Appl Intell (2015) 43:356–385
DOI 10.1007/s10489-015-0648-z

Performance-based ontology matching

A data-parallel approach for an effectiveness-independent performance-gain in ontology matching

Muhammad Bilal Amin · Wajahat Ali Khan · Sungyoung Lee · Byeong Ho Kang

Published online: 8 March 2015
© Springer Science+Business Media New York 2015

Abstract Ontology matching is among the core techniques used for heterogeneity resolution by information and knowledge-based systems. However, due to the excess and ever-evolving nature of data, ontologies are becoming large-scale and complex; consequently, leading to performance bottlenecks during ontology matching. In this paper, we present our performance-based ontology matching system. Today's desktop and cloud platforms are equipped with parallelism-enabled multicore processors. Our system benefits from this opportunity and provides effectiveness-independent data parallel ontology matching resolution over parallelism-enabled platforms. Our system decomposes complex ontologies into smaller, simpler, and scalable subsets depending upon the needs of matching algorithms. Matching process over these subsets is divided from granular to finer-level abstraction of independent matching requests, matching jobs, and matching tasks, running in parallel over parallelism-enabled platforms. Execution of matching algorithms is aligned for the minimization of the matching space during the matching process. We comprehensively evaluated our system over OAEI's dataset of fourteen real world ontologies from diverse domains, having different sizes and complexities. We have executed twenty different matching tasks over parallelism-enabled desktop and Microsoft Azure public cloud platform. In a single-node desktop environment, our system provides an impressive performance speedup of 4.1, 5.0, and 4.9 times for medium, large, and very large-scale ontologies. In a single-node cloud environment, our system provides an impressive performance speedup of 5.9, 7.4, and 7.0 times for medium, large, and very large-scale ontologies. In a multi-node (3 nodes) environment, our system provides an impressive performance speedup of 15.16 and 21.51 times over desktop and cloud platforms respectively.

Keywords Ontology matching · Heterogeneity resolution · Multithreading · Parallel processing · Parallel programming · Semantic web

M. B. Amin · 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

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

B. H. Kang
School of Computing and Information Systems, University of Tasmania, Tasmania, Australia
e-mail: Byeong.Kang@utas.edu.au

1 Introduction

In this era of automated knowledge aggregation, integration of data and information from heterogeneous sources is the key [1]. The excess of available information over ubiquitous platforms, contributed by various domains using various input devices has substantially increased the amount of disparate information; consequently, semantic heterogeneity issues have emerged. The primary solution for semantic heterogeneity problem is ontology matching. It determines correspondence between semantically related ontologies. This correspondence is termed as mappings or alignment [2]. These mappings are further used by information systems, electronic commerce systems, knowledge-based systems, search engines and social networking systems. Due

to the greater benefits of ontology matching, ontologies are extensively utilized in multiple domains. For example, in biomedicine, ontologies are used for representing medical knowledge and clinical guidelines [3], standardization of medical data formats [4], clinical data integration and medical decision-making [5]. Consequently, biomedical ontologies like the Gene Ontology (GO) [6], the National Cancer Institute Thesaurus (NCI) [7], the Foundation Model of Anatomy (FMA) [8], and the Systemized Nomenclature of Medicine (SNOMED-CT) [9] have emerged; furthermore, infrastructures like OBO Foundry [10] and BioPortal [11] are promoting the usage of ontologies in biomedicine. Similarly, in electronic commerce, ontologies are used for mediation among two or more web services [12] and their discovery [13]. The vast usage of ontologies has compelled researchers and experts to invest more in development of newer ontologies and provide continuity to the already created ones. As a result, ontologies are becoming larger in size, complex in structure, and their matching process has become computationally expensive.

Ontology matching is a two-fold problem where challenges and issues are classified into two categories; (i) accuracy that deals with the effectiveness of the matching algorithms and (ii) performance that is based upon scalability, resource utilization, and overall execution time of the whole matching process [14]. Although the trade-off between accuracy and overall execution time exists, by implementing scalable and optimal resource utilization techniques, performance of the ontology matching process can be largely improved with effectiveness independence.

Ontology matching is a computationally intensive task with quadratic computational complexity [15]. It is a Cartesian product of two candidate ontologies, which requires Resource-based element-level (String-based, Annotation-based, and Label-based) [2] and structural-level (Child-based, Graph-based, and Property-based) [2] matching algorithms to be executed over candidate ontologies for the generation of the required mappings. In our experiments, executing these matching algorithms over various size ontologies has taken from hours till days to generate desirable results. This delay in mapping results makes ontology matching ineffective for dynamic applications with in-time processing demands.

Ontology matching systems developed over the years have taken the execution time into consideration and have implemented possible resolutions. However, the performance aspect of these systems is tightly coupled with the accuracy and complexity of matching algorithms. Their implemented resolutions are more focused on optimization of the matching algorithms and partitioning of larger ontologies into smaller chunks for performance benefits [16]. In these implementations, a clear distinction between the resolutions for accuracy and performance does not exist. Furthermore, an explicit and decoupled runtime has not been proposed yet which can improve the performance factors without inflicting any changes in the effectiveness of matching algorithms. Therefore, these resolutions fall into the category of effectiveness-dependent solutions where a trade-off between matching effectiveness (accuracy measures, precision, recall, and F-Measure) and execution time (performance) exists. Moreover, the performance improvement based-on exploitation of newer hardware technologies has largely been missed. Among these technologies are affordable parallel systems that are easily available as stand-alone and distributed platforms [14]. Current ontology matching systems are design time tools which are not optimized for resource consumption [14]. Therefore, they have not provided substantial performance-gain by just deploying over parallelism-enabled stand-alone and distributed platforms.

In earlier years, parallelism and distributed platforms were associated with High Performance Computing (HPC) [17]. To support HPC, expensive platforms have been developed over the years. These platforms are not only scarce, but also have higher costs and skill-set requirements, making them incurious for average developers and platform administrators. However, more recently, parallelism has been applicable over personal computing devices like desktop PCs, laptops, and even over smartphones because of the advent of multicore processors [18]. These processors are equipped with multiple cores on a single die, enabling each core to serve as a virtual microprocessor, providing parallelism at the hardware level. Moreover, with the arrival of Cloud computing as a backbone platform for ubiquitous computing [19], these multicore processors are always available as distributed platforms of commodity machines with utility-based pricing model. With these readily available, yet affordable parallel platforms, an opportunity emerges for their utilization in ontology matching. Furthermore, their utilization can lead to an effectiveness-independent performance-gain ontology matching solution where the accuracy of the matching algorithms remains preserved and performance-gain is extracted from smarter use of available computing resources.

The innovation of hardware architecture has brought parallel computing over personal and ubiquitous platforms; however, the utilization of these resources requires parallel programming techniques. Ontology matching being a compute intensive task can be resolved by several parallel programming paradigms including Message Passing over high-end hardware and communications devices [20], Task Parallelism, and Data Parallelism [21]. Message passing requires inter-process communication that is appropriate for iterative problems where dependency between operations exists [22]. In task parallelism, independent

threads execute different operations on the same data. However, data parallelism is one such technique where multiple autonomous operations are performed repeatedly on multiple pieces of data [23]. Looking from the perspective of ontology matching problem, data parallelism is a candidate technique in which these ontologies can be divided into smaller pieces and assigned over to computing resources for executing matching algorithms in parallel. The parallel data implementation largely improves the ontology matching performance over other parallel paradigms. By implementing data parallelism, thread-level parallelism gets implemented with a set of independent matcher threads executing the same matching algorithm on a different part of candidate ontologies. This mechanism also enables matching space reduction as every following set of matching threads will only match ontology resources left by the previous set with another algorithm. Moreover, unlike message-passing, data parallelism resolves ontology matching by using independent threads with zero inter-thread communication and network I/O during matching. In case of task parallelism for matching ontologies, matcher threads cannot be truly independent because: (i) redundant matching on same part of candidate ontologies will occur, i.e., a concept matched by one algorithm will be matched in parallel by another algorithm, unless communicated otherwise. Redundant matching not only costs extra computation time at matching but also has an aggregation overhead; (ii) higher chances of idle cores, i.e., as the computational complexity and overall time taken by an algorithm running by a thread on same part of candidate ontologies will be different from other algorithms running by other threads. Therefore, one thread will finish early and wait in idle for others to finish unless a costly load redistribution is performed at runtime. Data parallelism with its better scalability, matching space reduction, and no communication overhead is more performance efficient than other parallel paradigms for ontology matching. In addition, work distribution among a set of threads running the same algorithm is based on having an equal amount of workload per thread, which reduces the chances of idle processing cores to a bare minimum, i.e., no runtime load redistribution required.

By accumulating the opportunities mentioned earlier, i.e., delay in ontology matching, effectiveness-dependent nature of current ontology matching systems, absence of exploitation of parallelism-enabled stand-alone and ubiquitous platforms for effectiveness-independent performance-gain during matching, and likelihood of data parallelism over these platforms for ontology matching, provides the motivation for a performance-based ontology matching system. This paper contributes by presenting one such system that implements data parallelism over parallelism-enabled platforms for parallel ontology matching. Utilization of these platforms leads to an effectiveness-independent performance-gain, as our system decouples the performance aspect from accuracy and explicitly provides resolution to earlier mentioned performance challenges of ontology matching. Consequently, no change is inflicted in the implementation of matching algorithms, keeping the accuracy preserved. Moreover, with the availability of better computational resources, faster-matched results are obtained. In our proposed system, we provide a resolution to performance challenges by:

- decomposing the complex ontologies into smaller Resource-based ontology subsets depending upon the needs of matching algorithms. These subsets are independent and simpler (reduced computational complexity) with performance and scalability-friendly data structures. This method contributes to our system's performance by only loading the ontology resources required by matching algorithms and data structures that can be easily partitioned for data parallel matching. These subsets are also preserved by serialization to reduce the matching effort for future matching requests of same ontologies;
- division of the matching process over these subsets into three levels of abstractions (independent Matching Requests, Matching Jobs, and Matching Tasks) depending upon the available parallelism-enabled platform. Matching Requests are assigned to participating node(s), matching jobs are the division of one matching request over available computing cores within a node, and each core is assigned with a set of equal numbers of matching tasks to complete the whole matching process. Matching task invokes assigned matching algorithm for effectiveness-independent matching. This method contributes to our system's performance by distributing matching tasks over participating computing cores and executing them in parallel at finer level with optimal computing resource utilization;
- aligning the execution of matching algorithms to minimize the matching space for every following matching algorithm execution during the whole matching process. This method contributes to performance by reducing the number of matching tasks to unmatched resources only, thus avoiding redundant expensive matching operations.

Performance-gain of our proposed system is substantially achieved from our data-parallel methodology. Therefore, our system requires the availability our system requires of parallelism-enabled platforms. We have used quad-core desktop PCs and Microsoft Azure public cloud platform configured in single- and multi-node environments for the deployment, execution, and evaluation of our system. In case of availability of subpar computational resources,

our system scales down to conventional sequential matching. Furthermore, our proposed system is independent of ontology types and domains. Any ontology following RDF/XML[1] syntax specification is processed by our system for matching; thus, no changes in the original structure of the candidate ontologies is required for our performance-based ontology matching solution.

We have comprehensively evaluated our system with Ontology Alignment Evaluation Initiative (OAEI)'s 2013 dataset of real world ontologies (14 ontologies in total) from diverse knowledge domains, having various sizes and complexities. For ontologies from Anatomy track (adult mouse anatomy with human anatomy part of NCI Thesaurus), our system has been able to achieve an impressive performance speedup of 4 times over the desktop and 5 times over the cloud platform (single-node). For ontologies from Library track, our system has been able to achieve an impressive performance speedup of 3.9 times over the desktop and 6.3 times over the cloud platform (single-node). For all six tasks of Large Biomedical Ontologies track, our system has been able to achieve an impressive performance speedup of 4.4, 4.7, and 5.3 times over the desktop and 6.5, 7.5, and 7.25 times over the cloud platform for tasks 1, 3, and 5 respectively (single-node). For tasks 2, 4, and 6, our system has been able to achieve an impressive performance speedup of 14.65, 15.64, and 15.19 times over desktop and 21.6, 21, and 21.93 times over cloud platform respectively (multi-node). We have also evaluated our system with small ontologies from Conference track over a dual-core Azure Virtual Machine. We have executed 12 different tasks from this track and recorded an average performance speedup of 1.25 times. Furthermore, we have compared our system with GOMMA's [24] parallel matching techniques. For large category, our system outperforms intra-matcher by 5.2 % on desktop and 55 % over cloud platform. For very large category, our system outperforms intra-matcher by 4.6 % on desktop and 47.7 % over cloud platform. Our system also outperforms Intra&Inter multi-node matcher by 12.8 %.

The rest of the paper is structured as follows. In Section 2, we describe the related work in the field of ontology matching from the perspective of performance. Section 3 describes our proposed methodology on which our system has been constructed. Implementation details of our system, including the stack design of components and their details are presented in Section 4. Section 5 provides a comprehensive evaluation of our system on real-world ontologies of various domains, types, and sizes over multicore desktop PC and Microsoft Azure's public cloud platform. Section 6 concludes this paper.

[1] http://www.w3.org/TR/REC-rdf-syntax/

2 Related work

Nowadays, Internet has grown to become a huge public resource for large and ever-growing heterogeneous data [25]. This excess of knowledge provides a great opportunity for integration by heterogeneity resolution; consequently, researchers have developed ontology matching systems and techniques. As our work is related to performance, In this section we have discussed performance aspect of two types of ontology matching systems, i.e., generic ontology matching systems and ontology matching systems implemented in particular for biomedical ontologies due to their usage, complexity, and size. Furthermore, we have also discussed candidate parallel techniques and their feasibility for ontology matching.

From the technique perspective, a considerable amount of research has been done towards optimizing ontology matching algorithms for better performance [16]. Consequently, various structural partitioning approaches for ontologies have emerged. Falcon-AO [26], a famous ontology matching tool provides a divide-and-conquer approach called PBM [27]. Similarly, an ontology segmentation approach called Anchor-Flood is proposed by [28]. However, in both of these techniques, performance is coupled with the complexity of the partitioning approach. Neither of these techniques benefits from readily available parallelism-enabled platforms for ontology matching.

Among the generic ontology matching strategies and systems, multi-agent systems based on the semantic negotiation have also been proposed in [29] and [30]. These works are based on semantic negotiation protocols HISENE [31] and HISENE2 [32]. In [29], an algorithm is proposed to compute the ontology-based similarity and an agent-based system to perform this computation in a distributed fashion called clustering method. For agent deployment, JADE (Java Agent DEvelopment Framework) [33] is utilized. Although the semantic negotiation has shown promising results in efficiency, its performance is dependent on the amount of communication over an asynchronous message passing protocol required for negotiation between distributed agents. In case of a homogenous cluster of agents, this mechanism is efficient; however, in case of increased heterogeneity, the communication among the agents will increase, adding to the network I/O overhead. In a decentralization approach proposed in [30], the communication cost for large multi-agent systems has been reduced but the semantic negotiation is a learning process that is based on strong collaboration among agents over iterative communication. Thus, communication overhead can be reduced but will fluctuate during the ontology evolution. Furthermore, behavior scheduling of an agent is not pre-emptive, making an agent to be a single Java threaded instance [33]. Although this can be efficient in limited

computational resource environment, it leads to underutilization of computational resources in current multicore systems.

In current state-of-the-art generic ontology matching systems, i.e., AgrMaker [34], LogMap [35], and GOMMA [24], performance has been given a considerable focus to complement accuracy of these systems. AgrMaker with its effectiveness-dependent performance-gain implementation tightly integrates matching algorithms and the system's user interface and relies on user interactions and feedback. Performance of AgrMaker depends upon the iterative execution of matching algorithms as the sample set for the following matching algorithms gets reduced. However, with no parallelism at all, baseline performance of AgrMaker depends upon the complexity of the first matching algorithm. From OAEI 2011.5 campaign, AgrMaker scored highest precision but lagged over performance. It did not participate in 2012 and 2013s OAEI campaign.

Analogous to AgrMaker, LogMap is another generic ontology matching system. Its implementation is claimed as highly scalable from the perspective of ontology matching; however, this scalability is not of any parallel or distributed nature. From the anatomy of LogMap described in [35], it is clear that LogMap is based on a step-by-step matching process (from the lexical indexation to compute overlapping) with a core iterative process for mapping repair and discovery. Although it uses highly optimized data structures for lexical and structural indexing, the whole matching process is sequential in nature. The performance of the system varies with the effectiveness of the matching process; thus, accuracy of the system cannot be preserved for performance-gain.

