A Parallel Approach for Matching Large-scale Ontologies

Tiago Brasileiro Araújo[1], Carlos Eduardo Pires[1], Thiago Pereira da Nobrega[2], Dimas C. Nascimento[3]

[1] Universidade Federal de Campina Grande, Brazil
tiagobrasileiro@copin.ufcg.edu.br, cesp@dsc.ufcg.edu.br
[2] Universidade Estadual da Paraíba, Brazil
thiagonobrega@uepb.edu.br
[3] Universidade Federal Rural de Pernambuco, Brazil
dimascnf@uag.ufrpe.br

Abstract. Recent years have seen an increasing use of large ontologies in various areas of knowledge, e.g. health and agriculture. In this scenario, ontology matching is an important way of establishing interoperability between applications that use different but related ontologies. Matching large ontologies is challenging since it involves a great number of comparisons between concepts which leads to high execution time and requires a large amount of computing resources. This work proposes a MapReduce-based approach that distributes the computation of comparisons between concepts among the nodes of a cloud computing infrastructure. Experimental results indicate that the proposed approach can reduce execution time without necessarily compromising the effectiveness of ontology alignments.

Categories and Subject Descriptors: C.1 [**Processor Architectures**]: Parallel Architectures

Keywords: Large Ontologies, MapReduce, Ontology Matching, Parallelism

1. INTRODUCTION

Ontology matching is a process that takes as input two different ontologies (O_1 and O_2) and identifies pairs of similar concepts (correspondences) in the two ontologies [Rahm 2011; Euzenat and Shvaiko 2013]. The process may accept parameters (e.g. weights, metrics and/or thresholds) and resources (e.g. dictionaries and other ontologies) to assist in the comparison of concepts. The set of correspondences forms the alignment (A') between ontologies O_1 and O_2. The similarity between the concepts is determined by employing matching algorithms (matchers). In each comparison involving a pair of concepts, multiple matchers can be applied to explore the various features of the concepts. Each matcher produces a similarity value that varies between 0 (no similarity) and 1 (complete similarity). The similarity values are combined using aggregation measures (e.g. average and weighted average). A similarity threshold is applied to identify potential matches (correspondences).

In traditional ontology matching, the comparison of concepts is guided by the Cartesian product, i.e., all concepts from ontology O_1 are compared with all concepts of ontology O_2. However, when dealing with the large ontology matching problem [Babalou et al. 2014], optimization techniques become necessary mainly to provide a reduction of the match search space [Rahm 2011] and/or parallel matching [Gross et al. 2010]. Partition-based matching [Hu et al. 2006] is used to limit the number of comparisons between concepts by partitioning the input ontologies into subontologies (partitions) and comparing only those concepts contained in similar pairs of subontologies. Parallel matching [Mestre and Pires 2014] consists in distributing the concept comparisons among various resources (e.g. computers or virtual machines). In this context, MapReduce [Dean and Ghemawat 2008] emerges as a major alternative for the efficient distributed data-intensive tasks due to its capability for being a

scalable parallel shared-nothing data-processing. This programming model hides the details of data distribution and allows users to focus on the data processing aspects.

In this paper, we investigate the parallelization for the complex problem of matching large ontologies. A MapReduce-based approach is proposed to coordinate the distributed computation of comparisons between concepts. We assume that the input ontologies are partitioned and the pairs of subontologies to be compared are defined. Then, our distributed approach is used to parallel the computation of concept comparisons aiming to reduce execution time. This paper is organized as follows. Section 2 provides a background on optimization techniques for large ontology matching. Section 3 discusses related work. Section 4 presents a formalization of partition-parallel-based ontology matching. Section 5 presents the MapReduce-based approach for large ontology matching. Section 6 presents the experiments and results. Section 7 presents our conclusions and suggestions for future work.

2. BACKGROUND

This section presents an overview of techniques commonly used to optimize large ontology matching: partitioning-based and parallel-based ontology matching.

2.1 Partition-based Ontology Matching

Partition-based ontology matching aims at partitioning the input ontologies O_1 and O_2 in such a way that the concepts inside a subontology (partition) of O_1 are matched against the concepts contained in another subontology of O_2 [Rahm 2011]. In general, such technique includes the following steps: i) *read files*: ontology files (e.g. in OWL format) are read and the content (e.g. concepts, relationships and ontology information) is loaded into the main memory; ii) *partition of ontologies*: concepts of an input ontology that are similar according to specific aspects (e.g. linguistic or structural) are grouped in subontologies. For each input ontology, a set of subontologies is produced; iii) *identify similar subontologies*: avoids comparing all subontologies of ontology O_1 with all subontologies of O_2. To

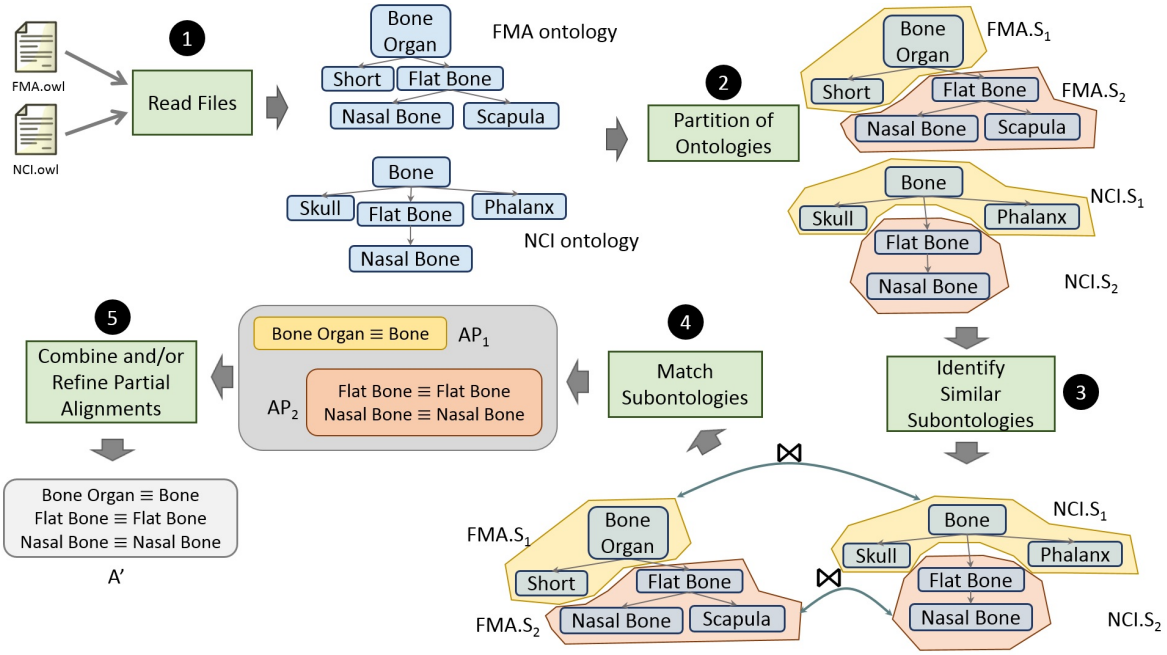

Fig. 1. Steps to match large ontologies.

