Thesis for the Degree of Doctor of Philosophy

MULTI-DIMENSIONAL PERFORMANCE-BASED ONTOLOGY MATCHING OVER PARALLEL PLATFORMS

Muhammad Bilal Amin

Department of Computer Engineering
Graduate School
Kyung Hee University
South Korea

August 2015

MULTI-DIMENSIONAL PERFORMANCE-BASED ONTOLOGY MATCHING OVER PARALLEL PLATFORMS

Muhammad Bilal Amin

Department of Computer Engineering
Graduate School
Kyung Hee University
South Korea

August 2015

Multi-Dimensional Performance-based Ontology Matching over Parallel Platforms

by

Muhammad Bilal Amin

Supervised by

Prof. Sungyoung Lee

Submitted to the Department of Computer Engineering and the Faculty of Graduate School of Kyung Hee University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Dissertation Committee:

Prof. Young-Koo Lee

Prof. Eui-Nam Huh

Prof. Hyonwoo Seung

Prof. Jinsung Cho

Prof. Sungyoung Lee

For the dreams of my father,
the efforts of my mother,
the trust of my wife,
and
the love of my daughter,

I succeed, I prevail, I achieve.

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, problems regarding information heterogeneity have emerged. Although, resolutions for data heterogeneity are quite trivial but in the case of semantic heterogeneity, resolutions involve data's intend, making the integration a challenging opportunity. The volume of data makes manual annotation of concepts unrealistic; consequently, automated solutions based on ontologies are used by software agents. The most prominent solution for semantic heterogeneity resolution is ontology matching; however, adjacent to the abundance of information, the ontologies representing these resources are also becoming large-scale and complex, leading to performance bottlenecks during the matching process.

Over the years, ontology matching systems and techniques have taken performance aspects of ontology matching into consideration and proposed various resolutions. However, these resolutions are matching effectiveness centric, i.e., accuracy of the matching algorithms. The performance aspect of matching these ontologies is concentrated on optimization of the matching algorithms and partitioning of larger ontologies into smaller chunks for performance benefits. Due to the trade-off between performance and effectiveness of matching algorithms (accuracy measures, precision, recall, and F-Measure), ontology-matching problem can go to a certain extent in gaining performance by optimizing only the matching algorithms. Furthermore, the performance improvement based-on exploitation of newer hardware technologies has largely been missed. Among these technologies are affordable parallelism-enabled systems, which are easily available as stand-alone (desktop) and distributed platforms (cloud).

In contrary to current state-of-the-art implementations for ontology matching, work presented

in this thesis regards with a multi-dimensional methodology to achieve performance-gain during the ontology matching process. The proposed methodology identifies candidate performance bottlenecks from end-to-end in the ontology matching process and presents resolutions without impacting the accuracy aspects of the matching algorithms. Thus, preserving the accuracy of the whole matching process in parallel to performance-gain. Presented methodology contributes from four dimensions towards the performance aspect of ontology matching: (i) Memory space, the memory stress on the matching environment. For resolution proposed methodology decomposes complex ontologies into smaller, simpler, and scalable subsets depending upon the needs of matching algorithms. Accessing ontology resources from these subsets in following stages is significantly faster due to their smaller size, independent nature, and data structures that can be readily partitioned for parallel and distributed matching. This approach effectively contributes to overall performance-gain especially when matching large-scale ontologies. Ontology subsets are serialized and cached in repositories, preventing the re-generation of ontology subsets of already serialized ontologies for future matching requests. Furthermore, execution of matching algorithms is aligned for the minimization of the matching space during the matching process. This method contributes to the performance by reducing the number of matching tasks to unmatched resources only, thus avoiding redundant expensive matching operations for subsequent algorithm executions; (ii) Execution time, effectiveness-independent performance-gain by parallel and distributed matching. Matching process over the candidate ontology 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. 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 systems performance by distributing matching tasks over participating computing cores and executing them in parallel at finer level with optimal computing resource utilization; (iii) Performance-based ontology matching runtime, the proposed methodology has been implemented as a performance-based parallel and distributed ontology matching runtime decoupled from ontology matching libraries;

(iv) service and a platform, non-monolithic ontology matching runtime is deployed as an ontology matching service and a platform with interface for matching algorithm libraries. Service is used by the heterogeneity resolution clients for ontology matching and performance-based ontology matching platform is used by semantic-web experts to evaluate their matching algorithms over parallel platforms.

Proposed methodology presented in this thesis has been evaluated comprehensively over ontology datasets of real-world ontology matching problems of diverse scope, complexity, and sizes provided by Ontology Alignment Evaluation Initiative (OAEI). Furthermore, an ontology matching runtime built on the proposed methodology has also been evaluated by OAEI campaign. Results from these evaluations have shown a substantial performance-gain achieved with effectiveness-independence by the whole ontology matching process by adopting the proposed methodology. Moreover, the implemented runtime has already been in production as a service, where it is utilized by semantic web experts for heterogeneity resolution.

Acknowledgement

First and foremost, I would like to thank the **Almighty Allah** for showering His blessings on me and my family. He gave me patience, strength, and courage during my doctoral study.

I would especially like to thank my advisor, Respected Professor Sungyoung Lee for providing me the opportunity to pursue my education further and become a Ph.D. His guidance, support, and encouragement has been the key to my success and achievements. Above of all, his kindness towards me as a father-figure is what I cherish the most. I am nothing but grateful.

I am grateful to Respected Professor Byeong Ho Kang from the University of Tasmania, Australia. He has been a huge source of guidance and ideas for me. His critique and evaluation have always kept me on track, resulting in the completion of my objectives.

I am also grateful to my dissertation evaluation committee (Prof. Jinsung Cho, Prof. Euinam huh, and Prof. Young-Koo Lee from KHU and Prof. Hyonwoo Seung from Seoul Women's University) for their time and valuable suggestions. Their input has been a great source of improvement not only in my dissertation but also in my skill-set.

I would like to thank Mrs. Kim for being available and helping me in many regards. Even though she has always been extremely busy, but always been kind when asked for help.

I would like to acknowledge my colleagues and seniors, especially Dr. Asad Masood Khattak and Dr. Zeeshan Pervez for their guidance and critique. Dr. Wajahat Ali Khan, Mr. Shujaat Hussain, and Mr. Mahmood for their help, support, and availability. Their kindness truly means a lot to me. I would also like to thank our UCLab team members, Dr. Banos, Dr. Han, Dr. Fahim, Mr. Muhammad Idris, Mr. Bilal Rizvi, Mr. Bui Dinh Mao, Mr. Maqbool Hussain, Mr. Muhammad Afzal, Mr. Taqdir Ali, Mr. Jamil Hussain, Mr. Rahman Ali, Mr. Tae Ho Hur, Mr. Jae Hun Bang, Mr. Byung Gill Go, and Mr. Hameed Siddiqi for their support. Working with them has been noth-

ing but pleasurable. I would also like to thank Mr. Omar Farooq from Biomedical Engineering, Dr. Waqas Nawaz and Mr. Kifayat ullah Khan from DKE Lab, Dr. Muhammad Aazam from ICNS Lab, Dr. Amjad Ali from Electronics and Communication Engineering, Mr. Dildar Hussain from Biomedical Engineering, and Dr. Muhammad Rizwan from Nuclear Engineering for giving me moments to enjoy and feel blessed.

In the end, I would like to express my sincere gratitude to my family. My brother Farrukh, my sister Nida and her husband Aitisam for taking care of our parents in my absence, something I have been missing greatly. My in-laws for their prayers and support. Two strong women of my life, my mother and my wife Rabia. One who brought me up to what I am and the other one who ensures that I stay on track towards our reach. Without their prayers, patience, and support this would not be possible. My daughter Zuhaa for just being there in my life, there is nothing more pleasurable and priceless than her presence. Finally, my father, who deserves my title more than me. There is no one I have more regards in my life than him. I wish i could be as selfless and kind as you, the rarest of the qualities a person can have. I hope to become a father like you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Overview

In the era of globalization and automation, integration of information has become a key tool for providing knowledge driven services. For automated knowledge aggregation, integration of data and information from heterogeneous sources is the key [1]. With the abundance of available information over ubiquitous platforms, contributed by various domains using various input devices has substantially increased the amount of disparate information; consequently, heterogeneity issues have emerged. The heterogeneity is classified into two types: data heterogeneity and semantic heterogeneity [2]. Data heterogeneity has solutions based on data definitions, types, formats, and precision [2]. Tools like Microsoft Biztalk Server [3] are also used for data integration and heterogeneity resolution as described in [4] and [5]. Semantic heterogeneity; however, involves data's intend [2], making it a challenging opportunity for integration [1]. The volume of data, makes manual annotation of concepts unrealistic; consequently, automated solutions based on ontologies are used by software agents [6]. The primary solution for semantic heterogeneity problem is ontology matching. It determines correspondence between semantically related ontologies. Concept behind ontology matching process is illustrated in Fig 1.1, where heterogenous data resources are annotated by ontologies and later matched by libraries of complex ontology matching algorithms ranging from entity-based matching to relationship-based matching for correspondence determination [7]. This correspondence is termed as mappings or alignment, whereas the encapsulation of these mappings is represented as a bridge ontology [8]. 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

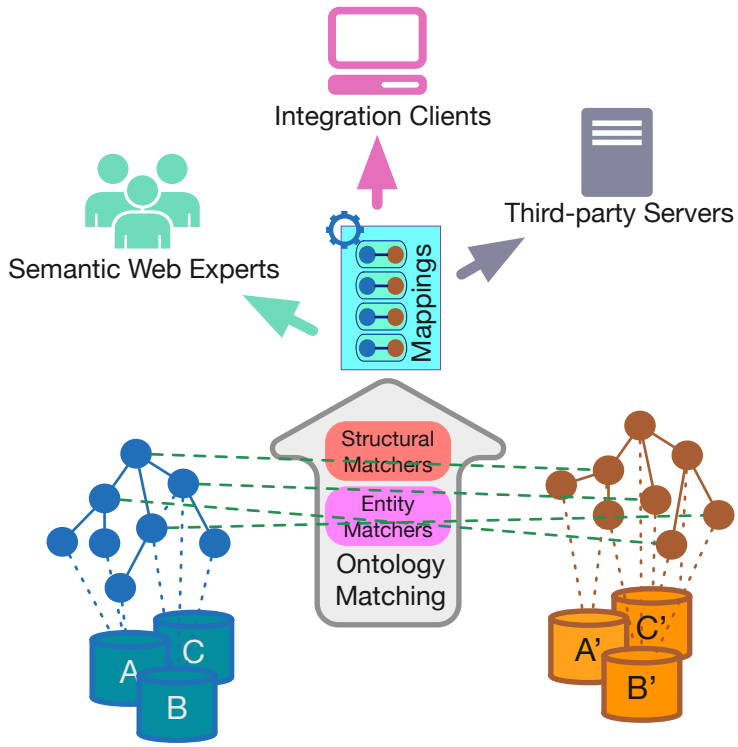


Figure 1.1: Ontology Matching Process

domains. For example, in biomedicine, ontologies are used for representing medical knowledge and clinical guidelines [9], standardization of medical data formats [10], clinical data integration and medical decision-making [11]. Consequently, biomedical ontologies like the Gene Ontology (GO) [12], the National Cancer Institute Thesaurus (NCI) [13], the Foundation Model of Anatomy (FMA) [14], and the Systemized Nomenclature of Medicine (SNOMED-CT) [15] have emerged; furthermore, infrastructures like OBO Foundry [16] and BioPortal [17] are promoting the usage of ontologies in biomedicine. Similarly, in electronic commerce, ontologies are used for mediation among two or more web services [18] and their discovery [19]. 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 systems developed over the years have taken the performance into consideration and have implemented possible resolutions. However, these systems are design time

resolutions with performance aspect focused only on optimization of the matching algorithms and partitioning of larger ontologies into smaller chunks for performance benefits [20]. These resolutions are categorized as effectiveness-dependent ontology matching solutions, as a distinctive trade-off between matching accuracy and performance exists. Furthermore, these resolutions are monolithic in nature, i.e., neither their platform nor results are sharable among the clients and semantic web experts for re-usability. Implementation of these resolutions fails to incorporate benefits available in the form of newer hardware technologies for the sake of performance-gain during the matching process. Among these technologies are affordable parallel systems that are easily available as stand-alone and distributed platforms [21].Current state-of-the-art ontology matching systems 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 [20]. 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 [21].

In earlier years, parallelism and distributed platforms were associated with High Performance Computing (HPC) [22]. 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 smart phones because of the advent of multicore processors [23]. These processors are equipped with multiple cores on a single die, enabling each core to serve as a virtual microproces-

sor, providing parallelism at the hardware level. Cloud computing [24] has recently emerged as a computing platform with reliability, ubiquity, and availability in focus [25]. The utility of cloud as a resource and service provider has already been investigated; however, the benefits of cloud's commodity hardware based distributed infrastructure are still overlooked [25]. Moreover, with the arrival of Cloud computing as a backbone platform for ubiquitous computing [26], 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. This 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. Furthermore, the ubiquitous access to cloud platform can provide ontology matching as a service and a platform, reusable among the heterogeneity resolution clients and semantic web experts.

1.1 Motivation

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 [21]. 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.

Effective ontology matching is a computationally intensive operation requiring Resource-based matching algorithms (Name-based, Hierarchy-based, Annotation-based, and Property-based) to be executed over candidate ontologies. As mentioned in [20], ontology matching between two ontologies is a Cartesian product of all the concepts and their relationships leading to quadratic complexity. In case of medium ($\sim$3,000 concepts) to large-scale (10,000+ concepts) ontologies [20], computation and memory utilization peaks due to the size of the ontology and relationships among its concepts. In our experiments on relatively medium size ontologies (Adult Mouse Anatomy [27] with anatomical part of the NCI Thesaurus [13]), matching algorithms have

taken 20 minutes to obtain required matched results. Over very large ontologies (whole FMA [14] with whole NCI [13]), executing matching algorithms have taken several days to produce desirable matched results. This delay makes ontology matching ineffective for applications like interactive semantic web systems and systems with intelligent information retrieval tasks where in-time processing is a must [28]. Apart from computation needs, memory requirements for matching operations are also higher. Matching algorithms evaluating over the relationships of concepts require concept-graphs in the main memory, producing memory strains in gigabytes during execution. During our experiments over the whole FMA with NCI matching, JVM heap crashes and out-of-memory errors have occurred quite often even with 2 GB of heap memory available. This delay in ontology matching and unreliable nature of the matching process due to memory issues, makes ontology matching ineffective for dynamic applications with performance-based processing demands.

Current state-of-the-art ontology matching systems 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 [20]. 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 are classified 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 [21]. Current ontology matching systems are design time tools which are not optimized for resource consumption [21]. Therefore, no substantial performance-gain is recorded by deploying them over parallelism-enabled stand-alone and distributed platforms. The monolithic implementation of these systems discourages the result and platform sharing. Bridge ontologies and mappings

generated by these systems can be manually shared by the individual user; however, they do not provide a ubiquitous service like platform which clients can use for bridge generation and semantic web experts can execute and evaluate their matching algorithms.

Approaches

Ontology Matching has been an area of interest for researchers in the last decade; consequently, many approaches have emerged over the years. From a performance perspective, current state-of-the-art ontology matching approaches can be classified into two broader categories, i.e., effectiveness-dependent performance-based and effectiveness-independent performance-gain ontology matching solutions.

In the effectiveness-dependent performance-based category, solutions exist where the core of performance extraction is primarily driven by the optimization of ontology matching algorithms only. Although secondary techniques like structural partitioning, fragmentation, and divide-and-conquer approaches are utilized; however, due to the strong coupling between accuracy and performance, the performance of the system always degrades with the increase of matching complexity. Furthermore, with the absence of explicit computational resource utilization, these systems do not scale even in the presence of parallelism enabled hardware [29]. Moreover, the memory consumption aspect of ontology matching has been entirely missed. Considering the fact that the volatile memory is considerable cheap, proposed solutions recommend its excessive utilization. Due to the fact that reading ontology resources from the memory is considerably fast, implementations tend load as many resources as possible, by increasing the possible heap size. This approach inflicts performance degradation instead due to garbage collection by the programming language runtime [30]. Thus, in case of commodity machines with optimal heap size of 2 Gb, completing the ontology matching process in one automated execution becomes a challenge. After a comprehensive review on apparent ontology matching techniques and approaches it is quite clear that most of the approaches exist in this category, including state-of-the-art systems like Falcon-AO [31] which uses a segmentation approach called Anchor-Flood is proposed by [32], LogMap [33] with its axiom classification technique, AROMA [34] with its incremental discovery of rules approach, and AgrMaker [35] with its iterative execution of matching algorithms.

