J Supercomput
DOI 10.1007/s11227-013-1037-1

SPHeRe

A Performance Initiative Towards Ontology Matching by Implementing Parallelism over Cloud Platform

Muhammad Bilal Amin · Rabia Batool · Wajahat Ali Khan · Sungyoung Lee · Eui-Nam Huh

Abstract The abundance of semantically related information has resulted in semantic heterogeneity. Ontology matching is among the utilized techniques implemented for semantic heterogeneity resolution; however, ontology matching being a computationally intensive problem can be a time-consuming process. Medium to large-scale ontologies can take from hours up to days of computation time depending upon the utilization of computational resources and complexity of matching algorithms. This delay in producing results, makes ontology matching unsuitable for semantic web-based interactive and semireal-time systems. This paper presents SPHeRe, a performance-based initiative that improves ontology matching performance by exploiting parallelism over multicore cloud platform. Parallelism has been overlooked by ontology matching systems. SPHeRe avails this opportunity and provides a solution by: (i) creating and caching serialized subsets of candidate ontologies with single-step parallel loading; (ii) lightweight matcher-based and redundancy-free subsets result in smaller memory footprints and faster load time; and (iii) implementing

M.B. Amin · R. Batool · W.A. Khan · S. Lee (✉)
Ubiquitous Computing Lab, Department of Computer Engineering, Kyung Hee University, Global Campus, 1 Seocheon-dong, Giheung-gu, Yongin-si, Gyeonggi-do 446-701, South Korea
e-mail: sylee@oslab.khu.ac.kr

M.B. Amin
e-mail: mbilalamin@oslab.khu.ac.kr

R. Batool
e-mail: rabia@oslab.khu.ac.kr

W.A. Khan
e-mail: wajahat.alikhan@oslab.khu.ac.kr

E.-N. Huh
Internet Computing and Network Security Lab, Department of Computer Engineering, Kyung Hee University, Global Campus, 1 Seocheon-dong, Giheung-gu, Yongin-si, Gyeonggi-do 446-701, South Korea
e-mail: johnhuh@khu.ac.kr

Published online: 18 October 2013

data parallelism based distribution over subsets of candidate ontologies by exploiting the multicore distributed hardware of cloud platform for parallel ontology matching and execution. Performance evaluation of SPHeRe on a trinode (12-core) private cloud infrastructure has shown up to 3 times faster ontology load time with up to 8 times smaller memory footprint than Web Ontology Language (OWL) frameworks Jena and OWLAPI. Furthermore, by utilizing the computation resources most efficiently, SPHeRe provides the best scalability in contrast with other ontology matching systems, i.e., GOMMA, LogMap, AROMA, and AgrMaker. On a private cloud instance with 8 cores, SPHeRe outperforms the most performance efficient ontology matching system GOMMA by 40 % in scalability and 4 times in performance.

Keywords Ontology matching · Semantic web · Matching performance · Parallel matching · Parallel programming · Cloud computing

1 Introduction

In the era of globalization and automation, integration of information has become a key tool for providing knowledge driven services [29]. With the abundance of information over the web, problems regarding information heterogeneity have emerged. The heterogeneity is classified into two types: data heterogeneity and semantic heterogeneity [36]. Data heterogeneity has solutions based on data definitions, types, formats, and precision [36]. Tools like the Microsoft Biztalk Server [49] are also used for data integration and heterogeneity resolution as described in [48] and [28]. Semantic heterogeneity; however, involves data's intend [36], making it a challenging opportunity for integration [29]. The volume of data makes manual annotation of concepts unrealistic; consequently, automated solutions based on ontologies are used by software agents [34]. The most prominent solution for semantic heterogeneity resolution is ontology matching, which determines conformity among semantically related ontologies. Mappings drawn from ontology matching can be further used in information systems and database integration, e-commerce systems, semantic web services, and social networks [30].

Effective ontology matching is a computationally intensive operation requiring Resource-based matching algorithms (Name-based, Hierarchy-based, Annotation-based, and Property-based) to be executed over candidate ontologies. As mentioned in [35], ontology matching between two ontologies is a Cartesian product of all the concepts and their relationships leading to quadratic complexity with respect to ontology size. In case of medium ($\sim$3,000 concepts) to large-scale (10,000+ concepts) ontologies [35], computation and memory utilization peaks due to the size of the ontology and relationships among its concepts. In our experiments on relatively medium-size ontologies (Adult Mouse Anatomy [37] with anatomical part of the NCI Thesaurus [57]), matching algorithms have taken 20 minutes to obtain desirable matched results. Over very large ontologies (whole FMA [62] with whole NCI [40]), executing matching algorithms have taken 3 days to produce desirable results. This delay makes ontology matching ineffective for applications with in-time processing demands like interactive semantic web systems and systems with intelligent information retrieval tasks [58]. Apart from computation needs, memory requirements for matching operations are also higher. Matching algorithms evaluating over the relationships of concepts require concept-graphs in the main memory, producing memory strains in gigabytes during execution. During our experiments over whole FMA with NCI matching, JVM heap crashes and out-of-memory errors have occurred quite often, even with 2 GB of heap memory available.

Several ontology matching systems have emerged over the years. Primary motivation of these tools is higher accuracy complemented by efficiency in terms of performance; however, the core techniques for achieving better performance are either related to the optimization of matching algorithms or the fragmentation of ontologies for matching algorithms [35]. The performance improvement based on exploitation of newer technology has largely been missing. Among these technologies are parallel and distributed platforms available for utilization [56].

In earlier years, parallel and distributed platforms were associated with High-Performance Computing (HPC) [47]; however, today's commodity hardware is equipped with multicore processors, enabling them to utilize individual cores as virtual processors, providing hardware parallelism. A great resource of affordable commodity-based hardware infrastructure is today's cloud platform. Cloud computing [23] has recently emerged as a computing platform with reliability, ubiquity, and availability in focus [22]. The utility of cloud as resource and service provider has already been investigated; however, the benefits of cloud's commodity hardware based distributed infrastructure are still overlooked [22]. Cloud being a collection of multicore processors and utility-based pricing model, can provide an affordable yet adequately efficient hardware infrastructure that can be used for ontology matching.

The problems mentioned in the above experiments and the opportunity of affordable commodity infrastructure in the form of cloud provides the motivation for an ontology matching system with components to perform parallel matching and execution on distributed multicore processors. A performance driven initiative is required that can be used in interactive and semi-real-time semantic-based systems. SPHeRe (*__S__ystem for __P__arallel __He__trogeneity __Re__solution*) is one such initiative that exploits multicore distributed computation resources of our private cloud [23] for parallel matching and execution; consequently, providing better performance. This paper contributes toward performance aspect of SPHeRe.[1] It scores better performance by:

- creating subsets of ontologies depending on the needs of matching algorithms and caches them in serialized formats, providing a single-step ontology loading for matching algorithms in parallel,
- each subset is lightweight due to matcher-based and redundancy-free creation, providing smaller memory footprints and contributing in overall system performance,
- implementing data parallelism [51] based distribution on subsets of candidate ontologies over multicore hardware and providing a collection of mappings among the ontologies as a bridge ontology file.

SPHeRe uses storage services of a public cloud [23] provider for bridge ontology delivery. Serialized subset creation and caching of candidate ontologies enables

[1] Accuracy aspect of SPHeRe is beyond the scope of this paper and has been catered in another publication; however, accuracy with performance results for FMA with NCI ontology are presented in Sect. 5.

SPHeRe to outperform Jena [3], and OWLAPI [19] by providing 3 times faster ontology load time and 8 times smaller memory footprint. SPHeRe's data parallelism based distribution exploits computational resources most efficiently and provides best scalability in contrast with GOMMA [9], LogMap [16], AROMA [4], and AgrMaker [1]. With respect to overall system performance, one our private cloud [23] infrastructure, SPHeRe outperforms GOMMA, most scalable and performance efficient ontology matching system available by 40 % in scalability and 4 times in performance.

The rest of the paper is organized as follows. In Sect. 2, we describe the related work in the field of ontology matching from the perspective of performance. In Sect. 3, we describe our proposed methodology of SPHeRe. In Sect. 4, we present the implementation details of SPHeRe's components. Section 5 describes the experimentation performed over a trinode private cloud for SPHeRe's performance evaluation. Section 6 discusses the parameters contributing to SPHeRe's performance in contrast with other OWL frameworks and ontology matching systems. Section 7 concludes this paper.

2 Related Work

Ontology matching has been an area of interest for researchers in the last decade. Many tools and techniques have evolved over the years. From techniques perspective, considerable amount of research has been driven toward optimization of ontology matching algorithms for better performance [35]. Consequently, various structural partitioning approaches have emerged. Falcon-AO [39], a famous tool for ontology matching proposes an effective divide-and-conquer approach called PBM [39]. Similarly a segmentation approach called Anchor-Flood is proposed by [55]. However, [39] and [55] does not benefit from the exploitation of newer hardware for ontology matching. For better performance during ontology matching, implementation of parallelism over multicore platform like cloud has been missed.

From the perspective of cloud and parallelism, a suggested solution can be the use of cloud-based data-intensive parallel platform called Hadoop [2]. Hadoop with its MapReduce [27] programming model, queries efficiently on distributed data. In case of large-scale ontology matching, it may be considered a candidate technology over cloud platform; however, the essence of performance efficiency in hadoop is coupled with the amount of available data typically in gigabytes and terabytes [11]. The ideal size of a single chunk of data in Hadoop is 64 MB, which is relatively equivalent to a single OWL file making an ontology too small to be distributed over hadoop file system (HDFS) [11], inflicting performance degradation instead. Moreover, Hadoop with MapReduce is still to be equipped with an efficient RDF and OWL plugin. Projects like Reasoning-Hadoop [17], Heart [12], and Hadooprdf [10] have yet to prove their usability and performance.