this end, it identifies which pairs of subontologies have concepts with a certain degree of similarity in such a manner that comparisons are performed only between concepts of such pairs. Depending on the partitioning algorithm, this step can be performed along with the previous one; iv) *match subontologies*: performs comparisons between concepts (Cartesian product) of subontology pairs identified as similar in the previous step. The similarity between concepts is calculated by matchers. For each pair of subontologies compared, a partial alignment is generated; v) *combine and/or refine partial alignments*: combines the partial alignments (e.g. by the union of correspondences) to produce the final alignment. Optionally, the partial alignments can also be refined. Refinement (post-processing step) aims to remove inconsistencies and duplication of correspondences of partial alignments.

To better understand the process of matching large ontologies an illustration is depicted in Figure 1 (adapted from [Algergawy et al. 2011]). The example uses an excerpt from two large ontologies available at the BioPortal repository: FMA (Foundational Model of Anatomy) and NCI (National Cancer Institute Thesaurus). In step 1, the OWL files FMA.owl and NCI.owl are read and their information is loaded into memory, resulting in the representation of ontologies O_1 and O_2, respectively. The ontologies contain the concepts: O_1 = {*Bone Organ, Short, Flat Bone, Nasal Bone, Scapula*} and O_2 = {*Bone, Skull, Flat Bone, Phalanx, Nasal Bone, Rib*}. In step 2, each ontology is partitioned into subontologies: $O_1.S_1$ = {*Bone Organ, Short*} and $O_1.S_2$ = {*Flat Bone, Nasal Bone, Scapula*}; $O_2.S_1$ = {*Bone, Skull, Phalanx*} and $O_2.S_2$ = {*Flat Bone, Nasal Bone, Rib*}. In step 3, similar pairs of subontologies are identified. The concepts belonging to these pairs of subontologies will be compared. In the example, the pairs of subontologies $O_1.S_1 \bowtie O_2.S_1$ and $O_1.S_2 \bowtie O_2.S_2$ are compared. In step 4, the concepts belonging to each pair of subontologies are matched resulting in two partial alignments AP_1 and AP_2, where AP_1 = {[*Bone Organ, Bone*]} and AP_2 = {[*Flat Bone, Flat Bone*], [*Nasal Bone, Nasal Bone*]}. In step 5, the partial alignments are combined by means of an union operator to form the final alignment A' = {[*Bone Organ, Bone*], [*Flat Bone, Flat Bone*], [*Nasal Bone, Nasal Bone*]}.

2.2 Parallel-based Ontology Matching

Effective ontology matching, i.e., the identification of high effectiveness correspondences, normally requires the combined execution of multiple matchers to determine the similarity between ontology concepts. Typical matchers, such as semantic and structural ones, are time-consuming and memory-intensive. In addition, matching is typically performed on memory representations (graph structures) of the ontologies and requires the maintenance of several similarity values for every pair of concepts. Such requirements are even more increased when dealing with large ontologies.

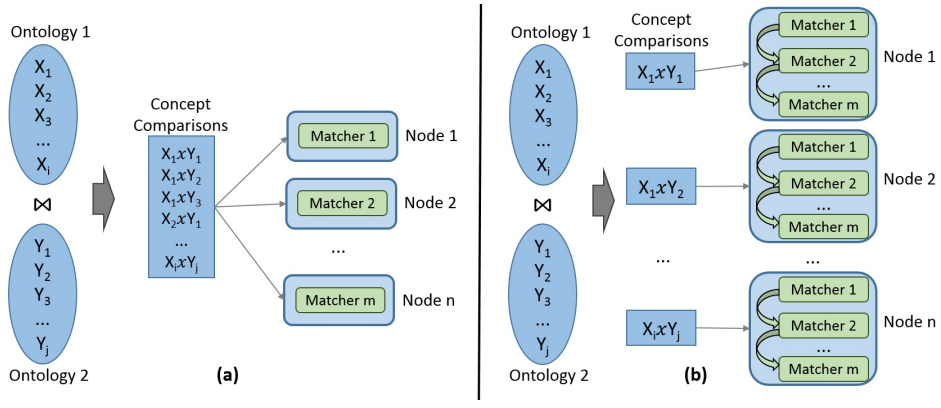

Fig. 2. Inter-matcher (a) and intra-matcher (b) parallelization.

The reduction of the search space (Section 2.1) does not guarantee that problems related to execution time and memory consumption will be solved [Dragisic et al. 2014]. As a result, techniques to parallel the process of ontology matching emerge as a complementary solution. According to Gross et al. [2010], there are essentially two ways to parallel the process of ontology matching which mainly differ in how matchers are run: *inter-matcher* and *intra-matcher*. In inter-matcher parallelization (Figure 2(a) – adapted from [Gross et al. 2010]), each matcher runs on a different node (computer, virtual machine or computer core) of a distributed computing infrastructure (e.g. a cloud or a cluster of machines). The pairs of concepts to be compared are sent to each matcher which performs comparisons in parallel. In intra-matcher parallelization (Figure 2(b) – adapted from [Gross et al. 2010]), the matching workflow runs in parallel. Each pair of concepts to be compared is sent to an available node where all matchers perform comparisons sequentially. In parallel, comparisons between other pairs of concepts are operated on different nodes.

A natural way of composing the matchers consists of improving the matching through the use of sequential composition [Euzenat and Shvaiko 2013]. For instance, a matcher based on labels can be used before running a matcher based on the structure of concepts. The previous matcher must send a partial alignment to the following matcher. The intra-matcher parallelization facilitates the communication between matchers since they are located at the same resource. For this reason, in this work we use intra-matcher to parallel the comparisons between concepts.

3. RELATED WORK

Challenges involving different aspects of ontology matching motivated the creation of competitions such as the one organized by the Ontology Alignment Evaluation Initiative (OAEI[1]). This competition brings together companies and universities around the world, in order to get faster and effectiveness solutions for the process of ontology matching. Concerning the OAEI track of large ontologies (Large Bio), in 2014, only six (LogMapLite, XMap, LogMap, AML, LogMap-C, and LogMap-Bio) of the fifteen algorithms submitted have completed all test cases. The majority burst memory ("out of memory") or exceeded the competition time threshold [Dragisic et al. 2014].