GOMMA is another ontology matching tool that is considered the most performance efficient. The researchers of GOMMA understand the benefit of parallelism-enabled platforms and provide an effectiveness-independent performance-gain implementation in [36] and [16]. In [36], authors acknowledge the fact that very little research has been performed in devising parallelism for matching problems; furthermore, it describes size-based partitioning scheme to perform parallel matching. Research presented in [36] discusses entity matching in general. However, in [16], authors specifically discuss parallelism techniques pertaining to life science ontologies. They propose inter- and intra-matcher parallelism techniques, which uses parallel and distributed infrastructure for ontology matching to improve performance. Inter-matcher parallelism processes independent matchers on a parallel platform. However, as acknowledged by the authors, inter-matcher has memory requirements as matchers evaluate on complete ontologies creating memory strains during execution. In this case, a matcher thread is loading the ontology information which may not be required for its matching algorithm (e.g., a synonym-based matcher does not require ontology's structure information). On the other hand, intra-matcher parallelism deals with the decomposition of ontology resources into several finer parts with limited complexity so that matcher on these parts can be executed in parallel (e.g., tokenization of concept names). However, defining the granularity for decomposition is not a one-size-fits-all solution. Some ontology concepts may not require to be decomposed. In this case, parallelism technique becomes subjective to the complexity of the ontology resource. Over or under decomposing ontology resources can end up inflicting performance degradation instead. Moreover, neither inter- nor intra-matcher guarantees the optimal computational resource utilization and ontologies used for their evaluation are only of smaller to medium size, i.e., AdultMouse-Anatomy MA (2,737 concepts) with anatomical part of NCI Thesaurus (3,289 concepts) and two GO sub-ontologies Molecular Function (9,395 concepts) with Biological Processes(17,104 concepts).

Due to the excessive utilization of ontologies in biomedical and bioinformatics, some of the ontology matching systems are developed particularly for matching biomedical ontologies [37]. Among them, SAMBO [38] is a pioneering system which provides a framework for aligning and merging biomedical ontologies. SAMBO's implementation is focused towards its matcher algorithms, i.e., a terminological matcher that uses WordNet [39] as thesaurus, a structural matcher that matches the hierarchies, a domain knowledge matcher that uses UMLS as Meta-thesaurus, a learning matcher that generates PubMed [40] abstracts for alignments, and a combination matcher for using more than one matcher for an integrated execution. Despite the fact that integration with third-party thesauri and resources is highly beneficial for the effectiveness, slow nature of these resources creates performance bottlenecks while matching over millions of concepts. Besides that, SAMBO's sequential nature of execution limits its abilities to overcome its performance bottlenecks with better and parallel platforms. In [38], authors failed to mention any performance related aspect of SAMBO while integrating third-party resources. Furthermore, authors have used very small subsets of biomedical ontologies GO (57 and 73 terms) with SigO (10 and 17 terms) [41], and MeSH (15, 39, and 45 terms) [42] with MA (18, 77, and 112 terms) [43] for system evaluation and have not provided any benchmarks regarding large-scale biomedical ontologies. However, results of OAEI 2008 [44] provides performance evaluation of SAMBO, it took 12 hours to complete the anatomy track of biomedical ontologies NCI and MA.

Similar to SAMBO, a hybrid ontology matching strategy for biomedical ontologies is explained in [45]. This technique also utilizes UMLS thesaurus for lexical matching during its sequential execution. The authors failed to

mention any aspect related to performance and repercussions associated while using a third-party thesaurus.

Another ontology matching system with the motivation of producing alignments for biomedical ontologies is ASMOV [25]. With its effectiveness dependent performance, authors of [25] acknowledged that effort is required to improve the computational complexity of the system. With high coupling between ASMOV's performance and computational complexity of matching algorithms and its sequential execution, it is incongruous for ASMOV to avail any performance benefits from parallel platforms. Evaluation of ASMOV is provided in [25]. It is evaluated over anatomy parts of NCI (3304 classes) with Adult Mouse Anatomy (2744 classes) which are far smaller subset of biomedical ontologies. Even for such a small matching task, ASMOV took 3 hours to complete the matching process.

ServOMap [46] is another biomedical ontology matching system but built with the motivation of matching large-scale biomedical ontologies. Instead of using lexical resources like WordNet and UMLS, ServOMap relies on information retrieval and an ontology repository technique. Ontology repository acts as a server of semantic indexes that later contributes to perform similarity operations between ontology entities. Moreover, ServOMap uses lexical and context-based matching algorithms for mapping generation. ServOMap has been able to record better performance over large-scale biomedical ontologies FMA, NCI, and SNOMED-CT; however, from [46] it is understood that this performance gain is because of the absence of third-party resources and thesauri. ServOMap does not implement any performance gain techniques that can exploit parallelism over available multicore platforms for the benefit of large-scale biomedical ontology matching.

From the perspective of data parallelism over distributed platforms, big data technologies like Hadoop, with its MapReduce programming model, query over distributed data with larger volumes. From this regard, it can be considered as a candidate technology for ontology matching; however, the performance benefits of Hadoop and MapReduce are primarily coupled with two aspects, i.e., the size of the data that is typically in gigabytes and terabytes [47] and the structure of the data as Hadoop is unsuitable in situations where structure of the data is as important as the data itself [48]. The ideal size of single chunk of data in Hadoop is 64 MB, which is relatively equal to whole larger-size ontologies; for example, large-scale biomedical ontologies like FMA = 46 MB, NCI = 50 MB, SNOMED extended = 142.6 MB, making an ontology too small to be distributed over HDFS (Hadoop File System). If distributed, it will inflict performance degradation instead. Furthermore, Hadoop is built for unstructured data, distributed in binary format over participating nodes. On the other hand, ontologies are graph like constructs. During matching, relationships among the ontology resources are of vital importance; in case of the binary distribution these relationships are lost. To preserve these relationships, the resources need to be labeled prior to distribution, adding an additional storage and processing overhead. In MapReduce, mappers have to classify whether an incoming ontology resource belongs to which candidate ontology before matching in the reducers at runtime, adding more processing overhead and increased memory footprint. In our experiments, Hadoop-MapReduce based matching has shown 5 times slower performance in contrast with our proposed system due to the stated reasons. From these aspects, Hadoop and Hadoop-like solutions (e.g., CloudBLAST [49]) are unsuitable for the ontology matching problem. Moreover, Hadoop-MapReduce has yet to be equipped with an efficient RDF and OWL plugin. Projects like Reasoning-Hadoop [50], Heart [51], and Hadoop Distributed RDF Store (HDRS) [52] have yet to prove their efficiency and performance.

Parallel ontology matching has been theoretically discussed in [20]. It provides a generic ontology distribution mechanism for selecting a priority ontology and matching it with other candidate ontologies over participating nodes. For parallelization, authors propose the data distribution from the standard parallelization provided by Flynn's taxonomy [53], i.e., SCMD, MCSD, and MCMD. For actual parallel implementation, authors recommend generic techniques like Message Passing and Hadoop-MapReduce. The limitations of both of these approaches in perspective of ontology matching have been discussed earlier; furthermore, [20] fails to provide any details of how an ontology matching system should be using Message Passing middleware or Hadoop-MapReduce platform. Also, it does not provide any evaluation to complement the proposed theoretical details.

In contrast with the above-mentioned techniques and systems, our proposed system implements data parallelism over parallelism-enabled platforms for effectiveness-independent performance-gain during ontology matching. It decomposes complex ontologies into smaller and simpler resource-based scalable subsets depending upon the needs of the matching algorithms. These subsets are serialized to preserve the parsing effort for future matching requests of the same ontologies, and their usage reduces memory strains during execution as subsets required by the matching algorithms are loaded instead of whole ontologies. Our system provides three levels of abstraction for the distribution of matching process, enabling every computing resource to be used at a finer level for effectiveness-independent parallel matching. Equal number of independent matching tasks is assigned to all matching jobs, reducing the chances of idle cores and ensuring the optimal utilization of computing cores during execution. Furthermore, our system aligns the

execution of matching algorithms to minimize the matching space significantly contributing to performance-gain.

3 Proposed methodology

This section provides an overview of our proposed methodology. The primary objective of this methodology is to implement effectiveness-independent performance-gain by drawing abstraction over ontology matching process. These abstractions are drawn to a primitive level such that an independent execution can invoke any matching algorithm without inflicting a change in the implementation of the algorithm. This independent execution is called a matching task (MT) which is the unit of matching process; defined as, a single independent execution of a matching algorithm over a resource from source (O_S) and target ontologies (O_T). These matching tasks are distributed over available computing cores and become the foundation of our data parallelism based ontology matching system. Equations (1, 2, and 3) describe this distribution process.

$$MT_i \cap MT_{i+1} \cap MT_{i+2} .... \cap MT_n = \emptyset \tag{1}$$

$$MT_{Total} \geq m \times n \qquad \forall \quad m \in O_S \quad \& \quad n \in O_T \tag{2}$$

$$MT_{Core} \leftarrow \frac{MT_{Total}}{Cores_{Total}} \tag{3}$$

A primitive example of MT is illustrated in Fig. 1, where a concept C_0 of a source ontology is matched with C_0 of target ontology. Four independent matching tasks perform the complete matching process, for example, MT_1, MT_2, and MT_3 perform element-level string-based, properties-based, and annotation-based matching, respectively, and MT_4 performs structural-level child-based relationship matching. All these matching tasks are mapped to individual cores available in a single- (e.g., multicore desktop) or multi-node platforms (e.g., cloud).

In a single-node, all the matching tasks execute within the computational capacity the node offers. On multi-node platform, the request receiving node becomes the primary node, and it communicates with other participating (secondary) node(s) by sending and receiving control messages for distributed matching.

As illustrated in Fig. 2, to complete the whole matching process, a request is processed through Pre-Matching, Parallel-Matching, and Post-Matching stages. By default ontologies are not scalable structures from the perspective of performance ([54, 56] and [55]). Therefore, the pre-matching stage is introduced where candidate ontologies O_S and O_T are converted into simple subsets with performance and scalability-friendly data structures (e.g., arrays and lists). Furthermore, these subsets are generated depending upon the needs of the matching algorithms making them encapsulated and independent (4, 5, and 6); for example, a string-based matching algorithm for concept names only requires a linear data structure of concepts. As a result, two subsets of candidate ontologies with only concept names will be loaded for matching tasks executing the string-based matching algorithm. Accessing ontology resources from these subsets in following stages is significantly faster due to their smaller size, independent nature, and data structures that can easily be partitioned for data parallelism. This approach effectively contributes in overall performance-gain especially when matching large-scale ontologies. In our experiments, we have recorded as much as 8 times faster ontology resource loading with 4 times smaller memory footprint working with ontology subsets instead of whole ontologies for matching. These subsets are serialized and persisted in repositories, preventing us from

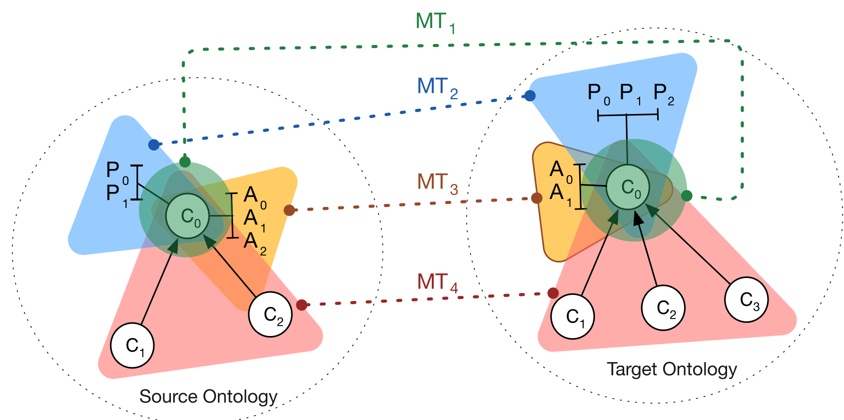


Fig. 1 Matching tasks between two concepts of candidate ontologies

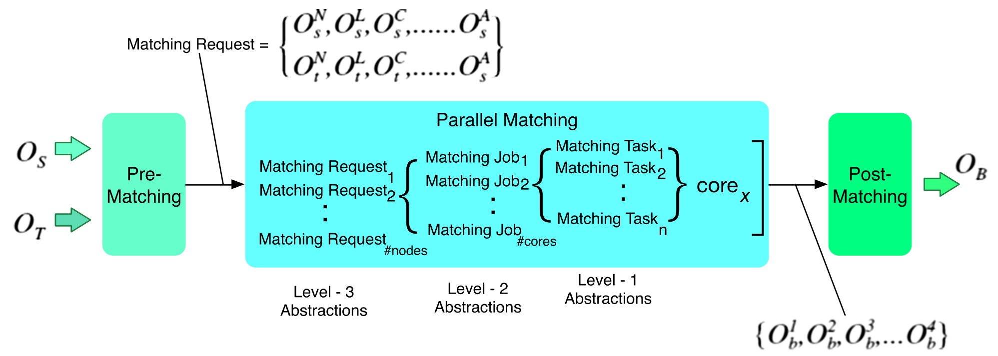


Fig. 2 Execution flow of the matching process

re-generating ontology subsets of already serialized ontologies for future matching requests.

$$i \in Algorithms : Algorithms = \{String, Label, Properties, ..., Child\} \tag{4}$$

$$O_x^i \leftarrow f^i(O_x) \; : \; x \in \{source, target\} \tag{5}$$

$$O_x = \bigcup_{j=1}^{n} O_x^j \; : \; n = NumberOfAlgorithms \tag{6}$$

After pre-matching, parallel-matching stage is invoked. Data parallelism requires each processing core to perform the matching task on a separate piece of candidate ontologies. To enable this, the total number of matching tasks is determined from serialized subsets of ontologies. As illustrated in Fig. 2, by distribution abstractions over matching process, these matching tasks are distributed among the participating nodes as matching requests (single request per node) and their cores (single job per core). As described in (3), number of matching tasks across all matching jobs is equal. This strategy ensures the reduced chances of having an idle processing core during later stages of parallel matching and optimal computing resource utilization. In a single-node platform, matching tasks are only distributed among existing cores as matching jobs; however, in a multi-node platform, distribution is among the participating nodes as matching requests. Each set of matching tasks is assigned to a computing core with knowledge of matching algorithm to be executed on them. Subsequently, all cores in participating nodes are invoked in parallel for the matching process. The following (7, 8, 9, and 10) describe this distribution abstraction implemented by our methodology:

$$MR \leftarrow \sum_{i=1}^{n} MR_i \; : \; n = TotalNodes \tag{7}$$

$$MR_i \leftarrow \sum_{i=1}^{c} MJ_i \; : \; c = TotalCoresPerNode \tag{8}$$

$$MJ_i \leftarrow \left\{ \bigcup_{i=1}^{t} MT_i \right\} \; : \; t = TotalTasksPerCore \tag{9}$$

$$MT_i \leftarrow m \times n \quad \forall \quad m \in O_S \quad \& \quad n \in O_T \tag{10}$$

Apart from distribution over the available computing cores, parallel-matching also aligns the execution of matching algorithms to minimize the matching space for every following matching algorithm execution (11 and 12).

$$O_b^1 \leftarrow (m \times n)_{i=1} \quad \forall \quad i \in Algorithms, \; m \in O_s^i \quad \& \quad n \in O_t^i \tag{11}$$

$$O_B \leftarrow \bigcup_{i=2}^{t} \left(\left(m^i - \left(m^i \cap O_b^{i-1} \right) \right) \times \left(n^i - \left(n^i \cap O_b^{i-1} \right) \right) \right) \quad | \quad O_b^i \geq O_b^{i+1} \tag{12}$$

As illustrated in Fig. 3, a matching process with two matching algorithms is described where element-level string-based matching algorithm determines more matching results than structural-level child-based matching algorithm. Therefore, string-based algorithm is executed in parallel first and generates its intermediate bridge ontology (O_b^1). In the following execution (child-based), ontology resources that are already matched and now part of O_b^1 are removed from loaded ontology subsets (O_s^C and O_t^C) prior

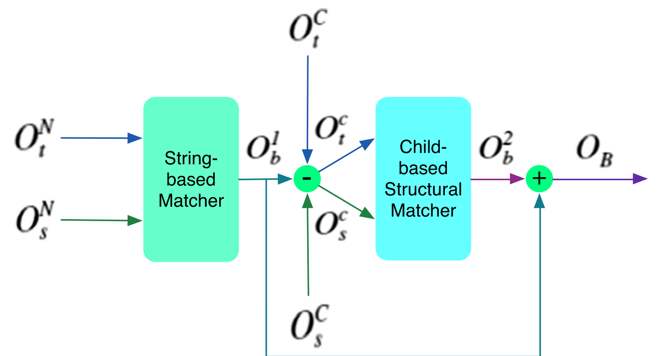


Fig. 3 Algorithm sequence to minimize matching space

to parallel-matching. By this method, the number of expensive matching operations is reduced as they only execute on ontology resources that are still unmatched; consequently, the chances of redundant matching tasks and overall matching performance during run-time are improved. Furthermore, this method also eliminates the chances of redundant matches in the final bridge ontology (O_B).