Although there are many ontology matching systems available; however, we are focused toward the ones explicitly built with performance as primary consideration without trading off precision. From the results of 2012s intermediary Ontology Alignment Evaluation Initiative (OAEI) campaign, i.e., OAEI 2011.5 [53], LogMap, GOMMA, and AROMA recorded the fastest time matching over multicore systems with adequate precision. LogMap, GOMMA, and AROMA have been able to scale from 1-core to 4-cores by reduction of 55 %, 53 % to 61 %, and 67 % respectively. AgrMaker, however, with highest precision and reduction of 55 %, lags over performance. Systems including CODI [5], MAPSSS [25], Lily [61], and CSA [59] with very high reduction prove to be poor in scalability and underutilizing the computational resources. MassMtch [54] provides best scalability with reduction of 37 %, produces fairly low precision. SPHeRe on the other hand equipped with its precision matching algorithm and explicit concurrency control for data parallelism, provides reduction of 32 % in scalability. This reduction rate proves that SPHeRe utilizes computational resources most efficiently.

LogMap, as described in [41] and [21], claims as highly scalable ontology matching system, yet it does not implement any parallelism. Their associated research group has proposed a concurrent classification approach for reasoning over ontologies in [42]; however, its utilization in LogMap is not clear. In a MapReduce like implementation, a stream of unclassified axioms is provided and classified as processed or unprocessed. The implemented loop that can be concurrently executed, maintains a container like data structure to store the unprocessed axioms; consequently, increasing the size of the container overtime that may contribute to unnecessary memory stress. This classification of axioms is further utilized by the matching techniques described in [41]. This classification is beneficial with respect to reasoning and inference, but from pure ontology matching perspective, this process is a performance overhead.

AROMA, as described in [26], is a simple and adaptable ontology matching tool which utilizes Knowledge Discovery in Databases (KDD) [31] model. AROMA itself does not implement any concurrency control; however, KDD discusses parallel clustering technique for incremental discovery of rules and structures in [32]. A presumption can be driven that such an implementation might be utilized by AROMA, which in fact contributes to its better reduction score, though [26] fails to mention any parallelism or concurrency involved for the benefit of AROMA's performance.

GOMMA, as described in [45], is one of the most performance efficient ontology matching tools. After a detailed research, it is safe to say that to the best of our knowledge, GOMMA is the only other system that utilizes multicore architecture for ontology matching. GOMMA's parallel matching and data partitioning techniques are mentioned in [35] and [20], respectively. GOMMA proposes inter and intramatcher parallelism, which utilizes parallel and distributed infrastructure to achieve better performance for ontology matching. As mentioned in [35], limitations of intermatcher include; varying complexity of matchers that might lead to slower performance than expected and higher memory requirements as all matchers evaluate complete ontologies. For intramatcher parallelism, internal decomposition of matcher parts is executed after loading and prior to matching. This partitioning is never persisted, leading to redundant partitioning operations for every matching request of a particular ontology. Ontology model utilized for this implementation is briefly explained in [35]; however, it fails to mention the model's thread-safety and mechanism of population from an ontology file.

SPHeRe in contrast with above mentioned systems, is an end-to-end parallel system. It avoids unnecessary memory strains by creating lightweight matcher-based and

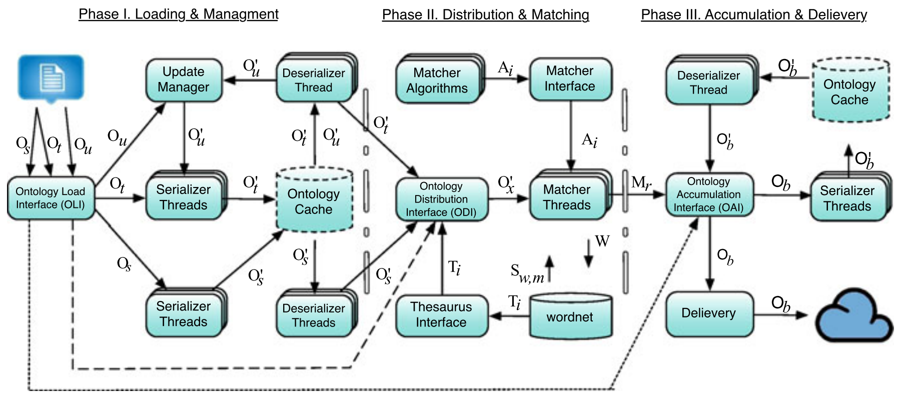

Phase I. Loading & Managment
Phase II. Distribution & Matching
Phase III. Accumulation & Delievery
Update Manager
Deserializer Thread
Matcher Algorithms
Matcher Interface
Ontology Cache
Ontology Load Interface (OLI)
Serializer Threads
Ontology Distribution Interface (ODI)
Matcher Threads
Ontology Accumulation Interface (OAI)
Deserializer Threads
Thesaurus Interface
wordnet
Delievery
Regular System Flow
Conditional System Flow I, if O's and O't exists in Ontology Cache
Conditional System Flow II, if O'b exists in Ontology Cache

Fig. 1 SPHeRe's proposed methodology

redundancy-free subsets of candidate ontologies. It provides faster loading by caching serialized subsets of candidate ontologies. It utilizes data parallelism based distribution for matching over available computational resources providing better scalability. All these attributes collectively contribute to overall system performance.

3 Proposed Methodology of SPHeRe

SPHeRe is an implementation of computation intensive data parallelism, i.e., each processing core performs ontology matching on a different piece of candidate ontologies. Matching algorithm executes across multiple cores and processors on a parallel computing enabled cloud platform. For performance improvement, SPHeRe implements parallelism at operational level, i.e., ontologies are not only matched in parallel, they are loaded, parsed, cached, and delivered in parallel too. Figure 1 illustrates the multiphase design of SPHeRe's execution flow and Table 1 describes the notations used. From left to right, it can be seen that system's execution has been divided into 3 phases: (i) loading and management; (ii) distribution and matching; and (iii) accumulation and delivery. All phases are equipped with components to perform parallel operations according to the requirements of the running tasks. This design eases the development and addition of newer components by following the standard input output interfaces among all the phases. Functionalities of each phase are described in following subsections.

3.1 Phase-I, Loading and Management

Source (O_s) and target (O_t) ontologies are provided to the system either by using SPHeRe's web-based UI or SPHeRe's ontology matching web-service. These ontologies are loaded in parallel by multithreaded ontology load interface (OLI). OLI

Table 1 Terminologies and Notations used in the paper

Notation	Description
O_s	Source Ontology provided by the user for matching
O_t	Target Ontology provided by the user for matching
$O_x \| x \in \{s, t\}$	Candidate Ontologies, set of ontologies to be matched
$O'_{xC} \| x \in \{s, t\}$, $O'_{xC} \subseteq O_x$	Serialized subset of candidate ontologies, collection of concept names
$O'_{xH} \| x \in \{s, t\}$, $O'_{xH} \subseteq O_x$	Serialized subset of candidate ontologies, Hierarchical data structure of concepts describing relationships
$O'_{xL} \| x \in \{s, t\}$, $O'_{xL} \subseteq O_x$	Serialized subset of candidate ontologies, collection of concepts with labels
$O'_{xP} \| x \in \{s, t\}$, $O'_{xP} \subseteq O_x$	Serialized subset of candidate ontologies, collection of concepts with properties
$O'_s = O'_{sC} \cup O'_{sH} \cup O'_{sL} \cup O'_{sP} \cdots \cup O'_{sn}$	Parsed source ontology, serialized to be persisted in ontology cache
$O'_t = O'_{tC} \cup O'_{tH} \cup O'_{tL} \cup O'_{sP} \cdots \cup O'_{sn}$	Parsed target ontology, serialized to be persisted in ontology cache
O_i	Updated ontology instance to be serialized and persisted in ontology cache
T_i	Thesaurus instance to be used for matching
A_i	Matching algorithm instance
$S_{w,m}$	Word-match set
M_r	Matched results, mappings between candidate ontologies
O_b	Bridge Ontology
O'_b	Aggregated Bridge Ontology from various matching algorithms and computing nodes
m_c	Control Message sent to participating nodes

is responsible for parallel loading of candidate ontologies (O_x), OWL file parsing to create object model (ontology model), and ontology model serialization and deserialization tasks. Prior to any parallel matching, necessity is a performance friendly and thread-safe ontology representation. Without such representation, data parallelism over multithreaded execution in Phase II cannot be achieved. OLI's parser owns this responsibility by generating a thread-safe performance friendly ontology model object. This ontology model facilitates our system with the following benefits. Firstly, an ontology representation with the knowledge of total associated information an OWL file is encapsulating (classes, class relationships, properties, axioms, and annotations). Distribution and matching phase effectively uses this knowledge while task distribution. Secondly, an ontology representation with thread-safety by providing immutable objects in a multithreaded environment. Thirdly, division of ontology into multiple subsets according to the needs of matching algorithms, preventing the system from loading information not required for matching. This technique reduces the memory strain, avoiding the possibility of JVM heap crashes during execution. Fourthly, no file IO during matching phase, avoiding the huge performance bottleneck of accessing OWL files numerously during matching process. Lastly, reducing the size of ontology by removing redundant information and unnecessary data, creating smaller memory footprint. For example, the namespace URI is redundant over an OWL file, keeping a single attribute in ontology model that stores the namespace URI reduces the actual size of concept names, which are essentially used entities while matching in Phase II.