On the other hand effectiveness-independent performance-gain [36] category includes solutions which evaluate ontology matching process from end-to-end and provide resolutions to candidate bottlenecks without inflicting any changes in the matching library of the system. In this category, a clear distinction between matching accuracy and performance of the execution exists logically and physically. Thus, accuracy of the matching process is preserved without the cost of substantial performance-gain. In current ontology matching systems, this category is largely unexplored. A plausible reason can be the background of researchers and developers building these systems, as most of them are from the semantic web background with a focus on highly accurate matching techniques. Consequently, current resolutions are more focused on optimization of the matching algorithms [20]. Thus, performance-gain from faster ontology loading, caching ontology resources. effective memory utilization through-out the matching process, matching space reduction, and scaling for parallel and distributed matching has been completely missed.

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 [37], Task Parallelism, and Data Parallelism [38]. Message passing requires inter-process communication that is appropriate for iterative problems where dependency between operations exists [39]. In task parallelism, independent threads execute different operations on same data. However, data parallelism is one such technique where the same or different autonomous operations are performed on the same or different pieces of data repeatedly [40]. 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 data parallel implementation over other parallel paradigms affects the ontology matching performance by large. By implementing data parallelism, thread-level parallelism gets implemented with a set of independent matcher threads executing the same matching algorithm on a separate part of candidate ontologies. This mechanism may enable matching space reduction as every following set of matching threads will only match ontology resources left unmatched by the previous set with another algorithm. Moreover, unlike message-

passing, data parallel resolves ontology matching by 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. 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. Due to considerable benefits of data parallelism for ontology matching, proposed methodology presented in this thesis utilizes it as the foundation for parallel and distributed ontology matching.

From the perspective of cloud and parallelism, a suggested solution can be the use of cloud-based data-intensive parallel platform called Hadoop [41]. Hadoop with its MapReduce [42] programming model, queries efficiently on distributed data. In case of large-scale ontology matching, it may be considered a candidate technology over cloud platform; however, the essence of performance efficiency in hadoop is coupled with the amount of available data typically in gigabytes and terabytes [43]. The ideal size of a single chunk of data in Hadoop is 64 MB, which is relatively equivalent to a single OWL file making an ontology too small to be distributed over hadoop file system (HDFS) [43], inflicting performance degradation instead.

Contrary to other ontology matching systems; GOMMA, is one of the most performance efficient ontology matching tools [44]. It proposes inter and intra-matcher parallelism which utilizes parallel and distributed infrastructure to achieve better performance for ontology matching [20]. However, its proposed parallelism is embedded within the matching algorithms; furthermore the

effective memory utilization has been ignored. Therefore the authors of [20] have stated the limitations which include, slower performance than expectation due to the complexity of the matchers with higher memory requirements.

Apart from the above discussed categorization, a missing element in the apparent approaches is the reusability aspect of performance runtime for ontology matching. The effectiveness-independent performance-gain approach over ubiquitous and parallel platforms like cloud brings an opportunity for sharable ontology platforms. The clear distinction between accuracy and performance, encourages the semantic web experts to develop their matching algorithms and deploy over the reusable platform for evaluation; furthermore, generating bridge ontologies without worrying for the performance-gain. This strategy can further be employed to matching algorithm evaluation and a resource for ontology matching as a service over cloud.

In contrast with the above-mentioned approaches, this thesis presents an effectiveness-independent performance-gain methodology for ontology matching. This methodology provides a multi-dimensional resolution to the performance aspects of ontology matching, i.e., space, time, re-usable runtime, and ontology matching as a service. Presented methodology provides end-to-end resolution from faster loading of ontologies, their decomposition for reducing structural complexity, memory utilization, matching space reduction to parallel and distributed matching. Furthermore, proposed methodology encapsulates presented resolutions as a high performance ontology matching runtime, deployed over cloud platforms, providing ontology matching as a service and a platform.

Problem Statement

Ontology matching is the most utilized and efficient resolution for semantic heterogeneity among diverse data and information resources. However, effective ontology matching is a two-fold problem where challenges exist in the accuracy and performance aspects. Due to the excess of data, the complexity of their represented ontologies has considerably increased; consequently, algorithms matching these ontologies have also become comprehensive and complex, making ontology matching a computationally intensive matching task. For example, in case of medium ($\sim$3,000 concepts) to large-scale (10,000+ concepts) ontologies [20], computation and memory utilization

peaks due to the size of the ontology and relationships among its concepts. On relatively medium size ontologies (Adult Mouse Anatomy [27] with anatomical part of the NCI Thesaurus [13]), matching algorithms have taken 20 minutes to obtain desirable matched results. Over very large ontologies (whole FMA [14] with whole NCI [13]), executing matching algorithms has taken several days to produce desirable results. This delay makes ontology matching ineffective for applications with in-time processing demands like interactive semantic web systems and systems with intelligent information retrieval tasks [28]. Apart from computation needs, memory requirements for matching operations are likewise higher. Matching algorithms evaluating over the relationships of concepts require concept-graphs in the main memory, producing memory strains in gigabytes during execution. During our experiments over the whole FMA with NCI matching, JVM heap crashes and out-of-memory errors have occurred quite often, even with 2 GB of heap memory available. Therefore; more recently, performance aspects of ontology matching have been under discussion as a separate research issues [21].

Primary motivation of current state-of-the-art ontology matching systems is higher accuracy complemented by efficiency in terms of performance; however, the core techniques for achieving better performance are either related to the optimization of matching algorithms or the fragmentation of ontologies for these matching algorithms [20]. These systems are design time monolithic solutions categorized as effectiveness-dependent matching systems as accuracy of the matching process varies inversely with variation in performance. From this retrospect, a methodology is required which implements explicit performance measures for end-to-end performance-gain with accuracy preservance. This methodology must identify the candidate bottleneck areas throughout the ontology matching process and propose resolutions without impacting the implementation of matching algorithms, thus presenting an effectiveness-independent performance-gain approach. Furthermore, this methodology must be available and non-monolithic, such that not only clients utilize it for heterogeneity resolution but also semantic-web experts can utilize it for plugging-in their matching algorithms for execution and evaluation.

To achieve the goal of effectiveness-independent performance-gain, a set of objectives are required to be achieved as a resolution. These objectives include; resolution for faster ontology loading and access, effective not excessive memory stress during ontology matching process, it-

erative matching process with possible matching space reduction for every subsequent matching algorithm execution, and exploitation of available computational resources for performance-gain benefits, i.e., parallel and distributed matching. Furthermore, all the resolutions must be incorporated as a non-monolithic ontology matching runtime, which can be shared as a service and platform.

The main challenges of successfully achieving the goal of performance-based ontology matching in perspective of above mentioned objectives are: (i) Effective memory utilization and completion of matching process within the bounds of optimal heap memory; (ii) Optimal computational resource utilization during the matching process for performance-gain in ontology matching process with accuracy preservance; (iii) Implementation of a high performance ontology matching runtime with parallel and distributed ontology matching (iv) availability of this runtime as non-monolithic ontology matching resolution as a service and a platform . This thesis presents one such multi-dimensional performance-based ontology matching methodology that caters these challenges and provides a substantial performance-gain for the whole matching process.

1.2 Contributions

In consideration to the problem statement and accumulating the mentioned opportunities, i.e., delay in ontology matching, design-time 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 monolithic nature of current systems, provides the motivation for a performance-based ontology matching methodology. This thesis contributes by presenting one such methodology which provides end-to-end performance resolutions for the ontology matching process with effectiveness-independence. Presented methodology 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 achieved. Furthermore, the proposed methodology is implemented as a high performance runtime for ontology matching and extended as non-monolithic ontology matching service, accessible over cloud

platform by clients for heterogeneity resolution and as a platform for ontology matching algorithms execution and evaluation. The main contributions of the thesis are briefly described in the following subsections.

Smaller Memory Footprint during Ontology Matching Process

Ontologies being structurally complex tend to be slow in loading and stress on the memory of the runtime during the matching process. As a resolution, the presented methodology decomposes the complex ontologies into smaller Resource-based ontology subsets depending upon the needs of matching algorithms. These subsets are independent and simpler (reduced structural complexity) with performance and scalability-friendly data structures. This method contributes to the matching performance by only loading the ontology resources required by matching algorithms as data structures that can be easily partitioned. These subsets are also preserved by serialization to reduce the matching effort for future matching requests of the same ontologies. Furthermore, matching process loads ontology subsets by deserialization in parallel over parallelism-enabled platforms which improves the ontology resource access substantially faster in later stages of execution.

Matching process represents an execution of several matching algorithms implementing their own strategy to deliver the matched results. However, due to decoupled execution of algorithms, redundant matching and late redundancy checking is probable for multiple bridge instances in the generated bridge ontology. The presented methodology aligns the execution of matching algorithms to reduce the matching space for every following matching algorithm execution during the whole matching process. This method contributes in performance by reducing the amount of matching tasks to unmatched resources only, thus avoiding redundant expensive matching operations with no requirement of redundancy checking for multiple bridge instances at the end of the process.

Performance-gain by Parallel and Distributed Ontology Matching

Ontologies by default are not parallelism-friendly due to their intensive cohesive nature. However, due to our scalability-friendly subsets, presented methodology implements division of a matching process over these subsets into three levels of abstractions (independent matching requests,

matching jobs, and matching tasks) over a parallel 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 matching performance by distributing matching tasks over participating computing cores and executing them in parallel at finer level with optimal computing resource utilization.

High Performance Ontology Matching Runtime

Current state-of-the-art ontology matching systems fail to encapsulate the performance aspects of matching as a re-usable component that is not dependent on matching libraries and algorithms. On the contrary, presented methodology implements a high performance ontology matching runtime that utilizes the above mentioned methods for effectiveness-independent performance-gain during matching. With explicit thread-level parallelism for data parallel ontology matching, presented runtime can effectively scale over parallel platforms including multicore desktop and multi-node cloud computing infrastructures. This runtime does not depend on the scope, complexities, and size of the ontologies to be matched.

Ontology Matching as a Service and a Platform

Current ontology matching approaches are design time tools with monolithic implementations. However, presented methodology implements a decoupled non-monolithic resolution where performance runtime is deployed over cloud platform with interfaces not only at the service level but also at the platform level. Clients for mapping generation as bridge ontology use an SOA-based ontology matching service to submit their matching requests. Semantic web and ontology matching algorithm experts use platform level *Matchable* interface to plug-in their matching algorithms. These algorithms are executed over parallelism-enabled cloud platform with above proposed performance-gain strategies. By this resolution, an author of the matching algorithm needs not to provide any parallel code within the implementation of the matching algorithm.

Presented methodology has been comprehensively evaluated with dataset of real world ontolo-

gies from diverse knowledge domains, having various sizes and complexities. Matching problems used for evaluation on these ontologies are carefully curated and particularly designed by Ontology Alignment Evaluation Initiative (OAEI) for evaluation of ontology matching systems. Presented methodology has been implemented as a runtime for our ontology matching system called SPHeRe (System for Parallel and Heterogeneity Resolution). For ontologies from Anatomy track (Adult Mouse Anatomy with human anatomy part of NCI Thesaurus), SPHeRe 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, SPHeRe 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, SPHeRe 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, SPHeRe 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 further evaluated SPHeRe with small ontologies from Conference track over a dual-core Microsoft Azure public cloud 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 SPHeRe with GOMMA's [45] parallel matching techniques. For large category, SPHeRe outperforms intra-matcher by 5.2% on desktop and 55% over cloud platform. For very large category, SPHeRe outperforms intra-matcher by 4.6% on desktop and 47.7% over cloud platform. SPHeRe also outperforms Intra&Inter multi-node matcher by 12.8%.

1.3 Thesis Organization

This dissertation is organized into chapters as following.

- **Chapter 1: Introduction**. Chapter 1 provides introduction of the research work on ontology matching and its performance bottleneck. Furthermore, it provides a review of current approaches for ontology matching performance resolutions. This chapter focuses on the problems in the area of performance during ontology matching process and discusses the

objectives for the resolution from a multi-dimensional prospective.

- **Chapter 2: Related Work**. The background detail is provided in this chapter about the ontology matching techniques and approaches where performance has been given a vital importance. Systems and techniques built from generic to specific ontology matching problems are reviewed with the limitations in contrast with proposed methodology.

- **Chapter 3: Effectiveness-independent Performance-gain in Ontology Matching**. This chapter describes the proposed methodology for memory space reduction and effectiveness-independent performance-gain by parallel and distributed matching. This chapter also provides overview of the concepts used in the thesis related to proposed methodology.

- **Chapter 4: Performance-based Ontology Matching Runtime**. This section presents a comprehensive runtime implemented for performance-based ontology matching using the approaches from chapter 3 as foundation steps. Details regarding the execution steps and technical implementation aspects of participating components is also presented in this chapter.

- **Chapter 5: Ontology Matching as a Service and a Platform**. This section extends the implementation of Performance-based Ontology Matching Runtime as a Service and a Platform. The runtime is deployed over cloud platform and used by clients for bridge ontology and semantic web experts for their ontology matching algorithm evaluation.

- **Chapter 6: Evaluations and Discussion**. The evaluations performed on the presented methodology are comprehensively described and discussed in this chapter. OAEI's standard datasets are used for this comprehensive evaluation.

- **Chapter 7: Conclusion and Future Directions**. This chapter concludes the thesis and also provides future directions in this research area. The main contribution of the thesis is also highlighted in this chapter.

Chapter 2

Related Work

Nowadays, Internet has grown to become a huge public resource for large and ever-growing heterogeneous data [46]. 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.

2.1 Generic Ontology Matching Systems, Tools, and Techniques

From the technique perspective, a considerable amount of research has been done towards optimizing ontology matching algorithms for better performance [20]. Consequently, various structural partitioning approaches for ontologies have emerged. Falcon-AO [47], a famous ontology matching tool provides a divide-and-conquer approach called PBM [48]. Similarly, an ontology segmentation approach called Anchor-Flood is proposed by [49]. However, in both of these techniques, performance is coupled with the complexity of the partitioning approach. None 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 [50] and [51]. These works are based on semantic negotiation protocols HISENE [52] and HISENE2 [53]. In [50], 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 DE-

velopment Framework) [54] is utilized. Although the semantic negotiation has shown promising results in efficiency; however, 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 the network I/O overhead. In a decentralization approach proposed in [51], 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 [54] . Although this can be efficient in limited computational resource environment, but under-utilization of computational resources in current multicore systems.

In current state-of-the-art generic ontology matching systems, i.e., AgrMaker [35], LogMap [33], and GOMMA [45], performance has been given a considerable focus to complement accuracy of these systems.

2.1.1 Falcon-AO

Ontology Matching has been an area of interest for researchers in the last decade. Many tools and techniques have evolved over the years. From techniques perspective, considerable amount of research has been driven towards optimization of ontology matching algorithms for better performance [20]. Consequently, various structural partitioning approaches have emerged. Falcon-AO [31], a famous tool for ontology matching proposes an effective divide-and-conquer approach called PBM [31]. Similarly a segmentation approach called Anchor-Flood is proposed by [32]. However, [31] and [32] does not benefit from the exploitation of newer hardware for ontology matching. For better performance during ontology matching, implementation of parallelism over multicore platform like cloud has been missed.

2.1.2 Agreement Maker

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.

2.1.3 LogMap

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 [33], 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.

2.1.4 AROMA

AROMA, as described in [34], is a simple and adaptable ontology matching tool which utilizes Knowledge Discovery in Databases (KDD) [55] model. AROMA itself does not implement any concurrency control; however, KDD discusses parallel clustering technique for incremental discovery of rules and structures in [56]. A presumption can be driven that such an implementation might be utilized by AROMA which in-fact contributes to its better reduction score, though [34] fails to mention any parallelism or concurrency involved for the benefit of AROMA's performance.

2.1.5 GOMMA

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 [57] and [20]. In [57], 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 [57] discusses entity matching in general. However, in [20], 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. By 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., AdultMouseAnatomy 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).

2.2 Biomedical Ontology Matching

Due to the excessive utilization of ontologies in biomedical and bioinformatics, some of the ontology matching systems are developed particularly for matching biomedical ontologies [58].