After completion of parallel-matching stage, i.e., all the parallel matchers have finished their respective matching jobs over their assigned cores in a single- or multi-node environment, post-matching stage is invoked. For this stage, all the matched results are aggregated and the final mediation bridge ontology is generated. In a multi-node environment, the primary node waits for all the secondary nodes to submit their match results before generating the aggregated bridge ontology. The following equations (13, 14, and 15) describe this process in a multi-node platform.

$$O_b^{Job} \leftarrow \bigcup_{i=1}^{t} (m \times n)_i : m \times n \neq \emptyset, \quad t = TotalTasksPerCore \tag{13}$$

$$O_b^{Node} \leftarrow \sum_{i=1}^{j} O_b^{Job=i} : j = TotalJobsPerNode \tag{14}$$

$$O_B \leftarrow \sum_{i=1}^{n} O_b^{Node=i} : n = TotalNodes \tag{15}$$

On a single-node environment, where utilization of computing resources scales down to multicore, generation of mediation bridge ontology is a two-step process (described in 16 and 17):

$$O_b^{Job} \leftarrow \bigcup_{i=1}^{t} (m \times n)_i \quad : \quad m \times n \neq \emptyset, \quad t = TotalTasksPerCore \tag{16}$$

$$O_B \leftarrow \sum_{i=1}^{j} O_b^{Job=i} \quad : \quad j = TotalJobs \tag{17}$$

Firstly, results of matching tasks are combined ($\bigcup$) to become an intermediate bridge ontology per matching job. Secondly, these intermediate bridge ontologies are accumulated ($\sum$) to generate a formal mediation bridge ontology (O_B). The finalized O_B is delivered to the client as the matching response.

4 Implementation details

This section provides the implementation details of our system based upon the proposed methodology. It includes the overall stack design of our system and the details regarding the core components.

4.1 Stack design

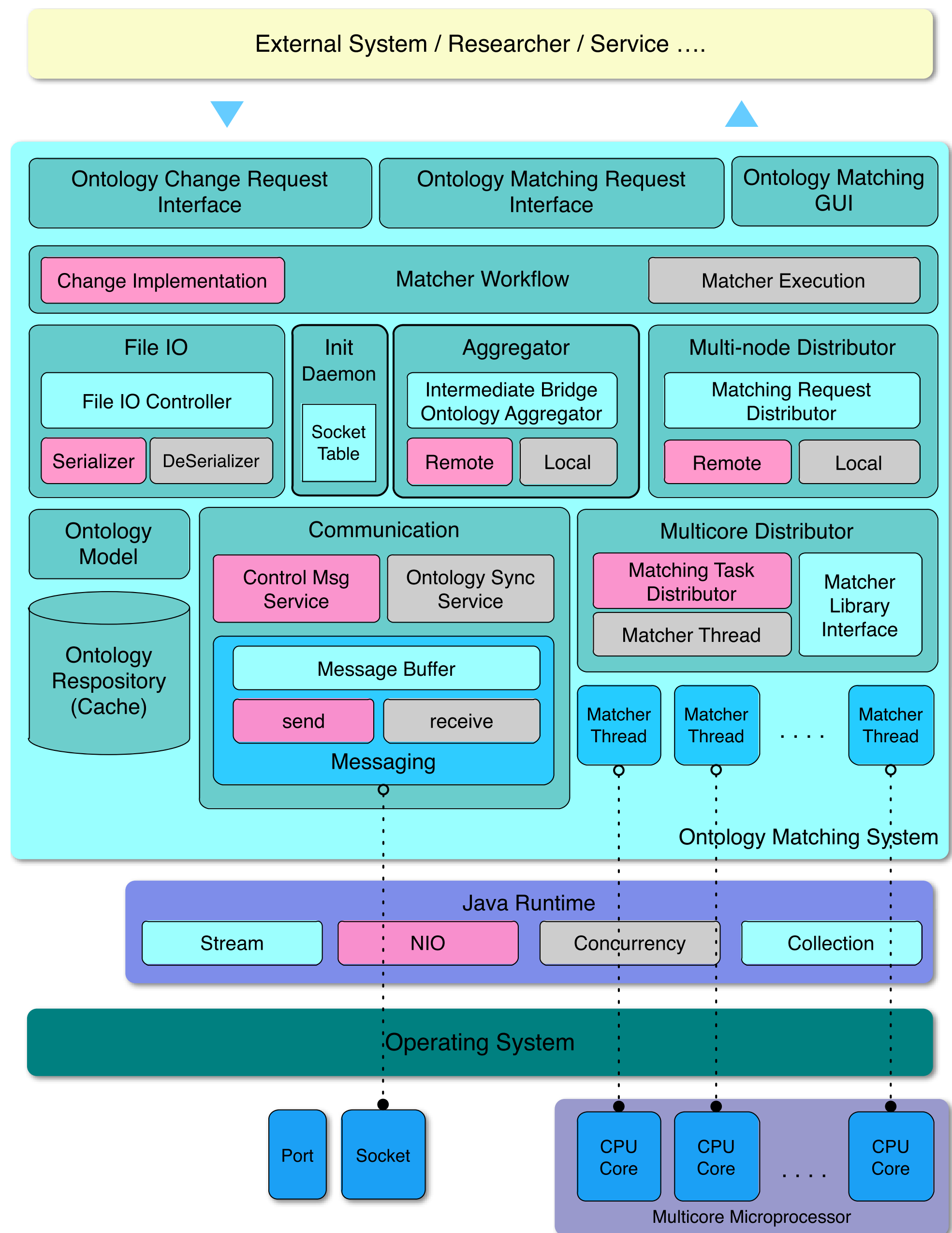
Our proposed system has a layered architecture, following a stack design. With agility in mind, this design supports incremental development and over-time updates without propagating implementation changes across the system. The stack view of system's layers and components is illustrated in Fig. 4. This stack is deployed as an integrated system on all participating nodes involved in ontology matching.

Our system provides two interfaces to interact with the client, i.e., a web service and a graphical user interface (GUI). If a third party system, service, or a client wants to use the parallel matching facility, they can interact by utilizing Ontology Matching Request Interface. This interface is hosted by a SOAP-based web service to be consumed by client programs and systems. Adjacent to the request interface is a GUI-based interaction component which facilitates the utilization of our system by an individual researcher via browser. In parallel, there is an Ontology Change Request interface that is used to implement the evolution process of ontology's design. Ontology change request interface receives the change updates for serialized ontologies to support continuity in ontology change management. These interfaces and GUI rely on lower-level core components for actual parallel matching and change implementation, executing over single- and multi-node platforms.

The core of our system consists upon six loosely coupled components (File IO, Init Daemon, Multi-node Distributor, Aggregator, Communication,[2] and a Multicore Distributor) and an ontology repository. These components with their focused responsibilities are integrated with an intermediate workflow layer called Matcher Workflow. This workflow layer hosts two paths for system execution, i.e., a matcher execution for parallel matching request and a change implementation to support ontology's design evolution. Among the core components, init daemon is responsible for setting up the multi-node environment by providing a socket table for all the participating nodes. This setup is required, prior to any distributed matching. File IO component is used for parsing and loading candidate ontologies. It is responsible for serializing candidate ontology subsets and implementing CRUD operations on these subsets for change implementation. Multi-node distributor is responsible for distribution of matching process as matching requests over participating nodes via control messages. These messages are sent and received by the communication component. This component also hosts an ontology synchronization service to replicate ontology changes over secondary repositories hosted by participating nodes. For local distribution of matching tasks such as matching jobs over available

[2]Utilization of communication by each core component is described in the components explanation.

Fig. 4 Stack design



cores, multicore distributor is used. This component exploits the existing cores by implementing thread-level parallelism. Each matcher thread is assigned to its matching job coupled with instance of matching algorithms over candidate ontologies.

For the utilization of multicore platforms, a programming language is required with a strong emphasizes on concurrency and platform independence. Java is one such language that is equipped with an effective multithreading model and is available for most of the computing platforms. Keeping these facts in perspective, we have provided our system's implementation in Java and used its concurrency, collection, NIO and stream libraries for our benefit.

4.2 Core component details

This section provides details regarding the inner workings of the core components of our proposed system.

4.2.1 Init daemon

Initialization Daemon (Init Daemon) is responsible for setting up the environment for the matching process. It executes in pre-matching stage of the system. In a multi-node platform, init daemon is responsible for providing communication objects of every participating node in a collection called socket table. This table is generated at every node

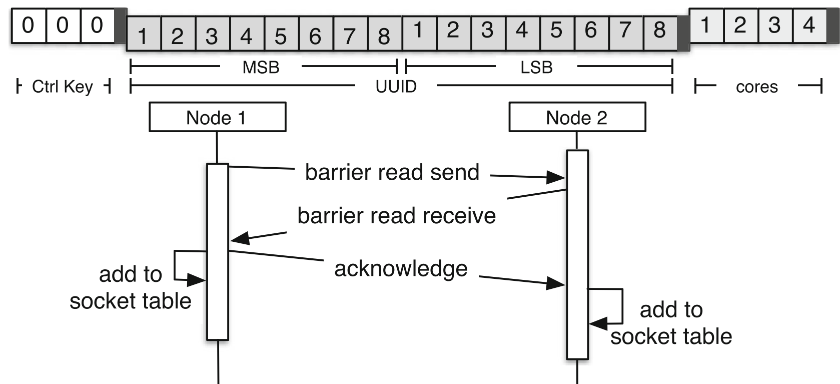


Fig. 5 Barrier read sequence diagram

and contains the collection of socket objects for every other node, distinguished by unique identifiers (UUID). From a higher level abstraction, each UUID represents a running instance of a participating node in a multi-node environment.

Algorithm 1 describes the details of init daemon's socket table creation. Prior to execution, each daemon holds a text file containing ranks (unique integer values) of participating nodes and their respective IP addresses. Algorithm 1 enables each node to generate its own UUID, attach it with information regarding available computational resources on that node and shares it among the participating nodes; consequently, each daemon receives a UUID with available number of cores over a particular node on a socket object. All the receiving UUIDs with their corresponding number of cores and socket objects are stored as a socket table in every node's main memory. At communication level, sharing of UUIDs among the nodes is performed by a barrier read. This communication is illustrated in the sequence diagram of Fig. 5. Barrier read is initiated by invoking a 24 byte control message, sent from one node to the other node(s). This message contains the respective UUID of the sending node (16 bytes), number of available computing cores (4 bytes), and ctrl key (3 bits). Every receiving node acknowledges the control message by similar reply and subsequently attaches the receiving port number with the received UUID and forwards it to its socket table. Figure 6 provides a depiction of socket tables in a tri-node environment after init daemon setup. This strategy enables the system to avoid unnecessary file access and re-creation of socket objects for every communication. Socket tables are further used by each node to send and receive control, ontology change and synchronization messages during system execution.

Algorithm 1 Generate socket table

Require: $node \geq 2$,
  $temp = 1$
  $uuidMsg \leftarrow$ generateUUID()
  $cores \leftarrow$ Runtime.getNumberofCores()
  $rank \leftarrow$ getRankforThisNode()
  while $temp < ShiftLeft(1, nodes)$ **do**
    if $temp \geq nodes$ **then**
      $stop$
    end if
    $sender = rank$
    $receiver \leftarrow$XOR$(rank, temp)$
    $socket \leftarrow$getSocket$(receiver)$
    if $sender > receiver$ **then**
      sendMessage$(socket, uuidMsg, cores)$
      socketTable.add($socket$,receiveMessage($socket$))
    else
      socketTable.add($socket$,receiveMessage($socket$))
      sendMessage$(socket, uuidMsg)$
    end if
    $temp = temp + 1$
  end while

4.2.2 File IO

File IO component is responsible for ontology loading, subset creation, and providing an interface to ontology repository for ontology persistence. It also executes in prematching stage of the system. File IO provides serialization and deserialization operations. When the system receives a new ontology, i.e., a candidate ontology that has not been converted into subsets, file IO parses it to create a respective object model. This object model is persisted as serialized subsets according to the needs of matching algorithms along with the ontology hash value. For matching request of already serialized ontology, deserializer loads the required subsets into respective ontology models and provides these models to distributor component for parallel matching operations.

To facilitate parallel matching in a multi-node platform, ontology subsets need to be available on every participating node; therefore, the ontology subsets are replicated over secondary repositories with the help of connectivity

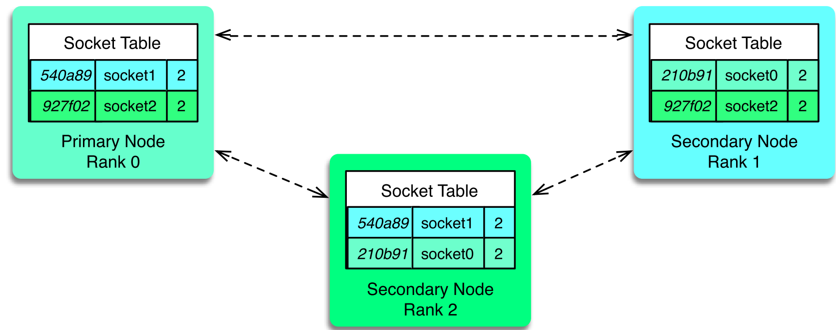


Fig. 6 Socket tables in a tri-node (2 cores/node) environment

information provided by the init daemon. Communication between primary and secondary node(s) for subset replication is illustrated in the sequence diagram of Fig. 7. Unlike barrier read, control messaging for ontology subset replication is a two-step process. Firstly, primary node sends a 24 byte message to secondary node(s) containing ontology UUID (16 byte), size of the subset to be sent (4 bytes) and ctrl key (3 bits). Secondary node(s) receive this message and create receiving buffers of size of the subset and send prompt acknowledgments to the primary node. Secondly, the primary node sends the subset to the secondary node(s). By this method, matching threads only load subsets from their local repositories, avoiding the internode communication during matching.

File IO is also responsible for implementing ontology changes. To implement a change, the ontology must be loaded inside nodes memory as an instance of the ontology model. Ontology change request interface through matcher workflow provides file IO with the UUID for ontology to be updated. Deserializer loads the required ontology from the repository into an ontology model instance. This instance is returned to file IO for change implementation. Matcher workflow receives the instance of the ontology model to be updated from ontology change request interface. A change can be of many types, from a triple update to an addition of an entirely new hierarchy. Operations for change implementation are classified into Create, Update, and Delete types. These operations are used by file IO over ontology model instance for change implementation. ChangeManager is a command pattern [57] implementation. Apart from agility, this pattern provides undo and redo operations for change implementation. After the change implementation, updated subsets are serialized back in the repository and in case of multi-node platform, these changes are replicated over repositories of secondary nodes. Communication between primary and secondary node(s) for change implementation is illustrated in the sequence diagram of Fig. 8. Similar to subset replication, change implementation request is also a two-step process. Firstly, primary node sends a 24 byte message to secondary nodes containing information regarding the ontology that needs to be updated (16 bytes), the size of updates that needs to be implemented (4 bytes), and ctrl key (3 bits). Secondary node(s) receive this message and deserialize the candidate ontology into an ontology model object; subsequently, they create receiving buffers of size of the updates and send a prompt acknowledgment to the primary node. Secondly, the primary node sends the actual changes to the secondary nodes. After the change implementation, updated ontology model instance is sent to file IO for persistence. File IO serializes the ontology model and stores it back in the ontology repository.