After parsing the candidate ontologies, OLI invokes parallel threads of custom serializer (described in [24]) for ontology model caching. A single ontology is serialized into subsets based on matching algorithms. In current implementation, four subsets of O_x are created: (i) collection of class names for name-based ontology matching (O'_{xC}); (ii) hierarchical data-structure for relationship-based ontology matching (O'_{xH}); (iii) collection of classes with their corresponding labels for label-based ontology matching (O'_{xL}); and (iv) collection of classes with their properties for property-based ontology matching (O'_{xP}). All these serialized subsets are individually persisted in ontology cache. OLI's parser and serializer only get executed when a new ontology is provided by matching request, i.e., in case of O'_s and O'_t already been available in ontology cache, Conditional System Flow I (illustrated in Fig. 1) gets executed. Parsing and serialization steps are skipped until a massively updated version of a previously serialized ontology is received. Ontology cache provides persistence to serialized ontologies and their corresponding MD5 [52] hash values. OLI verifies whether the ontologies have already been serialized by calculating the hash values of candidate ontologies. In case of smaller partial updates, update manager renews ontology contents of serialized ontologies and persists update ontology instance O_i in ontology cache. An O_i can be obtained following the strategies mentioned in [43] and [44]. Multiple serialized versions of a particular O_s and O_t are maintained by ontology cache. In case of multiple SPHeRe nodes, ontology cache is replicated over every node, keeping all nodes synchronized. O'_s and O'_t are loaded in parallel by deserializer threads, providing a single-step ontology loading and feeding to distribution and matching phase.

3.2 Phase-II: Distribution and Matching

Serialized subsets of source (O'_s) and target (O'_t) ontologies are loaded in parallel by multithreaded ontology distribution interface (ODI). ODI is responsible for task distribution of ontology matching over parallel threads (Matcher Threads). ODI currently implements size-based distribution scheme to assign partitions of candidate ontologies to be matched by matcher threads. These threads can be running over multicore or multiprocessor architecture on single and multiple computing nodes. In a single node, matcher threads correspond to the number of available cores for the running instance. In multinodes, each node performs is its own parallel loading and internode control messages (m_c) are used to communicate regarding the ontology distribution and matching algorithms. A set of matcher threads is assigned by ODI to execute matching algorithm instance (A_i) over individual ontology partitions. ODI also facilitates matcher threads with a thesaurus interface to assign thesaurus instance (T_i). Multiple dictionaries and thesaurus can be plugged into ODI via thesaurus interface. SPHeRe is currently using word-net dictionary [60] and also provides an open-end web-service interface to plugin web-based and remote dictionaries. Remote resources; however, can result in bandwidth issues and inflict performance degradation for matcher threads as multiple http requests will be generated for an individual case of possible concept matching. ODI also facilitates matching threads with a matching algorithm interface. Multiple algorithms can be plugged into ODI via matcher interface. Matcher threads are customized to use single and multiple matching algorithms concurrently on O_s' and O_t' subsets. Matched results (M_r) provided by matcher threads are submitted to accumulation and delivery phase for creation and delivery of bridge ontology (O_b).

3.3 Phase-III: Accumulation and Delivery

Ontology Aggregation Interface (OAI) accumulates M_r provided by matcher threads. OAI is responsible for O_b creation by combining M_r as mappings and delivering O_b via cloud storage platform. OAI provides a thread-safe mechanism for all matcher threads to submit their matched results. After the completion of all matched threads, OAI invokes O_b creation process which accumulates all the matched results in a single O_b instance. In case of multinode distribution, OAI also accumulates results from remote nodes after completion of their local matcher threads. O_b creation is customizable from single O_b per matching algorithm to an aggregated bridge ontology O_b' from all matching algorithms running locally and remotely. In case of aggregated O_b', OAI is also responsible for redundancy resolution among mappings drawn by multiple matching algorithms. After the creation of O_b, it is delivered by utilizing Storage-as-a-Service based cloud service platform. An active link (URL) is provided for O_b to be shared or downloaded by the user.

OAI also persists O_b in ontology cache for future matching requests. For unchanged O_s and O_t matching requests, Conditional System Flow II (illustrated in Fig. 1) gets executed. In this execution flow, loading and management, and distribution and matching phases are skipped and URL to O_b is provided to the user. This mechanism prevents the system from performing redundant operations and preserves memory usage and CPU cycles.

4 SPHeRe's Implementation

This section provides the implementation details of SPHeRe. Its components implement several object-oriented design patterns [33] for high cohesiveness and minimal coupling. Utilization of design patterns also contributes largely towards SPHeRe's extensibility and iterative development. To explain the working details, UML Class Diagrams [46] and algorithms are used.

4.1 SPHeRe's Ontology Model

SPHeRe's Ontology Model is an object-oriented representation of an ontology file (OWL). Ontology model's design has been kept generic yet concise to support the requirements of the system. It is reused by all phases of SPHeRe as it encapsulates the OWL file by providing higher-level abstraction to ontology resources, annotations, and axioms for matching algorithms. For correctness, expert evaluation of this ontology model has been done at design, implementation, and testing stages. Ontology

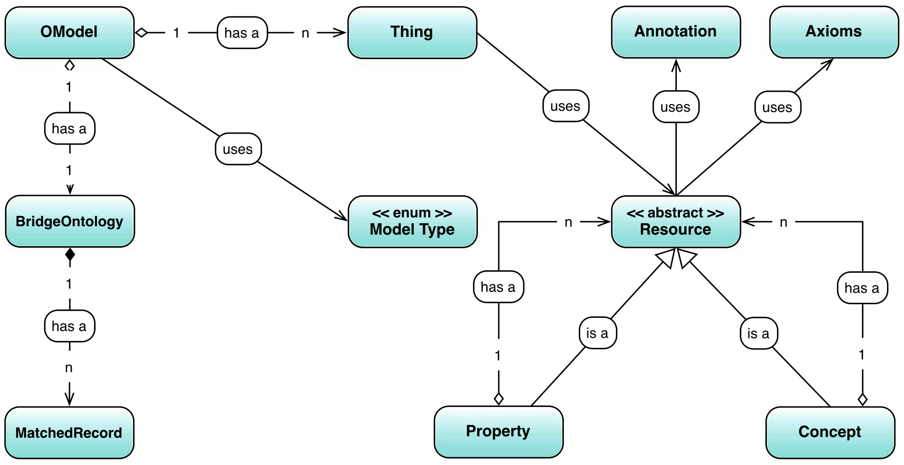


Fig. 2 Ontology Model class diagram

model represents the object model for serialized subsets of candidate ontologies. Furthermore, finalized bridge ontology is also serialized as ontology model by ontology cache. Ontology model's class diagram is illustrated in Fig. 2.

Ontologies are structures with an abstract root concept called Thing. All the resources exist under the umbrella of Thing, which is defined, but not a usable concept. The proposed ontology model follows the similar representation and provides *Thing* object that aggregates triples of individual ontologies. *ModelType* enumeration classifies a *Thing* object as collection of concepts, collection of concepts with annotations, collection of concepts with properties, and a hierarchical data structure for triples. For a single ontology file, subsets of ontology models exist simultaneously. This technique stores redundant concepts in all subsets; however, a set of matching threads will load the subset required by its matching algorithm. For distribution and matching phase, ontology model provides accessor methods to all resources via read-only interface preventing mutability for thread-safety, avoiding the possibility of inconsistency in a multithreaded execution environment.

Resource, *Property*, and *Concept* implements Composite design pattern [33]. This implementation facilitates *Concept* to mimic triples and *Property* to aggregate objects of its own type, providing a self-containing-object data structure. *Resource* abstract class is extended by *Concept* and *Property* object, which aggregates itself as subconcepts and subproperties, respectively. *Concept* and *Property* object also provides iterators to their respective associated instances, i.e., providing *Concept* with a concept name will return the required *Concept* object, its subconcepts, annotations, axioms, and associated properties. *Annotation* object encapsulates associated labels and comments to *Resource*. *Axiom* object encapsulates associated constraints to *Resource*.

MatchedRecord represents a single matched result (concept names and similarity) evaluated by matcher threads in Phase II. *BridgeOntology* contains a thread-safe collection of *MatchedRecord* objects fed by individual matcher threads. OAI in Phase III iterates over *BridgeOntology* object to create finalized O_b. *Thing* object is an

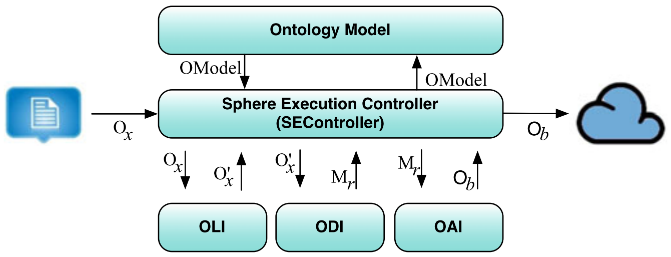


Fig. 3 SPHeRe's execution controller

aggregation to a single ontology; however, *OModel* object provides a higher level aggregation over multiple ontologies (collection of *Thing*) and their corresponding O_b.

4.2 SPHeRe's Execution Flow

To provide a flow sequence to systems execution, a workflow object called SPHeRe Execution Controller (*SEController*) is implemented and illustrated in Fig. 3. *SEController* is a shared object binding the communication between all three phases (Loading and Management, Distribution, and Matching, and Accumulation and Delivery) of the system. *SEController* is an implementation of Singleton design pattern [33]. *SEController* is responsible for invoking OWL loading via OLI in Phase I, distribution via ODI in Phase II, and accumulation and delivery of O_b via OAI in Phase III. *SEController* also implements a socket server to send m_c and receive M_r from remote SPHeRe instances.

4.3 Loading and Management component

Class diagram for loading and management component (LMC) is described in Fig. 4. Classes are packaged into two categories; (i) ontology loading and (ii) ontology management. Ontology loading package is responsible for parsing new ontologies into ontology models, serializing, and deserializing them for distribution and matching in Phase II.

SEController routes candidate ontologies to LMC via *OLI*. *Utility* class is used by *OLI* to calculate hash values for candidate ontologies and match the values against persisted ontologies hash values in ontology cache. If serialized versions of any or both candidate ontologies are already present in ontology cache, *OLI* invokes *Deserializer* object for loading O'_s and O'_t in parallel for distribution and matching phase. Algorithm 1 for the owlLoad method describes this process.