2.2.1 SAMBO

Among them, SAMBO [59] 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 [60] 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 [61] 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 [59], 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) [62], and MeSH (15, 39, and 45 terms) [63] with MA (18, 77, and 112 terms) [27] for system evaluation and have not provided any benchmarks regarding large-scale biomedical ontologies. However, results of OAEI 2008 [64] 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 [65]. 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.

2.2.2 ASMOV

Another ontology matching system with the motivation of producing alignments for biomedical ontologies is ASMOV [46]. With its effectiveness dependent performance, authors of [46] 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 [46]. 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.

2.2.3 ServOMap

ServOMap [66] 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 [66] 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.

2.3 Candidate platforms for Parallel Ontology Matching

2.3.1 Big Data for Ontology Matching

From the perspective of data parallelism over distributed platforms, BigData technologies like Hadoop with its MapReduce programming model, queries 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 [67] and the structure of the data as Hadoop is unsuitable in situations where structure of the data is important as the data itself [68]. 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 [69]) 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 [70], Heart [71], and Hadoop Distributed RDF Store (HDRS) [72] have yet to prove their efficiency and performance.

2.3.2 Flynn's Taxonomy and Ontology Matching

Parallel ontology matching has been theoretically discussed in [37]. 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 [73], 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, [37] 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 methodology 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. Furthermore, our approach aligns the execution of matching algorithms to minimize the matching space; consequently, contributing in performance-gain by large. Our methodology also 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. Moreover, the proposed methodology is implemented as a performance-based ontology matching runtime and exposed at service and platform level for usage.

Chapter 3

Effectiveness-independent Performance-gain in Ontology Matching

This chapter provides details regarding the two foundation dimensions, i.e., space and time, catered by the presented methodology for an effectiveness-independent performance-gain in ontology matching process.

3.1 Memory Footprint Reduction

The space dimension of ontology matching performance is catered by the presented methodology with two approaches, i.e., Ontology subset generation and eager space reduction technique.

3.1.1 Matching Algorithm-based Ontology Subset Generation

By default ontologies are not scalable structures from the perspective of performance ([74], [75], and [76]). Therefore, the ontology subset generation approach is proposed where candidate ontologies (Source Ontology: O_S) and (Target Ontology: O_T) are converted into simple subsets with performance and scalability-friendly data structures. These subsets are generated depending upon the needs of the matching algorithms making them encapsulated and independent (equation 3.1, 3.2, and 3.3).

$$i \quad \in \quad Algorithms : Algorithms = \{String, Label, Properties, ..., Child\} \tag{3.1}$$

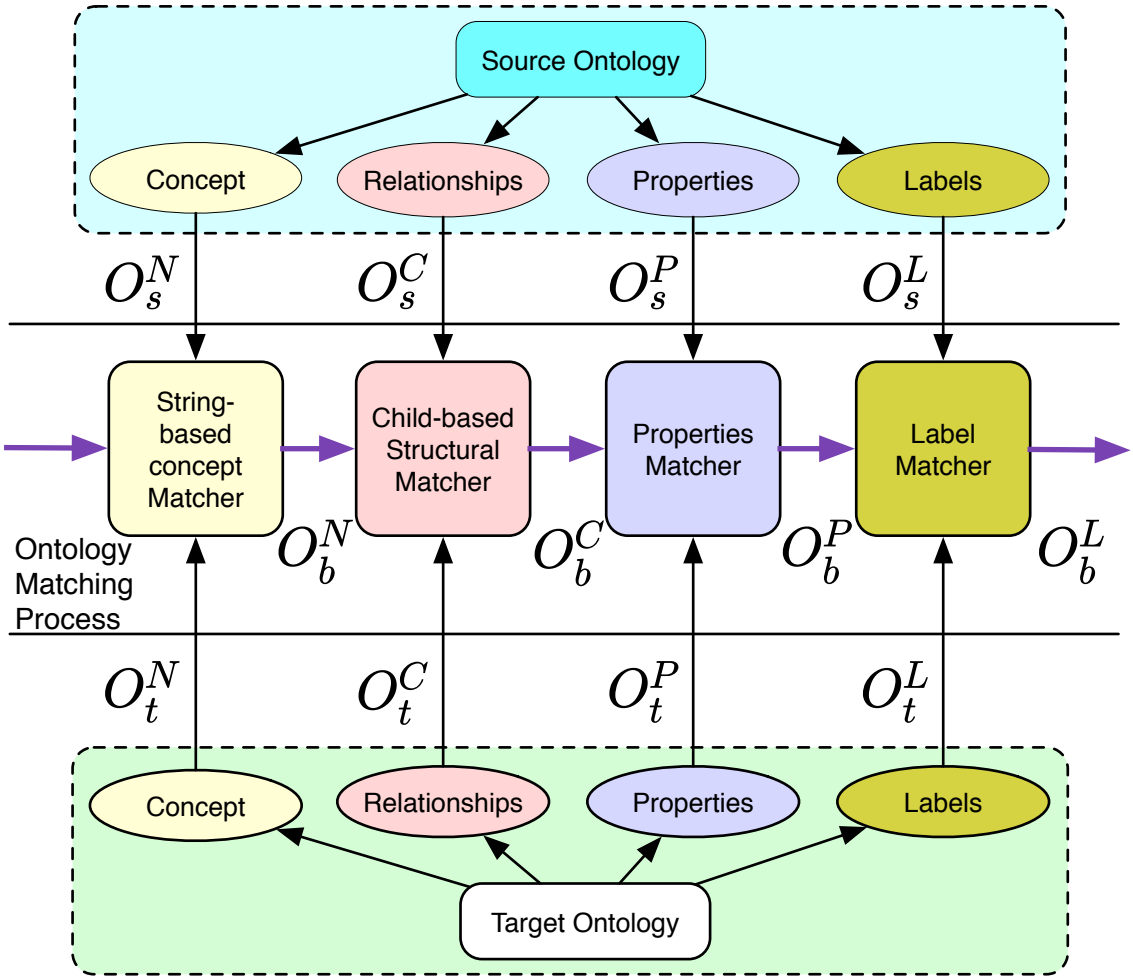


Figure 3.1: Algorithm-based Ontology Subsets for Matching

$$O_x^i \leftarrow f^i\left(O_x\right) \quad : \quad x \quad \in \quad \{source : s, target : t\} \tag{3.2}$$

$$O_x = \bigcup_{j=1}^{n} O_x^j \qquad : \qquad n = NumberOfAlgorithms \tag{3.3}$$

For example, consider a matching process between two candidate ontologies (source and tar-

get) as illustrated in Fig 3.1. To complete the whole matching process, a library consisting upon four matching algorithms (string-based concept matcher, child-based structural matcher, properties matcher, and label matcher) is evaluated over candidate ontologies. Instead of loading whole ontologies four times, presented approach creates subsets of candidate ontologies and loads the required subset(s) needed for matching algorithm to execute. String-based concept 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 in the memory. At any given execution of matching algorithm, the memory of the runtime will host the subset(s) required based on the needs of the current matching algorithm. Accessing ontology resources from these subsets is significantly faster due to their smaller size, independent nature, and data structures that can be readily partitioned. This approach effectively contributes in overall performance-gain especially when matching large-scale ontologies. Theoretical evidence regarding the completeness of this matching algorithm based subset generation is provided by the following theorem (Theorem 3.1.1).

Theorem 3.1.1 *For a candidate ontology O_S with N subsets, if a set of Algorithms A completely matches O_S with O_T and produces the Accurate bridge Ontology O_B, then the generation of the subsets provide no loss of required ontology information for matching*

Proof *Consider a candidate ontology O_S to be completely matched with O_T by a set of matching algorithms A.*
O_S has x subsets such that for every algorithm $a\exists n$. Although $n1 \cup n2 \cup n3..nx \subseteq O_S$; however, as $(A) \mapsto$ Bridge Ontology O_B. Thus, generation of N subsets provide adequate ontology information.
Conclusively, Generating the subsets according to the matching algorithms for an accurate bridge ontology provides no loss of ontology information

In the experiments, working with ontology subsets instead of whole ontologies for matching, we have recorded as much as 8 times faster ontology resource loading with 4 times smaller memory footprint. These subsets are serialized and persisted in repositories, preventing the regeneration of ontology subsets for already serialized ontologies for future matching requests.

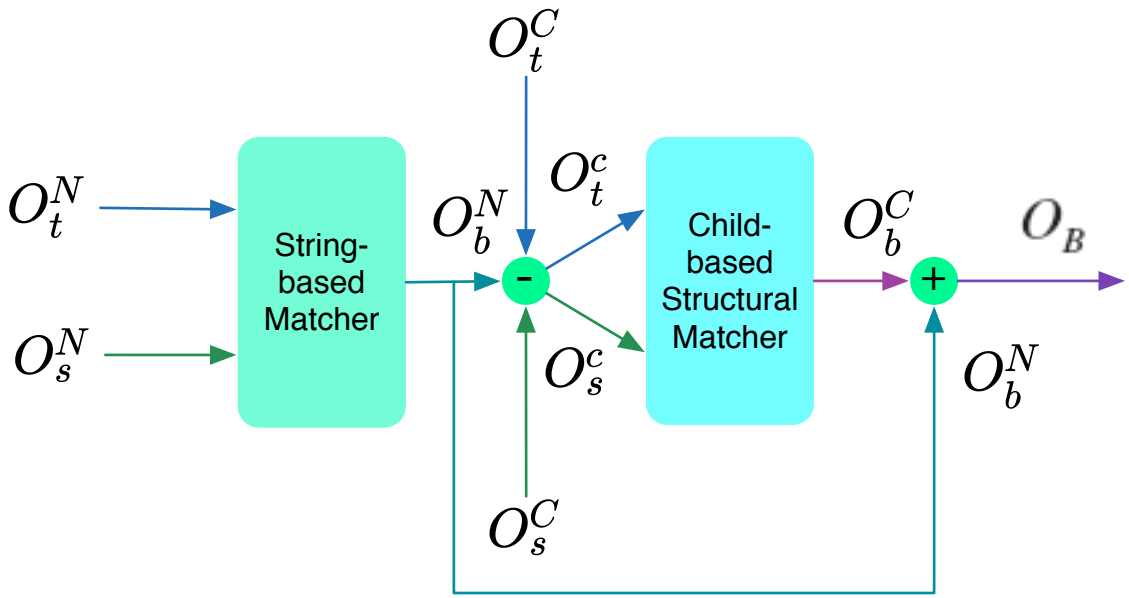


Figure 3.2: Eager Matching Space Reduction

3.1.2 Eager Matching Space Reduction

Apart from matching algorithm-based subset generation, presented methodology proposes eager matching space reduction approach. This approach aligns the execution of matching algorithms to minimize the matching space for every following matching algorithm execution (equation 3.4 and 3.5).

$$O_b^1 \leftarrow (m \times n)_{i=1} \quad \forall \quad i \in Algorithms, \quad m \in O_s^i \quad \& \quad n \in O_t^i \tag{3.4}$$

$$O_B \leftarrow \bigcup_{i=2}^{t} \left(\left(m^i - \left(m^i \cap O_b^{i-1} \right) \right) \times \left(n^i - \left(n^i \cap O_b^{i-1} \right) \right) \right) \quad | \quad O_b^i \geq O_b^{i+1} \tag{3.5}$$

As illustrated in Fig 3.2, 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 first and gen-

erates its intermediate bridge ontology (O_b^N). In the following execution (child-based), ontology resources that are already matched and now part of O_b^N are removed from ontology subsets, generating much smaller subsets (O_s^C and O_t^C) prior to matching. By this method, the number of expensive matching operations is reduced as they only execute on ontology resources that are still unmatched; consequently, reducing the matching space, eliminating chances of redundant matching task, and improving overall matching performance during run-time. Furthermore, this approach also eliminates the chances of redundant matches in the final bridge ontology (O_B). Theoretical evidence for reduced matching space by the presented approach is provided by the following theorem (Theorem 3.1.2).

Theorem 3.1.2 *For a matching space of n matching tasks (MT_n), matched by a set of Algorithms A. If intermediate bridge ontology O_b is subtracted from MT_n then every following execution of an algorithm will have a reduced matching space.*

Proof *Consider a matching space MT_n, matched by a set of Algorithms A.*
For A_1, intermediate bridge ontology is $f(m \times n)_1 = O_b^1$. For following executions of A, the concepts yet to be matched in source ontology O_S are $m^i - (m^i \cap O_b^{i-1})$, Similarly the concepts yet to be matched in target ontology O_T are $n^i - (n^i \cap O_b^{i-1})$. For every $O_b^i \geq O_b^{i+1}$, $f(m^i - (m^i \cap O_b^{i-1}) \times n^i - (n^i \cap O_b^{i-1})) < f(m^i - (m^i \cap O_b^i) \times n^i - (n^i \cap O_b^i))$
Conclusively, By Eager Matching space reduction every following matching algorithm will execute on a smaller matching space

3.2 Parallel and Distributed Ontology Matching

The time dimension of ontology matching is catered by the presented methodology with parallel and distributed matching. The primary objective of this approach is to implement effectiveness-independent performance-gain by drawing abstractions over the ontology matching process. Depending upon these abstractions the matching process is distributed among available computing resources for parallel and distributed matching. These abstractions are drawn from higher to a primitive level such that an independent execution can invoke any matching algorithm without inflicting a change in the implementation of the algorithm.

3.2.1 Matching Task

Matching Task (MT) is the unit of the 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 parallel and distributed ontology matching. Equations (3.6, 3.7, and 3.8) describe this distribution

$$MT_i \cap MT_{i+1} \cap MT_{i+2} .... \cap MT_n = \emptyset \tag{3.6}$$

$$MT_{Total} \geq m \times n \qquad \forall \quad m \in O_S \quad \& \quad n \in O_T \tag{3.7}$$

$$MT_{Core} \leftarrow \frac{MT_{Total}}{Cores_{Total}} \tag{3.8}$$

A primitive example of MT is illustration in Fig 3.3, 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, i.e., 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 parallel 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.

Apart from gaining performance from parallel and distributed matching, the independent nature of the matching tasks benefits the ontology matching process by avoiding the inter-matching

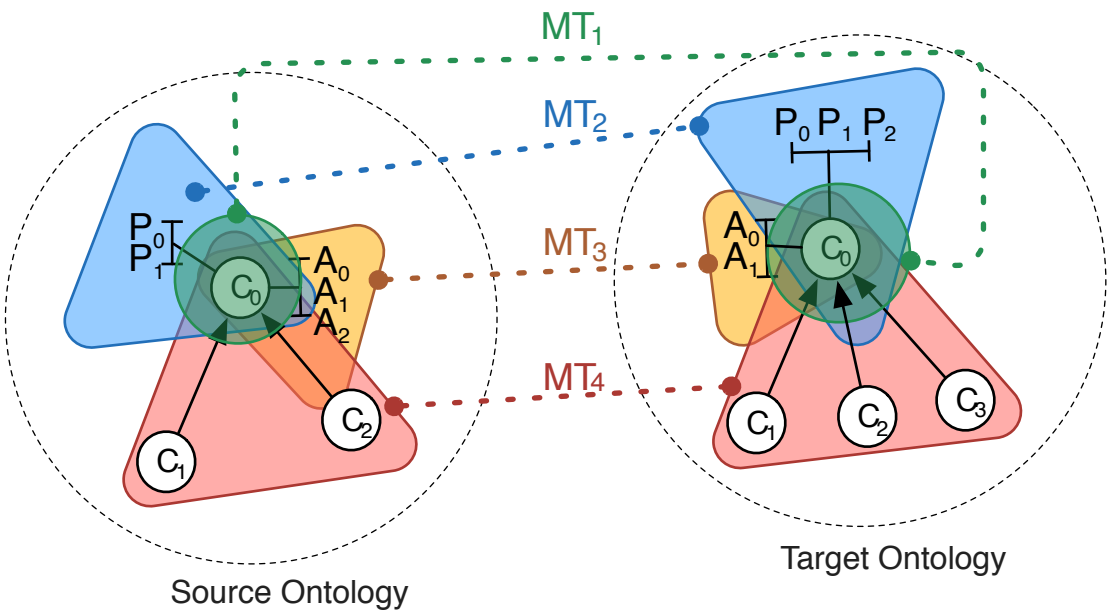


Figure 3.3: Matching Tasks between two concepts of Candidate Ontologies

communication during execution of a matching algorithm. With zero communication overhead, no matching task waits for any other task to be completed. Theoretical evidence of this approach is provided by the following theorem (Theorem 3.2.1).