4.2.3 Distributor

Distributor components (multicore and multi-node distributors) are collectively responsible for the distribution of

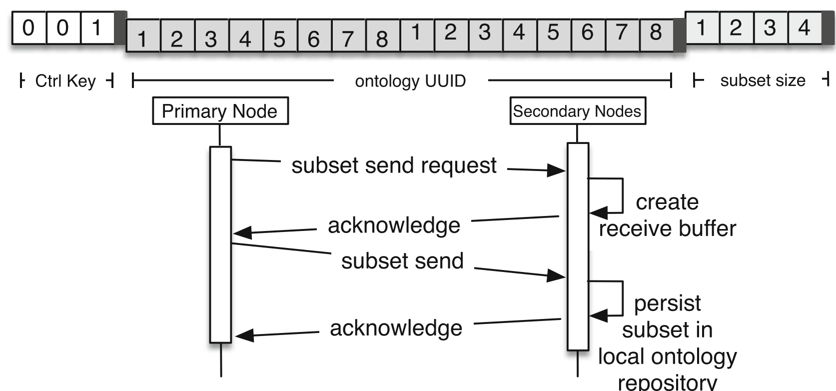


Fig. 7 Ontology subset replication sequence diagram

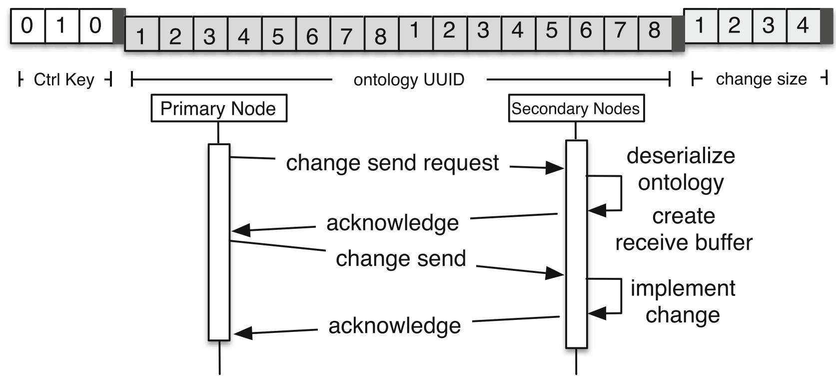


Fig. 8 Ontology change request sequence diagram

matching process over computational resources for invoking parallelism on candidate ontologies (O_S, O_T) in parallel matching stage. To accomplish this responsibility, the matching process is layered into three levels of abstraction, i.e., from macro-level matching request (MR) and grainer-level matching jobs (MJ) to finer-level matching task (MT).

The distribution process for implementing data parallelism in a multi-node environment is illustrated in Fig. 9. A whole matching request (classified as a matching process) received by the primary node is divided among participating nodes depending upon their computational resources. A matching request received by an individual node is further subdivided into matching jobs such that each job on a node contains an equal number of matching tasks. Subsequently, a matching job is assigned to execute over a processing core available on a participating node. This technique provides three major benefits to our system: (i) better scalability, as chances of idle cores are minimal because each core is assigned with equal number of matching tasks; (ii) implementing the most efficient scenario of parallel execution, i.e., one job per core; and (iii) matching tasks are independent among themselves, other matching jobs, and other matching requests running remotely, ensuring no communication required between nodes during parallel matching. These three characteristics of the distribution are the foundation of achieving data parallelism for parallel matching.

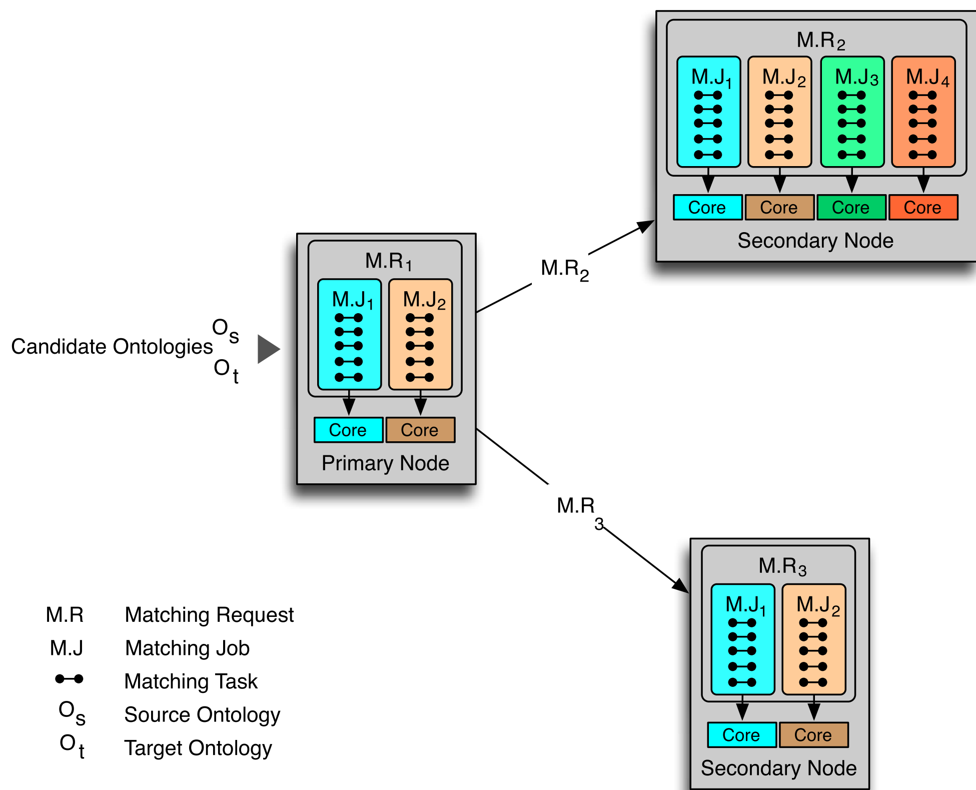


Fig. 9 Matching request distribution in a tri-node environment

In the case of single-node systems, the distribution process scales down to multiple cores on one node. The multicore distributor divides a whole matching request into matching jobs with an equal number of independent matching tasks. Each job is assigned to run over a particular core, consequently achieving data parallelism.

Algorithms 2, 3, and 4 describe the distribution of matching tasks in single- and multi-node environments. In the case of single-node platform, multicore distributor (Algorithm 3) is invoked. It identifies the number of participating cores from the native runtime and calculates the partition slab by dividing the size of the bigger ontology with the number of cores and taking its ceiling value in case of fraction. A matching job per core is created and invoked by thread-level parallelism. For example, in case of matching conference ontology "iasted" having 140 concepts with another conference ontology "cmt" with 29 concepts over a quad-core single-node platform, Algorithm 3 first calculates the partition slab (140/4 = 35). First 35 concepts of iasted ontology are assigned to be matched with all the 29 concepts of cmt ontology as first matching job with a total number of $35 \times 29 = 1015$ matching tasks. This matching job is invoked as first matching thread. In parallel, next 35 concepts of iasted ontology are matched with all the 29 concepts of cmt ontology as second matching job with the same number of 1015 matching tasks, invoked as second matching thread. Similarly, third and fourth matching threads are also assigned in parallel with their respective matching jobs of 1015 matching tasks each, thus distributing the whole matching process of 4060 matching tasks evenly among 4 cores for parallel matching.

In multi-node environments, distribution algorithm invokes the multi-node distributor (Algorithm 4) which receives the information regarding the available computational resource of participating nodes from init daemon. Distribution slab is calculated and control messages are created sent with matching requests to the secondary nodes. The size of these control messages is 64 bytes containing information regarding source and target ontologies (32 bytes), start index (4 bytes), partition slab (4 bytes), matcher algorithm id (16 bytes), and ctrl key (3 bits). In reply, a single byte acknowledge message is received by the primary node. This process is illustrated in the sequence diagram of Fig. 10. To elaborate the execution of Algorithm 4, consider

Algorithm 2 Distributor algorithm

Require: $nodes > 0$
  if $nodes$=1 **then**
    MulticoreDistributor(O_S,O_T)
  else
    Multi-nodeDistributor(O_S, O_T)
  end if

Algorithm 3 Multicore distributor algorithm

Require: $nodes > 0$
  $cores$ ←Runtime.getNumberOfCores()
  if $nodes$=1 **then**
    $start$=0
    $bigOnt \leftarrow (size_S \geq size_T)?O_S : O_T$
    $smallOnt \leftarrow (size_S < size_T)?O_S : O_T$
    $Partition_{slab} = \lceil bigOnt.size/cores \rceil$
    SPAWN MATCHER THREADS:
    for do$i = 1$ **to** $cores$ **do**
      $end = start + Partition_{slab}$
      if $end \leq bigOnt.size$ **then**
        $end = bigOnt.size$
      end if
      MatchingJob.create($MatchingTasks[start, end)$, big, $small$, $matcher$)
      thread.run($matchingJob$)
      $start = end$
    end for
  else
    RECEIVE MATCHING REQUEST:
    controlMessage.receive($matchingRequest$)
    $Partition_{slab} = (end - start)/cores$
    GOTO SPAWN MATCHER THREADS
  end if

Algorithm 4 Multi-node distributor algorithm

Require: $nodes > 1$
  $nodes$ ←initDaemon.getNoOfNodes()
  $participatingCores = \sum node.\#cores$
  $start$=0
  end=0
  $bigOnt \leftarrow (size_S \geq size_T)?O_S : O_T$
  $smallOnt \leftarrow (size_S < size_T)?O_S : O_T$
  $Distribution_{slab} = \lceil bigOnt.size/participatingCores \rceil$
  for $node \leftarrow nodes$ **do**
    $end = start + Distribution_{slab} \times node.\#cores$
    if $end \leq bigOnt.size$ **then**
      $end = bigOnt.size$
    end if
    MatchingRequest.create($[start, end)$, big, $small$, $matcher$)
    if $node.isLocal$ **then**
      local.MulticoreDistributor($matchingRequest$)
    else
      controlMessage.send($matchingRequest$)
    end if
    $start = end$
  end for

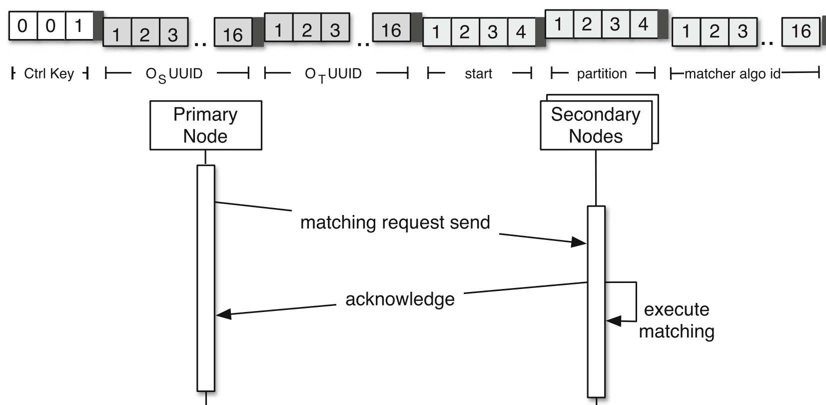


Fig. 10 Ontology matching request sequence diagram

the example of matching process between two biomedical ontologies, “adult mouse anatomy (2,744 concepts)” with “NCI human anatomy (3,304 concepts)” over the tri-node environment illustrated in Fig. 9. Algorithm 4 first calculates the distribution slab by dividing the size of the bigger ontology (NCI human anatomy) with the total number of participating cores (3,304/8=413). Request for matching first 826 concepts of NCI human anatomy ontology with all the concepts of adult mouse anatomy is created. This matching request is distributed over the local node by calling multicore distributor (Algorithm 3) which calculates the partition slab for 2 available cores ((826-0)/2=413). Consequently, two matching jobs are invoked by thread-level parallelism starting from concepts [0 to 413) and [413 to 826) of NCI human anatomy ontology respectively. As first secondary node is a quad-core resource, second matching request is generated for matching next 1,652 concepts of NCI human anatomy ontology starting from [826 to 2,477) with all the concepts of mouse anatomy. This matching request is sent via control message using communication protocol illustrated in Fig. 10. and received by the multicore distributors (Algorithm 3) of first secondary node. Matching request is extracted from the control message and four matching jobs are created each with 413 concepts of NCI human anatomy ontology ([826 to 1,239), [1,239 to 1,652), [1,652 to 2,065), and [2,065 to 2,478)) to be matched with all the concepts of mouse anatomy. Similar to second matching request, third matching request is generated for the other secondary node which distributes it between two matching jobs ([2,478 to 2,891), and [2,891 to 3304)), thus distributing the whole matching process of over nine million matching tasks (9,066,176), evenly among 3 nodes for parallel matching.

From the description of multi-node distributor algorithm, it is quite clear that our distribution component assumes the multi-node environment to be homogenous. The distribution slab calculated by Algorithm 4 precisely considers the parallelism ability of participating nodes, i.e., number of computing cores per node; however, in case of heterogeneity among the computational ability (processor frequency, memory size, and IO performance) of participating nodes, an idle core can exist as one node might complete its the matching request prior to the others.

Distributor components also provide an interface to the matching library. Matching algorithms can be plugged in and out of the system or can be executed as suites based on software engineering design principles. This interface ensures the effectiveness-independent performance-gain aspect of our system and decouples the performance of the system from the effectiveness and accuracy of the system. By default, the system provides a library of element-level and structural-level matching algorithms. Furthermore, matching algorithms provided by various semantic web experts have been incorporated for evaluation.

4.2.4 Aggregator

This component is responsible for aggregating matched results from participating nodes and generating the required mappings in post-matching stage. Depending upon the deployment environment (single- or multi-node), aggregator accumulates matched results from two different interfaces (local and remote) and creates a formal representation of mappings called Mediation Bridge Ontology (MBO). MBO is a pattern-based bridge ontology that provides mediation between different candidate ontologies. Type and structure of the MBO can be changed depending upon the needs by customization of bridge ontology definition.

In a single-node, aggregator receives the intermediate bridge ontologies from each core as a result of a matching job via local interface. All the intermediate bridge ontologies are aggregated to generate the formal and final mediation bridge ontology.

In the case of multi-node environment, the primary node receives the intermediate bridge ontologies from local interface and remote interface where secondary nodes send their intermediate bridge ontologies as matching response. Aggregator at primary node aggregates all these

intermediate bridge ontologies to generate the formal and final mediation bridge ontology.

5 Evaluation and discussion

In this section, we describe a comprehensive experimentation performed on our proposed system. For the evaluation, we have used the OAEI 2013 dataset of real world ontologies. Our system is evaluated over Anatomy, Library, Large Biomedical, and Conference tracks of OAEI 2013's dataset. The candidate ontologies used in these tracks are of various sizes, covering the different magnitudes of ontology matching problem.

We have executed three different libraries of ontology matching algorithms (computational complexity $\geq O(n^2)$) provided to us by different semantic web experts. Our system is evaluated over two platforms: (i) a single-node quad-core desktop PC, equipped with 3.4 GHz Intel(R) Core i7(R) Hyper-Threaded (Intel(R) HT Technology) [58] CPU (2 threads/core) with 16 GB memory, Java 1.8 and Windows 7 64 bit OS, and (ii) a public cloud Microsoft Azure instance with two virtual machine (VM) configurations, i.e., standard A4 VM instances with 8 cores, 14 GB of memory, Java 1.8, and Windows 2012 R2 Guest OS running over an AMD Opteron(TM) 2.1 GHz CPU and A2 VM instance with 2 cores, 3.5 GB memory, Java 1.8, and Windows 2012 R2 Guest OS running over an Intel(R) Xeon(R) 2.1 GHz CPU.

5.1 Anatomy track

The anatomy track consists of mapping generation between the Adult Mouse Anatomy (2,744 concepts) [59] and part of NCI Thesaurus describing human anatomy (3,304 concepts). Beside their larger size, these ontologies are carefully harmonized by OAEI experts such that a rather high number of mappings can be found by trivial string matching techniques and a good share of non-trivial mappings require complex analysis over ontology structures. To generate the bridge ontology we have used the default matching library with String-based, Label-based, and Child-based Structural matching algorithms.

We have executed our system in both multicore desktop and cloud scenario as a single-node execution (illustrated in Fig. 11). Matching requests are generated from the client; consequently, adult mouse anatomy (O_S) and human anatomy (O_T) ontologies are loaded in parallel by file IO and provided to multicore distributor component. With the knowledge of available computing resources and ontology subsets (O_s, O_t) required by matching algorithms, distributor creates 8 independent matching jobs. Each job is allocated with a set of equal numbers of independent matching tasks $\left(\frac{AdultMouseAnatomy_{classes} \times HumanAnatomy_{classes}}{8}\right)$.