Parallel-based ontology matching has been treated in [Gross et al. 2010; Thayasivam and Doshi 2013; Amin et al. 2015; Zhang et al. 2012; 2011; Torre-Bastida et al. 2014], mainly to minimize the execution time of ontology matching by distributing concept comparisons among the resources (e.g. computers or virtual machines) of a distributed system. Some authors [Gross et al. 2010; Amin et al. 2014; Amin et al. 2015] propose their own architecture. The architecture proposed in [Gross et al. 2010] is divided into three services: data (extracts the information from ontologies), workflow (forwards the comparisons to available resources), and match (compares concepts and determines the correspondences). In [Amin et al. 2014; Amin et al. 2015], the strategy used to parallel ontology matching is similar to inter-matcher except that a node can execute any matcher. The work presents two dedicated services to distribute the comparisons among nodes and cores. Other authors [Thayasivam and Doshi 2013; Zhang et al. 2012; 2011; Torre-Bastida et al. 2014] use MapReduce to parallel the process of matching ontology. In [Thayasivam and Doshi 2013], MapReduce is used to minimize the execution time of four large ontology matching systems: Falcon-AO, Logmap, Optima+, and YAM++. Each node runs a specific ontology matching system. In [Zhang et al. 2012; 2011], the authors propose an approach that converts the input ontologies into virtual documents and analyzes cyclically the similarities between documents to define the correspondences. The work of Torre-Bastida et al. [2014] employs ontology matching techniques in the context of Linked Open Data (LOD). The techniques are used to extract information from entities while similarity metrics define the links between them.

Unlike all works discussed earlier, our approach considers as input pairs of subontologies (produced by a partitioning algorithm) in order to minimize the amount of comparisons between concepts.

[1]http://oaei.ontologymatching.org/

Regarding data transmission and fault tolerance, two relevant issues in distributed systems [Dean and Ghemawat 2008], the works of Gross et al. [2010], Amin et al. [2014] and Amin et al. [2015] do not describe how these issues are treated. The works that use MapReduce framework are benefited from its architecture features in order to tackle these problems. Unlike the works proposed by Zhang et al. [2012], Zhang et al. [2011] and Torre-Bastida et al. [2014], which uses five and four MapReduce jobs to generate the ontology alignment, respectively, our approach uses only one job. The higher the number of jobs, the higher tends to be the execution time of the matching process. Although it is not the main focus of our work, another difference between the aforementioned works and our approach is that we employ a (basic) load balancing strategy. We analyze the amount of comparisons to be performed in each pair of subontologies, based on the information provided by a partitioning algorithm, to distribute evenly the comparison blocks between the reduce tasks without executing another MapReduce job as in the previously mentioned works. Works related to LOD use the resulting correspondences of the ontology matching process to select potential entities to be linked, which are analyzed at the instance level (instance matching) [Nentwig et al. 2015]. Thus, solutions like ours can be combined with the work [Torre-Bastida et al. 2014] in order to improve the efficiency of the process.

In the literature, there are several algorithms to partition large ontologies: PBM (Partition-based block matching) [Hu et al. 2008], SCAN (Structural Clustering Algorithm for Networks) [Xu et al. 2007], DenGraph-HO [Schlitter et al. 2014], APP (Anchor, Partition, Partition) and PAP (Partition, Anchor, Partition) [Hamdi et al. 2010]. Among them, only PAP and APP partition one ontology based on the partitioning result of the other ontology. In addition, both algorithms produce a set of pairs of similar subontologies to be compared, where each subontology in a pair belongs to a different ontology. These are helpful characteristics for ontology matching since they avoid the comparison of concepts that have few chance of resulting in correspondences. In this work, we employ PAP to generate the input of our MapReduce-based approach, i.e., the pairs of subontologies to be compared. The PAP algorithm work as follows: one ontology is partitioned and before partitioning the other input ontology, anchor concepts (concepts considered similar in both ontologies) are determined using a linguistic similarity function. The anchor concepts, selected between two ontologies, are used to guide the partitioning of the other ontology and define which subontologies should be compared.

4. PROBLEM STATEMENT

In this section, we formalize the problem of partition-parallel-based Ontology Matching (OM). Let O_1 and O_2 be the input ontologies to be matched. The ontologies are composed by a set of concepts denoted by $O_1 = \{c_1, c_2, \ldots, c_j\}$ and $O_2 = \{c_1, c_2, \ldots, c_k\}$. Let $sim(c, c')$ be the similarity value between the concepts c and c' according to the aggregation (e.g., average) of similarity scores computed by matchers. The OM goal is to generate the partial alignments $PA(O_1, O_2)$ as a set of tuples in the form $\langle c, c', sim(c, c')\rangle$, s.t. $c \in O_1$, $c' \in O_2$ and $sim(c, c')$ is above a predefined input threshold ϕ.

To avoid computing $sim(c, c')$ for each $c \in O_1$ and $c' \in O_2$ (which leads to both high execution time and computational costs), the input ontologies are first submitted to a partitioning algorithm denoted by $\mathcal{C}l_{alg}$. By doing so, for each input ontology O_i, it is generated a set of subontologies denoted by $O_i.S = \{S_1, S_2, \ldots, S_m\}$, such that $\bigcap_{S \in O_i.S} = \emptyset$ and $\bigcup_{S \in O_i.S} = O_i$. Moreover, $\mathcal{C}l_{alg}$ also generates a list of subontology pairs denoted by $Pairs(O_1.S, O_2.S) = \{\langle S, S'\rangle_1, \langle S, S'\rangle_2, ..., \langle S, S'\rangle_p\}$, such that $S \in O_1.S$ and $S' \in O_2.S$, in order to indicate the pairs of subontologies that should be matched. After that, for each $\langle S, S'\rangle \in Pairs(O_1.S, O_2.S)$: $S \bowtie S'$ is computed (i.e., $sim(c, c')$ is evaluated for each $c \in S$ and $c' \in S'$). For each $\langle S, S'\rangle \in Pairs(O_1.S, O_2.S)$, a partial alignment PA is generated. Thus, the output of this task is a set of partial alignments denoted by $PA(O_1, O_2) = \{PA_1, PA_2, \ldots, PA_p\}$, such that each $PA \in PA(O_1, O_2)$ is a set of tuples in the form $\langle c, c', sim(c, c')\rangle$.

In order to optimize even more the OM process, the computation of $Pairs(O_1, O_2)$ can be executed in distributed computational resources. Let $E = \{n_1, n_2, \ldots, n_N\}$ be the set of nodes (e.g. virtual or physical machines) with homogeneous configuration in a cloud computing environment that are

Table I. Summary of the partition-parallel-based OM steps and goals.