In case of absence of serialized versions for candidate ontologies, *Parser* object is invoked. *Parser* object has a composition relationship with *ParserThread* and *SerializerThread* objects, i.e., parser and serializer threads are created with the creation of *Parser* object. Number of threads to be created for parsing and serialization can be customized by request, by default it is the number of cores on an available processor; however, a single thread parses a single subset of O_x. *Parser* object loads the ontology in memory and assign parsing algorithms to parser threads via *Parsable*

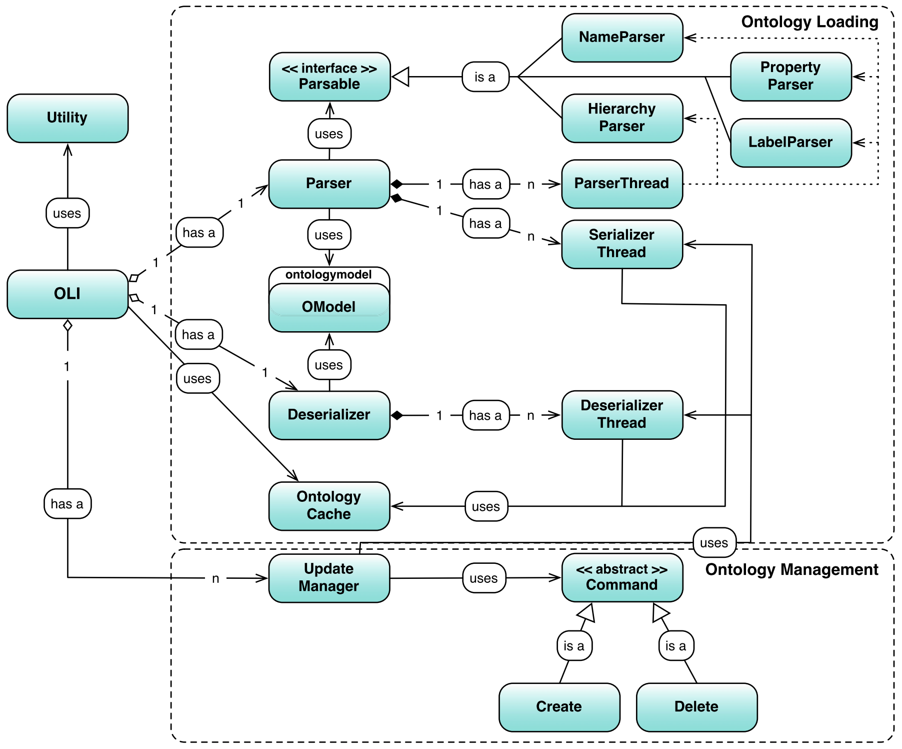


Fig. 4 Class diagram for loading and management component

```
Algorithm 1 Method owlLoad
Require: $O_s \neq NULL$ and $O_t \neq NULL$
  $Hash_s \leftarrow$ Utility.calculateHash($O_s$)
  $Hash_t \leftarrow$ Utility.calculateHash($O_t$)
  $ontologyCache \leftarrow$ OntologyCache.getInstance()
  $parser \leftarrow$ Parser.createInstance()
  if $Hash_s, Hash_t$ !$in$ $ontologyCache$ then
    parser.parse($O_s$, $O_t$)
    parser.serialize($O_s$, $O_t$)
  else
    if $Hash_t$ !$in$ $ontologyCache$ and $Hash_s$ $in$ $ontologyCache$ then
      parser.parse($O_t$)
      parser.serialize($O_t$)
    else if $Hash_s$ !$in$ $ontologyCache$ and $Hash_t$ $in$ $ontologyCache$ then
      parser.parse($O_s$)
      parser.serialize($O_s$)
    end if
  end if
  deserialize($Hash_s$, $Hash_t$)
  return
```

Algorithm 2 Method nameParser

```
Require: $O_x \neq NULL, x \in \{s, t\}$
  $thing$ ← Thing.createInstance(url)
  while $O_x$ has classes do
    $concept$ ← OClass.createInstance($currentClassName$)
    thing.addConcept(&$concept$)
  end while
  return thing
```

Algorithm 3 Method labelParser

```
Require: $O_x \neq NULL, x \in \{s, t\}$
  $thing$ ← Thing.createInstance(url)
  while $O_x$ has classes do
    $concept$ ← OClass.createInstance($currentClassName$)
    while $currentClass$ has labels do
      $label$ ← Annotation.createLabel($labelName$)
      concept.addAnnotation(&$label$)
    end while
    thing.addConcept(&$concept$)
  end while
  return thing
```

Interface. *Parsable* is an implementation of Strategy design pattern [33]. Parsing algorithms can be plugged according to the needs, adding to the agility, extensibility, and customization of the system over time.

Four algorithms currently implement *Parsable* interface: (i) *Name* parser; (ii) *Hierarchy* parser; (iii) *Label* parser; and (iv) *Property* parser. *Name*, *Label*, and *Property* parser provide implementation by reading the class-names (described in Algorithm 2), their labels (described in Algorithm 3) and properties (described in Algorithm 4), respectively. However, the hierarchy parser implements multistep bottom-up ontology parsing approach. All classes from the ontology are read with reference to their parent classes. A class may not have a child; however, every class has a parent. Parent class references are maintained by every class in the hierarchy. Algorithm 5 describes the implementation of hierarchy parser.

After parsing of candidate ontologies in ontology models, parser object forks serializer threads to persist in parallel the serialized versions of ontology models in ontology cache via *OntologyCache* object. *OLI* requests *Deserializer* object to invoke deserializer threads for loading the serialized subsets of O_s and O_t from ontology cache. *SEController* receives O_s and O_t from *OLI* and passes them to ODI for distribution and matching phase.

Ontology management package is responsible for partial updates in ontology model over time. Instead of serializing the whole ontology for every slightest update, *UpdateManager* receives the updated instances on candidate ontologies from *OLI* and implements the change by: firstly, deserializing the candidate ontology; secondly,

Algorithm 4 Method propertyParser

```
Require: $O_x \neq NULL, x \in \{s, t\}$
  $thing$ ← Thing.createInstance(url)
  while $O_x$ has classes do
    $concept$ ← OClass.createInstance($currentClassName$)
    while $currentClass$ has properties do
      $property$ ← OProperty.createInstance($propertyName$)
      concept.addProperty(&$property$)
    end while
    thing.addConcept(&$concept$)
  end while
  return thing
```

Algorithm 5 Method hierarchyParser

```
Require: $O_x \neq NULL, x \in \{s, t\}$
  $thing$ ← Thing.createInstance(url)
  while $O_x$ has classes do
    $concept$ ← OClass.createInstance($currentClassName$)
    while $currentClass$ has parents do
      if !$thing.exists(parent)$ then
        $parent$ ← OClass.createInstance($parentName$)
        thing.addConcept(&$parent$)
      else
        $parent$ ← thing.getConcept($parentName$)
      end if
      concept.addConcept(&$parent$)
    end while
    thing.addConcept(&$concept$)
  end while
  return thing
```

implementing the changed instance. A change can be an addition of new triple(s), update in original triple(s), and deletion of triple(s) via *Create* and *Delete* objects; and thirdly, serializing the updated ontology and placing it back into ontology cache. Ontology management package implements the Command design pattern [33], which facilitates the update manager to undo and redo ontology changes.

4.4 Distribution and Matching Component

Distribution and matching (DMC) component is responsible for distributing O'_x over available computational resources and invoking matching algorithms on them. The class diagram of DMC is described in Fig. 5. O'_x are fed to *ODI* that invokes a distribution strategy from *Distributable* interface. *Distributor* object implements *Distributable* interface and encapsulates distribution algorithm. *Distributor* object

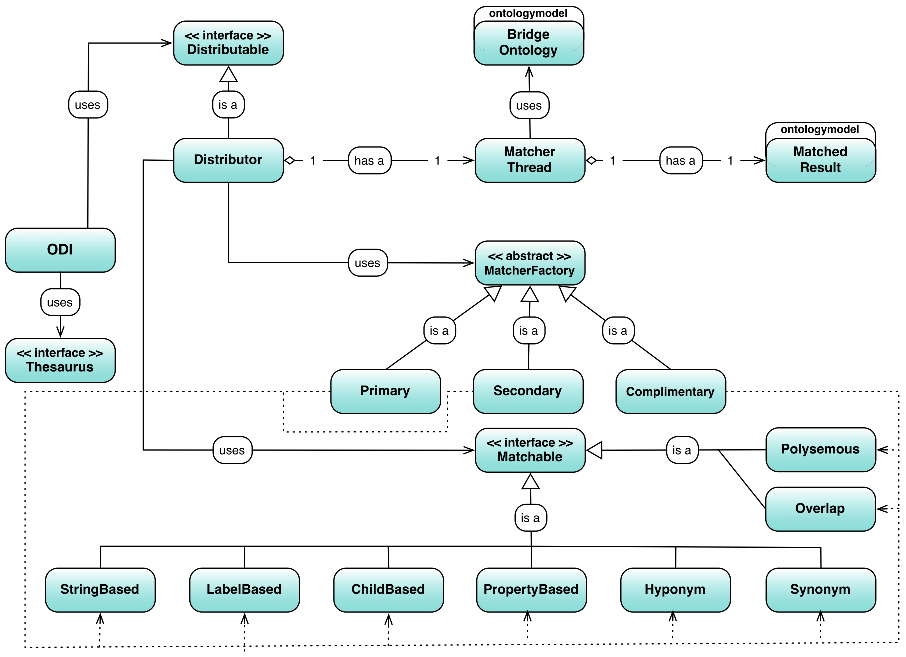


Fig. 5 Class diagram for distribution and matching component

currently implements static distributor based on ontology size (described in Algorithm 6). Size-based partitioning is an optimal mechanism for distributing matching tasks among available computational resources, as every resource gets an equal share of workload. *Distributor* object and *Distributable* interface is an implementation of Strategy design pattern, which increases the flexibility and extensibility of the system by providing interface to add more distributor objects over time without propagating changes.