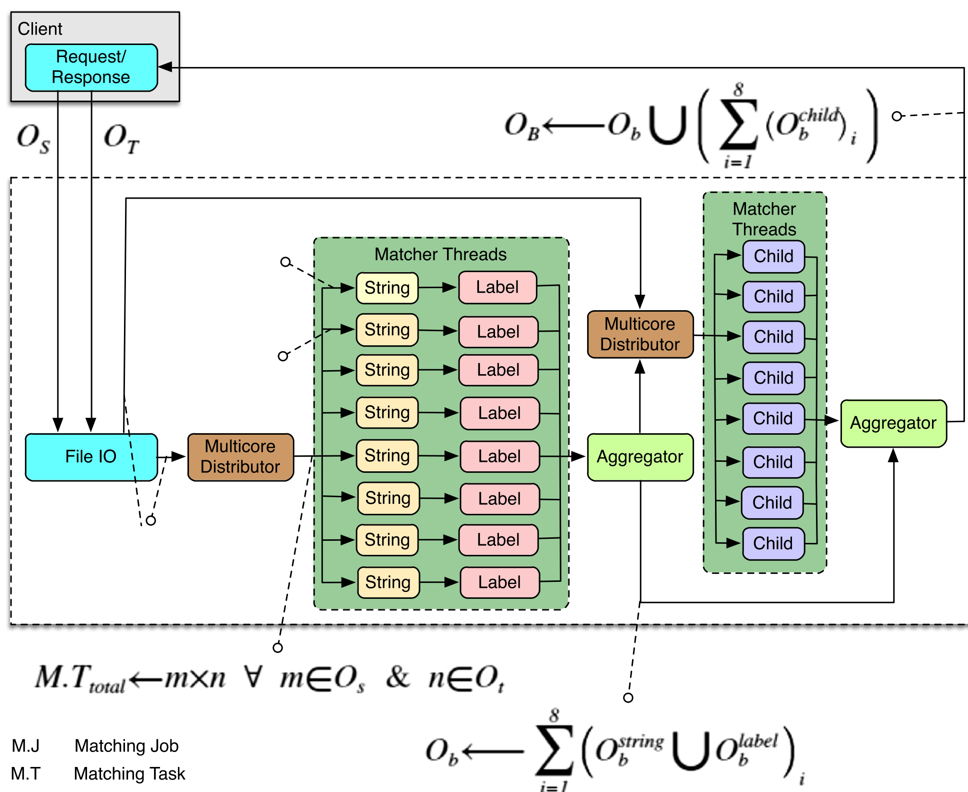


Fig. 11 Parallel flow for anatomy track over single-node

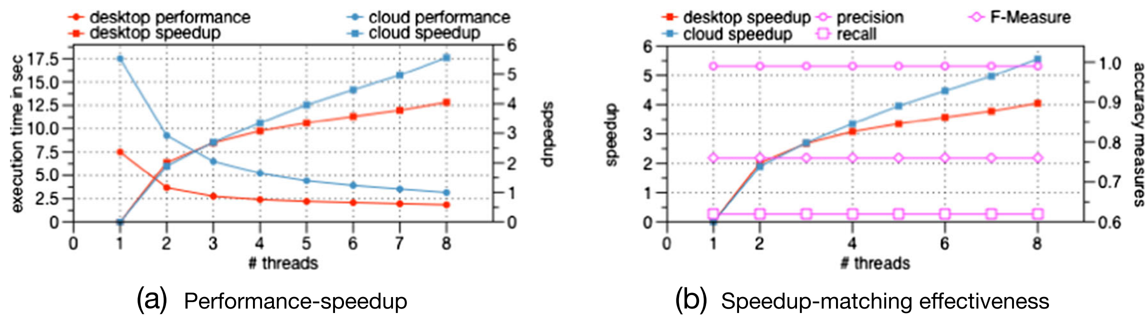


Fig. 12 Results from anatomy track

As String and Label-based matching algorithms execute on the same subsets of the respective ontologies, the distributor assigns these two algorithms to every matching job. Subsequently, the distributor allocates each matching job to a single-core for matching. After completion of all jobs, an intermediate bridge ontology (O_b) is created by the aggregator. Thereafter, the distributor loads the subsets of adult mouse anatomy and human anatomy required for the Child-based structural matching algorithm through file IO and follows the same procedure as before. After the completion of the Child-based structural matching algorithm, the aggregator accumulates its results with the intermediate bridge ontology (O_b) and generates the formal mediation bridge ontology (O_B). This bridge ontology is finally delivered to the client as a response.

Results from both the scenarios (desktop and cloud) are illustrated in Fig. 12. For the desktop, scenario, the matching request executes over the quad-core desktop, and the results are described in Fig. 12a. The sequential process (illustrated in Fig. 13) takes 7.5 seconds to complete the matching request; however, with the use of our data parallelism enabled system over multiple cores, total matching time starts improving as more cores are introduced. Our system completes the matching process in less than 2 seconds over 4 cores (= 8 threads) with a speedup of 4 times. The same matching request is executed for the second scenario over the Azure VM. The sequential process over the VM takes 17.5 seconds to complete; however, our system completes the whole matching process over 8 threads within 3.1 seconds with an impressive speedup of 5.5 times. The overall performance of the matching process is slightly slower over the Azure VM due to the virtualization layer (Hyper-V).

Accuracy preservation throughout the performance speedup is illustrated in Fig. 12b. As stated earlier, for effectiveness-independent performance-gain, the performance is extracted from parallel threads over multiple cores, and no changes in matching library have been made for performance reasons. Consequently, the matching effectiveness (e.g., precision, recall, F-Measure) stays the same throughout the performance speedup.

The same matching track was evaluated by [16] as a medium-scale problem by its intra-matcher parallelization on a single node. Matchers are evaluated individually and possibly generate individual alignments. These alignments are later to be aggregated for a comprehensive bridge ontology. A performance speedup of 3.6–4.2 times (depending upon the matching algorithm) have been achieved by intra-matcher of [16]. In our system, matchers execute as a combined matching process; consequently, it efficiently generates a single comprehensive bridge ontology instead. Even with an inferior hardware platform, our system slightly outperforms the performance speedup of [16] on the desktop scenario, i.e., 4 times (vs. mean(3.6–4.2)) and largely outperforms it by 41 % when executed in the cloud scenario, i.e., 5.5 times (vs. mean(3.6–4.2 times)).

5.2 Library track

The library track consists of mapping generation between the STW [60] and the TheSoz thesaurus [61] ontologies. Both ontologies provide a vocabulary for economics with respect to social science subjects. These ontologies are primarily used by libraries for indexation and retrieval. Although lightweight, these ontologies are large with STW containing 6,575 concepts and TheSoz containing 8,376

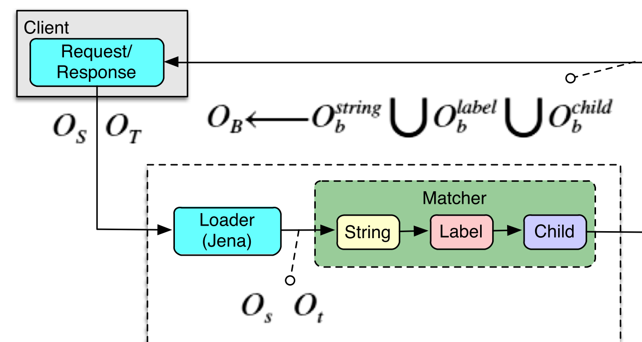


Fig. 13 Sequential flow on a single-node

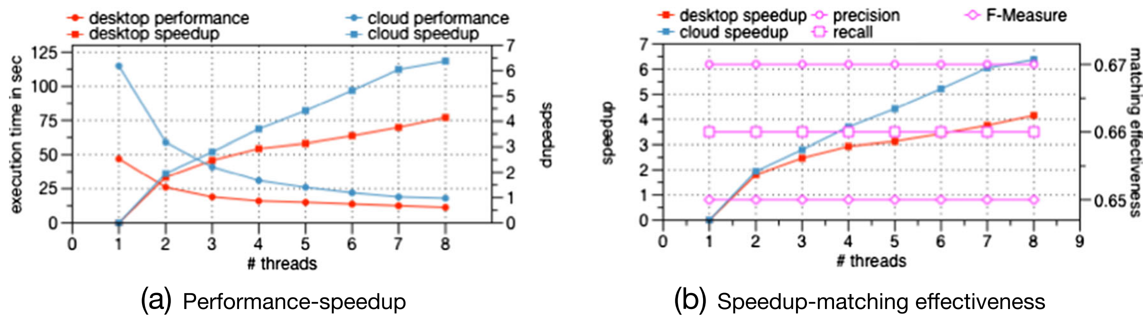


Fig. 14 Results from library track

concepts. To generate the bridge ontology we have used the same matching library used earlier in anatomy track.

Similar to anatomy track, we have executed our system in both multicore desktop and cloud scenario as a single-node execution. Results from these scenarios are illustrated in Fig. 14. For the single-node desktop scenario, the sequential process takes close to 47 seconds to complete the matching request; however, our system completes the matching process around 11 seconds over 8 threads with an impressive performance speedup of 4.15 times. Same matching request is executed for the second scenario over the single-node Azure VM. The sequential process over the VM takes close to two minutes to complete; however, our system completes the whole matching process over 8 threads in 18 seconds with an impressive speedup of 6.38 times. Furthermore, similar to anatomy tasks, the accuracy of the matching process stays preserved with the same effectiveness throughout the performance speedup (illustrated in Fig. 14b).

5.3 Large biomedical ontologies track

The large biomedical ontologies track consists upon finding mappings between FMA, SNOMED-CT, and the NCI ontologies. These ontologies are semantically rich, substantially complex, and significantly large containing thousands of concepts. For this track we have used a matching library with String-based, Annotation-based, and Child-based structural matching algorithms for bridge ontology generation. This track consists upon 6 tasks that are described in following subsections.

5.3.1 Task 1: FMA-NCI small fragments

This task consists upon matching relatively smaller fragments of FMA and NCI ontologies. The FMA fragment consists upon 5 % of the whole FMA ontology (3,696 concepts) while the NCI fragment consists upon 10 % of the whole NCI ontology (6,488 concepts).

We have executed our system in both multicore desktop and cloud scenario illustrated in Fig. 15 as a single-node execution. Matching requests are generated from the client; consequently, smaller fragments of FMA (O_S) and NCI (O_T) ontologies are loaded in parallel by file IO and provided to multicore distributor component. With the knowledge of available computing resources and ontology subsets (O_s, O_t) required by matching algorithms, distributor creates 8 independent matching jobs. Each job is allocated with a set of equal numbers of independent matching tasks $(\frac{FMA_{classes} \times NCI_{classes}}{8})$. As String and Annotation-based matching algorithms execute on the same subsets of the respective ontologies, distributor assigns these two algorithms to every matching job. Subsequently, distributor allocates each matching job to a single-core for matching. After completion of all jobs, an intermediate bridge ontology (O_b) is created by aggregator. Thereafter, distributor loads the subsets of adult mouse anatomy and human anatomy required for Child-based structural matching algorithm through file IO and follows the same procedure as before. After the completion of Child-based structural matching algorithm, aggregator accumulates its results with the intermediate bridge ontology (O_b) and generates the formal mediation bridge ontology (O_B). This bridge ontology is finally delivered to the client as a response.

Results for this track from both the scenarios (desktop and cloud) are illustrated in Fig. 16. For the desktop scenario, the matching request executes over the quad-core desktop. The sequential process (similar to the illustration in Fig. 13) takes 48 seconds to complete the matching request; however, our system completes the matching process in slightly over 11 seconds over 4 cores (= 8 threads) with the performance speedup 4.2 times. Same matching request is executed for the second scenario over the Azure VM. The sequential process over the VM takes 100 seconds to complete; however, our system completes the whole matching process over 8 threads in slightly over 15 seconds with an impressive speedup of 6.5 times. Furthermore, similar to anatomy track the accuracy of the matching process stays preserved with the same effectiveness throughout the performance speedup (illustrated in Fig. 16b).

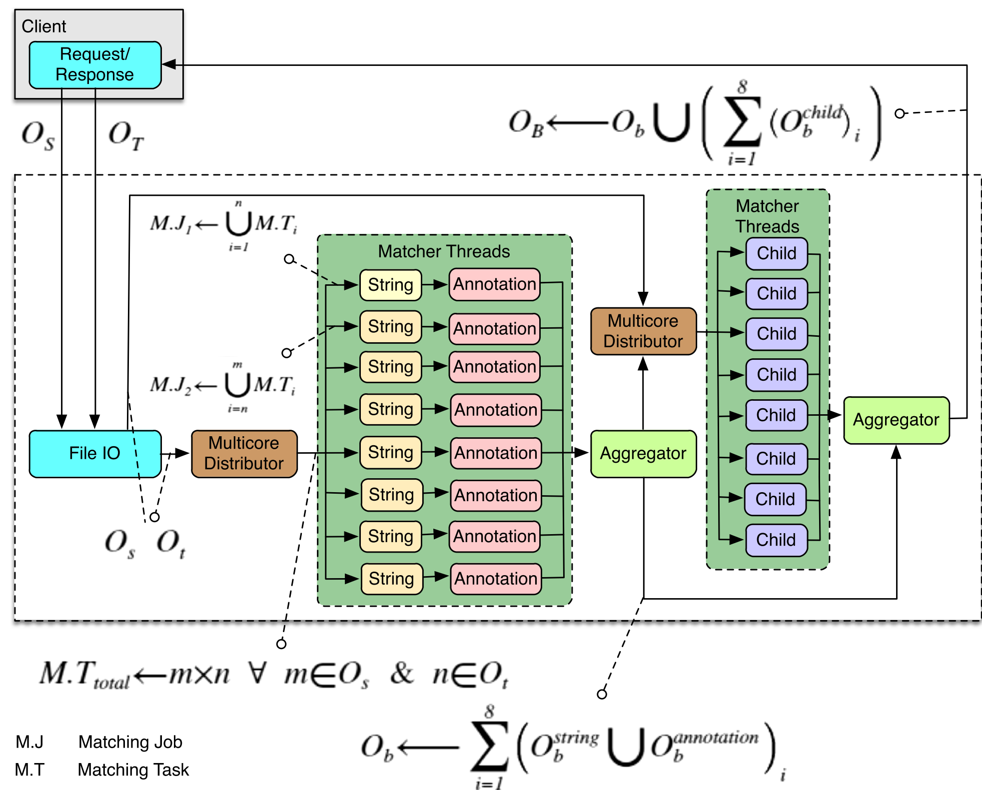


Fig. 15 Parallel flow for large biomedical track over single-node

5.3.2 Task 2: FMA-NCI whole ontologies

This task consists upon matching the whole FMA and NCI ontologies. The FMA ontology consists upon 78,989 concepts while the NCI ontology consists upon 66,724 concepts. Due to the very large size of the ontologies, the matching process is scaled over multi-node environment, i.e., 3 desktops and Azure VMs with above-stated specification for the first and second scenarios respectively.