	Input	$O_1, O_2, \mathcal{C}l_{alg}, \phi$
task 1	***Generate***	$O_1.S, O_2.S, Pairs(O_1.S, O_2.S)$
task 2	***Compute***	$S \bowtie S', s.t.\ \langle S, S' \rangle \in Pairs(O_1.S, O_2.S)$ using $E = \{n_1, n_2, \ldots, n_N\}$
goal 1	***Maximize***	$Recall(PA(O_1, O_2))$, $Precision(PA(O_1, O_2))$
goal 2	***Maximize***	$T_{exec}(O_1 \bowtie O_2) - T_{exec}(Pairs(O_1, O_2))$
goal 3	***Minimize***	$N - \left(\frac{T_{exec}(Pairs(O_1,O_2))}{T^E_{exec}(Pairs(O_1,O_2))} \right)$
	Output	$PA(O_1, O_2)$

available to compute $PA(O_1, O_2)$. We denote by $T_{exec}(Pairs(O_1, O_2))$ the execution time of computing $PA(O_1, O_2)$ by using a single node $n \in E$. In turn, we denote by $T^E_{exec}(Pairs(O_1, O_2))$ the execution time of computing $PA(O_1, O_2)$ by employing a cloud computing environment represented as E. Basically, we can summarize the problem as stated in Table I.

5. MATCHING LARGE ONTOLOGIES WITH MAPREDUCE

In this section, we describe our approach to match large ontologies in parallel. Parallelization is achieved using MapReduce [Dean and Ghemawat 2008], a programming model designed for parallel data-intensive computing in shared-nothing clusters. The approach is based on intra-matcher parallelization and focus on the step 4 of Figure 1. Thus, we assume that the set of subontology pairs has been previously generated by a partitioning algorithm ($\mathcal{C}l_{alg}$) (step 3) which also provides useful information (e.g. number of concepts in each subontology) that are used to evenly distribute the comparisons over the available computational resources. Analogously, since our approach produces a set of partial alignments, different strategies can be used to produce and refine the final alignment (step 5), such as trimming correspondences under a particular threshold or simply combining the partial alignments using an union operator [Euzenat and Shvaiko 2013].

Briefly, the MapReduce-based approach works as follows: for each pair of subontologies, the concepts of the smaller subontology (in terms of number of concepts) are replicated[2] w times in the map phase, where w is the number of reduce tasks. The concepts of the larger subontology are sent together with the replicated concepts of the smaller subontology to be compared in the same reduce task (reduce phase). Matchers compute the degree of similarity between each pair of concepts. The pairs of concepts that present a similarity value above a threshold (ϕ) are considered correspondences. The approach is divided into four phases: reading, map, shuffle, and reduce. Next, we describe each phase in more details. For the ease of understanding, we use the motivating example illustrated in Figure 3 throughout the section.

Reading Phase. Firstly, the concepts contained in each subontology are read and receive an identification key. Then, it is performed a statistical analysis of the amount of concept comparisons to be executed for each pair of subontologies. This analysis will help guiding the distribution of comparisons evenly among reduce tasks.

Each pair of subontologies is associated to an identifier (*pair_id*). The concepts are read and the information of each concept (e.g. label of concept, annotations, and label of children) are extracted to form a tuple. Each concept (tuple) receives a key composed as follows: $\langle$*pair_id.concept_id.replicated*$\rangle$, where *concept_id* is the concept identifier in the corresponding subontology and *replicated* indicates whether the concept will be replicated (*replicated* = 1) or not (*replicated* = 0). In Figure 3, there are two pairs of subontologies: $Pair\ 0 = \langle O_1.S_1, O_2.S_1 \rangle$ and $Pair\ 1 = \langle O_1.S_2, O_2.S_2 \rangle$. In $Pair\ 1$, $O_2.S_2$ is the smaller subontology since it has fewer concepts (2) than subontology $O_1.S_2$ (3). The key of

[2]If both subontologies have an identical size, the concepts of the second subontology are replicated.

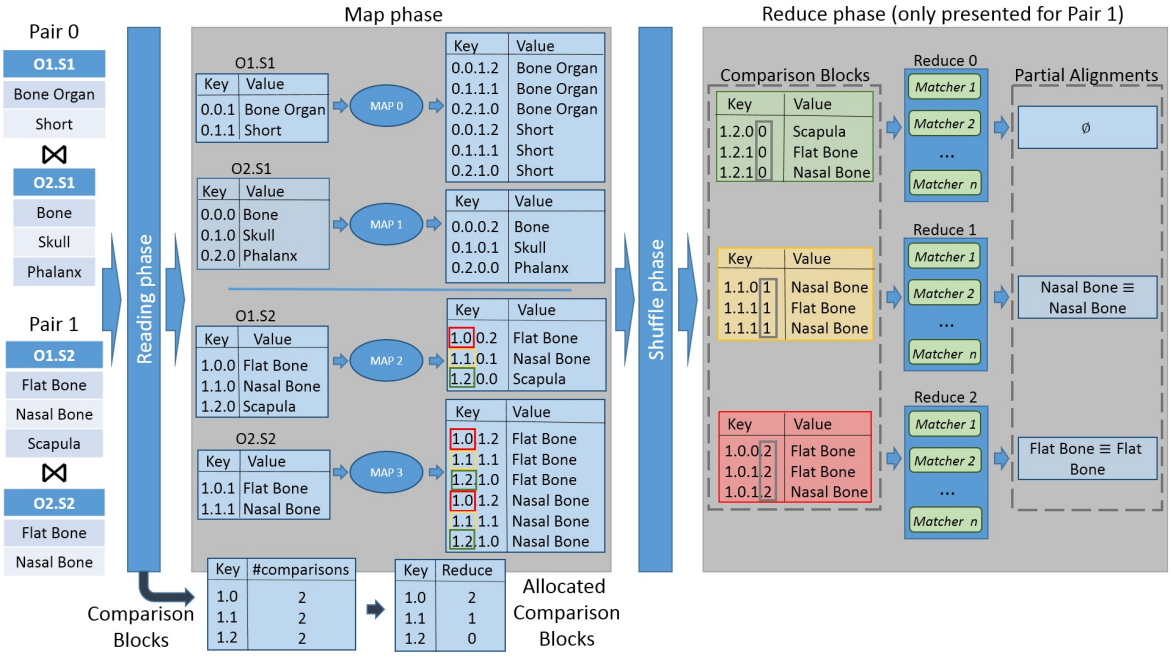

Fig. 3. Matching subontology pairs using MapReduce.

concept *FlatBone* is "1.0.1", where character "1" represents the *Pair* 1, "0" is the concept identifier in subontology $O_2.S_2$ and "1" indicates that the concept will be replicated.