Distributor object has two responsibilities: (i) Create matcher threads. *Distributor* evaluates the current runtime for the availability of computation resources and creates multithreaded environment of matcher threads to execute. Each *MatcherThread* records a *MatchedResult* object of ontology model in a thread-safe collection of matched records in *BridgeOntology* object. (ii) Assign matching algorithms to matcher threads. *Distributor* implements two design patterns, i.e., Abstract Factory [33] and strategy to assign instances of matcher algorithms to individual matcher threads.

Matching algorithms are implemented with two levels of abstraction, a fine grain implementation of matching algorithms and a coarse grain aggregation of family of matching algorithms based on utilization. Individual matching algorithms implement *Matchable* interface. Strategy pattern is again utilized here for systems flexibility and extensibility. Plug-n-play nature of strategy pattern facilities the system to add more algorithms for matching to improve accuracy. Coarse grain aggregation of individual algorithms is implemented via *MatcherFactory* (Abstract Factory). Individual algorithms are classified into three categories; *Primary*, *Secondary*, and

Algorithm 6 Method distributor

```
Require: $start = 0$, $thingOne \neq 0$, $thingTwo \neq 0$
  $cpuCount \leftarrow$ Runtime.getNumberOfProcessorse()
  $executorService \leftarrow$ Executors.newFixedThreadPool(cpuCount)
  $sizeOfOne \leftarrow$ thingOne.getSize()
  $sizeOfTwo \leftarrow$ thingTwo.getSize()
  if $sizeOfOne > sizeOfTwo$ then
    $big \leftarrow$ &thingOne
    $partitionSlab \leftarrow$ sizeOfOne/cpuCount
    $small \leftarrow$ &thingTwo
  else
    $big \leftarrow$ &thingTwo
    $partitionSlab \leftarrow$ sizeOfTwo/cpuCount
    $small \leftarrow$ &thingOne
  end if
  for $i = 0$ to $cpuCount$ do
    SPAWN THREAD:
    executorService.submit($start$, $partitionSlab$, $big$, $small$)
    $start \leftarrow$ start + slab
  end for
```

Complimentary algorithms. *Primary* family instantiates a set of algorithms that are executed for every matching request that currently are; *StringBased*, *LabelBased*, and *ChildBased*. *Secondary* family instantiates a set of algorithms that are executed by request to dig deeper into ontologies for higher accuracy; this set currently includes the *PropertyBased*, *Synonym*, and *Hyponym* algorithms. *Complementary* family instantiates a set of algorithms that are executed by request in context with domain, for example, two ontologies from medical domain have higher chances of matchable concepts, so by executing complimentary set of algorithms might contribute in the accuracy. These algorithms currently are *Overlap* and *Polysemous*. The research findings for the accuracy of the different categories of matching algorithms are currently under review in another paper.

4.5 Accumulation and Delivery component

Accumulation and delivery component (ADC) is responsible for accumulating the matched results, creating a corresponding O_b, and delivering it via cloud storage services. Its class diagram is described in Fig. 6. *OAI* invokes *Accumulator* object to create a corresponding O_b for candidate ontologies by reading matched records from *BridgeOntology* object via *OModel*. *Accumulator* object writes O_b as a mapping file and spawns a *SerializerThread* to store its serialized version in ontology cache for future matching requests of same candidate ontologies. In parallel, the mapping file object is returned to *OAI* for delivery via cloud storage services. SHPeRe is currently using a public cloud storage provider Dropbox for O_b delivery. *DropboxInterface* implements dropbox's REST API for ontology delivery. A URL to the mapping file of O_b is shared with the client via browser and email. In case of O_b already present

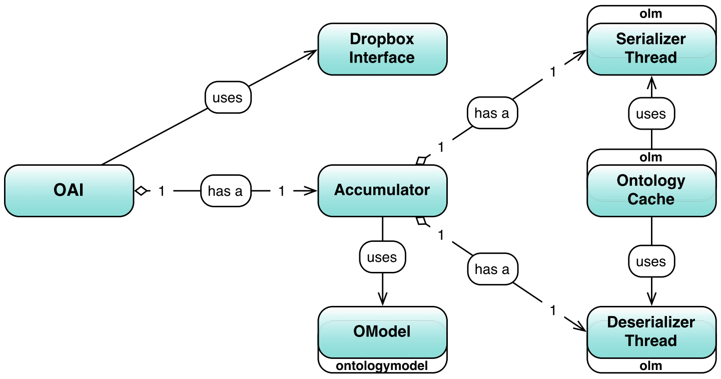


Fig. 6 Class diagram for accumulation and delivery component

for candidate ontologies, Phase II and III are skipped and corresponding mapping file is generated by *Accumulator* via *DeserializerThread* and is delivered via the same process. Similar to candidate ontologies, bridge ontologies can also be updated by *UpdateManager* in load and management component.

4.6 SPHeRe's Deployment

SPHeRe can be deployed in single-node and multinode configurations. In single-node configuration, deployment of SPHeRe is fairly straightforward, i.e., components, UI, and web service as a single application instance. An ideal deployment of multinode configuration is a cloud-based implementation that fully avails the distributed, yet collaborative nature of cloud platform. SPHeRe is a SaaS-based service deployment that currently uses the hybrid cloud [23] deployment model for ontology matching and bridge ontology delivery; however, SPHeRe's implementation is not bound to hybrid cloud model only, any cloud-oriented distributed platform will be adequate for SPHeRe's deployment and execution.

Our current deployment model is illustrated in Fig. 7. SPHeRe is deployed in multinode configuration on virtual instances (VMs) over a trinode private cloud equipped with commodity hardware. Each node is equipped with Intel(R) Core(TM) i7 CPU, 16 GB memory with Xen Hypervisor. For ease of description, in the rest of the paper, term node is used to describe a running VM over a physical cloud node. All nodes are equipped with complete deployment of SPHeRe components except the user interface and web service, which are hosted over an Apache server on primary node. Ontology cache is also replicated over each node via cloud-based file-system synchronization services. For multinode configuration an initialization process is executed, which shares the connectivity information among the participating nodes. Each running instance of SPHeRe creates a socket table with a Universal(ly) Unique Identifier (UUID) and socket object of every participating node. These objects can be used to send m_c to the peers regarding ontology distribution and assignment of matching algorithms.

SPHeRe's agility facilitates the system to process under dual execution models; (i) Stand-alone model in which matching requests are received by primary node, seri-

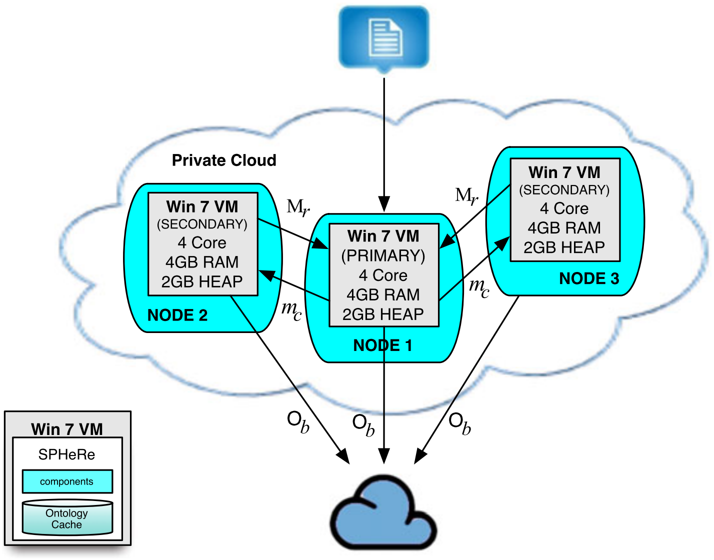


Fig. 7 SPHeRe's deployment setup

alized in parallel if required and synchronized with all available instances of ontology cache. Sets of matcher threads are assigned with matching algorithm in each participating node, with every node responsible for one or more bridge ontologies. Every O_b corresponds to an algorithm executing over a particular node by multiple threads. Each node is responsible for delivering their O_b to the client via dropbox. (ii) Contribution model, similar to (i), except all participating nodes deliver their corresponding bridge ontologies to primary node. Primary node accumulates these ontologies to generate an aggregated O'_b for delivery via dropbox.

5 Experiment

In this section, we describe the experimentation performed for end-to-end evaluation of SPHeRe by matching FMA with NCI ontologies from OAEI 2012 as a concrete example. We have used the cloud infrastructure illustrated in Fig. 7. Results drawn from this experimentation are described in Table 2.

To evaluate SPHeRe, we have performed experimentation on small, extended and full ontologies of FMA and NCI as real-world test cases. As SPHeRe implementation is performance focused by exploiting parallelism for its benefit, every component during the execution flow contributes to performance efficiency by large. Table 2 provides results of individual components and overall system performance. For precision, end-to-end evaluation process has been repeatedly executed for 20, 10, and 5 times over small, extended, and whole versions of FMA and NCI ontologies respectively.

For an end-to-end execution, SPHeRe has to perform four computationally expensive operations. First of these operations is to load the candidate ontology files,

Table 2 FMA with NCI evaluation over SPHeRe

	Parameter	Small	Ext.	Whole
Component Performance	Parsing Time	10.73 s	8.40 m	51.4 m
	Loading Time	0.49 s	2.10 s	4.98 s
	Matching Time	19.24 s	16.46 m	228 m
	Accumulation Time	0.06 s	0.11 s	1.05 s
Overall Execution Time	Regular System Flow	30.47 s	24.23 m	4.65 hr
	Conditional System Flow I	19.73 s	16.83 m	3.80 hr
F-Measure	Refined UMLS	0.857	0.739	0.710
	Original UMLS	0.863	0.744	0.715

parse them to generate ontology model objects, and persist these objects as serialized subsets. This operation is performed by LMC (Sect. 4.3). Maximum time taken by a candidate ontology to be parsed into ontology model and serialized to ontology cache is defined as the parsing time. In Table 2, the Parsing Time parameter represents the time, taken by SPHeRe to perform parsing task over small, extended, and full ontologies of FMA and NCI. Parsing of both of the ontologies is done in parallel by serializer threads.