As illustrated in Fig. 17, the primary node receives the matching request for candidate ontologies, whole FMA (O_S) and NCI (O_T) from the client. Candidate ontologies are loaded in parallel by file IO of the primary node which consequently invokes the multi-node distributor for distributed matching. Socket table provides the multi-node distributor with socket objects for secondary nodes. With the knowledge of available computing resources (3 nodes, 1 primary and 2 secondary, each with 8 cores available) and the ontology subsets (O_s, O_t) required by matching algorithms, the multi-node distributor of the primary node creates 3 independent matching requests of equal size. The first matching request is forwarded to the local multicore distributor where 8 independent matching jobs with an equal number of independent matching tasks are created. Subsequently, multi-node distributor sends control messages to other secondary nodes with their respective matching requests. At receiving nodes, these matching requests are forwarded to their local multicore distributor. Assigned

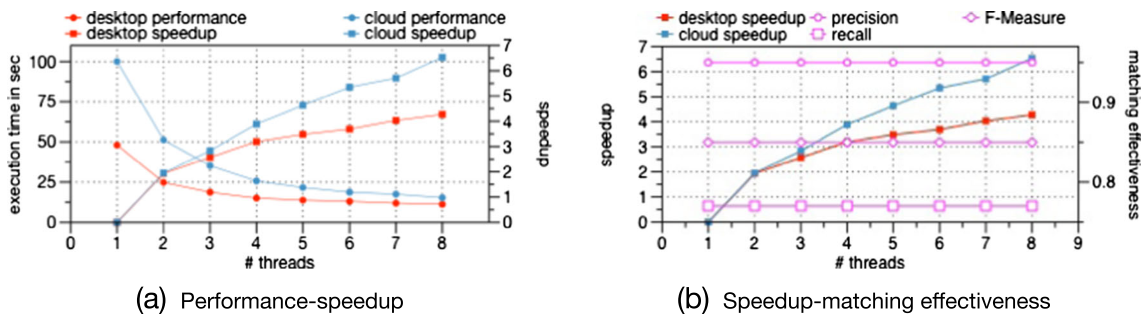


Fig. 16 Results from large biomedical ontologies track, task 1

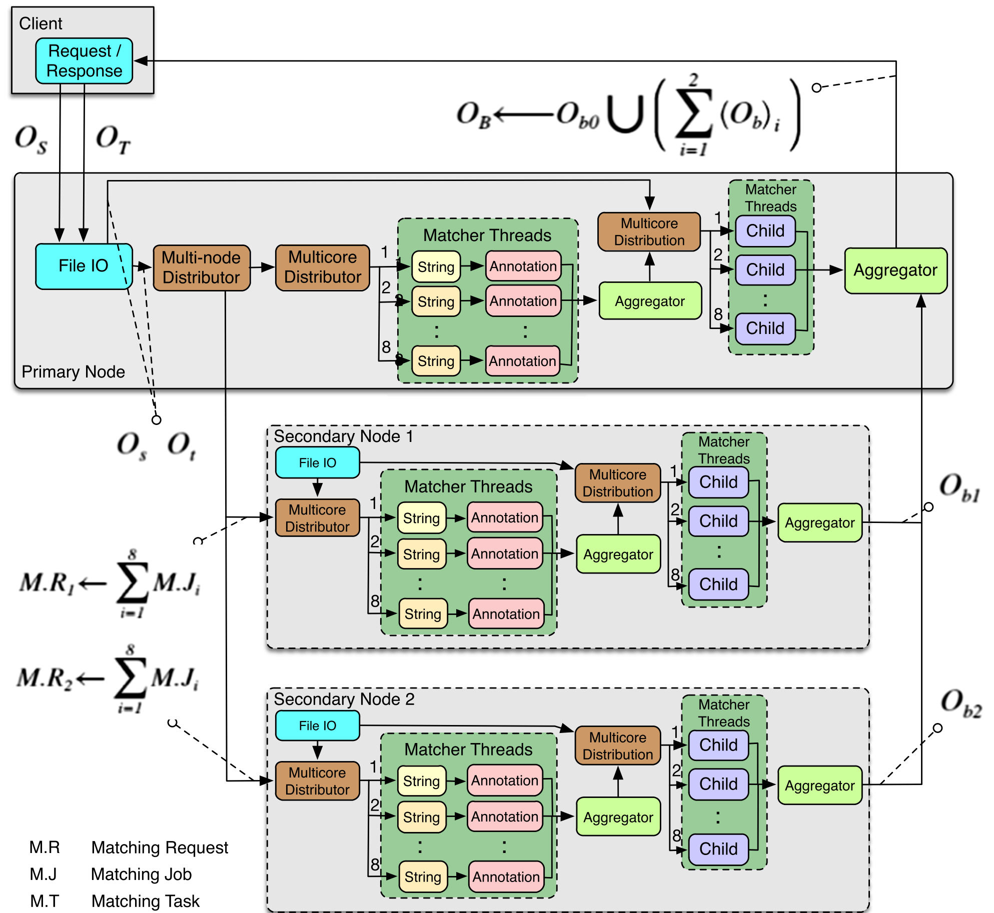


Fig. 17 Parallel flow for large biomedical track over multi-node

with their respective matching requests, all 3 participating nodes load serialized subsets of the whole FMA and NCI required by matching algorithms from their respective ontology repositories. From this point forward, every participating node executes independently, similar to the execution of task 1 until an intermediate bridge ontology is generated by every node (O_{b0}, O_{b1}, and O_{b2}). The aggregators at secondary nodes send their respective intermediate ontologies to the primary node. These bridge ontologies are accumulated by the aggregator at the primary node and finally delivered to the client as the formal mediation bridge ontology (O_B).

Results for this task from both the scenarios (desktop and cloud) are illustrated in Fig. 18. For the multi-node desktop scenario, the sequential process takes around 7 hours to complete the matching request; however, our system completes the matching process within half-an hour over 24 threads with an impressive performance speedup of 14.75 times. The same matching request is executed for the second scenario over the multi-node Azure VM. The

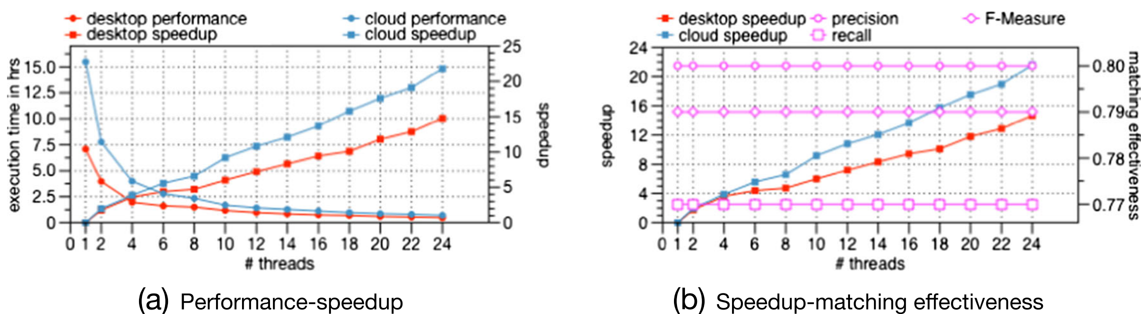


Fig. 18 Results from large biomedical ontologies track, task 2

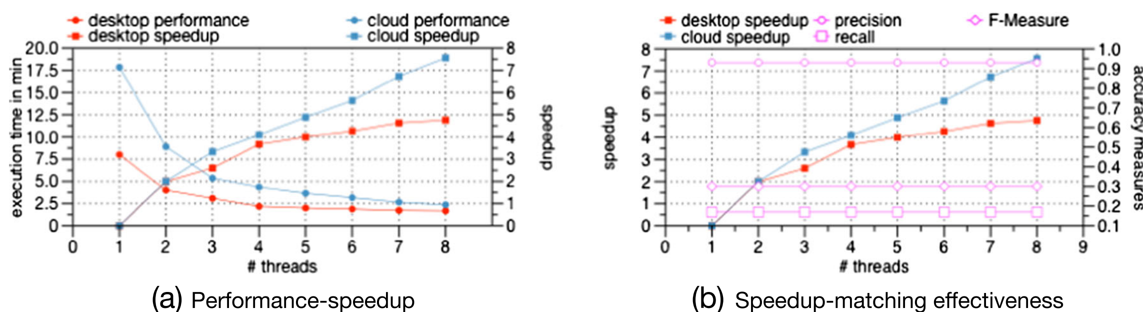


Fig. 19 Results from large biomedical ontologies track, task 3

sequential process over the VM takes 15.5 hours to complete; however, our system completes the whole matching process over 24 threads in slightly over 40 minutes with an impressive speedup of 21.8 times. Furthermore, similar to task 1 the accuracy of the matching process stays preserved with the same effectiveness throughout the performance speedup (illustrated in Fig. 18b).

5.3.3 Task 3: FMA-SNOMED small fragments

This task consists upon matching relatively smaller fragments of FMA and SNOMED ontologies. The FMA fragment consists upon 13 % of whole FMA ontology (10,157 concepts) while the SNOMED fragment consists upon 5 % of whole NCI ontology (13,412 concepts).

Similar to task 1, we have executed our system in both multicore desktop and cloud scenario as a single-node execution. Results from these scenarios are illustrated in Fig. 19. For the single-node desktop scenario, the sequential process takes around 8 minutes to complete the matching request; however, our system completes the matching process in slightly over one and a half minutes over 8 threads with an impressive performance speedup of 4.76 times. Same matching request is executed for the second scenario over the single-node Azure VM. The sequential process over the VM takes around 18 minutes to complete; however, our system completes the whole matching process over 8 threads in slightly less than two and half minutes with an impressive speedup of 7.56 times. Furthermore, similar to previous tasks, the accuracy of the matching process stays preserved with the same effectiveness throughout the performance speedup (illustrated in Fig 19b).

5.3.4 Task 4: FMA whole ontology with SNOMED large fragment

This task consists upon matching the whole FMA ontology with a large fragment of SNOMED ontology. The FMA ontology consists upon 78,989 concepts while the SNOMED fragment consists upon 40 % of SNOMED ontology (122,464 concepts).

Similar to task 2, we have executed our system in both multicore desktop and cloud scenario as multi-node execution. Results from these scenarios are illustrated in Fig. 20. For the multi-node desktop scenario, the sequential process takes about 14 hours to complete the matching request; however, our system completes the matching process in less than an hour over 24 threads with an impressive performance speedup of 15.64 times. Same matching request is executed for the second scenario over the multi-node Azure VM. The sequential process over the VM takes over 26 hours to complete; however, our system completes the whole matching process over 24 threads in slightly over an hour with an impressive speedup of 21 times.

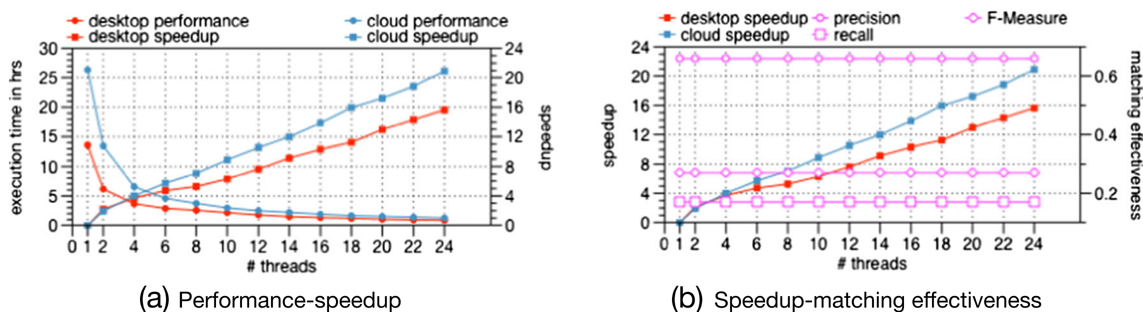


Fig. 20 Results from large biomedical ontologies track, task 4

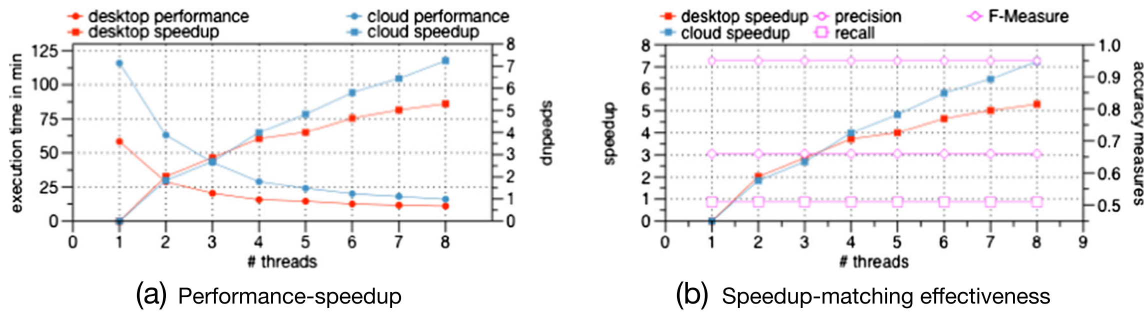


Fig. 21 Results from large biomedical ontologies track, task 5

Furthermore, similar to the previous tasks, the accuracy of the matching process stays preserved with the same effectiveness throughout the performance speedup (illustrated in Fig. 20b).

5.3.5 Task 5: SNOMED-NCI small fragments

This task consists upon matching relatively smaller fragments of SNOMED and NCI ontologies. The SNOMED fragment consists upon 17 % of SNOMED ontology (51,128 concepts), while the NCI fragment consists upon 36 % of whole NCI ontology (23,958 concepts).

Similar to task 1 and 3, we have executed our system in both multicore desktop and cloud scenario as a single-node execution. Results from these scenarios are illustrated in Fig. 21. For the single-node desktop scenario, the sequential process takes around an hour to complete the matching request; however, our system completes the matching process in 11 minutes over 8 threads with an impressive performance speedup of 5.31 times. Same matching request is executed for the second scenario over the single-node Azure VM. The sequential process over the VM takes around 116 minutes to complete; however, our system completes the whole matching process over 8 threads in 16 minutes with an impressive speedup of 7.25 times. Furthermore, similar to previous tasks, the accuracy of the matching process stays preserved with the same effectiveness throughout the performance speedup (illustrated in Fig. 21b).

5.3.6 Task 6: NCI whole ontology with SNOMED large fragment

This task consists upon matching the whole NCI ontology with a large fragment of SNOMED ontology. The NCI ontology consists upon 66,724 concepts while the SNOMED fragment consists upon 40 % of SNOMED ontology (122,464 concepts).

Similar to task 2 and 4, we have executed our system in both multicore desktop and cloud scenario as multi-node execution. Results from these scenarios are illustrated in Fig. 22. For the multi-node desktop scenario, the sequential process takes close to 8 hours to complete the matching request; however, our system completes the matching process in half-an-hour over 24 threads with an impressive performance speedup of 15.19 times. Same matching request is executed for the second scenario over the multi-node Azure VM. The sequential process over the VM takes over 17 hours to complete; however, our system completes the whole matching process over 24 threads in less than an hour with an impressive speedup of 22 times. Furthermore, similar to the previous tasks, the accuracy of the matching process stays preserved with the same effectiveness throughout the performance speedup (illustrated in Fig. 22b).