After reading the concepts, for each pair of subontologies $\langle S, S' \rangle \in Pairs(O_1, O_2)$ the amount of comparisons is computed as $|S| * |S'|$. Also, the smaller and bigger subontologies (S_s and S_b, respectively) are determined, regarding the amount of concepts. Considering that $R = \{r_1, r_2, ..., r_w\}$ is the set of reduce tasks, since the comparisons in each pair of subontologies are guided by the Cartesian product ($S_s \bowtie S_b$), we divide the amount of concepts in the bigger subontology $|S_b|$ by the number of reduces tasks w. As a result, each subset of the bigger subontology forms a comparison block (CB). A CB contains $|S_b|/w$ concepts of S_b and all concepts of S_s. The number of comparisons in a CB is computed as $\#comparisons = (|S_b|/w) * |S_s|$. Each CB of a subontology pair receives a key in the format: $\langle pair\_id.comparison\_block\_id \rangle$, where $pair\_id$ identifies the subontology pair and $comparison\_block\_id$ is the comparison block identifier based on the number of reduce tasks (ranging from 0 to the $w - 1$). $T = \{CB_1, CB_2, ..., CB_g\}$ is the list of comparison blocks of all pairs in $Pairs(O_1.S, O_2.S)$. In Figure 3, the comparison blocks are formed as follows: $T = \{\langle\langle 1.0\rangle, 2\rangle, \langle\langle 1.1\rangle, 2\rangle, \langle\langle 1.2\rangle, 2\rangle\}$. T is then ordered in descending order by the amount of comparisons ($\#comparisons$) in each comparison block. We denote the resulting list by T_{sort}. The function $emptiestReduceTask(R) = r$ returns the index of the reduce task (r) with fewest comparisons to be allocated. T_{sort} is iterated and each comparison block is guided to a reduce task to form the set of tuples $CB_{Allocated} = \{a_1, a_2, ..., a_h\}$, s.t. each $a \in CB_{Allocated}$ is represented as $a = \langle\langle pair\_id.comparison\_block\_id\rangle, emptiestReduceTask(R)\rangle$. The set $CB_{Allocated}$ is saved to guide the allocation of comparisons between the reduce tasks in the next phases. In Figure 3, the allocation of CBs is done as follows: $CB_{Allocated} = \{\langle\langle 1.0\rangle, 2\rangle, \langle\langle 1.1\rangle, 1\rangle, \langle\langle 1.2\rangle, 0\rangle\}$. Thus, the three comparison blocks will be evenly distributed among the available reduce tasks (shuffle phase).

Map Phase. This phase modifies the concept keys and replicates the concepts in such a manner that the keys can guide the formation of comparison blocks. The map tasks receive the key-values ⟨key, concept⟩ and replicate the concepts in which the *replicated* key character is equal to 1. For each replicated concept, the $concept\_id$ of the key is replaced by the $concept\_id$ of the concept belonging

to the larger subontology. The map tasks also append a *reduce_id* character, returned by the set $CB_{Allocated}$ (generated in the previous phase) that determines to which reduce task the concept will be forwarded. In the motivating example, the concept *Flat Bone* (subontology $O_2.S_2$) is replicated three times, where three is the size of the larger subontology ($O_1.S_2$). For each replication, the *concept_id* of *Flat Bone* is replaced by the *concept_id* of the concepts *Flat Bone*, *Nasal Bone*, and *Scapula*. Thus, *Flat Bone* (subontology $O_1.S_2$) and *Flat Bone* (subontology $O_2.S_2$) have the same two first characters of key that identifies a comparison block. Also, note that the map task appends the value 1 to the character *reduce_id* of concept *Nasal Bone* key.

On the other hand, if the concept must not be replicated (*replicated* = 0), only the reduce number is appended to the concept key. In the example, the key of *Scapula* is "1.2.0", where "0" indicates that the concept will not be replicated. Thus, the value "0" (*reduce_id*) is appended to the key resulting in "1.2.0.0". Finally, the key-values ⟨key, concept⟩ are emitted to the next phase (shuffle). The map algorithm is detailed in Appendix A.

Shuffle Phase. In this phase, the set of key-values ⟨key, concept⟩ is partitioned, sorted, and grouped. To form the comparison blocks, the key-values ⟨key, concept⟩ are grouped and sorted by the key. The concepts are grouped by the first two characters of their keys (*pair_id* and *concept_id*). Figure 3 illustrates the formation of a comparison block. The concepts *FlatBone*, *FlatBone* and *NasalBone* (red comparison block) are grouped because they have the same first two characters of their keys ("1.0") and form *Comparison Block* 0. To sort the concepts in a comparison block, all key characters are considered. Finally, the set of ⟨key, concepts⟩ are sent to the reduce phase. To this end, in the partition task, the last key character (*reduce_id*) of the concepts determines to which reduce task the concepts that compose a comparison block must be allocated. In Figure 3, the concepts *Scapula*, *Flat Bone* and *Nasal Bone* of *Comparison Block* 0 are allocated to the reduce task *reduce* 0, since the character *reduce_id* of each concept's key is equal to 0.

Reduce Phase. Each reduce task receives several comparison blocks. In a comparison block, the subset of concepts of the larger subontology are compared with all concepts of the smaller subontology. The first concepts of a comparison block belong to the larger subontology, since the third character of its key is "0". The other concepts belong to the smaller ontology (third character is equal to "1"). The comparisons are executed by the matchers that return a similarity value for each comparison. The pair of concepts with similarity above a threshold (ϕ) is considered a correspondence. The output of each reduce task is the set of correspondences that forms a partial alignment.

In Figure 3, due to space restrictions only the comparisons of *Pair* 1 are shown. The comparison block formed by the concepts *Scapula*, *Flat Bone* and *Nasal Bone* is allocated to *reduce* 0 (*reduce_id* is equal to 0), where *Scapula* (the first concept) belongs to $O_1.S_2$ (larger subontology) and the other concepts belong to $O_2.S_2$ (smaller subontology). Thus, the concept *Scapula* is compared with the concepts *Flat Bone* and *Nasal Bone*. Each comparison produces a similarity value: *Scapula* × *Flat Bone* = 0.1 and *Scapula* × *Nasal Bone* = 0.12. Since the similarity values are below the threshold ($\phi = 0.7$), *reduce* 0 produces an empty partial alignment while *reduce* 1 and *reduce* 2 identify the correspondences $NasalBone \equiv NasalBone$ and $FlatBone \equiv FlatBone$, respectively. The reduce algorithm is detailed in Appendix A.