Theorem 3.2.1 *For a matching process of n independent matching tasks in a distributed environment, if matching tasks are independent, then the communication overhead is none*

Proof *Consider a matching process of candidate ontologies O_S and O_T with n matching tasks (MT) to be completed.*

As all the matching tasks are independent, i.e., $MT_1 \perp MT_2 \perp MT_3, . \perp MT_n$. A matching job (MJ_x) collection of $MT_x \subseteq MT$. Intersection of multiple matching job $MT_x \cap MT_y = \phi$, Thus a matching process based on independent matching jobs having independent matching tasks require no dependency among themselves whether on same machine or another.

Conclusively, Set of independent matching tasks in a matching process with require zero or no communication between participating nodes or jobs.

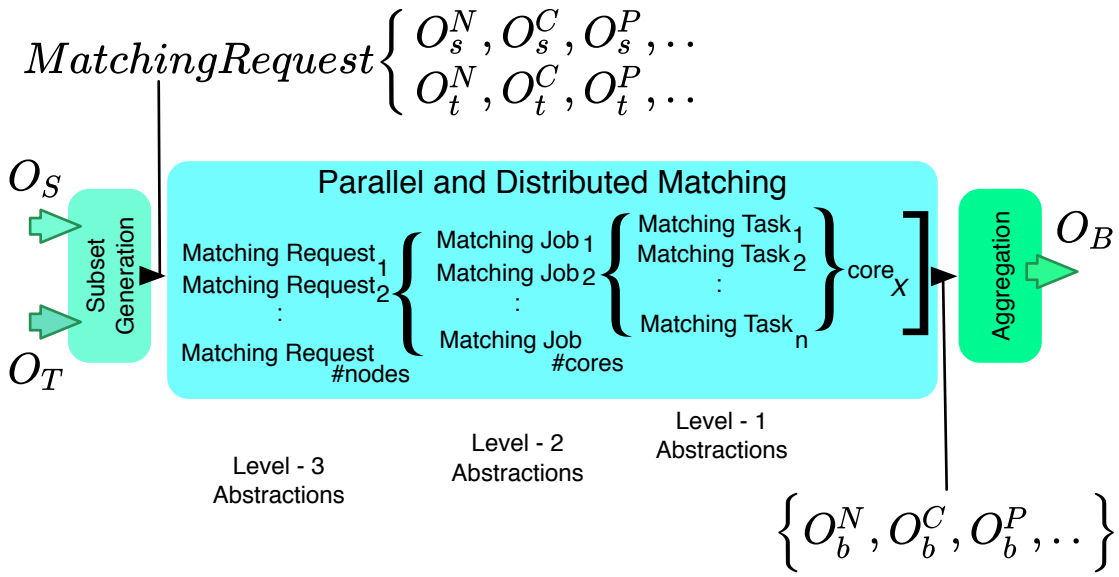


Figure 3.4: Matching Abstractions for Parallel and Distributed Matching

3.2.2 Matching Process Abstractions

Depending upon the available computing resources, abstractions are defined on the ontology matching process. Data parallelism, being the foundation of presented parallel and distributed matching approach, 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 3.4, 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 equation 3.8, 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. Due to size based partitioning, in case of uneven distribution, the number of undistributed matchable tasks will always be less than the amount of computing resources. These undistributed tasks can be matched by any matching job within the matching process. Theorem 3.2.2 and corollary 3.2.3, and theorem 3.2.4 and corollary 3.2.5 provide theoretical evidence for this approach.

Theorem 3.2.2 *For all matching problems on a single-node multicore platform, if total ontology matching tasks* $(MT_T) \geq$ *available computing cores* (C_T) *then a Multicore distribution algorithm distributes no more than* $MT_T/C_T + R$*, matching tasks per core where* $0 < R < C_T$

Proof *Consider a participating node P with* C_T *number of cores.*
Candidate ontologies O_S *and* O_T *with m and n matchable concepts respectively have total matching tasks* $(MT_T = m \times n)$*. Distribution of* MT_T *over P is* MT_T/C_T *= matching task per core* $MTpC + remainder R$*, where* $0 \leq R < C_T$
Conclusively, All cores will have no more than $MT_T/C_T + R$ *Matching Tasks to compare*

Corollary 3.2.3 *For Large scale matching problems on a single-node multicore platform, if total ontology matching tasks* $(MT_T) >>$ *available computing cores* (C_T) *then a multicore distribution algorithm distributes the matching tasks evenly.*

Proof *From Theorem 3.1.4, Distribution of* MT_T *over P is* MT_T/C_T*= Matching Task per Core MTpC + remainder R. If* $C_T << MT_T => R << C_T, thus R << MT_T$*, The Computational Cost of R is so small in* $MTpC + R$ *such that it can be ignored, thus* $MT_T/C_T = MTpC$*.*
Conclusively, For very large scale matching problems, Multicore Distribution distributes all matching tasks even between the computing cores

Theorem 3.2.4 *For all matching problems on a multi-node multicore platform, if total ontology matching tasks* $(MT_T) \geq$ *available computing cores* (N_T) *then a Multi-node distribution algorithm distributes no more than* $MT_T/N_T + R$*, matching tasks per node where* $0 < R < \sum_1^N C_T$

Proof *Consider a multi-node platform with participating nodes N. having* C_T *number of cores =* $\sum C_T$*.*
Candidate ontologies O_S *and* O_T *with m and n matchable concepts as total respectively have total matching tasks* $(MT_T = mxn)$*.Distribution of* MT_T *over N is* MT_T/C_T *= matching task per core per node MTpCpN + remainder R, where* $0 \leq R < C_T$
Conclusively, All cores will have no more than $MT_T/C_T + R$ *Matching Tasks to compare*

Corollary 3.2.5 *For Large scale matching problems on a multi-node multicore platform, if total ontology matching tasks* $(MT_T) >>$ *available computing cores* $\sum(C_T)$ *then a multi-node distribution algorithm distributes the matching tasks evenly.*

Proof *From Theorem 3.1.6, Distribution of* MT_T *over N is* MT_T/C_T *= Matching Task per Node MTpCpN + remainder R. If* $C_T << MT_T => R << C_T, thus R << MT_T$*, The Computational Cost of R is so small in* $MTpCpN + R$ *such that it can be ignored, thus* $MT_T/C_T = MTpC$*. Conclusively, For very large scale matching problems, Multi-node Distribution distributes all matching tasks even between the computing cores of participating nodes*

On a single-node parallel platform, matching tasks are only distributed among existing cores as matching jobs; however, on a multi-node parallel platform, distribution is among the participating nodes as matching requests. Each set of matching tasks is assigned to a computing core with knowledge of the matching algorithm to be executed on them. Subsequently, all cores in participating nodes are invoked in parallel for the matching process. Following equations (3.9, 3.10, 3.11, and 3.12) describe this distribution abstraction implemented by presented methodology:

$$MR \leftarrow \sum_{i=1}^{n} MR_i \quad : \quad n = TotalNodes \tag{3.9}$$

$$MR_i \leftarrow \sum_{i=1}^{c} MJ_i \quad : \quad c = TotalCoresPerNode \tag{3.10}$$

$$MJ_i \leftarrow \left\{ \bigcup_{i=1}^{t} MT_i \right\} \quad : \quad t = TotalTasksPerCore \tag{3.11}$$

$$MT_i \leftarrow m \times n \quad \forall \quad m \in O_S \quad \& \quad n \in O_T \tag{3.12}$$

After completion of parallel and distributed matching, i.e., all the parallel matchers have finished their respective matching jobs over their assigned cores in a single- or multi-node platform, aggregation of all bridge instances is invoked. For this step, all the matched results are aggregated and the final mediation bridge ontology is generated. In a multi-node platform, the primary node waits for all the secondary nodes to submit their match results before generating the aggregated bridge ontology. Following equations (3.13, 3.14, and 3.15) describe this process in a multi-node platform.

$$O_b^{Job} \leftarrow \bigcup_{i=1}^{t} (m \times n)_i \quad : \quad m \times n \neq \emptyset, \quad t = TotalTasksPerCore \tag{3.13}$$

$$O_b^{Node} \leftarrow \sum_{i=1}^{j} O_b^{Job=i} \quad : \quad j = TotalJobsPerNode \tag{3.14}$$

$$O_B \leftarrow \sum_{i=1}^{n} O_b^{Node=i} \quad : \quad n = TotalNodes \tag{3.15}$$

On a single-node platform, where utilization of computing resources scales down to multicore, generation of mediation bridge ontology is a two-step process (described in equations 3.16 and 3.17):

$$O_b^{Job} \leftarrow \bigcup_{i=1}^{t} (m \times n)_i \quad : \quad m \times n \neq \emptyset, \quad t = TotalTasksPerCore \tag{3.16}$$

$$O_B \leftarrow \sum_{i=1}^{j} O_b^{Job=i} \quad : \quad j = TotalJobs \tag{3.17}$$

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.

3.3 Summary

This chapter presented the overview of the two approaches dealing with space and time dimensions of effectiveness-independent performance-based ontology matching methodology. For space, presented methodology provides resolutions for reduced memory footprint during the ontology matching process. For time, presented methodology provides a data parallel approach with abstractions over the matching process for parallel and distributed ontology matching. In both resolutions, accuracy of the ontology matching process has been preserved as performance is gained by effective resource utilization.

These approaches becomes the foundation for the implementation of the comprehensive High Performance Ontology Matching Runtime presented in chapter 4.

Chapter 4

Performance-based Ontology Matching Runtime

This chapter covers the third dimension of the presented methodology, i.e., performance-based ontology matching runtime. In this chapter implementation details of the runtime based upon the space and time resolutions of our methodology are discussed. This includes the overall execution flow, the stack design of the runtime and the details regarding its core components. This runtime is implemented and incorporated as the core of our ontology matching system called SPHeRe.

4.1 SPHeRe: System for Parallel Heterogeneity Resolution

SPHeRe is an implementation of computation intensive data parallelism, i.e., each processing core performs ontology matching on a separate piece of candidate ontologies. Matching algorithm executes across multiple cores and processors over parallel computing platforms. For performance improvement, SPHeRe implements parallelism at operational level, i.e., ontologies are not just matched in parallel, they are loaded, parsed, cached, and delivered in parallel too. Fig 4.1 illustrates the multiphase design of SPHeRe's execution flow and table 4.1 describes the notations used. From left to right, it can be seen that system's execution has been divided into 3 phases: (i) Pre-Matching; (ii) Parallel Matching; and (iii) Post-Matching. All phases are equipped with components to perform parallel operations according to the requirements of the running tasks. This design eases the development and addition of newer components by following the standard input output interfaces among all the phases. Functionalities of each phase are described in following subsections.

Table 4.1: Terminologies and Notations used

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

4.2 Execution Phases

4.2.1 Phase-I: Pre-Matching

Source (O_S) and target (O_T) ontologies are provided to the system either by using SPHeRe's web-based UI or SPHeRe's ontology matching web-service. These ontologies are loaded in parallel by multithreaded ontology load interface (OLI). OLI is responsible for parallel loading of candidate ontologies (O_x), OWL file parsing to create object model (ontology model), and ontology model serialization and deserialization tasks. Prior to any parallel matching, necessity is a performance friendly and thread-safe ontology representation. Without such representation, data parallelism over multithreaded execution in Phase II cannot be achieved. OLI's parser owns this responsibility by generating a thread-safe performance friendly ontology model object. This ontology

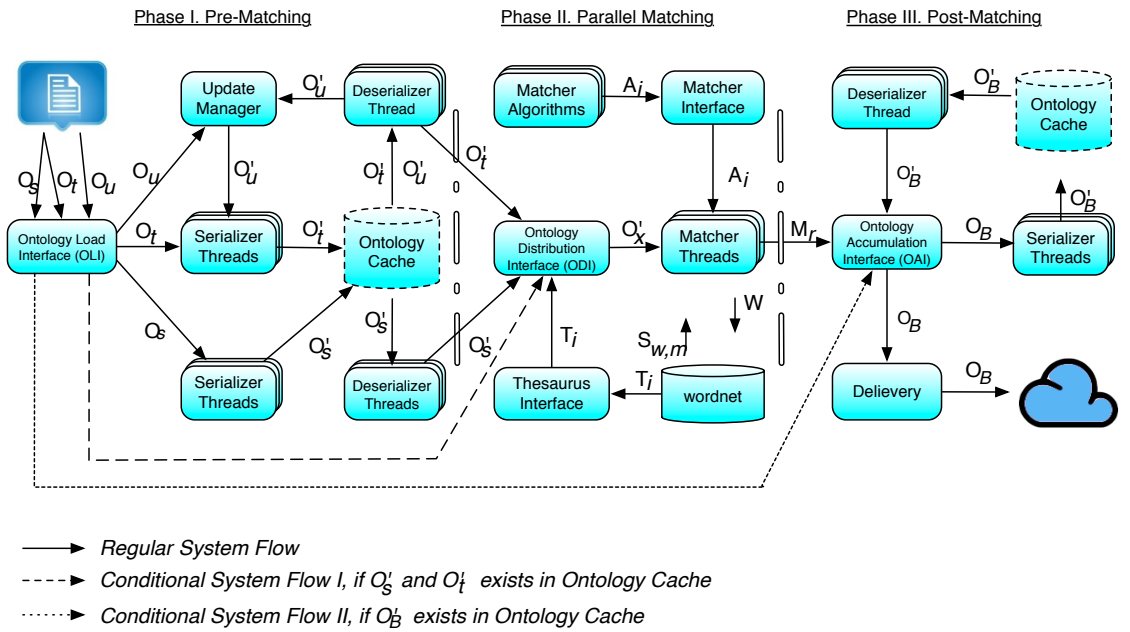


Figure 4.1: Execution flow of SPHeRe

model facilitates our system with the following benefits. Firstly, an ontology representation with the knowledge of total associated information an OWL file is encapsulating (classes, class relationships, properties, axioms and annotations). Distribution and matching phase effectively uses this knowledge while task distribution. Secondly, an ontology representation with thread-safety by providing immutable objects in a multithreaded environment. Thirdly, division of ontology into multiple subsets according to the needs of matching algorithms, preventing the system from loading information not required for matching. This technique reduces the memory strain, avoiding the possibility of JVM heap crashes during execution. Fourthly, no file IO during matching phase, avoiding the huge performance bottleneck of accessing OWL files numerously during the matching process. Lastly, reducing the size of ontology by removing redundant information and unnecessary data, creating smaller memory footprint. For example, the namespace URI is redundant over an OWL file, keeping a single attribute in an ontology model that stores the namespace URI reduces the actual size of concept names which are essentially used entities while matching in Phase II.

After parsing the candidate ontologies, OLI invokes parallel threads of custom serializer (de-

scribed in [77]) for ontology model caching. A single ontology is serialized into subsets based on matching algorithms. In current implementation, four subsets of O_x are created: (i) collection of class names for name-based ontology matching (O'_{xC}); (ii) hierarchical data-structure for relationship-based ontology matching (O'_{xH}); (iii) collection of classes with their corresponding labels for label-based ontology matching (O'_{xL}); and (iv) collection of classes with their properties for property-based ontology matching (O'_{xP}). All these serialized subsets are individually persisted in ontology cache. OLI's parser and serializer only get executed when a new ontology is provided by matching request, i.e., in case of O'_s and O'_t already been available in ontology cache, Conditional System Flow I (illustrated in Fig 4.1) gets executed. Parsing and serialization steps are skipped until a massively updated version of a previously serialized ontology is received. Ontology cache provides persistence to serialized ontologies and their corresponding MD5 [78] hash values. OLI verifies whether the ontologies have already been serialized by calculating the hash values of candidate ontologies. In case of smaller partial updates, update manager renews ontology contents of serialized ontologies and persists update ontology instance O_i in ontology cache. An O_i can be obtained following the strategies mentioned in [79] and [80]. Multiple serialized versions of a particular O_s and O_t are maintained by ontology cache. In case of multiple SPHeRe nodes, ontology cache is replicated over every node, keeping all nodes synchronized. O'_s and O'_t are loaded in parallel by deserializer threads, providing a single-step ontology loading and feeding to distribution and matching phase.