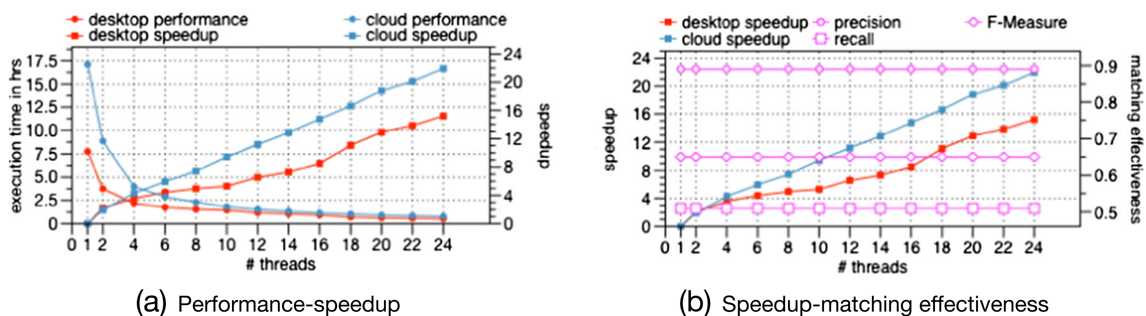


Fig. 22 Results from large biomedical ontologies track, task 6

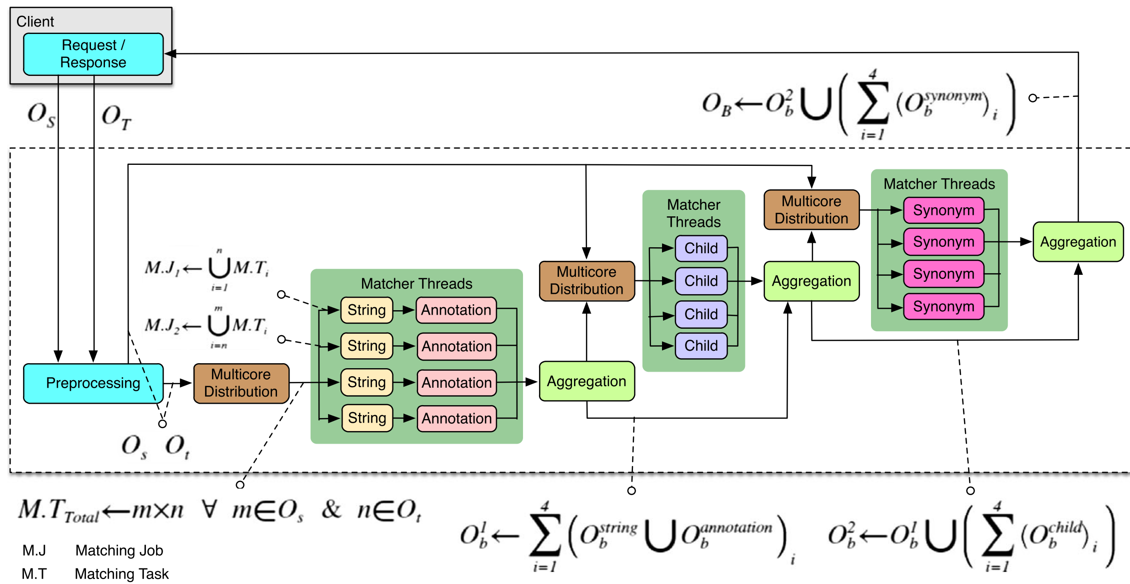


Fig. 23 Parallel flow for conference track over dual core single-node Azure VM

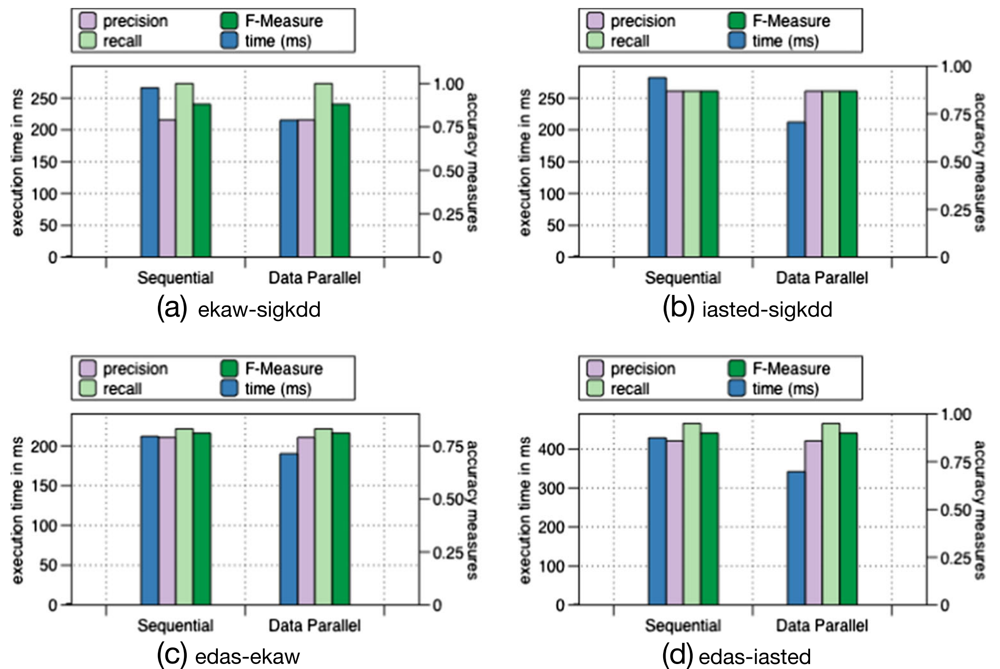
5.4 Conference track

The conference track consists of mapping generation within a collection of ontologies describing the domain of organizing conferences. From trivial string-based correspondence, bridging these ontologies also requires semantic-based matching. Therefore, to generate bridge ontology we have used a matching library with String-based, Annotation-based, Child-based Structural matching, and Synonym-based matching algorithm which utilizes a static dictionary file (illustrated in Fig. 23). Due to the smaller size of these ontologies we have used the A2 (dual core) Azure VM for evaluation. We have executed 12 different mapping tasks on cmt, conference, confOf, edas, ekaw, iasted, and sigkdd ontologies. Results from these tasks are illustrated in Figs. 24 and 25.

5.5 Results summary

In our evaluation, we have used the dataset of real-world ontologies provided by OAEI's 2013 campaign. The key strength of this dataset is its comprehensiveness

Fig. 24 Results from conference track over dual core single-node Azure VM part-I



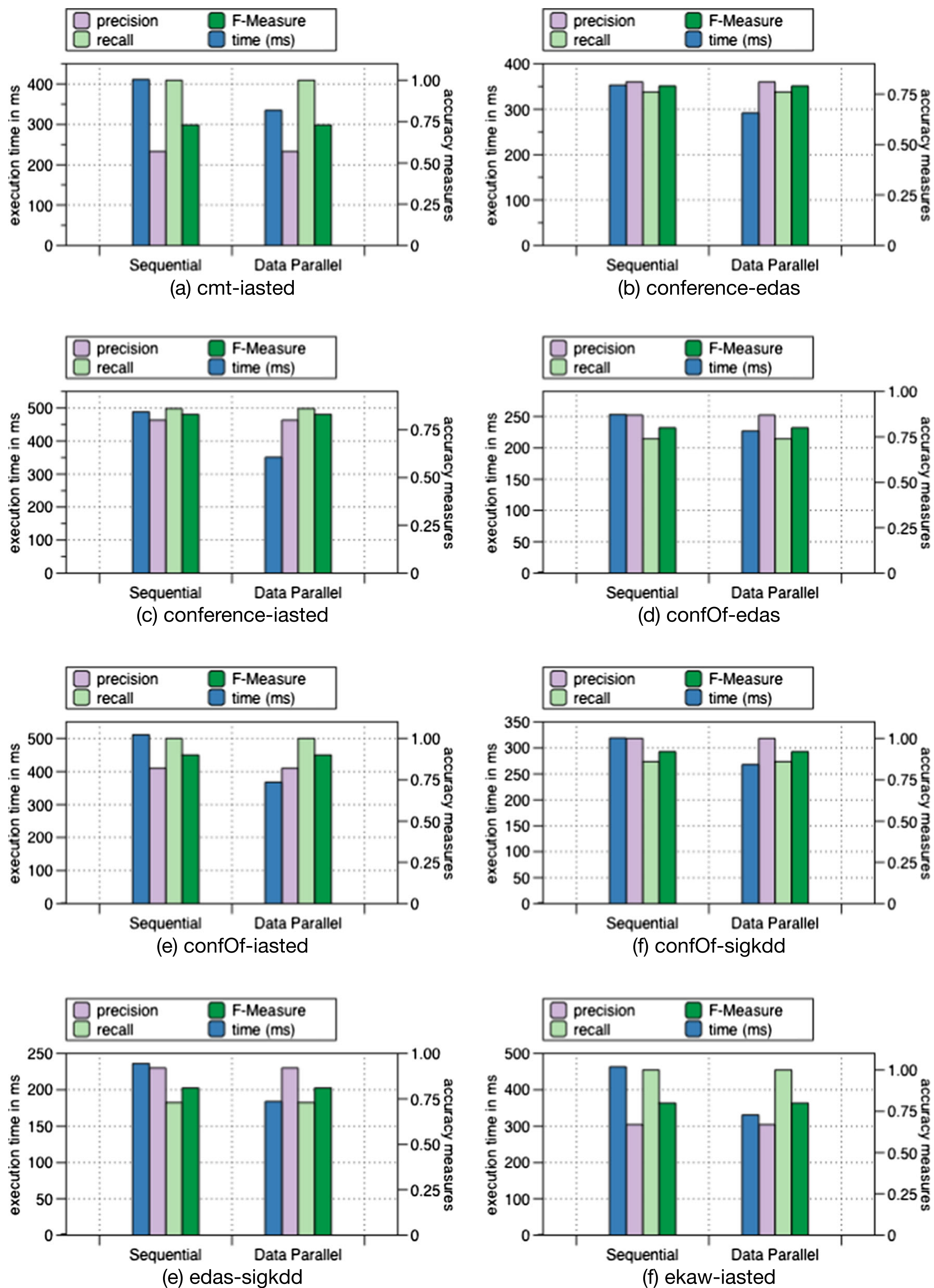


Fig. 25 Results from conference track over dual core single-node Azure VM part-II

that cannot be achieved in datasets comprised of synthetic and custom-built ontologies. The results from the matching problems of OAEI's dataset are summarized in Table 1. These results provide evidence for four major

characteristics of our system, described in the following subsections.

5.5.1 Independent of ontology domain

As stated in the related work section, some of the matching systems are built specific for ontology domains, particularly systems for matching biomedical ontologies. However, the longevity and applicability of an ontology matching system increases with its support to a larger set of ontologies. Therefore, a state-of-the-art ontology matching systems must be independent of ontology domain. The candidateontologies used in the matching problems evaluated by our system are of diverse domains. No change has been inflicted in the structure of the candidate ontologies, yet our system scores impressive performance speedup on all the matching problems. For example, problem of matching library ontologies and small FMA with small NCI ontologies are from different domains of knowledge; furthermore, different matching libraries are used for their mediation. However, due to the ontology subsets generated based on the type of matching algorithms and independent nature of the matching tasks, both of the matching problems score similar performance speedup on the same platform.

Table 1 Result Summary

	Matching problem	Domain	Platform	Speed up	Precision
Small	cmt-iasted	Conference	Single-node Cloud VM	1.22	0.57
	conference-edas		Single-node Cloud VM	1.25	0.81
	conference-iasted		Single-node Cloud VM	1.39	0.80
	confof-edas		Single-node Cloud VM	1.11	0.87
	confof-iasted		Single-node Cloud VM	1.38	0.82
	confof-sigkdd		Single-node Cloud VM	1.19	1.00
	edas-sigkdd		Single-node Cloud VM	1.28	0.92
	ekaw-iasted		Single-node Cloud VM	1.39	0.67
	ekaw-sigkdd		Single-node Cloud VM	1.23	0.79
	iasted-sigkdd		Single-node Cloud VM	1.33	0.87
	edas-ekaw		Single-node Cloud VM	1.11	0.79
	edas-iasted		Single-node Cloud VM	1.25	0.86
Medium	human-mouse	Anatomy	Single-node Desktop	4.05	0.99
			Single-node Cloud VM	5.56	0.99
	STW-TheSoz	Library	Single-node Desktop	4.15	0.67
			Single-node Cloud VM	6.38	0.67
	FMA_s-NCI_s	Biomedical	Single-node Desktop	4.27	0.95
			Single-node Cloud VM	6.53	0.95
Large	FMA_w-$SNOMED_s$		Single-node Desktop	4.76	0.93
			Single-node Cloud VM	7.56	0.93
	NCI_w-$SNOMED_s$		Single-node Desktop	5.31	0.95
			Single-node Cloud VM	7.25	0.95
Very Large	FMA_w-NCI_w		Multi-node Desktop	14.75	0.80
			Multi-node Cloud VM	21.80	0.80
	FMA_w-$SNOMED_l$		Multi-node Desktop	15.64	0.66
			Multi-node Cloud VM	20.91	0.66
	NCI_w-$SNOMED_l$		Multi-node Desktop	15.19	0.89
			Multi-node Cloud VM	21.93	0.89

5.5.2 Performance-based ontology matching over various size of matching problems

As described in Table 1, we can classify the matching problems in four categories: (i) small, containing conference ontologies track; (ii) medium, containing anatomy, library, and task 1 (FMA with NCI small fragments) from large biomedical ontologies track; (iii) large, containing task 3 (FMA-SNOMED small fragments) and 5 (SNOMED-NCI small fragments) from large biomedical ontologies track and (iv) very large, containing task 2 (FMA-NCI whole ontologies), 4 (FMA whole ontology with SNOMED large fragment), and 6 (NCI whole ontology with SNOMED large fragment) from large biomedical track. The average speedup by a category is illustrated in Fig. 26. It is quite evident from the figure that our system is more beneficial to the ontology matching problems with a medium to large and very large sizes. Results for the small category containing conference track are obtained from a dual core Azure VM. Although the sequential process does complete the matching process quite efficiently due to the small nature of the matching problem, yet our system was able to improve the performance by an average speedup of 1.25 times (≈ 20 % more efficient than sequential matching process).

The medium category was evaluated on a quad-core Hyper-Threaded desktop and an Azure VM with 8 cores. The average performance speedup is 4.1 and 5.9 times on desktop and cloud respectively. Comparing these results to the average speedup of medium-scale problem of [16], even with an inferior hardware our system outperforms the intra-matcher by 5 % on the desktop and 51 % over cloud platform.

The large category was also evaluated on a quad-core Hyper-Threaded desktop and an Azure VM with 8 cores. The average performance speedup is 5.0 and 7.4 on desktop and cloud respectively. Comparing these results to the average speedup of the large-scale problem of [16] on a single node, our system outperforms the intra-matcher by 5.2 % on desktop and 55 % over the cloud platform.

For the very large category, average speedup has been calculated over single-node (8 cores), and multi-node (16 and 24 cores). On a single-node the results are quite similar to single-node large category, i.e., 4.97 and 7.02 times on desktop and cloud respectively. Our system outperforms the intra-matcher of [16] by 4.6 % on desktop and 47.78 % over cloud platform. In case of multi-node platform with dual nodes (8 cores each), our system completes the matching process with average speedup of 9.42 and 14.1 on desktop and cloud respectively. Comparing these results with Intra&Inter multi-node matcher of [16] over 16 cores, our system running over Azure VMs outperforms Intra&Inter matcher by 12.8 %. Scaling the same matching problem to 3 nodes (8 cores each), our system completes the matching process with average speedup of 15.16 times on a desktop and 21.51 times over cloud platform.

5.5.3 Effectiveness-independent performance-gain

As described earlier, ontology matching systems developed over the years have taken performance into consideration; however, it is tightly coupled with the effectiveness of their matching algorithms. On the other hand, methodology proposed by our system extracts performance-gain without inflicting any changes in the accuracy of the matching algorithm. From the results, it is clear that the accuracy

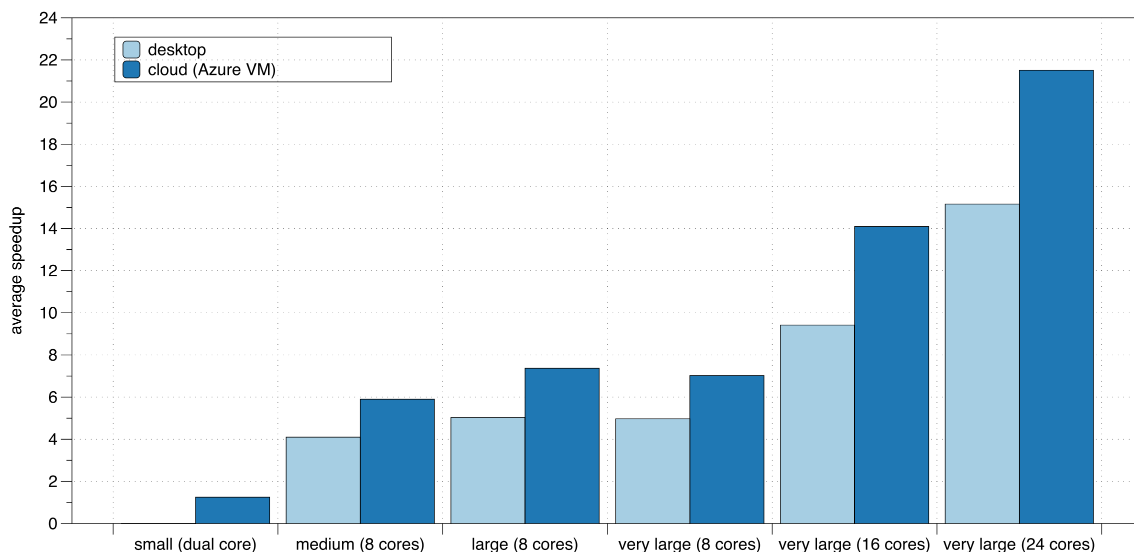


Fig. 26 Results summary

of the matched results remains preserved even when scaling up to multiple cores for parallel matching. In all the performed evaluations, the effectiveness measures remain constant even with substantial gain in performance.

5.5.4 Matching library interface

To implement effectiveness-independent performance-gain, distributor components of our system, decouples the matching library from the performance runtime with the help of a matching library interface. This approach provides an additional benefit of plug-n-play matching algorithms and libraries. In our evaluation, we have used three different ontology matching libraries with different accuracy measures, provided to us by different semantic web experts. For anatomy and library matching problem, same matching library of String, Label, and Child-based algorithms is used. For large-biomedical tracks, a matching library with String, Annotation, and Child-based algorithms is used. For conference matching problems, another library with four matching algorithms, i.e., String, Annotation, Child, and Synonym-based matching algorithm is used. This characteristic of our system provides an exclusive performance-based ontology matching runtime that can host and execute matching algorithms and libraries, developed by semantic web experts without worries of accuracy loss or platform-level maintenance.

6 Conclusion

In this paper, we presented our performance-based ontology matching system which implements effectiveness-independent data-parallel approach for matching. Ontology matching is a widely used technique for heterogeneity resolution among information and knowledge-based systems; however, size, complexity, and availability of these ontologies require solutions that are built from a performance aspect. With the availability of affordable parallelism-enabled multicore platforms like desktop and cloud, our system is built to exploit their performance benefits by data parallelism for ontology matching.