Second of the above mentioned operations is loading of serialized subsets of candidate ontologies for matching algorithms. This operation is performed by DMC (Sect. 4.4). Maximum time taken by a serialized subset of candidate ontologies to be loaded in main memory is defined as the loading time. In Table 2, Loading Time parameter represents the time taken by SPHeRe to load all serialized subsets of FMA and NCI ontologies required by matching algorithms. Loading of these subsets is done in parallel by deserializer threads.

Third of the above mentioned operations is to distribute the matching tasks among multiple computational resources (cores) across the cloud platform. This operation is also performed by DMC. Parallel matcher threads, assigned with instance of matching algorithms, execute over designated partition of candidate ontologies. Maximum time taken by these threads is defined as the matching time. In Table 2, the Matching Time parameter represents the time taken by SPHeRe to distribute the subsets of FMA and NCI ontologies over multicore cloud platform where each core is exploited to perform parallel matching.

Last of the above mentioned operations is to accumulate matched results from matcher threads running across cloud platform. These results are aggregated to create the final bridge ontology. This operation is performed by ADC (Sect. 4.5). Time taken to accumulate all results and persist them as bridge ontology is defined as the accumulation time. In Table 2, Accumulation Time parameter represents the time taken by SPHeRe to accumulate matched results of FMA and NCI ontologies; consequently, creating bridge ontology and persisting it in ontology cache.

Time taken by individual operations is aggregated to measure the overall execution time. In Table 2, Overall Execution Time parameter represents the end-to-end execution time of SPHeRe for a regular system flow. In case of Conditional System Flow I (illustrated in Fig. 1), the execution time is improved as parsing operation is skipped.

Execution time is largely improved in case of Conditional System Flow II (illustrated in Fig. 1), as the computationally expensive operations of parsing and matching are skipped and URL for already processed bridge ontology is delivered in a fraction of a second.

From the perspective of accuracy, we obtained the results using large biomed track provided by OAEI 2012 to calculate F-Measure. Table 2 describes SPHeRe's F-Measure on FMA and NCI ontologies calculated on refined and original UMLS mappings. The results reflect SPHeRe's primary algorithms such as StringBased, LabelBased, and ChildBased, executed in parallel over cloud platform.

In this experimentation, we evaluated SPHeRe to resolve a real-world ontology matching problem by executing over small, extended, and whole ontologies of FMA and NCI. End-to-end evaluation has been performed over multicore cloud platform. The results provided in Table 2, describe the performance of the individual components and over-all execution time taken by SPHeRe. These results provide a concrete proof of SPHeRe, evaluating over a concrete ontology matching problem. Furthermore, SPHeRe's individual components implement parallelism to contribute toward overall performance efficiency of the system.

6 Discussion

In this section, we discuss the evaluation of SPHeRe by conferring the contributions of different aspects of SPHeRe's implementation. For improved performance, components from end-to-end execute in parallel and contribute in overall performance by large. We evaluate the performance related aspects (loading, memory strain, and matching) of SPHeRe over cloud infrastructure illustrated in Fig. 7 and discuss in contrast with representative OWL frameworks and ontology matching systems. In our experiments, we have considered three large scale ontologies, whole FMA with 78,989 concepts, whole NCI with 66,724 concepts, and small SNOMED CT [18] with 49,622 concepts; and two medium scale ontologies Adult Mouse Anatomy with 2,737 concepts and anatomy part of NCI Thesaurus with 3,289 concepts as candidates for SPHeRe's benchmarking. These are the most widely used ontologies for benchmarking and evaluating ontology matching tools. OAEI also uses these ontologies for evaluation purposes. SPHeRe is evaluated against three types of experiments: (i) ontology's load time and memory footprint; (ii) scalability; and (iii) parallel matching's performance.

6.1 Load Time and Memory Footprint Evaluation

In this subsection, we compare SPHeRe's ontology loading time and memory footprint created by ontology model, against two most widely used OWL frameworks Jena and OWL API. Whole NCI, whole FMA, and small SNOMED CT are used as candidate ontologies for this evaluation. For precision during evaluation, loading process has been repeatedly executed for 100 iterations over primary node (illustrated in Fig. 7).

As mentioned in Sect. 3.1, SPHeRe has its own implementation of ontology model, built for performance and thread-safety. *Parser* and *Deserializer* objects of loading and management component are equipped to parse OWL files and its serialized subsets to populate ontology model. Already available and most widely used Apache's Java-based framework for semantic web applications Jena is equipped with an ontology parser. Development of Jena is focused around its strongest component, i.e., Inference API. To facilitate Inference API, all other components including the parser are built for inference support. Jena provides an object model (OntModel); however, firstly, due to its parser's memory hungry implementation, overflow in JVM heap while working with larger scale ontologies occurs quite often. During our stress testing on Jena's parser by loading FMA with NCI ontology for matching, JVM heap crashes occurred even after providing a 2 GB of heap memory to the virtual machine. Secondly, Jena's OntModel and APIs are not thread-safe [6] resulting inconsistency and throughput bottleneck issues for applications utilizing multithreaded execution models. Concurrency in this respect can lead to reduction in performance instead. Thirdly, Jena is built with graph-based OntModel, resulting an overload of not required information even for trivial operations like class retrieval. The cost of information retrieval with respect to memory footprint and retrieval time in this regard is too high [15]. These bottlenecks and implausible nature of Jena's parser and OntModel makes it ill-equipped to integrate with SPHeRe's performance oriented parallel ontology matching techniques and provides a valid justification to continue research in this regard.

Another available and most widely used framework for parsing OWL is OWL API [38], which also provides an ontology model for information retrieval. Contrary to Jena, OWL API and its ontology model has a lighter memory footprint and cost of ontology loading and information retrieval is far efficient. OWL API can be an excellent candidate; however, the ontology model for OWL API is also not thread-safe [8], leading us to the options of either building a thread-safe ontology model with an associated parser for parallelism needs or updating OWL API and its ontology model for thread-safety. Although the first option enables us to have more control over the model, its accessibility, and extensibility. We have an ongoing research for the resolution of this aspect which includes evaluation of both techniques at different scales. For our current implementation, we have created our own thread-safe ontology model by using OWL API's parsing components.

SPHeRe implements parallel loading for OWL files and their serialized subsets. Figure 8 describes the loading time of serialized subsets of candidate ontologies in parallel. The longest time taken is compared to the load time taken by Jena and OWL API, described in Fig. 9. Loading time in this evaluation is the sum of time taken by a system to load ontology in the memory and retrieve all of its classes. Performance of class retrieve time is directly related to the time consumed over concept matching, providing a significant impact on overall system's performance. Because of the subsets and serialized nature of SPHeRe's ontology cache, it achieves better performance to its comparable systems. SPHeRe only parses the candidate ontologies for the first time. For every following matching request, SPHeRe uses the cached candidate ontologies avoiding the re-parsing of OWL files. Results from Fig. 9 validates that loading from cached serialized subsets take less time to load ontologies and retrieve classes. This technique ensures that following requests for matching of candidate ontologies will have better performance.

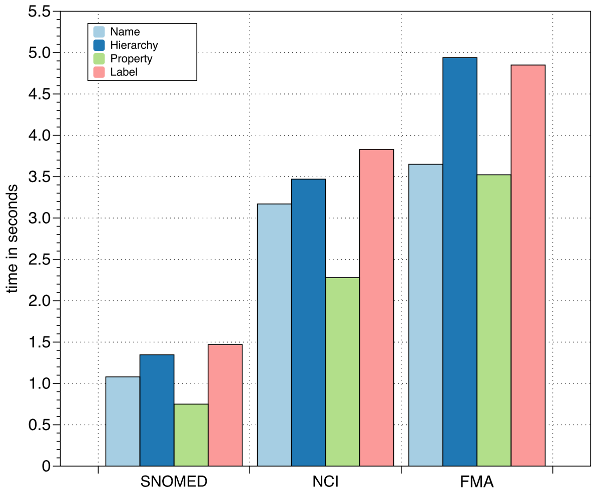


Fig. 8 Ontology load time for serialized subsets

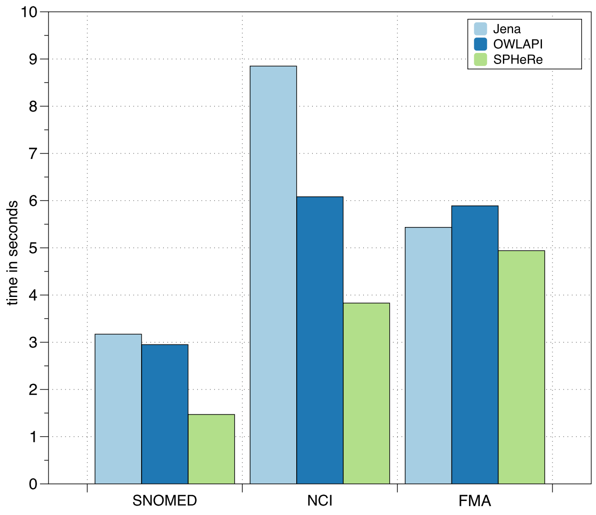


Fig. 9 Ontology load time comparison

For memory footprint experiment on same ontologies, results from SPHeRe, Jena, and OWL API are described in Fig. 10. Memory footprint has been calculated by measuring the difference between the amounts of free memory available in Java heap after the ontology load. Java's runtime library is used for this evaluation as described in [7], [13], and [14]. SPHeRe's cumulative memory footprint of all subsets is evaluated in contrast with memory footprint by Jena and OWL API. With the removal of redundancy of information and classification of ontology into subsets, SPHeRe produces up to 8 times smaller memory footprint than Jena and OWL API. This evaluation also complements to our remarks regarding the lightweight nature of SPHeRe, giving the system an edge over the existing systems to be a better option for ontology matching using commodity hardware and commodity hardware based distributed systems like cloud platforms.