4.2.2 Phase-II: Parallel Matching

Serialized subsets of source (O'_s) and target (O'_t) ontologies are loaded in parallel by multithreaded ontology distribution interface (ODI). ODI is responsible for task distribution of ontology matching over parallel threads (Matcher Threads). ODI currently implements size-based distribution scheme to assign partitions of candidate ontologies to be matched by matcher threads. These threads can be running over multicore or multi-node platforms on single and multiple computing nodes. In a single node, matcher threads correspond to the number of available cores for the running instance. In multi-nodes, each node performs is its own parallel loading and internode control messages (m_c) are used to communicate regarding the ontology distribution and matching algo-

rithms. A set of matcher threads is assigned by ODI to execute matching algorithm instance (A_i) over individual ontology partitions. ODI also facilitates matcher threads with a thesaurus interface to assign thesaurus instance (T_i). Multiple dictionaries and thesaurus can be plugged into ODI via thesaurus interface. SPHeRe is currently using word-net dictionary [60] and also provides an open-end web-service interface to plugin web-based and remote dictionaries. Remote resources; however, can result in bandwidth issues and inflict performance degradation for matcher threads as multiple http requests will be generated for an individual case of possible concept matching. ODI also facilitates matching threads with a matching algorithm interface. Multiple algorithms can be plugged into ODI via matcher interface. Matcher threads are customized to use single and multiple matching algorithms concurrently on O'_s and O'_t subsets. Matched results (M_r) provided by matcher threads are submitted to accumulation and delivery phase for creation and delivery of bridge ontology (O_B).

4.2.3 Phase-III: Post-Matching

Ontology Aggregation Interface (OAI) accumulates M_r provided by matcher threads. OAI is responsible for O_B creation by combining M_r as mappings and delivering O_B via cloud storage platform. OAI offers a thread-safe mechanism for all matcher threads to submit their matched results. After completion of all matched threads, OAI invokes O_B creation process which accumulates all the matched results in a single O_B instance. In case of multi-node distribution, OAI also accumulates results from remote nodes after completion of their local matcher threads. O_B creation is customizable from single O_B per matching algorithm to an aggregated bridge ontology O_B from all matching algorithms running locally and remotely.

OAI also persists O_B in ontology cache for future matching requests. For unchanged O_S and O_T matching requests, Conditional System Flow II (illustrated in Fig 4.1) gets executed. In this execution flow; loading and management, and distribution and matching phases are skipped and URL to O_B is provided to the user. This mechanism prevents the system from performing redundant operations and preserves memory usage and CPU cycles.

4.3 Stack Design

Our proposed runtime has a layered architecture, following a stack design. With agility in mind, this design supports incremental development and over-the-time updates without propagating implementation changes across the system. The stack view of runtime's layers and components is illustrated in Fig 4.2. This stack is deployed as an integrated system on all participating nodes involved in ontology matching.

Our runtime 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 parallel platforms.

The core of our runtime consists upon six loosely coupled components (File IO, Init Daemon, Multi-node Distributor, Aggregator, Communication[1], 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 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 the matching process as matching requests over participating

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

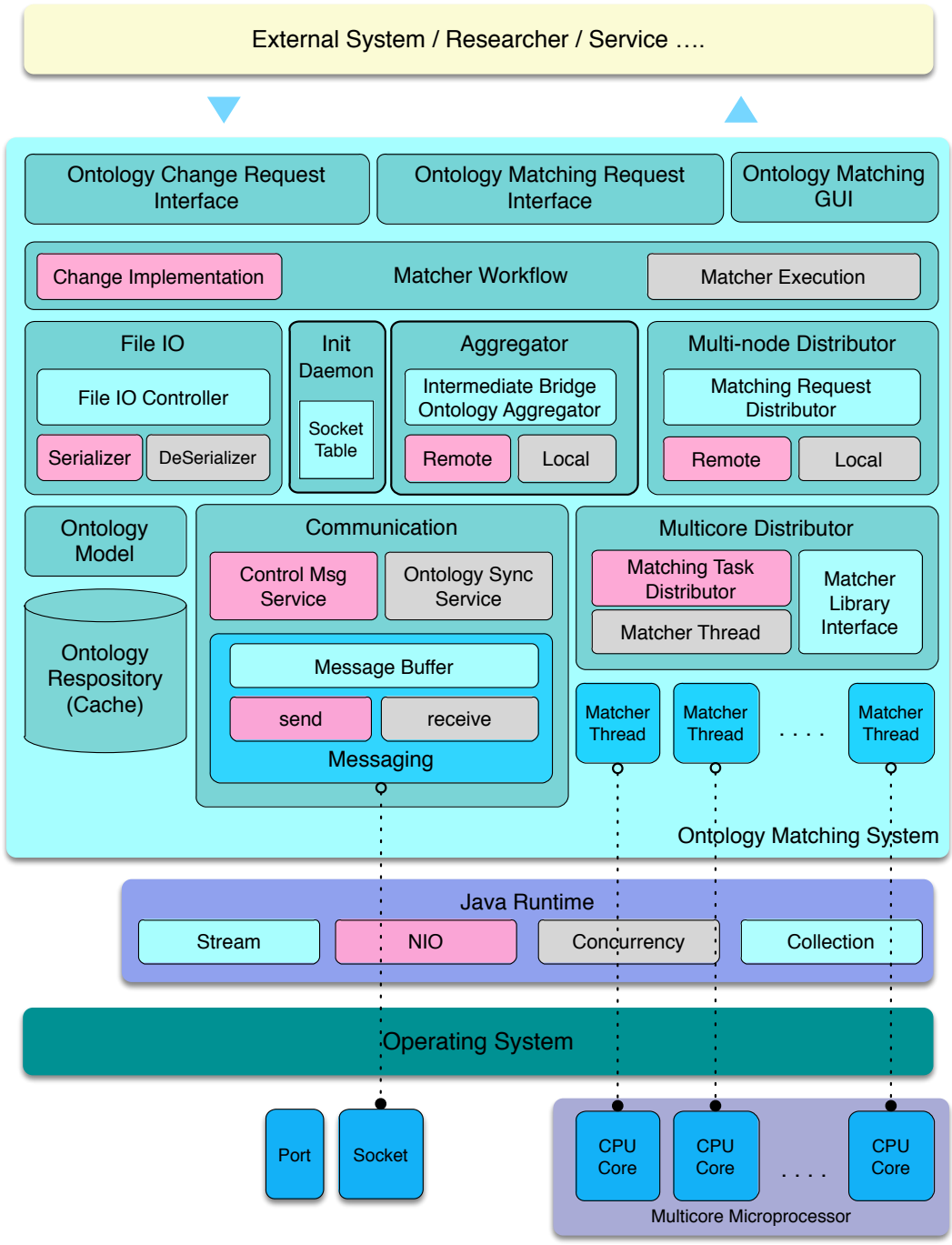


Figure 4.2: Stack Design

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 the participating nodes. For local distribution of matching tasks as matching jobs over available 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 parallel 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 runtime's implementation in Java and used its concurrency, collection, NIO and stream libraries for our benefit.

4.4 Core Component Details

This section provides details regarding the inner workings of the core components of the presented runtime.

4.4.1 Init Daemon

Initialization Daemon (Init Daemon) is responsible for creating the environment for the matching process. It executes in pre-matching stage of the system. On 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 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

Algorithm 1 Generate Socket Table algorithm

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

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 the communication level, sharing of UUIDs among the nodes is performed by a barrier read. This communication is illustrated in the sequence diagram of Fig 4.3. 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. Fig 4.4 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.

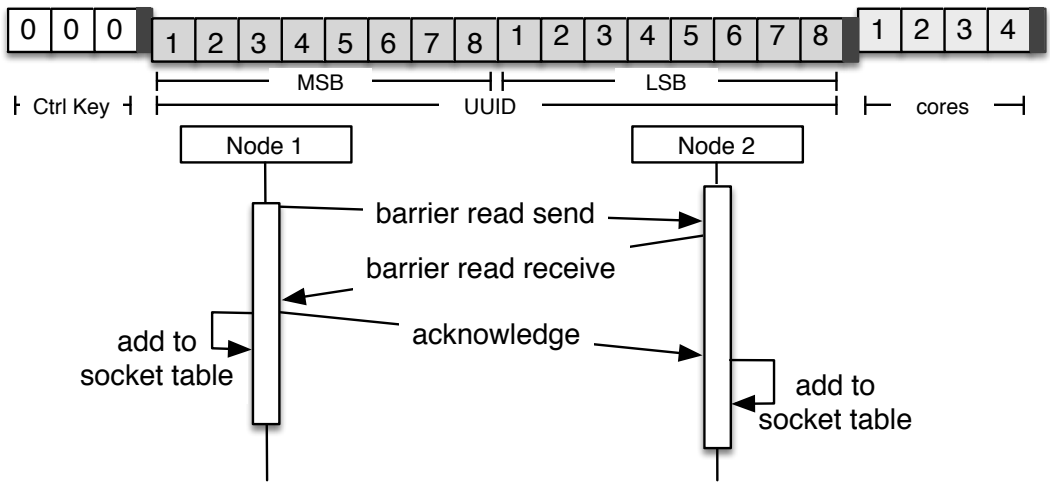


Figure 4.3: Barrier Read sequence diagram

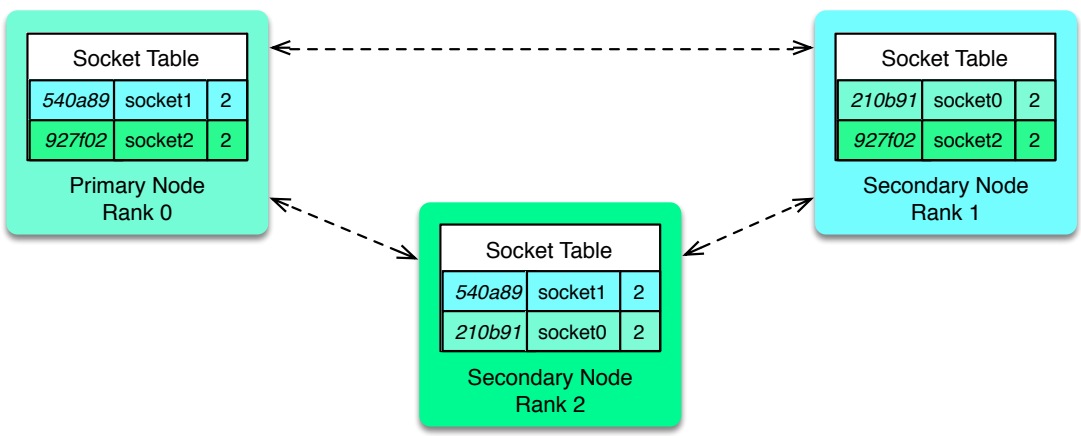


Figure 4.4: Socket Tables in a tri-node (2 cores/node) environment

4.4.2 Ontology Model

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

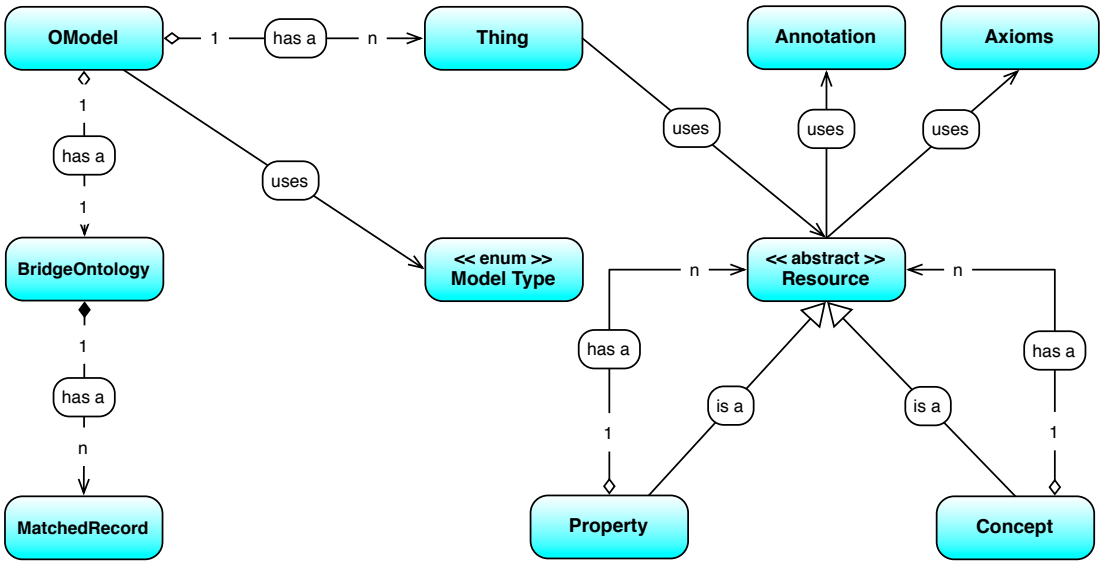


Figure 4.5: Ontology Model class diagram

tation, and testing stages. Ontology model represents the object model for serialized subsets of candidate ontologies. Furthermore, finalized bridge ontology is also serialized as ontology model by ontology cache. Ontology model's class diagram is illustrated in Fig 4.5.

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

$Resource$, $Property$, and $Concept$ implements Composite design pattern [81]. This implementation facilitates $Concept$ to mimic triples and $Property$ to aggregate objects of its own

type, providing a self-containing-object data structure. $Resource$ abstract class is extended by $Concept$ and $Property$ object which aggregates itself as sub-concepts and sub-properties respectively. $Concept$ and $Property$ object also provides iterators to their respective associated instances, i.e., providing $Concept$ with a concept name will return the required $Concept$ object, its sub-concepts, annotations, axioms, and associated properties. $Annotation$ object encapsulates associated labels and comments to $Resource$. $Axiom$ object encapsulates associated constraints to $Resource$.

$MatchedRecord$ represents a single matched result (concept names and similarity) evaluated as a matching task. $BridgeOntology$ contains a thread-safe collection of $MatchedRecord$ objects fed by individual matcher threads. $Thing$ object is an aggregation to a single ontology; however, $OModel$ object provides a higher level aggregation over multiple ontologies (collection of $Thing$) and their corresponding O_B.

4.4.3 File IO

File IO component is responsible for ontology loading, subset creation, and providing an interface to the ontology repository for ontology persistence. It also executes in pre-matching stage. File IO provides serialization and deserialization operations. When the runtime receives a new ontology, i.e., a candidate ontology that has not been converted into subsets, file IO parses it to create its 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 on 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 information provided by the init daemon. Communication between primary and secondary node(s) for subset replication is illustrated in the sequence diagram of 4.6. Unlike barrier read, control messaging for ontology subset replication is a two-step process. Firstly, primary node sends a 24 byte message to the secondary node(s) containing ontology UUID (16 byte), size of the subset to be sent (4 bytes) and ctrl key (3 bits). Secondary

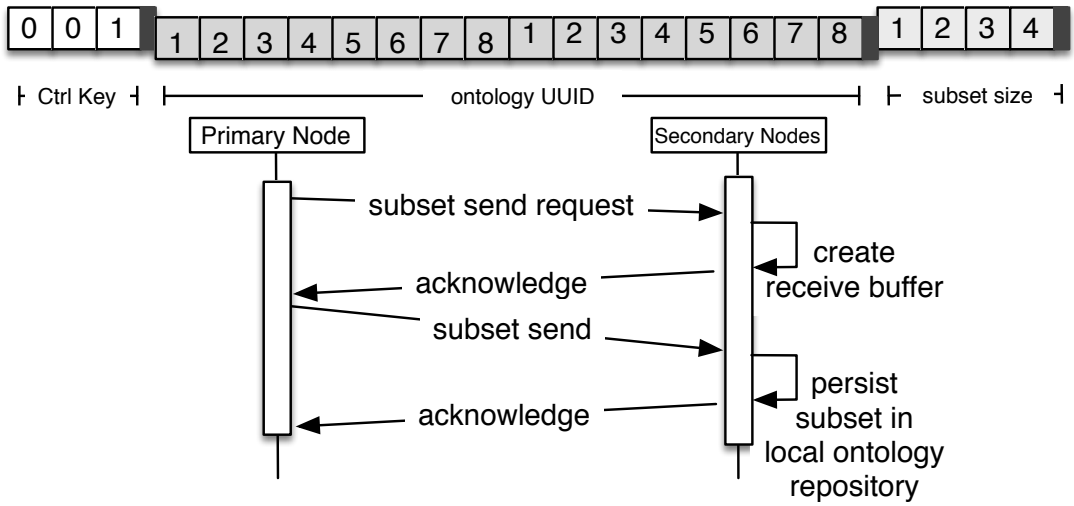


Figure 4.6: Ontology Subset Replication sequence diagram

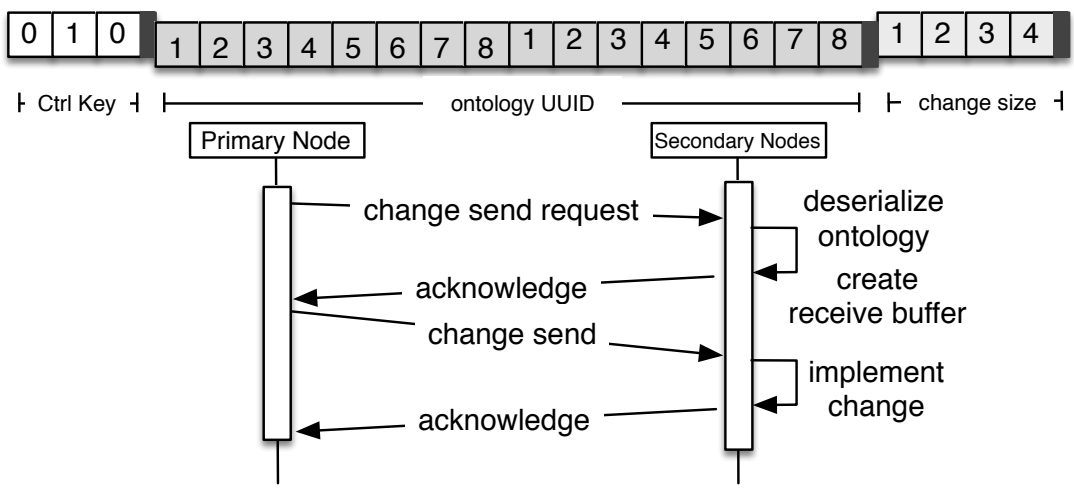


Figure 4.7: Ontology Change Request sequence diagram

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. 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 4.7. 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.