Our system divides the whole matching process into three stages, i.e., pre-matching, parallel matching, and post-matching, with each stage designed to contribute in the overall performance aspect of ontology matching. By default, ontologies are not scalable structures; therefore, pre-matching stage converts the candidate ontologies into smaller, simpler, and scalable resource-based ontology subsets, based on the requirements of matching algorithms. This method provides the resolution to the scalability challenge of ontology matching by providing ontology subsets that are distribution friendly. Furthermore, due to the smaller size, independence and scalable nature of these subsets, accessing ontology resources is significantly faster than loading directly from the ontology files. We have recorded 8 times faster ontology resource loading with 4 times smaller memory footprint working with ontology subsets instead of whole ontologies. These subsets are also serialized and persisted by our system for reuse. Moreover, pre-matching stage also acts as an adapter for ontologies to be plugged into our system. No change is inflicted in the original structure of candidate ontologies to make them compatible with parallel matching.

In the parallel-matching stage, ontology subsets generated by the pre-matching are further used by distributor components for implementing the three-layer distribution abstraction. This abstraction constitutes upon independent matching requests generated for each participating node by multi-node distributor, and matching jobs and matching tasks generated by multicore distributor for participating cores per node. These abstractions are independent in nature and provide the foundation for data-parallel ontology matching. This method of distributing the matching process from grainer level matching request to finer level matching tasks provides a way to improve the performance of ontology matching. Consequently, over parallelism-enabled platforms we have recorded a performance speedup of 4.1 to 7.5 times on single-node multicore platforms and up to 21.5 times on multi-node platforms. Furthermore, distribution components provide the interface to matching libraries and algorithms. Matching tasks are assigned with instances of matching algorithms to be executed at runtime with no change inflicted in the implementation of the algorithm. This method decouples the performance aspects of ontology matching from accuracy, providing an effectiveness-independent approach. We have recorded no change in the accuracy measures while scaling up the matching process for data-parallel matching. Matched results from matched tasks distributed over computing resources are aggregated to generate the required mappings as mediation bridge ontology by the post-matching stage.

To further contribute in overall performance of the ontology matching process, our system also aligns the execution of matching algorithms such that the matching space of every following algorithm execution gets minimized. This method speeds up the matching process by only matching the unmatched ontology resources; consequently, avoiding redundant matching operations.

For benchmarking of our system, we have used OAEI's real-world ontology dataset. The evaluation tracks and their tasks provided by OAEI's semantic web experts are specifically designed to evaluate the state-of-the-art ontology matching systems. This dataset includes fourteen ontologies from diverse domains, different sizes, and complexities.

Evaluation on such a diverse dataset of ontologies have validated the generic nature of our system, i.e., performance-based ontology matching process executes regardless of the type and scope of candidate ontologies. Furthermore, matching problems for evaluation are classified into different sizes, varying from small to very large-scale ontologies. Although our system provides a small performance speedup (1.25 times) on smaller ontologies however, it shows impressive performance-gain where it matters the most, i.e., in solving medium to large-scale ontology matching problems. For medium scale ontology matching problems, the average performance speedup is 4.1 and 5.9 times over single-node desktop and Microsoft Azure VM respectively. For large-scale ontology matching problems, the average performance speed is 5.0 and 7.4 times over single-node desktop and Azure VM respectively. For very-large-scale ontology matching problems, the average performance speedup is 15.16 and 21.51 times over multi-node desktop and Azure cloud platform. For further evaluation, our system is currently deployed over Microsoft Azure public cloud environment and being used by a running instance of a Clinical Decision Support System (CDSS). The interoperability engine of the CDSS is using our systems performance-based ontology matching for finding mappings between biomedical ontologies.

From the recorded results drawn by our system working with real-world ontologies, it is apparent that our approach offers a comprehensive solution to the performance challenges of ontology matching problems. Moreover, our system is generic, effectiveness-independent, and aligned with the use of new generation computing platforms. Due to the extensive use of ontologies, the size and complexity of ontology matching problems will increase. From the results, it is evident that our system performs impressively well on medium to very large-scale ontology matching problems. Thus, our proposed system has the required longevity for future ontology matching problems. We have on-going research in the area of performance-based ontology matching, with the proposed system as a research outcome. We have on-going research in the area of performance-based ontology matching, with the proposed system as a research outcome. Although our system scores impressively during evaluation, there are few limitations of the current implementation. Apparently our system presumes the multi-node environment to be homogenous, which might not stay true in the longer run as heterogeneous computing environments are becoming available with the excessive use of cloud computing. Furthermore, relationship needs to be identified between the size of the matching problem and acquisition of computing resources, such that optimal distribution slab for matching tasks can be identified automatically. These two important aspects are among the scope of our future work.

From the application and usability perspective of our system, it can greatly benefit semantic web experts, researchers, and dynamic systems, which rely on ontology matching to provide heterogeneity resolution. Due to the computational complexity and increasing size of these ontologies, a client has to wait for in-time results. Our system provides a resolution to these clients by performing matching operations in parallel over affordable platforms for fast results. Our system is built to scale from multi-core desktops PCs to ubiquitous and affordable distributed multi-node platforms like clouds for better performance. Furthermore, semantic-web experts who are focused on building matching algorithms can integrate their encapsulated algorithms and benefit from the parallel execution without writing any parallelism code with-in or to complement the matching algorithms. Moreover, due to the effectiveness-independent data-parallel approach of our system, these experts do not have to worry about any accuracy loss with performance speedup.

Acknowledgments This work was supported by a post-doctoral fellowship grant from the Kyung Hee University Korea in 2011 (KHU-20110219).

References

1. Doan A, Halevy A, Ives Z (2012) Principles of data integration. Addision-Wesley, Reading
2. Euzenat J, Shvaiko P (2013) Ontology Matching, 2nd Edn. Springer, Berlin
3. Isern D, Sanchez D, Moreno A (2012) Ontology-driven execution of clinical guidelines. Comput Methods Prog Biomed 107:122–139
4. Cimino J, Zhu X (2006) IMIA Yearbook of Medical 1:124–135
5. De Potter P, Cools H, Depraetere K, Mels G, Debevere P, De Roo J, Huszka C, Colaert D, Mannens E, Van de Walle R (2012) Semantic patient information aggregation and medicinal decision support. Comput Methods Prog Biomed 2:724–735
6. Gene Ontology Consortium (2004) The Gene Ontology (GO) database and informatics resource, Nucleic Acid Research, Database issue, 32
7. Golbeck J, Fragoso G, Hartel F, Hendler J, Oberthaler J, Parsia B (2003) The National Cancer Institute's Thesaurus and Ontology, Web Semantics: Science, Services and Agents on the World Wide Web, 1
8. (2003) A reference ontology for biomedical informatics: the Foundational Model of Anatomy, Journal of Biomedical Informatics, vol 36, pp 478–500
9. Schulz S, Cornet R, Spackman K (2011) Consolidating SNOMED CT's ontological commitment. Appl Ontol 1:1–11
10. Smith B, Ashburner M, Rosse C, Bard J, Bug W, Ceusters W, Goldberg LJ, Eilbeck K, Ireland A, Mungall CJ, Consortium OBI, Leontis N, Rocca-Serra P, Ruttenberg A, Sansone SA, Scheuermann RH, Shah N, Whetzel PL, Lewis S (2007) The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. Nat Biotech 25:1251–1255
11. Whetzel PL, Noy NF, Shah NH, Alexander PR, Nyulas C, Tudorache T, Musen MA (2011) BioPortal: enhanced

functionality via new Web services from the National Center for Biomedical Ontology to access and use ontologies in software applications, Nucleic Acids Res 2011 Jul;39(Web Server issue):W541-5. Epub
12. Algergawy A, Siegmund N, Saake G (2010) Combining schema and level-based matching for web service discovery. In: Proceedings of the 10th international conference of web engineering ICWE
13. Fensel D, Lausen H, Polleres A, de Bruijn J, Stollberg M, Roman D, Domingue J (2006) Enabling semantic web services: the web service modeling ontology. Springer, Heidelberg
14. Shvaiko P., Euzenat J. (2013) Ontology matching: state of the art and future challenges. IEEE Trans Knowl Data Eng 25:158–176
15. van Hage WR, Katrenko S, Schreiber G (2005) A method to combine linguistic ontology-mapping techniques, the semantic web ISWC 2005, pp 732–744. Springer, Berlin
16. Gross A, Hartung M, Kirsten T, Rahm E (2010) On matching large life science ontologies in parallel. Springer, Berlin Heidelberg
17. LeBlanc T. J., Friedberg S. A. (1985) HPC: a model of structure and change in distributed systems. IEEE Trans Comput C-34:1114–1129
18. Han S., Choi HG (2013) Investigation of the parallel efficiency of a PC cluster for the simulation of a CFD problem. Pers Ubiquit Comput:1–12
19. Buyyaa R, Yeoa CS, Venugopala S, Broberga J, Brandicc I (2009) Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comput Syst 25:599–616
20. Tenschert A, Assel M, Cheptsov A, Gallizo G, Della Valle E, Celino I (2009) Parallelization and distribution techniques for ontology matching in urban computing environments. In: Proceedings of the 4th international workshop on ontology matching (OM-2009) collocated with the 8th international semantic web conference (ISWC-2009) Chantilly, USA, October 25, 2009, volume 551 of CEUR Workshop Proceedings, CEUR-WS.org
21. Andrade D, Fraguela BB, Brodman J, Padua D (2009) Task-parallel versus data-parallel library-based programming in multicore systems, 2009 17th euromicro international conference on parallel, distributed and network-based processing, pp 101–110
22. Chen W-Y, Song Y, Bai H, Lin C-J, Chang EY (2011) Parallel spectral clustering in distributed systems. IEEE Trans Pattern Anal Mach Intell 33(3):568–586
23. Intel Developer Zone (2011) Choose the right threading model (task-parallel or data-parallel threading)
24. 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
25. Yves R, Jean-Mary E, Shironoshita P, Kabuka MR (2009) Ontology matching with semantic verification. Web Semant 7:235–251
26. Wei H, Yuzhong Q (2008) Falcon-AO: a practical ontology matching system. Web Semant 6:237–239
27. Wei H, Yuzhong Q, Cheng G (2008) Matching large ontologies: a divide-and-conquer approach. Data & Knowl Eng 67:140–160
28. Hanif Seddiqui Md, Aono M (2009) An efficient and scalable algorithm for segmented alignment of ontologies of arbitrary size. Web Semant Sci Serv Agents World Wide Web 7:344–356
29. Garruzzo S, Rosaci D (2008) Agent clustering based on semantic negotiation. ACM Trans Auton Adapt Syst 3:7:1–7:40
30. De Meo P, Quattrone G, Rosaci D, Ursino D (2012) Bilateral semantic negotiation: a decentralised approach to ontology enrichment in open multiagent systems. Int J Data Model Manag 4:1–38
31. Garruzzo S, Rosaci D (2006) Information agents that learn to understand each other via semantic negotiation. Distrib Appl Interoperable Syst 4025:99–112
32. Garruzzo S, Rosaci D (2006) HISENE2: a reputation-based protocol for supporting semantic negotiation. Move Meaningful Internet Syst 2006: CoopIS, DOA, GADA, and ODBASE 4275:949–966
33. Caire G (2007) (TILAB, formerly CSELT), JADE TUTORIAL, JADE PROGRAMMING For BEGINNERS. http://www.cs.uu.nl/docs/vakken/map/JADEProgramming-Tutorial-for-beginners.pdf
34. Cruz If, Antonelli FP, Stroe C (2009) AgreementMaker: efficient matching for large real-world schemas and ontologies. Proc VLDB Endow 2:1586–1589
35. enez-Ruiz EJ, Grau BC (2011) LogMap: logic-based and scalable ontology matching. Semant Web ISWC 7031:273–288
36. Kirsten T, Kolb L, Hartung M, Gross A, Köpcke H, Rahm E (2010) Data partitioning for parallel entity matching, 8th international workshop on quality in databases
37. Ernesto J-R, Meilicke C, Grau BC, Horrocks I (2013) Evaluating mapping repair systems with large biomedical ontologies, 26th international workshop on description logics. Springer LNCS, Berlin
38. Lambrix P, Tan H (2006) SAMBO-A system for aligning and merging biomedical ontologies. Web Semant 4:196–206
39. What is WordNet? (2013), Princeton University
40. National Center for Biotechnology Information, U.S. National Library of Medicine, PubMed, 2013
41. Takai-Igarashi T, Takagi T (2000) SIGNAL-ONTOLOGY: ontology for cell signaling. Genome Inform 11:440–441
42. U.S. National Library of Medicine (2013) National Institute of Health, Medical Subject Headings
43. Hayamizu TF, Mangan M, Corradi JP, Kadin JA, Ringwald M (2005) The adult mouse anatomical dictionary: a tool for annotating and integrating data. Genome Biol 6:1–8
44. Lambrix P, Tan H, Liu Q (2008) SAMBO and SAMBOdtf results for the ontology alignment evaluation initiative 2008. CEUR Workshop Proc 431:1114–1129
45. Zhang S, Bodenreider O (2007) Hybrid alignment strategy for anatomical ontologies: results of the 2007 ontology alignment contest. CEUR Workshop Proc:304
46. Ba M, Diallo G (2011) Large-scale biomedical ontology matching with ServOMap. IRBM 34:56–59
47. HDFS Architecture Guide. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
48. Krishnaswamy A (2013) To hadoop or not to hadoop? http://www.thoughtworks.com/insights/blog/hadoop-or-not-hadoop
49. Matsunaga A, Tsugawa M, Fortes J (2008) CloudBLAST: combining MapReduce and virtualization on distributed resources for bioinformatics applications, fourth IEEE international conference on eScience
50. Reasoning-Hadoop. http://www.jacopourbani.it/reasoning-hadoop.html
51. Heart Project. http://rdf-proj.blogspot.kr/
52. Hadoop Distributed RDF Store. https://code.google.com/p/hdrs/
53. Flynn's Taxonomy. http://en.wikipedia.org/wiki/Flynn's_taxonomy
54. Park M-J, Lee J, Lee C-H, Lin J, Serres O, Chung C-W (2007) An efficient and scalable management of ontology. In: Proceedings of the 12th international conference on database systems for advanced applications. Springer, Berlin
55. Zhao G, Meersman R (2005) Architecting ontology for scalability and versatility. In: Proceedings of the 2005 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, COA, and ODBASE - Volume Part II. Springer, Berlin
56. Zhou J, Ma L, Liu Q, Zhang L, Yu Y, Pan Y (2006) Minerva: a scalable OWL ontology storage and inference system, ASWC
57. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston

58. Intel Corporation (2013) Intel hyper-threading technology
59. Adult Mouse Anatomy. http://www.informatics.jax.org/searches/AMA_form.shtml
60. STW Thesaurus of Economics Ontology. http://zbw.eu/stw/versions/8.10/descriptor/29234-2/about.en.html
61. Thesaurus for the Social Sciences. http://www.gesis.org/en/services/research/thesauri-und-klassifikationen/social-science-thesaurus/

Muhammad Bilal Amin received his M.S. from DePaul University, Chicago, USA in 2006. He is currently a PhD. scholar at Ubiquitous Computing Lab, Department of Computer Engineering, Kyung Hee University, South Korea. He has a working experience of more than 10 years in software industry, working for Fortune 500 companies in USA. His research interest include, cloud computing, parallel programming, distributed systems, software architecture, semantic web, and performance-based ontology matching.

Wajahat Ali Khan received his B.S. in Information Technology with distinction from Kohat University of Science and Technology, Pakistan in 2007. He got M.S. in Information Technology from National University of Science and Technology, Pakistan in 2010. He got PhD in Computer Engineering from Kyung Hee University in 2015. He has been working as Postdoctoral Research Scientist in Ubiquitous Computing lab since 2015. His research interests include health-care interoperability, ontology mapping, semantic reasoning, and semantic web services.

Sungyoung Lee received his B.S. from Korea University, Seoul, Korea. He got his M.S. and Ph.D. degrees in Computer Science from Illinois Institute of Technology (IIT), Chicago, Illinois, USA in 1987 and 1991 respectively. He has been a professor in the Department of Computer Engineering, Kyung Hee University, Korea since 1993. He is a founding director of the Ubiquitous Computing Laboratory, and has been affiliated with a director of Neo Medical ubiquitous-Life Care Information Technology Research Center, Kyung Hee University since 2006. He is a member of ACM and IEEE.

Byeong Ho Kang received his B.S. from Pusan University, Pusan, Korea in 1988. He got his M.S degrees in Computer Science from University of Tasmania, Australia in 1990 and his Ph.D degrees in Computer Science from University of New South Wales, Australia in 1995. From 1996 to 1999, he has been an Assistant professor at School of Computer science, Hoseo University, Korea. Since 2008, he has been an Associate professor at School of Information and Communication Technology, University of Tasmania, Australia. His research interests include expert systems, artificial intelligence, knowledge acquisition, and Internet applications.