6. EXPERIMENTS

In this section, we evaluate the proposed MapReduce-based approach on a cloud computing infrastructure. The focus of the experiments[3] addresses the following research questions: i) *Is the proposed MapReduce-based approach comparable to existing state-of-the-art approaches in terms of performance?* (Goal 3 of Table I); ii) *Do partitioning algorithms improve the performance of the MapReduce-based*

[3]Available at https://sites.google.com/site/dqgroupufcg/mapreduce-approach-for-matching-large-scale-ontologies

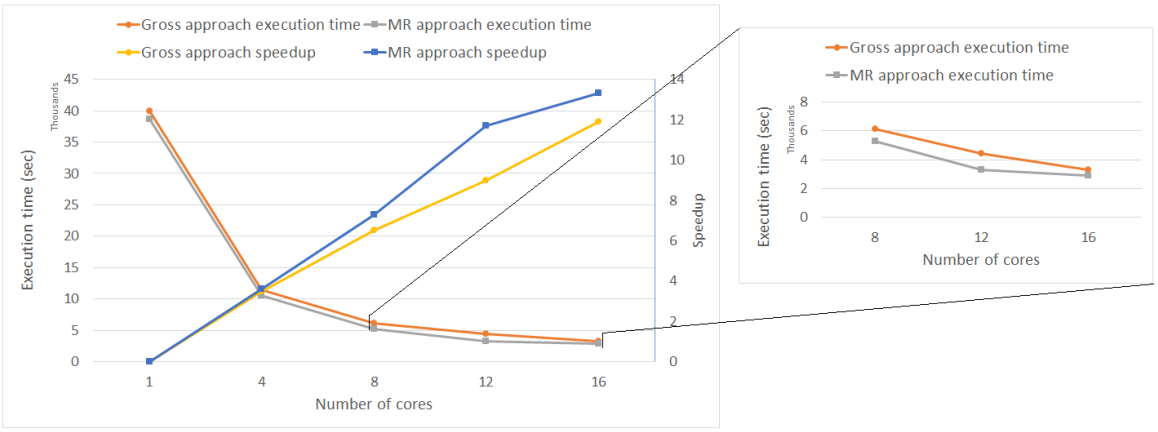

Fig. 4. Execution time and speedup of the Gross and MapReduce approaches.

approach without compromising its effectiveness? (Goals 1 and 2); and iii) *Is the MapReduce-based approach scalable in a context of multiple matchers?* (Goals 1 and 3). To answer these questions, the following experiments were proposed.

6.1 Performance Analysis

In this experiment, we perform a comparative performance analysis of our approach and the one proposed by Gross et al. [2010] (not based on MapReduce). As input ontologies, we use Molecular Functions (MF) and Biological Processes (BP), consisting of 9,395 and 17,104 concepts, respectively (versions of June 2009). Both ontologies were extracted from the GO[4] ontology. For a fair comparison, both approaches need to execute the same amount of comparisons. Thus, since the Gross' approach follows the Cartesian product to perform the comparisons between concepts, we ignored the partitioning step in this experiment. The similarity values are computed by applying three linguistic matchers [Gross et al. 2010] namely: Label (compares the labels of concepts), Children (compares the labels of children concepts), and Name Path (compares the labels of ancestor concepts). Trigram similarity is used by the matchers to determine the similarity between the labels. The experiments were performed on a cloud infrastructure with four nodes, each one with 4 cores (we use up to 16 cores). Each node has an Intel(R) Xeon(R) 4x1.0GHz CPU, 4GB memory and runs the 64-bit Debian GNU/Linux OS with a 64-bit JVM. Regarding MapReduce, we used the implementation Hadoop 2.6.0 configured with two map tasks and four reduce tasks per node (following Hadoop's documentation).

The results of the comparative performance analysis are shown in Figure 4. Regarding execution time, our MapReduce-based approach outperformed the Gross' approach for all variations of the number of cores, enabling a reduction time of 38,650 seconds (with 1 core) and 2,910 seconds (16 cores). For the range between 8 and 16 cores, our approach obtained, on average, a performance of 803 seconds faster than the Gross' approach. Concerning speedup which measures how much faster a process runs in parallel (i.e., the ratio of time it takes to run a process on 1 core (or node) and the time it takes to run the same process on C cores, where C is the amount of cores), our approach also obtained better results than the Gross' approach, for all variations of cores. In the range between 8 and 16 cores, we achieved the greatest differences of speedup. The MapReduce-based approach obtained, on average, a speedup of 1.63 greater than Gross' approach. Thus, based on the experimental results, we conclude that our approach achieved a higher performance than the Gross' approach.

[4]http://geneontology.org/

6.2 Performance and Effectiveness Analysis with(out) Partitioning

The purpose of this experiment is to evaluate if the partitioning step can improve execution time whilst maintaining the effectiveness of alignments. To this end, we executed our approach with and without (in this case, following the Cartesian product) partitioning. We use F-measure to measure the effectiveness of alignments, that compares the final alignment to a reference one (gold standard). The final alignment contains 1:1 correspondences (i.e., a concept of an ontology is related to only one concept of the other ontology). This was obtained by trimming correspondences below the similarity threshold. Particularly, for the approach with partitioning, the execution time includes both partitioning and matching times.

In this experiment, we used the pair of ontologies Adult Mouse Anatomy (2,744 concepts) and NCI Thesaurus (3,304 concepts), provided by OAEI[5]. To determine the similarity between concepts, we used three linguistic matchers [Euzenat and Shvaiko 2013]: Label, Children, and Annotation (compares the labels of concepts' annotations). The label similarity was determined by averaging the results of Levenshtein distance and Trigram. To partition the ontologies we used the PAP clustering algorithm described in related work. Two pairs of subontologies were generated by PAP. The resulting subontologies contain 1,256 and 1,379 concepts from NCI Thesaurus. In turn, the subontologies from the Adult Mouse Anatomy have 1,440 and 1,228 concepts. Following the PAP algorithm, the concepts that were not included in subontologies to be compared were not considered in the matching process. The experiments were performed on a cloud infrastructure with 32 nodes, each one with 1 core. The configuration of each node is the same as described in the previous experiment. We used the same implementation of MapReduce of the previous experiment, however, configured with one map task for each two nodes and one reduce task per node, since each node had one core.

According to the results shown in Figure 5(a), it is possible to identify a considerable reduction in execution time and a small decrease in effectiveness (F-measure) when using the partitioning algorithm. The time required to partition the ontologies sequentially (1 node) is around 2 seconds. Up to 12 nodes, the execution time decreased on average 100 seconds. When we used 4 nodes, the execution time was reduced by more than half. Regarding the effectiveness of the generated alignment (with threshold $\phi = 0.8$), while the approach without partitioning (Cartesian product) obtained a F-measure of 80%, the one with partitioning reported a F-measure of 78%. Even with the small decrease of F-measure, the usage of partitioning algorithms appears promising taking into account the gains obtained regarding execution time. Thus, we can infer that the use of a partitioning algorithm can reduce considerably the execution time, without necessarily compromising significantly the effectiveness of generated alignments.