Class diagram for file IO component described in Fig 4.8. Classes are packaged into two categories; (i) ontology loading and (ii) ontology management. Ontology loading package is responsible for parsing new ontologies into ontology models, serializing, and deserializing them for distribution and matching.

Candidate ontologies are received via OLI. $Utility$ class is used by OLI to calculate hash values for candidate ontologies and match the values against persisted ontologies hash values in ontology cache. If serialized versions of any or both candidate ontologies are already present in ontology cache, OLI invokes $Deserializer$ object for loading O'_s and O'_t in parallel for distribution and matching phase. Algorithm 2 for owlLoad method describes this process.

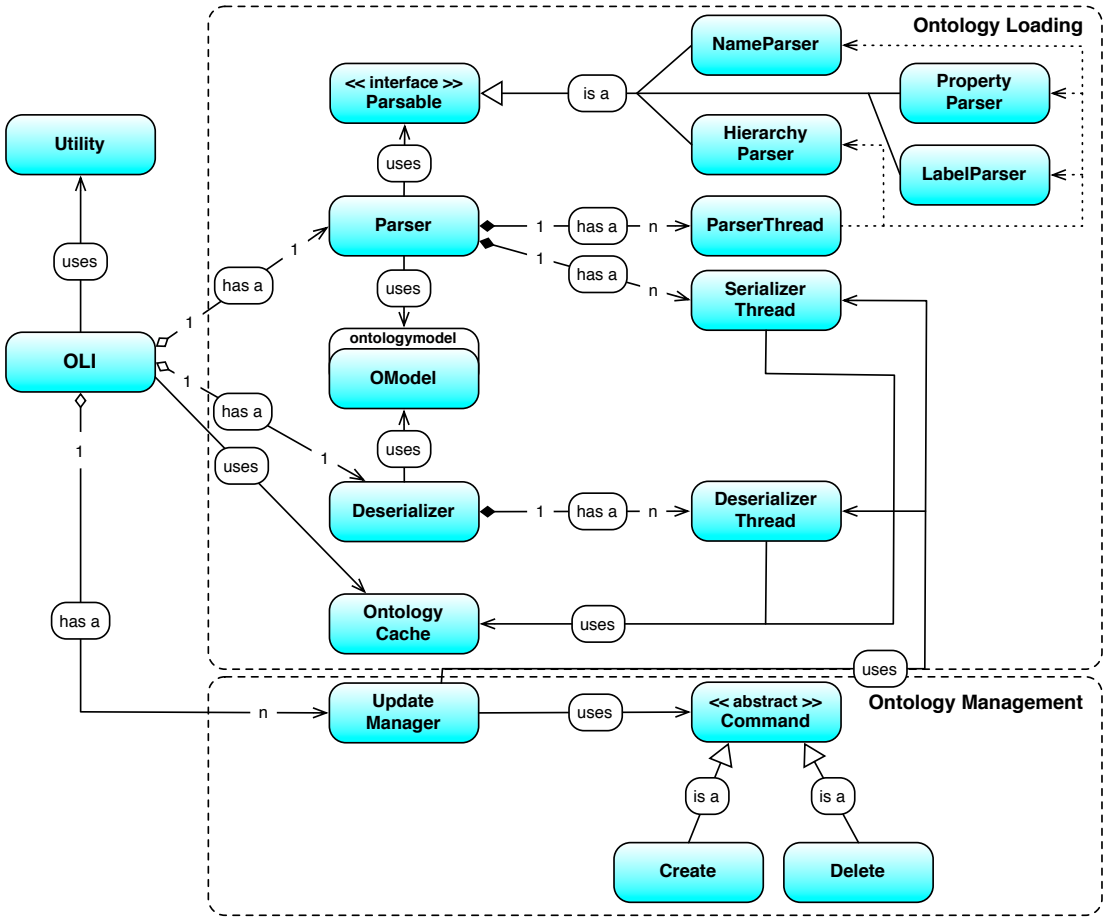


Figure 4.8: File IO class diagram

In case of absence of serialized versions for candidate ontologies, $Parser$ object is invoked. $Parser$ object has a composition relationship with $ParserThread$ and $SerializerThread$ objects, i.e., parser and serializer threads are created with the creation of $Parser$ object. Number of threads to be created for parsing and serialization can be customized by request, by default it is the number of cores on an available processor; however, a single thread parses a single subset of O_x. $Parser$ object loads the ontology in memory and assign parsing algorithms to parser threads via $Parsable$ Interface. $Parsable$ is an implementation of Strategy design pattern [81]. Parsing algorithms can be plugged according to the needs, adding to the agility, extensibility, and customization of the system over time.

Four algorithms currently implement $Parsable$ interface: (i) $Name$ parser; (ii) $Hierarchy$

Algorithm 2 OwlLoad algorithm

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

Algorithm 3 NameParser algorithm

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

parser; (iii) $Label$ parser; and (iv) $Property$ parser. $Name$, $Label$, and $Property$ parser provide implementation by reading the class-names (described in Algorithm 3, their labels (described in Algorithm 4) and properties (described in Algorithm 5) respectively. However, the hierarchy parser implements multi-step bottom-up ontology parsing approach. All classes from the ontology are read with reference to their parent classes. A class may not have a child; however, every class has a parent. Parent class references are maintained by every class in the hierarchy. Algorithm 6 describes the implementation of hierarchy parser.

UpdateManager is part of ontology management package with a command pattern [81] implementation. Apart from agility, this pattern provides undo and redo operations for ontology subset

Algorithm 4 LabelParser algorithm

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

Algorithm 5 PropertyParser algorithm

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

change implementation.

Algorithm 6 HierarchyParser algorithm

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

4.4.4 Distributor

Distributor components (multicore and multi-node distributors) are collectively responsible for the distribution of 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 4.9. 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

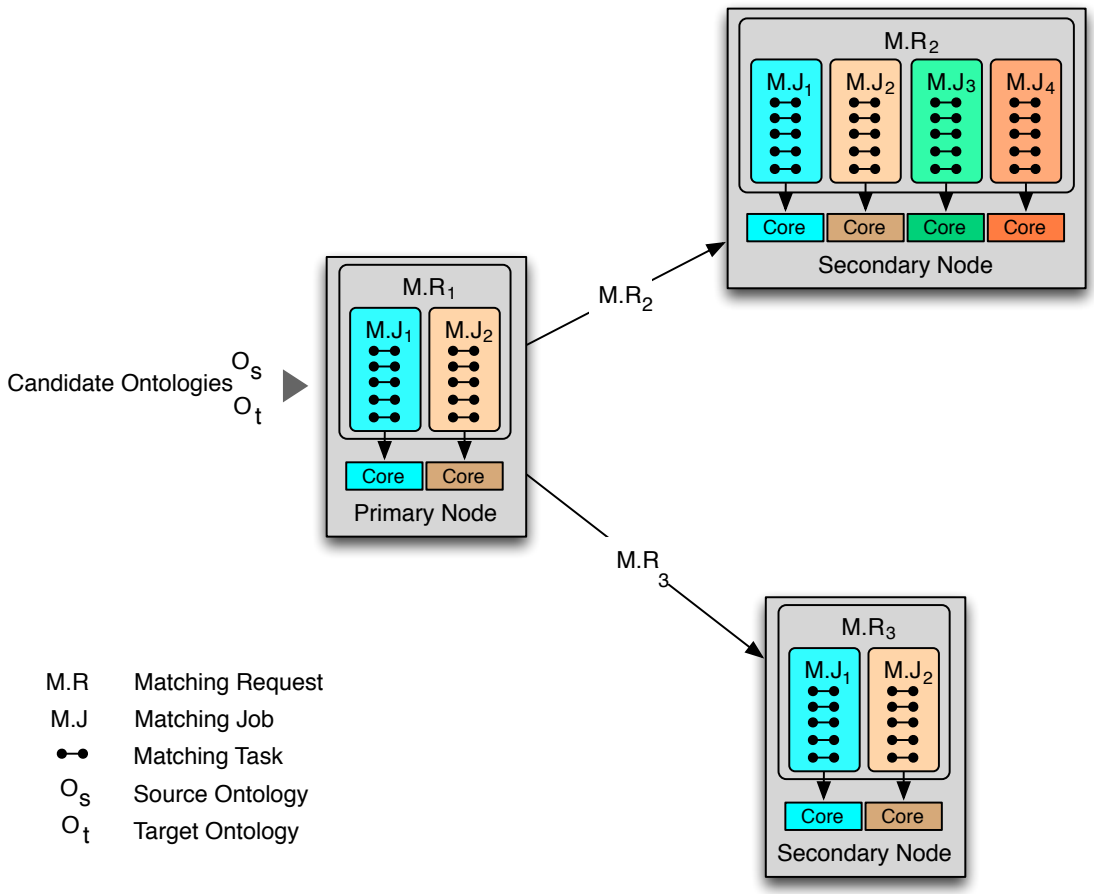


Figure 4.9: Matching Request Distribution in a tri-node environment

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.

In the case of single-node, distribution process scales down to multiple cores on one node. 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 7, 8, and 9 describe the distribution of matching tasks on single- and multi-node platforms. In the case of single-node platform, multicore distributor (Algorithm 8) is invoked. It identifies the number of participating cores from the native runtime and calculates the partition slab

Algorithm 7 Distributor algorithm

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

Algorithm 8 Multicore Distributor algorithm

Require: $nodes > 0$
  $cores \leftarrow$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 $i = 1$ **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

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 8 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,

Algorithm 9 Multi-node Distributor algorithm

Require: $nodes > 1$
  $nodes \leftarrow$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

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 9) 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 4.10. To elaborate the execution of Algorithm 9, consider the example of matching process between two biomedical ontologies, “adult mouse anatomy (2,744 concepts)” with “NCI

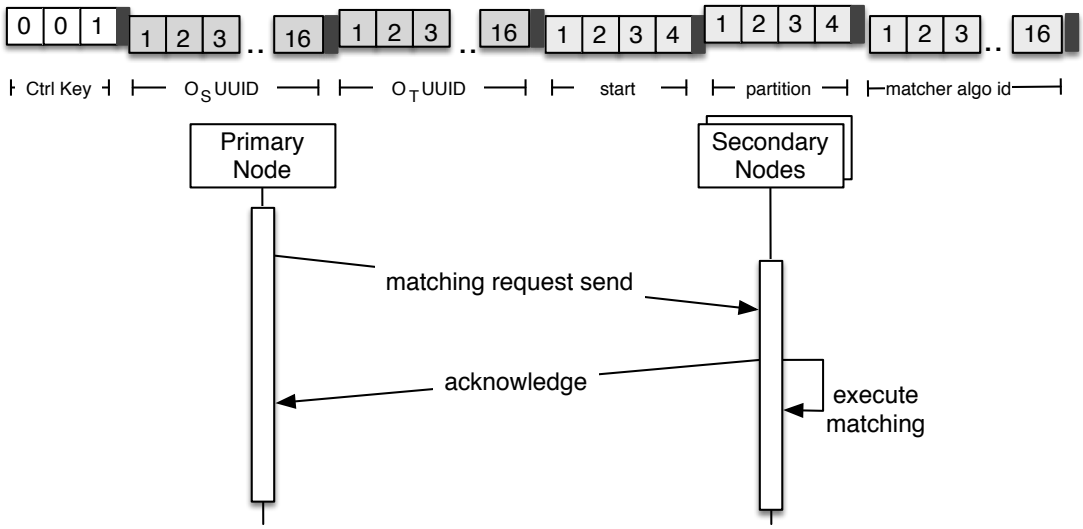


Figure 4.10: Ontology Matching Request sequence diagram

human anatomy (3,304 concepts)" over the tri-node environment illustrated in Fig 4.9. Algorithm 9 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 8) 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 4.10. and received by the multicore distributors (Algorithm 8) 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 distribut-

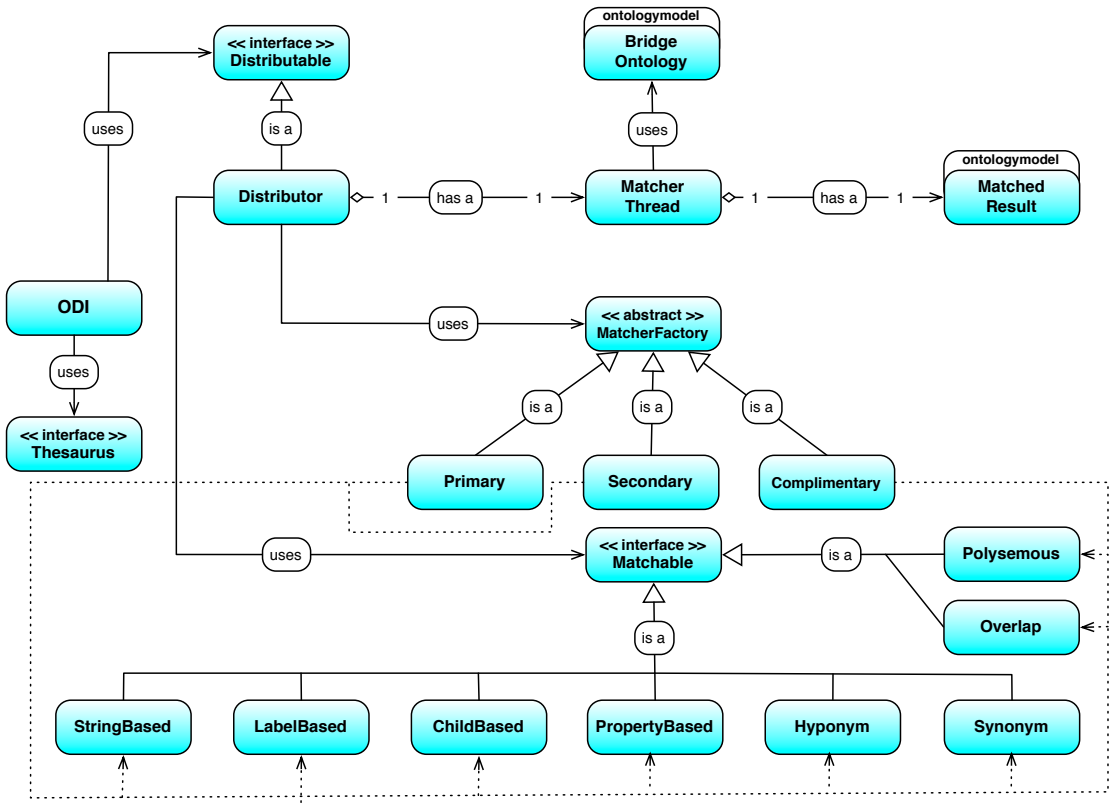


Figure 4.11: Distributor Components class diagram

ing 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. Although distribution slab calculated by Algorithm 9 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, idle core can exist as one node might complete its matching request prior to the others.