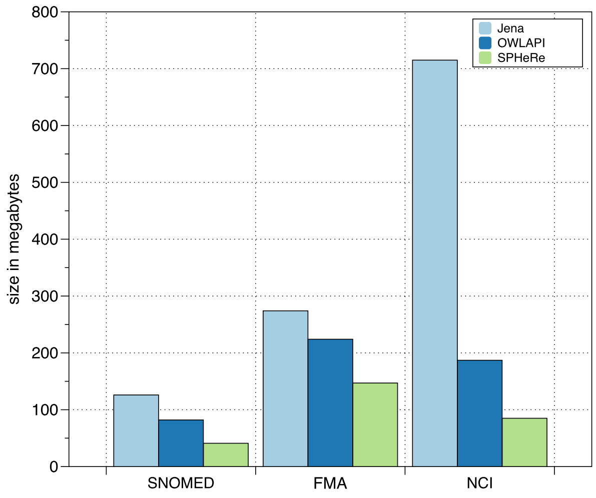


Fig. 10 Memory footprint comparison

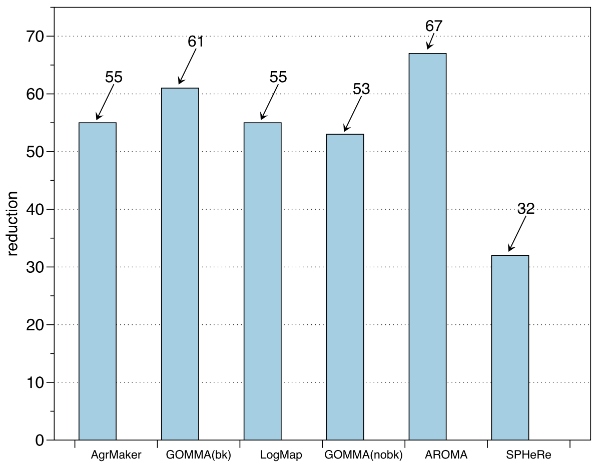


Fig. 11 Scalability comparison

6.2 Scalability Evaluation

In this subsection, we evaluate SPHeRe's scalability performance in contrast with existing scalable ontology matching systems from OAEI 2011.5 campaign. For measurement, as explained in [53], SPHeRe was executed over virtual instances with one, two, and four cores; however, each with lesser memory (4 GB) in contrast with [53]. Figure 11 indicates the reduction rate achieved by SPHeRe when executing over 4-core environment. Reduction value is computed by dividing the execution time on four cores by execution time on one core. System with the best scalability will score a value around 25 %. SPHeRe outperforms other scalable systems by scoring 32 %, which is closest to the optimal value. SPHeRe outperforms the most scalable ontology matching system GOMMA by 40 %. For precision, this evaluation has been repeatedly executed for 20 iterations.

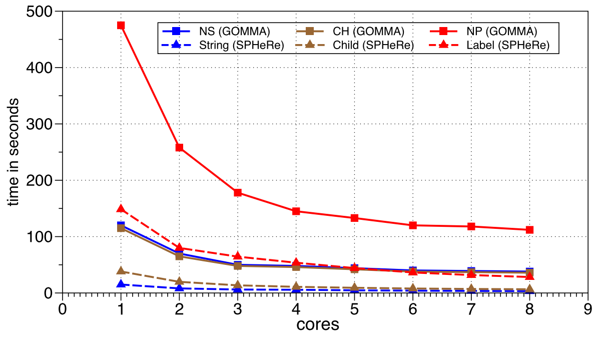


Fig. 12 Performance comparison with GOMMA

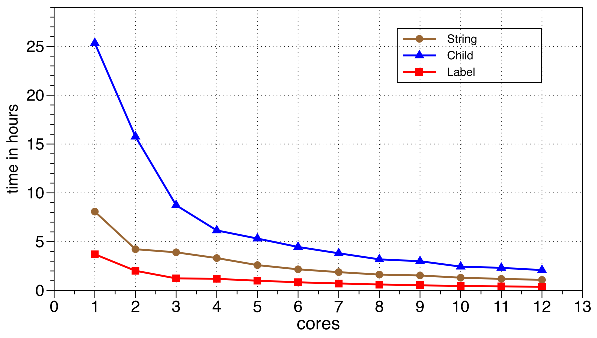
Size-based partitioning of static distributor of SPHeRe is a major contributor in achieving better scalability. This partitioning technique ensures that every matcher thread gets equal share of tasks by distribution. *Distributor* invokes matcher threads depending upon the runtime availability of the computing cores and assigns equal number of matching tasks with matching algorithm instances among all the available matcher threads. Each matcher thread is responsible for ensuring the execution of matching algorithm on its own partition, preserving the overall accuracy of the system.

6.3 Performance Comparison with GOMMA

In this subsection, we evaluate SPHeRe's performance against one of the most performance efficient ontology matching system GOMMA. Figure 12 describes SPHeRe's overall performance results in contrast with GOMMA's as presented in [35]. GOMMA's matching algorithms NameSynonym (NS), Children (CH), and NamePath (NP) are compared with similar (in complexity) matching algorithms of SPHeRe, i.e., *StringBased* algorithm that calculates similarity by measuring edit distance [50] between concept names, *ChildBased* algorithm that calculates similarity by comparing children of concepts, and a specialized *LabelBased* algorithms that calculates similarity over tokenized and normalized labels of concepts. Algorithms of SPHeRe scaling from 1 up to 8 threads (= number of cores) outperform GOMMA by 4 times. For precision, this evaluation has been repeatedly executed for 100 iterations.

SPHeRe is able to achieve this performance by creating subsets of ontologies and distributing these subsets over sets of matcher threads. This technique enables a matcher thread to avoid loading information in the memory that is out of matcher's scope by loading only the information it requires. In complement, all the subsets of ontologies are serialized and optimized to redundancy free ontology model by SPHeRe's ontology loading and management component, resulting in a far smaller memory footprint and much faster information retrieval for matching. Unlike GOMMA, these subsets are created depending upon the matcher algorithms and cached, avoiding the repartitioning of ontology for the following matching requests, providing a much efficient solution.

Fig. 13 FMA-NCI matching problem, performance evaluation



6.4 Performance evaluation with respect to FMA-NCI matching problem

Performance evaluation of FMA-NCI matching problem is described in Fig. 13. All primary algorithms (*StringBased*, *ChildBased*, and generic *LabelBased* algorithm that compares the concept labels) are executed over multinode configuration, utilizing all 12-cores provided by our trinode cloud platform. In case of sequential access, matching process takes around 25 hours to complete; however, by utilizing SPHeRe's scalability features (distributor and matcher threads), this time has been reduced to couple of hours. In an optimal distribution over our tri-node cloud platform for this problem, i.e., the distribution of 65 % of computation resources (8-cores) to *ChildBased* matching algorithm, which is by far the highest in complexity, 25 % of the computation resources (3-cores) to *StringBased* matching algorithm and 10 % of the computation resources (1-core) to a generic *LabelBased* algorithm that compares the concept labels, took less than 4 hours to complete the whole matching process. For precision, this evaluation has been repeatedly executed for 5 iterations. Same problem has been considered as one of tasks evaluated by [53]. Systems have solved this quicker than 4 hours; however, to the best of our knowledge the difference of performance is because of the underlying hardware. As mentioned in [53], experiments have utilized high-performance computational resources, equipped with 16 CPU's and 10 GB RAM. In contrast, our experimental setup is fairly primitive, evaluating over commodity hardware of 3 CPUs only.

7 Conclusion and Future Work

In this paper, we presented SPHeRe, a lightweight, scalable, and performance efficient ontology matching tool for commodity infrastructure of cloud platform. The increasing interest in heterogeneity resolution by ontology matching over web-based interactive and semireal-time systems, demands a performance-oriented lightweight ontology matching tool that can exploit the low-cost yet effective computational resources for its benefit. SPHeRe implements end-to-end parallelism across its execution flow. It provides better performance by caching redundancy-free lightweight serialized subsets of candidate ontologies. These subsets are distributed by size-based partitioning and matched by matcher threads executing in parallel over cloud-based commodity hardware, i.e., multicore processors. We have benchmarked SPHeRe against various ontology matching systems and OWL frameworks. Our results have shown that SPHeRe is able to achieve better performance to Jena and OWL API in ontology loading with smaller memory footprint. Our results have also shown that SPHeRe outperforms other ontology matching systems (AgrMaker, GOMMA, LogMap, and AROMA) in scalability. In comparison with most performance efficient ontology matching system GOMMA, SPHeRe is 40 % more scalable and provides better overall performance. SPHeRe has also been evaluated over commodity hardware to evaluate over large-scale ontologies, whole FMA with whole NCI. By evaluating in parallel, substantial performance improvement has been achieved by SPHeRe over a cloud platform.

SPHeRe is an in-progress implementation with opportunity areas still under research. We have initiatives working over the accuracy of matching algorithms, ontology repositories with synchronization services, and several data partitioning schemes for ontology distribution over parallel hardware. The outcomes of these initiatives will be covered in future research papers. Further information regarding SPHeRe and its progress are available at SPHeRe's website, i.e., http://uclab.khu.ac.kr/sphere.

Acknowledgements This research was supported by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2013-(H0301-13-2001)).

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2010-0020725) and the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2013-H0301-13-4006) supervised by the NIPA (National IT Industry Promotion Agency).