6.3 Performance and Effectiveness Analysis using Multiple Matchers

In this experiment, we evaluated the impact of combining matchers in the effectiveness and performance in the MapReduce-based ontology matching. We use the same refinement strategy of partial alignment used in the previous experiment to generate the final alignment. To prevent that partitioning algorithm interfere in the analysis of the effectiveness, in this experiment the comparison of concepts is guided by the Cartesian product of the following pair of ontologies: Foundational Model of Anatomy (FMA - 78,989 concepts) and National Cancer Institute Thesaurus (NCI - 66,724 concepts), provided by OAEI. We evaluated the execution time and effectiveness of MapReduce-based approach using the Label matcher and the combination of the matchers Label and Annotation. To determine the linguistic similarity in the Label and Annotation matchers, it was used the Levenshtein distance algorithm. The combination is performed executing firstly the Label matcher. If the similarity value is greater than the similarity threshold ($\phi = 0.8$), the concepts are considered similar and the Annotation matcher is not executed. Otherwise, the Annotation matcher executes and checks if any of the

[5]http://oaei.ontologymatching.org/2013/

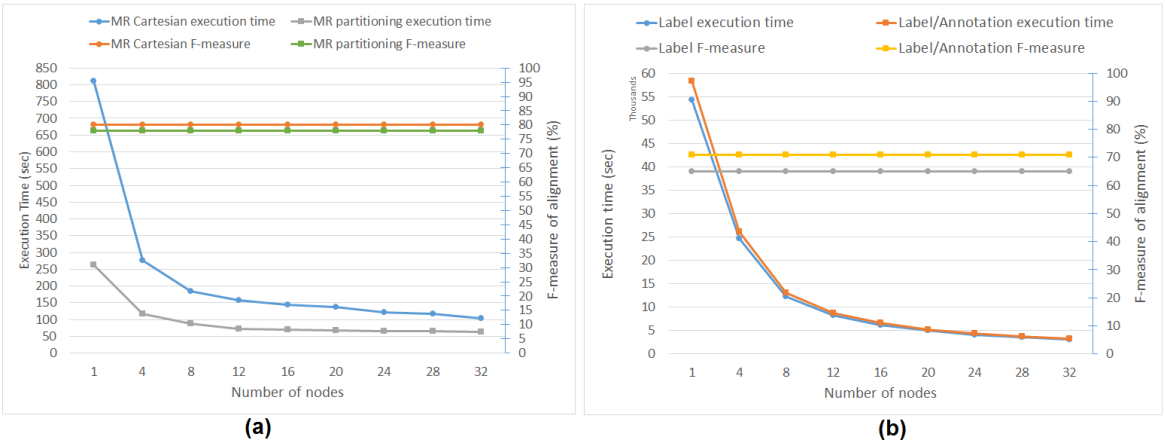

Fig. 5. (a) Reported execution time and F-Measure of our approach with and without partitioning; (b) Performance and effectiveness of our approach using Label (single) and Label/Annotation (aggregated) matchers.

annotations (in the concept) have a similarity value greater or equal than the similarity threshold. In this experiment, we employed the same cloud settings of the previous experiment.

According to Figure 5(b), we can observe that with the increase the number of nodes, the difference of execution time between the Label (single) and Label/Annotation (aggregated) matchers tend to decrease. Initially (1 node), the difference of execution time is 3,960 seconds; with 32 nodes this difference is reduced to 259 seconds. Regarding the effectiveness of the alignments, we realize an increase in F-measure from 65% to 71% by using of the combination of the Label and Annotation matchers. Clearly, the combination of multiple matchers tends to improve the effectiveness of alignments generated by the ontology matching process. However, its usage increases significantly the execution time of the process. Thus, our MapReduce-based approach emerges as a scalable solution which benefits the use of multiple matchers to maintain reasonable execution times.

7. CONCLUSIONS AND FUTURE WORK

Areas such as health have several large ontologies, for example, Open Biomedical Ontologies Foundry (OBO), Gene Ontology (GO) and OpenGalen. In the area of astronomy stands out the International Virtual Observatory ontology repository Alliance (IVOA). In agriculture, the large ontology AGROVOC created by the Food and Agriculture Organization (FAO) is considered one of the most important of the area. With the growing number of large ontologies, there is also a need to match related ontologies and still maintain reasonable execution time. In this context, this work proposes an approach, based on MapReduce, in order to reduce the execution time of the matching process involving large ontologies. Based on the experimental results, we observed that the MapReduce-based ontology matching approach (with partitioning) can provide reasonable execution times and effective alignments. In addition, the MapReduce-based approach benefits the usage of multiple matchers without greatly increasing the execution time.

In future work, we will investigate better strategies to distribute the comparisons between concepts evenly in order to improve even more the efficiency of the proposed matching approach. We also intend to investigate how large ontologies can be partitioned in parallel using the PAP algorithm.

REFERENCES

Algergawy, A., Massmann, S., and Rahm, E. A Clustering-Based Approach for Large-Scale Ontology Matching. In *Proceedings of 15th International Conference on Advances in Databases and Information Systems, ADBIS*. Vienna,

Austria, pp. 415–428, 2011.

Amin, M. B., Batool, R., Khan, W. A., Lee, S., and Huh, E.-N. SPHeRe - A Performance Initiative Towards Ontology Matching by Implementing Parallelism over Cloud Platform. *The Journal of Supercomputing* 68 (1): 274–301, 2014.

Amin, M. B., Khan, W. A., Lee, S., and Kang, B. H. Performance-based Ontology Matching. *Applied Intelligence* 43 (2): 356–385, 2015.

Babalou, S., Kargar, M. J., and Davarpana, S. H. A Comprehensive Review Of The Ontology Matching Systems By A Focus On Large Ontologies. In *International Congress of Chemical and Process Engineering*. Prague, Czech Republic, pp. 357–373, 2014.

Dean, J. and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 51 (1): 107–113, 2008.

Dragisic, Z., Eckert, K., Euzenat, J., Faria, D., Ferrara, A., Granada, R., Ivanova, V., Jimenez-Ruiz, E., Kempf, A. O., Lambrix, P., Montanelli, S., Paulheim, H., Ritze, D., Shvaiko, P., Solimando, A., Trojahn, C., Zamazal, O., and Grau, B. C. Results of the Ontology Alignment Evaluation Initiative 2014. In *International Workshop on Ontology Matching*. Trentino, Italy, pp. 61–104, 2014.

Euzenat, J. and Shvaiko, P. *Ontology Matching*. Springer-Verlag, Heidelberg (DE), 2013.