Distributor components also provide an interface to 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, 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.

The class diagram of distributor compoenents is described in Fig 4.11. O'_x are fed to ODI that invokes a distribution strategy from $Distributable$ interface. $Distributor$ object implements $Distributable$ interface and encapsulates distribution algorithm. $Distributor$ object currently implements static distributor based on ontology size. Size-based partitioning is an optimal mechanism for distributing matching tasks among available computational resources, as every resource gets an equal share of workload. $Distributor$ object and $Distributable$ interface is an implementation of Strategy design pattern which increases the flexibility and extensibility of the system by providing interface to add more distributor objects over time without propagating changes.

$Distributor$ object has two responsibilities: (i) Create matcher threads. $Distributor$ evaluates the current runtime for the availability of computation resources and creates multithreaded environment of matcher threads to execute. Each $MatcherThread$ records a $MatchedResult$ object of ontology model in a thread-safe collection of matched records in $BridgeOntology$ object. (ii) Assign matching algorithms to matcher threads. $Distributor$ implements two design patterns, i.e., Abstract Factory [81] and strategy to assign instances of matcher algorithms to individual matcher threads.

Matching algorithms are implemented with two levels of abstraction, a fine grain implementation of matching algorithms and a coarse grain aggregation of family of matching algorithms based on utilization. Individual matching algorithms implement $Matchable$ interface. Strategy pattern is again utilized here for systems flexibility and extensibility. Plug-n-play nature of strategy pattern facilities the system to add more algorithms for matching to improve accuracy. Coarse grain aggregation of individual algorithms is implemented via $MatcherFactory$ (Abstract Factory). Individual algorithms are classified into three categories; $Primary$, $Secondary$, and $Complimentary$ algorithms. $Primary$ family instantiates a set of algorithms that are executed for every matching request that currently are; $StringBased$, $LabelBased$, and $ChildBased$. $Secondary$ family instantiates a set of algorithms that are executed by request to dig deeper into ontologies for higher accuracy, this set currently includes, $PropertyBased$, $Synonym$, and $Hyponym$ algorithms.

$Complementary$ family instantiates a set of algorithms that are executed by request in context with domain; for example, two ontologies from medical domain have higher chances of matchable concepts, so by executing complimentary set of algorithms might contribute in the accuracy. These algorithms currently are $Overlap$ and $Polysemous$.

4.4.5 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 platform (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.

4.5 Summary

This chapter presented the implementation details of the high performance ontology matching runtime. The core components built for performance of the runtime are based on space and time efficiencies achieved by the presented methodology. This runtime is particularly designed and built for parallel platforms with abstractions of parallelism at the ontology matching process and thread-level parallelism at the core.

Chapter 5

Ontology Matching as a Service and a Platform

Over the recent years, semantic web technologies especially ontologies are contributing in data and information systems for greater benefit. For example, in the field of biomedical sciences, these ontologies are getting used for annotation of medical records [82], standardization of medical data formats [10], medical knowledge representation and sharing, clinical guidelines (CG) management [9], clinical data integration and medical decision making [11]. Therefore, biomedical community has in depth ontology repository like Open Biomedical Ontologies (OBO) [16]; furthermore, biomedical ontologies like the Gene Ontology (GO) [12], the National Cancer Institute Thesaurus (NCI) [13], the Foundation Model of Anatomy (FMA) [14], and the Systemized Nomenclature of Medicine (SNOMED-CT) [15] have emerged.

Similarly in other domains of sciences, use of ontologies has increased by large [36]. They are vastly becoming accustomed in information systems, social networks, search engines, and e-commerce. As a consequence of this vast usage, researchers and developers are investing more time in generating more and comprehensive ontologies. Adjacent to this vast usage of ontologies, their matching requests have also gained momentum to drive overlapping information. Utilization of this information is necessary for the integration, aggregation, and interoperability; for example, the plethora of web-based medical information resources provides related information over the Internet. If these resources are annotated by ontologies, software agents can automatically aggregate information for biomedical professionals and biomedical querying systems. For example, NCI ontology defines the concept of “Myocardium” related to the concept “Cardiac Muscle Tissue”, which describes the muscles surrounding the human heart. Concept “Cardiac Muscle Tissue” is defined in FMA ontology; therefore, a biomedical professional or a system integrating knowledge regarding human heart requires mappings between candidate ontologies FMA and

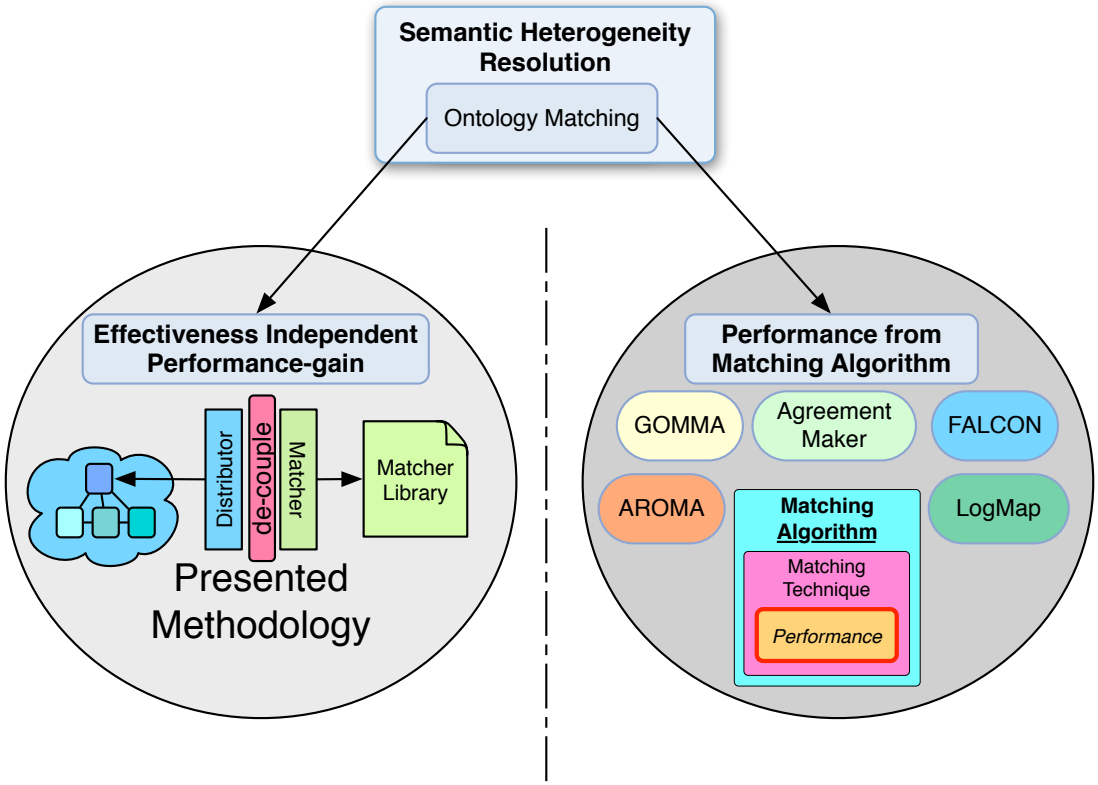


Figure 5.1: Classification between Matching Library and Performance

NCI [58]. Likewise, GO is a highly organized structure of medical knowledge facilitating medical genetics. It is widely used by biomedical researchers in numerous genetical research fields including gene group-based analysis for discovering the hidden links overlooked by the single-gene analysis [83]. Finding mappings between GO ontology and FMA ontology can be used by molecular biologist in understanding the outcome of proteomics and genomics in a large-scale anatomic view [84]. Moreover, mappings by ontology matching have also been used for heterogeneity resolution among various health standards [85].

Ontology matching systems developed over the years have resolutions for the ontology matching problem in isolated execution environments. Most the systems are monolithic implementations with least reusability in terms of result and platform sharing. Consequently, initiatives like SOMET [86] and OntoMediate [87] were proposed. These initiatives encourage collaborative

ontology matching environments, where semantic-web experts can participate with interventions. These projects did not evolve and were decommissioned due to their semi-automatic and purely design time implementation. Furthermore, handling complex and large ontologies was not catered. In contrast, technology has changed considerably over the years with the implementation of cloud computing. Ontology matching systems with confined deployments, limited computational abilities, and scalability of local computational resources can extend themselves for ubiquitous access. Therefore, an opportunity emerges for implementing an ontology matching platform that should not be confined as a localized deployment. Presented methodology avails this opportunity and extends its high performance ontology matching runtime with interfaces at service level and platform level. Consequently, this approach is non-monolithic in terms of resource and result sharing for researchers, developers, semantic web experts and systems to benefit from.

Fig 5.1 illustrates presented approach that provides explicit classification between ontology matching algorithms and performance-gain initiatives in contrast with current ontology matching systems. Presented approach provides an ideal environment where matching runtime deployed over cloud platforms is exposed by interfaces at service and platform level, and matching libraries and algorithms can be plugged-in for execution and evaluation.

5.1 Ontology Matching over Cloud Platforms

The deployment of presented runtime over cloud is illustrated as a high-level stack-like diagram in Fig 5.2. The primary objective of this runtime is to exploit the available computational resources of cloud's parallel platform and provide a service-based interaction, i.e., ontology matching as a service, taking the benefit of the ubiquitous nature of the cloud platform. Furthermore, providing decoupled interface between performance runtime and matching algorithms, i.e., ontology matching as a platform.

5.1.1 Ontology Matching as a Service

Request of matching ontologies can be generated from several resources including, developers and researchers, information systems, or even third-party information services running over other plat-

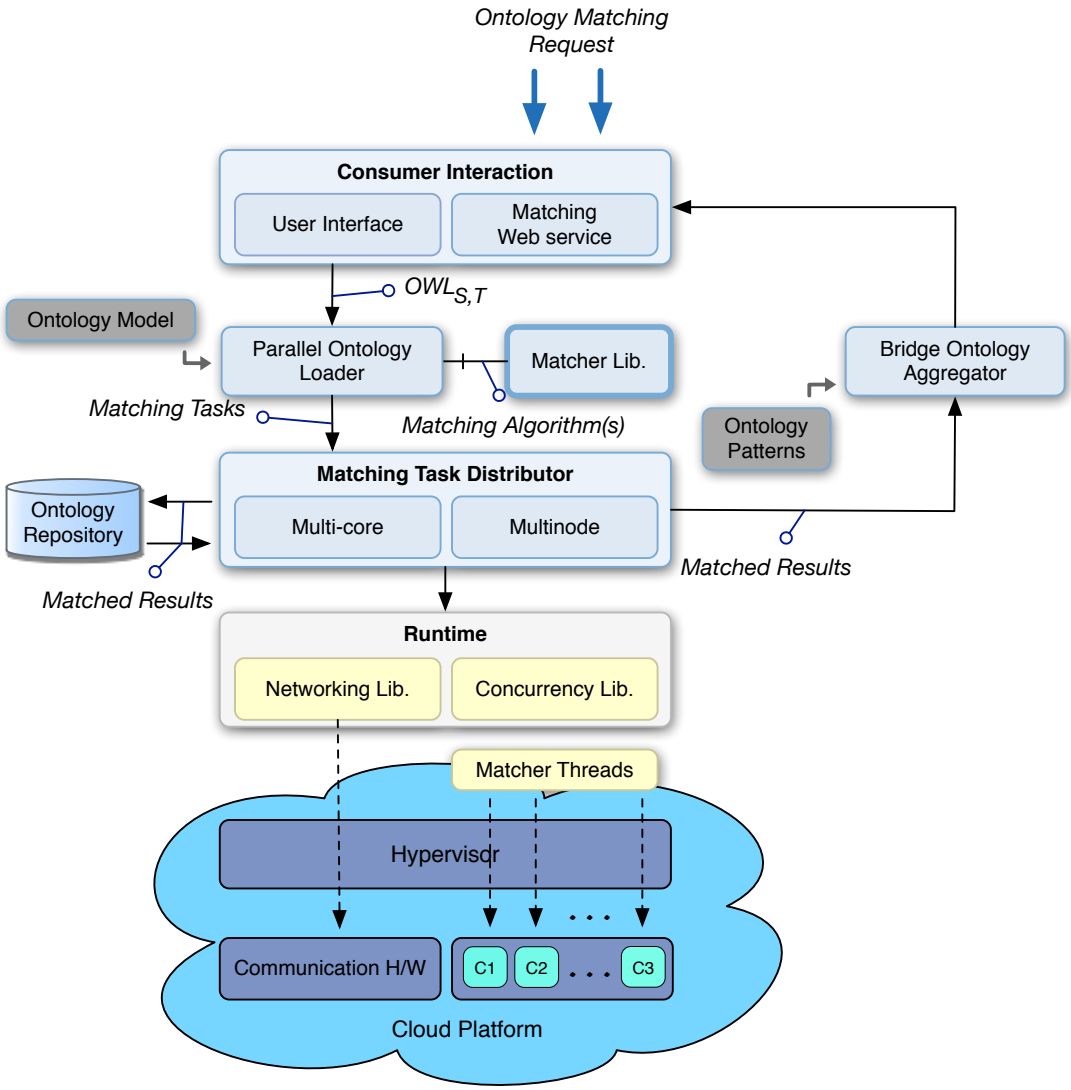


Figure 5.2: Ontology Matching as a Service and a Platform

forms. Match request encapsulates the ontologies to be matched as source and target ontologies. Matched results are returned to the consumer as bridge ontology.

Starting from the top of the stack illustrated in 5.2, Consumer Interaction component provides an ontology matching RESTful [88] web service for clients to consume. The matching service provides four trivial methods as service bindings for consumption.

1. match (sourceOntologyURI/File, targetOntologyURI/File)

2. match (sourceOntologyURI/File, targetOntologyURI/File, returnEmail)

3. match (sourceOntologyURI/File, targetOntologyURI/File, matchingAlgorithms [ ])

4. match (sourceOntologyURI/File, targetOntologyURI/File, matchingAlgorithms [ ], returnEmail)

Among the arguments, collection of matching algorithms and return email are extended parameters used for matching request customization. In case of first request, all the algorithms present in the matching library will execute. This matching will take more time; however, will have higher accuracy. Incase of trivial and far less complicated ontologies, consumer can select the matching algorithms to be executed as collection of matching algorithms. For large-scale ontologies, where the evaluation time can exceed from 20 minutes or later, URL of the bridge ontology to-be is provided and can be returned over a particular email address. After matching, the active URL will reference the bridge ontology.

Adjacent to the Web service, Consumer Interaction component encapsulates the matching web service in a user interface (UI). This UI provides a web-based direct interaction between a developer or a researcher who wants to benefit from matching service.

Parallel Ontology Loading benefits from the multicore nature of cloud instances and loads the source and target ontologies by utilizing the performance-based ontology loading and subset generation, i.e., ontologies are parsed in parallel and populated in multiple thread-safe ontology model objects. Each object encapsulates the information required by a single matching algorithm during runtime. Furthermore, redundancy like URI based names of concepts etc., is removed during this process. This prevents the runtime from loading un-necessary and redundant information in main memory during execution, avoiding memory strains during the matching process.

Matcher Library provides a library of ontology matching algorithm. These algorithms are classified into primary, secondary, and complementary type. Primary algorithms execute for every matching request, secondary algorithms execute for higher accuracy, and complimentary algorithms execute with respect of ontology scope. Matcher Library also utilizes external third-party resources, i.e., WordNet [60] and UMLS [89] for higher accuracy in secondary and complementary type algorithms. However, by service, client can override the default matching algorithm

execution model of the runtime.

Matching Task Distributor partitions the candidate ontologies as subsets and assigns over to the computing cores available. For local resources, matcher threads are assigned to perform parallel matching invoking available cores by multicore distributor. For remote resources, control messages are generated for participating nodes regarding their chunk of partition to work and matching algorithm to execute by multi-node distributor. Each node after receiving the control message loads performs parallel matching over their available computing cores.

Every participating node(s) generates their respective matched results. Bridge ontology aggregator, accumulates these results and generate a bridge ontology file. Bridge ontology aggregator provides an interface to bridge ontology patterns to be used for pattern-based bridge ontology generation. Bridge ontology file is returned as a response or a URL to physical file to the consumer. This ontology is also be persisted in ontology repository for future use in case of same matching requests.

