 37th International Conference on Parallel Processing*, pages 75–82. IEEE, 2008.

[18] Jacopo Urbani, Jason Maassen, and Henri Bal. Massive Semantic Web Data Compression with Mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 795–802, 2010.

[19] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, and Henri Bal. WebPIE: A Web-Scale Parallel Inference Engine Using MapReduce. *Journal of Web Semantics*, 10:59–75, 2012.

[20] Rob V Van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, Rutger FH Hofman, Ceriel JH Jacobs, Thilo Kielmann, and Henri E Bal. Ibis: A Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005.

[21] Jesse Weaver and James A Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *International Semantic Web Conference*, pages 682–697. Springer, 2009.