Gross, A., Hartung, M., Kirsten, T., and Rahm, E. On Matching Large Life Science Ontologies in Parallel. In P. Lambrix and G. Kemp (Eds.), *Data Integration in the Life Sciences*. Lecture Notes in Computer Science, vol. 6254. Springer, Berlin Heidelberg, pp. 35–49, 2010.

Hamdi, F., Safar, B., Reynaud, C., and Zargayouna, H. Alignment-Based Partitioning of Large-Scale Ontologies. In F. Guillet, G. Ritschard, D. A. Zighed, and H. Briand (Eds.), *Advances in Knowledge Discovery and Management*. Studies in Computational Intelligence, vol. 292. Springer, Berlin Heidelberg, pp. 251–269, 2010.

Hu, W., Qu, Y., and Cheng, G. Matching large ontologies: a divide-and-conquer approach. *Data & Knowledge Engineering* 67 (1): 140–160, 2008.

Hu, W., Zhao, Y., and Qu, Y. Partition-Based Block Matching of Large Class Hierarchies. In R. Mizoguchi, Z. Shi, and F. Giunchiglia (Eds.), *The Semantic Web*. Lecture Notes in Computer Science, vol. 4185. Springer, Berlin Heidelberg, pp. 72–83, 2006.

Mestre, D. G. and Pires, C. E. S. Efficient Entity Matching over Multiple Data Sources with MapReduce. *Journal of Information and Data Management* 5 (1): 40–51, 2014.

Nentwig, M., Hartung, M., Ngomo, A.-C. N., and Rahm, E. A Survey of Current Link Discovery Frameworks. *Semantic Web Journal*, 2015.

Rahm, E. Towards Large-Scale Schema and Ontology Matching. In Z. Bellahsene, A. Bonifati, and E. Rahm (Eds.), *Schema Matching and Mapping*. Data-Centric Systems and Applications, vol. 9. Springer, Berlin Heidelberg, pp. 3–27, 2011.

Schlitter, N., Falkowski, T., and Lassig, J. DenGraph-HO: a density-based hierarchical graph clustering algorithm. *Proceedings of the International Conference on Database and Expert Systems Applications* 31 (5): 469–479, 2014.

Thayasivam, U. and Doshi, P. Speeding Up Batch Alignment of Large Ontologies Using MapReduce. In *IEEE International Conference on Semantic Computing*. Irvine, USA, pp. 110–113, 2013.

Torre-Bastida, A., Villar-Rodriguez, E., Ser, J. D., Camacho, D., and Gonzalez-Rodriguez, M. On Interlinking Linked Data Sources by Using Ontology Matching Techniques and the Map-Reduce Framework. In *Proceedings of the 15th International Conference on Intelligent Data Engineering and Automated Learning*. Salamanca, Spain, pp. 53–60, 2014.

Xu, X., Nurcan Yuruk, Z. F., and Schweiger, T. A. J. SCAN: a structural clustering algorithm for networks. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. San Jose, USA, pp. 824–833, 2007.

Zhang, H., Hu, W., and Qu, Y. Constructing Virtual Documents for Ontology Matching Using MapReduce. In *Proceedings of The Joint International Semantic Technology Conference on Semantic Web*. Hangzhou, China, pp. 48–63, 2011.

Zhang, H., Hu, W., and Qu, Y. VDoc+: a virtual document based approach for matching large ontologies using MapReduce. *Journal of Zhejiang University - Science C* 13 (4): 257–267, 2012.

APPENDIX A. MAP AND REDUCE ALGORITHMS

As illustrated in Algorithm 1, the key concept is initially subdivided into the variables: *pair_id*, *concept_id* and *replicated* (lines 2 to 4). Considering the number that identifies the subontology pair to which the concept belongs to (line 5), the number of replications is recovered in the global variable ($configuration$). If the concept must not be replicated (line 6), the reduce task identifier is recovered

from the list ($matchTasks$) that stores the pairs: comparison block identifier ($pair\_id.concept\_id$) and reduce task identifier (line 7). The variable *reduce* is appended to the key (line 8) and the pair $\langle key, concept \rangle$ is emitted to the next phase (line 9). Otherwise, the concept is replicated (line 11) and the $concept\_id$ (line 12) is changed. The reduce task identifier is recovered (line 13) and appended to the key (line 14). Finally, the pair $\langle key, concept \rangle$ is emitted (line 15).

Algorithm 1 Map algorithm

```
1:  function MAP(key, concept)
2:      pair_id ← key[0]
3:      concept_id ← key[1]
4:      replicated ← key[2]
5:      numberReplications ← configuration.get(pair_id)
6:      if replicate = 0 then
7:          reduce ← matchTasks.get(pair_id.concept_id)
8:          key ← key.append(reduce)
9:          emit(key, concept)
10:     else
11:         for i = 0 → numberReplications do
12:             key ← pair_id.i.replicated
13:             reduce ← matchTasks.get(pair_id.i)
14:             key ← key.append(reduce)
15:             emit(key, concept)
```

Algorithm 2 Reduce algorithm

```
1:  function REDUCE(key, list(concepts))
2:      threshold ← configuration.get(Threshold_Value)
3:      conceptsLargerOntology ← {}
4:      result ← null
5:      for each concept in list(concepts) do
6:          if concept.isNoReplicated() then
7:              conceptsLargerOntology.add(concept)
8:          else
9:              conceptSmallerOntology ← concept
10:             maxSimilarity ← 0
11:             for each conceptLarge in conceptsLargerOntology do
12:                 similarity ← computeMatchers(conceptLarge, conceptSmallerOntology)
13:                 if similarity > threshold and similarity > maxSimilarity then
14:                     maxSimilarity ← similarity
15:                     result ← conceptLarger × conceptSmallerOntology : maxSimilarity
16:             if result ≠ null then
17:                 emit(key, result)
```

As detailed in Algorithm 2, with the parameter $Threshold\_Value$ (line 2), the value of *threshold* is recovered from the global variable ($configuration$) and the variables are initialized (lines 3 and 4). The list of concepts is iterated (line 5), the concepts belongs to the larger ontology are separated of the concepts belongs to the smaller ontology (lines 6 to 9). The variable $maxSimilarity$ (line 10) indicates the highest value of similarity between the concepts which were already compared. For each concept of the smaller ontology, it is computed the value of *similarity* with the concepts of the larger ontology (lines 11 and 12). If the *similarity* is larger than *threshold* and $maxSimilarity$, the variable $maxSimilarity$ receives the new value of similarity and *result* assumes the new pair of concepts with

the value of similarity between them (lines 13 to 15). If there is a pair of concepts with the value of similarity above *threshold* (line 16), the result is emitted to the output of the reduce task (line 17).