References

1. Agreement maker. http://agreementmaker.org/
2. Apache Hadoop. https://hadoop.apache.org
3. Apache Jena. http://jena.apache.org/
4. AROMA project. http://aroma.gforge.inria.fr/
5. Combinatorial optimization for data integration (CODI). http://code.google.com/p/codi-matcher/
6. Concurrent access to models. http://jena.apache.org/documentation/notes/concurrency-howto.html
7. Do you know your data size? Vladimir Roubtsov, JavaWorld. http://www.javaworld.com/javatips/jw-javatip130.html
8. Dropbox—simplify your life. http://ontolog.cim3.net/file/work/OWL2/OWL-2_Tools-n-Applications/owl-api-presentation–MatthewHorridge_20100805.pdf
9. Generic ontology matching and mapping management (GOMMA). http://dbs.uni-leipzig.de/GOMMA
10. Hadooprdf. http://74.207.237.15/product/hadooprdf
11. HDFS architecture guide. http://hadoop.apache.org/docs/hdfs/current/hdfs_design.html
12. HeartProposal. http://wiki.apache.org/incubator/HeartProposal
13. How to get max memory, free memory and total memory in Java, Javin Paul, Javarevisited. http://javarevisited.blogspot.kr/2012/01/find-max-free-total-memory-in-java.html
14. Java performance—memory and runtime analysis—tutorial, Lars Vogel, vogella.com. http://www.vogella.com/articles/JavaPerformance/article.html
15. Jena, a framework for developing semantic web applications. http://semanticwebbuzz.blogspot.kr/2009/10/jena-framework-for-developing-semantic.html
16. LogMap: logic-based methods for ontology mapping. http://www.cs.ox.ac.uk/isg/projects/LogMap/
17. Reasoning-Hadoop. http://www.jacopourbani.it/reasoning-hadoop.html
18. SNOMED clinical terms. http://bioportal.bioontology.org/ontologies/1353

19. The OWL API. http://owlapi.sourceforge.net/
20. Data partitioning for parallel entity matching. CoRR abs/1006.5309 (2010). http://dblp.uni-trier.de/db/journals/corr/corr1006.html#abs-1006-5309. Informal publication
21. Large-scale interactive ontology matching: algorithms and implementation. In: ECAI'12, pp. 444–449 (2012)
22. Amin M, Shafi A, Hussain S, Khan W, Lee S (2012) High performance Java sockets (HPJS) for scientific health clouds. In: 2012 IEEE 14th international conference on e-health networking, applications and services (Healthcom), pp 477–480. doi:10.1109/HealthCom.2012.6379466
23. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2010) A view of cloud computing. Commun ACM 53(4):50–58. doi:10.1145/1721654.1721672
24. Bloch J (2008) Effective Java, 2nd edn. Addison-Wesley, Reading
25. Cheatham M (2011) MAPSSS results for oaei 2011. In: OM, CEUR workshop proceedings, vol 814. CEUR-WS.org. http://dblp.uni-trier.de/db/conf/semweb/om2011.html#Cheatham11
26. David J, Guillet F, Briand H (2006) Matching directories and owl ontologies with aroma. In: Proceedings of the 15th ACM international conference on information and knowledge management, CIKM '06. ACM, New York, pp 830–831. doi:10.1145/1183614.1183752. http://doi.acm.org/10.1145/1183614.1183752
27. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51(1):107–113. doi:10.1145/1327452.1327492
28. Dilong Z (2001) Analysis of XML and COIN as solutions for data heterogeneity in insurance system Integration. Master's thesis, Sloan School of Management, Massachusetts Institute of Technology (MIT), Cambridge, MA
29. Doan A, Halevy A, Ives Z (2012) Principles of data integration. Addison-Wesley, Reading
30. Euzenat J, Shvaiko P (2007) Ontology matching. Springer, Berlin (DE)
31. Fayyad UM, Piatetsky-Shapiro G, Smyth P, Uthurusamy R (eds) (1996) Advances in knowledge discovery and data mining. American Association for Artificial Intelligence, Menlo Park
32. Frawley WJ, Piatetsky-Shapiro G, Matheus CJ (1992). Knowledge discovery in databases: an overview
33. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley/Longman, Boston
34. Gracia J, Mena E (2012) Semantic heterogeneity issues on the web. IEEE Internet Comput 16(5):60–67. doi:10.1109/MIC.2012.116
35. Gross A, Hartung M, Kirsten T, Rahm E (2010) On matching large life science ontologies in parallel. In: Proceedings of the 7th international conference on data integration in the life sciences, DILS'10. Springer, Berlin, pp 35–49. http://dl.acm.org/citation.cfm?id=1884477.1884483
36. Hakimpour F, Geppert A (2001) Resolving semantic heterogeneity in schema integration. In: Proceedings of the international conference on formal ontology in information systems, FOIS'01, vol 2001. ACM, New York, pp 297–308. doi:10.1145/505168.505196
37. Hayamizu T, Mangan M, Corradi J, Kadin J, Ringwald M (2005) The adult mouse anatomical dictionary: a tool for annotating and integrating data. Genome Biol 6:1–8. http://dx.doi.org/10.1186/gb-2005-6-3-r29. doi:10.1186/gb-2005-6-3-r29
38. Horridge M, Bechhofer S (2011) The owl api: a Java api for owl ontologies. Semant Web 2(1):11–21. http://dl.acm.org/citation.cfm?id=2019470.2019471
39. Hu W (2010) Falcon-AO. http://ws.nju.edu.cn/falcon-ao/
40. Institute NC (2012) National Cancer Institute thesaurus. http://ncit.nci.nih.gov/
41. Jiménez-Ruiz E, Grau BC (2011) Logmap: logic-based and scalable ontology matching. In: Proceedings of the 10th international conference on the semantic web, (ISWC'11). Springer, Berlin, pp 273–288. http://dl.acm.org/citation.cfm?id=2063016.2063035
42. Kazakov Y, Krötzsch M, Simancík F (2011) Concurrent classification of el ontologies. In: Proceedings of the 10th international conference on the semantic web, (ISWC'11). Springer, Berlin, pp 305–320. http://dl.acm.org/citation.cfm?id=2063016.2063037
43. Khattak AM, Latif K, Lee S (2013) Change management in evolving web ontologies. Knowl-Based Syst 37(0):1–18. doi:10.1016/j.knosys.2012.05.005. http://www.sciencedirect.com/science/article/pii/S0950705112001323
44. Khattak AM, Pervez Z, Latif K, Lee S (2012) Time efficient reconciliation of mappings in dynamic web ontologies. Knowl-Based Syst 35:369–374. http://dblp.uni-trier.de/db/journals/kbs/kbs35.html#KhattakPLL12

45. Kirsten T, Gross A, Hartung M, Rahm E (2011) Gomma: a component-based infrastructure for managing and analyzing life science ontologies and their evolution. J Biomed Semant 2:6
46. Larman C (2004) Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3rd edn. Prentice Hall, New York
47. LeBlanc T, Friedberg S (1985) Hpc: a model of structure and change in distributed systems. IEEE Trans Comput C-34(12):1114–1129. doi:10.1109/TC.1985.6312210
48. Microsoft (2004) Applying Microsoft Patterns to Solve EAI Problems. http://msdn.microsoft.com/en-us/library/ee265635(v=bts.10).aspx
49. Microsoft (2012) Microsoft BizTalk Server. http://www.microsoft.com/biztalk/en/us/default.aspx
50. Navarro G (2001) A guided tour to approximate string matching. ACM Comput Surv 33(1):31–88. doi:10.1145/375360.375365. http://doi.acm.org/10.1145/375360.375365
51. Rauber T, Runger G (2010) Parallel programming for mulitcore and cluster systems. Springer, Berlin
52. Rivest R (1992) The MD5 message-digest algorithm. Tech. rep. 1321, RFC editor, Fremont, CA, USA. http://www.rfc-editor.org/rfc/rfc1321.txt
53. Ruiz EJ, Grau BC, Horrocks I (2012) Evaluating ontology matching systems on large, multilingual and real-world test cases. http://www.cs.ox.ac.uk/isg/projects/SEALS/oaei/
54. Schadd FC, Roos N (2011) Maasmatch results for oaei 2011. In: OM, CEUR workshop proceedings, vol. 814. CEUR-WS.org. http://dblp.uni-trier.de/db/conf/semweb/om2011.html#SchaddR11
55. Seddiqui H, Aono M (2009) An efficient and scalable algorithm for segmented alignment of ontologies of arbitrary size. Web semantics: science, services and agents on the world wide web 7(4). http://www.websemanticsjournal.org/index.php/ps/article/view/272
56. Shvaiko P, Euzenat J (2013) Ontology matching: state of the art and future challenges. IEEE Trans Knowl Data Eng 25(1):158–176. doi:10.1109/TKDE.2011.253
57. Sioutos N, Coronado Sd, Haber MW, Hartel FW, Shaiu WL, Wright LW (2007) Nci thesaurus: a semantic model integrating cancer-related clinical and molecular information. J Biomed Inform 40(1):30–43. http://dx.doi.org/10.1016/j.jbi.2006.02.013. doi:10.1016/j.jbi.2006.02.013
58. Stoilos G, Stamou G, Kollias S (2005) A string metric for ontology alignment. In: Proceedings of the 4th international conference on the semantic web, ISWC'05. Springer, Berlin, pp 624–637. doi:10.1007/11574620_45. http://dx.doi.org/10.1007/11574620_45
59. Tran QV, Ichise R, Ho BQ (2011) Cluster-based similarity aggregation for ontology matching. In: OM, CEUR workshop proceedings, vol. 814. CEUR-WS.org. http://dblp.uni-trier.de/db/conf/semweb/om2011.html#TranIH11
60. University, P. (2013) WordNet A lexical database for English. http://wordnet.princeton.edu/
61. Wang P, Xu B (2009) Lily: Ontology alignment results for oaei 2009
62. University of Washington, S.o.M. (2007) Foundation Model of Anatomy. http://sig.biostr.washington.edu/projects/fm/