5.1.2 Ontology Matching as a Platform

Ontology matching as a platform focuses on the actual deployment and decoupling interface between the library of matching algorithms and the performance runtime. This interface is called *Matchable* interface which provides the implementation of the Strategy pattern [81]. As illustrated in the class diagram 5.3, strategy context of matchable interface is a polymorphic instance that is initialized by the implementation of the matching algorithm to be executed at runtime. At design time, matching algorithm experts implement the *Matchable* interface by overriding match methods. The matchable interface provides weak association relationship between the ontology model and the matching algorithm implementation.

Source code in Fig 5.4 describes the implementation of a string based matching algorithm built by using the *Matchable* interface. The overloaded method *match* is implemented by the matching algorithm author. From the signature, arguments *start* and *end* are provided as iterator boundaries. These boundaries play a significant role during the creation of a matching task; however, it is quite clear from the code that the implementation details of partitioning by start and end are completely hidden from the algorithm author.

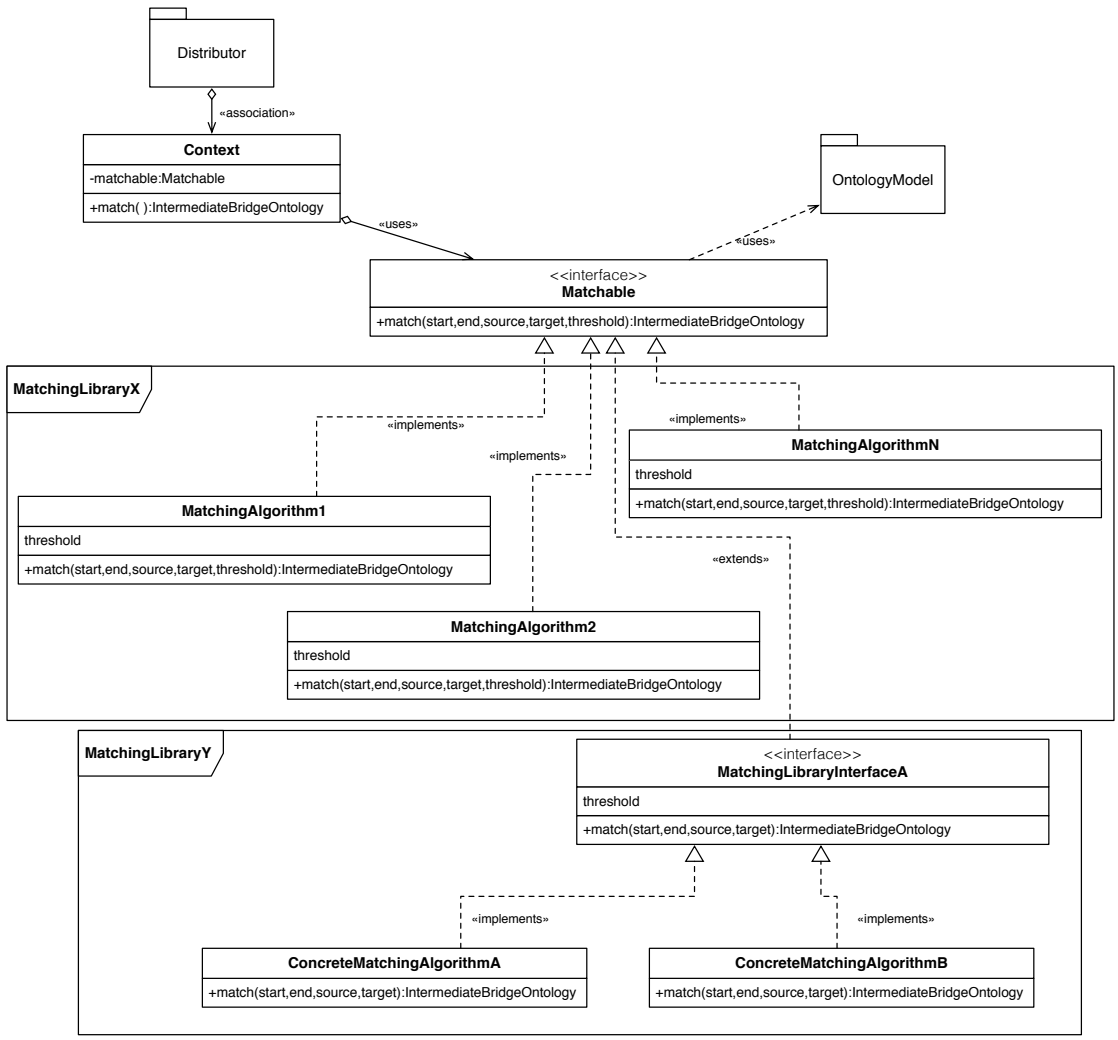


Figure 5.3: Matchable Interface and Matching Libraries

Signature arguments *source* and *target* provide reference to the candidate ontologies to be matched by the implemented matching algorithm. These references are instances of ontology model objects representing the individual candidate ontologies. Algorithm authors can implement iterators and utilize accessor methods of the ontology model to retrieve ontology resources for matching.

The *threshold* argument is overridden by the author to provide the break point where an entity is considered either to be as a mapping or not. In this implementation where editdistance [90] based matching is used, threshold has been considered an instance variable where a client of this

```
package com.sphere.matchinglibrary;

import com.sphere.aggregator.bridge.IntermediateBridgeOntology;
import com.sphere.aggregator.bridge.BridgeRecord;
import com.sphere.matchinglibrary.utility.EditDistance;
import com.sphere.ontology.model.Concept;
import com.sphere.ontology.model.Root;

/**
 * Created by Jason TY on 6/30/2014.
 */
public class StringBridge implements Matchable {

    private double threshold = 0.0;
    private String id = "StringBridge";
    IntermediateBridgeOntology intermediateBridgeOntology = new IntermediateBridgeOntology();

    @Override
    public IntermediateBridgeOntology match(int start, int end, Root source, Root target, double threshold) {
        this.threshold = threshold;
        try {
            for (int i=start;i<end;i++){
                Concept sourceConcept = source.getConcepts().get(i);
                if (sourceConcept.IsMatched()) continue;
                double editDistance = 0.0;
                String sourceName = sourceConcept.getName();
                BridgeRecord bridgeRecord = new BridgeRecord("","","","", 0.0,id);
                for (Concept targetConcept : target.getConcepts()) {
                    if (targetConcept.IsMatched()) continue;
                    String targetName = targetConcept.getName();

                    if (sourceName.length() == targetName.length()) {
                        if (sourceName.equals(targetName)) {
                            // its a bridge
                            bridgeRecord.setBridgeRecord(sourceConcept.getNameWithUri(), sourceName,
                                    targetConcept.getNameWithUri(), targetName, 1.0, id);
                            targetConcept.setMatched(true);
                            break;
                        } else {
                            editDistance = EditDistance.getSimilarity(sourceName, targetName);
                            if (editDistance >= threshold) {
                                if (editDistance > bridgeRecord.getEditDistance()) {
                                    bridgeRecord.setBridgeRecord(sourceConcept.getNameWithUri(), sourceName,
                                            targetConcept.getNameWithUri(),targetName, editDistance, id);
                                }
                                continue;
                            }
                        }
                    } else {
                        editDistance = EditDistance.getSimilarity(sourceName, targetName);
                        if (editDistance >= threshold) {
                            if (editDistance > bridgeRecord.getEditDistance()) {
                                bridgeRecord.setBridgeRecord(sourceConcept.getNameWithUri(), sourceName,
                                        targetConcept.getNameWithUri(), targetName, editDistance, id);
                            }
                            continue;
                        }
                    }
                }
                if (bridgeRecord.getEditDistance() != 0.0) {
                    sourceConcept.setMatched(true);
                    intermediateBridgeOntology.addBridgeRecord(bridgeRecord);
                }
            }
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
        return intermediateBridgeOntology;
    }
}
```

Figure 5.4: String Bridge Matching algorithm using Matchable Interface

algorithm can provide its own threshold value.

Return type of *match* method is an object of *IntermediateBridgeOntology* from ontology model. *IntermediateBridgeOntology* is a collection object with mapping instances. Each mapping instance is an abstraction of a source concept, target concept, matchable operation, and threshold.

Matching algorithm experts and authors can use provided templates for their own algorithm implementation or can use matchable interface signatures for their custom implementations. From the provided example in Fig 5.4, it is quite evident that even though this algorithm will execute over a performance-based ontology matching runtime presented in earlier chapter with parallel and distributed matching; however, the authors of the algorithms do not provide any parallelism or performance-like implementation in their algorithm. The performance-based execution is completely hidden by a facade of *Matchable* interface, thus preserving the accuracy of the algorithm with performance-gain at execution time.

5.2 Summary

In this chapter, we presented our non-monolithic resolution for ontology matching for sharing matched results with clients and platform with semantic-web experts. Our approach provides a RESTful matching service for heterogeneity resolution clients and a Matchable interface for matching libraries to be plugged-in for execution and evaluation. With the parallel and distributed matching ability of the runtime, service requests are executed and evaluated over parallel platforms.

Chapter 6

Evaluations and Discussions

In this section, we describe a comprehensive experimentation performed on our proposed methodology. For the evaluation, we have used the datasets of real world ontologies by OAEI 2012 [91] and 2013 [92]. Our methodology is implemented as a runtime to our ontology matching system SPHeRe. This system is evaluated over Anatomy, Library, Large-scale Biomedical, and Small-scale Conference Tracks of OAEI 2012 and 2013's datasets. The candidate ontologies used in these tracks are of various sizes, covering the different magnitudes of ontology matching problems. Candidate bottleneck areas of performance-based ontology matching are benchmarked individually in complement to the end-to-end performance measurement of SPHeRe's runtime.

We have executed three different libraries of ontology matching algorithms (computational complexity $\geq O(n^2)$) provided to us by different semantic web experts. Evaluation is performed over two parallel 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) [93] CPU (2 threads/core) with 16 GB memory, Java 1.8 [94] and Windows 7 64 bit OS, and (ii) a public cloud Microsoft Azure instance with two virtual machine (VM) configurations: (i) 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 (ii) Standard 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.

6.1 Load Time and Memory footprint evaluation

Ontology loading time is a candidate performance bottleneck that is resolved in presented methodology by subset generation, serialization and deserialization of candidate ontologies in parallel. Furthermore, the algorithm-based ontology subset generation and eager matching space reduction,

minimizes the memory stress with a smaller memory footprint throughout the ontology matching process. In this subsection, we compare SPHeRe's ontology loading time and memory footprint against two most widely used OWL frameworks Jena and OWL API. Whole NCI, whole FMA, and small SNOMED CT are used as candidate ontologies for this evaluation. For precision, this evaluation has been repeatedly executed for 100 iterations.

As mentioned in Section 4.4.2, SPHeRe has its own implementation of the ontology model, built for performance, thread-safety and scalability. *Parser* and *Deserializer* objects of loading and management component are equipped to parse OWL files and its serialized subsets to populate the ontology model. Already available and most widely used Apache's Java-based framework for semantic web applications Jena is provided with an ontology parser. Development of Jena is focused around its strongest component, i.e., Inference API. To facilitate Inference API, all other components including the parser are built for inference support. Jena provides an object model (OntModel); however, firstly, due to its parser's memory hungry implementation, overflow in JVM heap while working with larger scale ontologies occurs quite often. During our stress testing on Jena's parser by loading FMA with NCI ontology for matching, JVM heap crashes occurred even after providing a 2GB of heap memory to the virtual machine. Secondly, Jena's OntModel and APIs are not thread-safe [95] resulting in consistency and throughput bottleneck issues for applications utilizing multithreaded execution models. Concurrency in this respect can lead to reduction in performance instead. Thirdly, Jena is built with graph-based OntModel, resulting an overload of not required information even for trivial operations like class retrieval. The cost of information retrieval with respect to memory footprint and retrieval time in this regard is very high [96]. These bottlenecks and implausible nature of Jena's parser and OntModel makes it ill-equipped to integrate with SPHeRe's performance oriented parallel ontology matching techniques and provides a valid justification to implement a generic yet precise ontology model with performance in mind.

Another available and most widely used framework for parsing OWL is OWL API [97], which also provides an ontology model for information retrieval. Contrary to Jena, OWL API and its ontology model has a lighter memory footprint, and cost of ontology loading and information retrieval is far efficient. OWL API can be an excellent candidate; however, the ontology model for OWL API is also not thread-safe [98], leading to the options of either building a thread-safe

Table 6.1: Completeness of Ontology Model

Ontology Name	Type	**Actual No. of Concepts**	**No. of Concepts Loaded**
cmt	Tool	36	36
iasted	Web	140	140
conference	Tool	57	57
edas	Tool	104	104
confof	Tool	38	38
sigkdd	Web	49	49
ekaw	Insider	74	74
Human Anatomy (2012-13)	Anatomy	3,289	3,289
Human Anatomy (2013-14)	Anatomy	3,304	3,304
Mouse Anatomy (2012-13)	Anatomy	2,737	2,737
Mouse Anatomy (2012-13)	Anatomy	2,744	2,744
TheSoz	Library	8,376	8,376
STW	Library	6,575	6,575
Whole FMA	Biomedical	78,989	78,989
Whole NCI	Biomedical	66,724	66,724
SNOMED-CT Small Seg.	Biomedical	49,622	49,622
Whole FMA 5%	Biomedical	3,696	3,696
Whole NCI 10%	Biomedical	6,488	6,488
Whole FMA 13%	Biomedical	10,157	10,157
SNOMED-CT 5%	Biomedical	13,412	13,412
SNOMED-CT 40%	Biomedical	122,464	122,464
SNOMED-CT 17%	Biomedical	51,128	51,128
Whole NCI 36%	Biomedical	23,958	23,958

ontology model with an associated parser for parallelism needs, this approach allows to have more control over the model or updating OWL API and its ontology model for thread-safety. For our current implementation, SPHeRe provides its own thread-safe ontology model (described in Section 4.2.2) by using OWL API's parsing components.

SPHeRe's ontology model implementation has already been evaluated by experts for completeness. Table 6.1 provides details regarding its completeness over OAEI's ontology dataset.

SPHeRe implements parallel loading for serialized subsets of the OWL files. Fig 6.1 describes the loading time of serialized subsets of candidate ontologies in parallel. The longest time taken is compared to the load time taken by Jena and OWL API, described in Fig 6.2. Loading time in this evaluation is the total of time taken by a system to load ontology in the memory and retrieve all of its classes. Performance of class retrieval time is directly related to the time consumed over concept matching, providing a significant impact on overall system's performance. Because of the subsets and serialized nature of SPHeRe's ontology cache, it achieves better performance with its comparable systems. SPHeRe only parses the candidate ontologies for the first time. For every

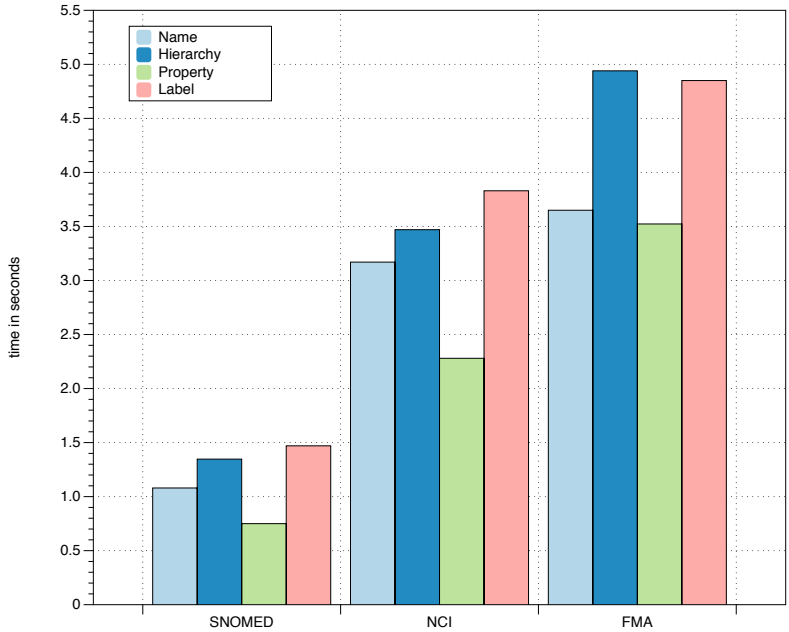


Figure 6.1: Ontology Load time for Serialized Subsets

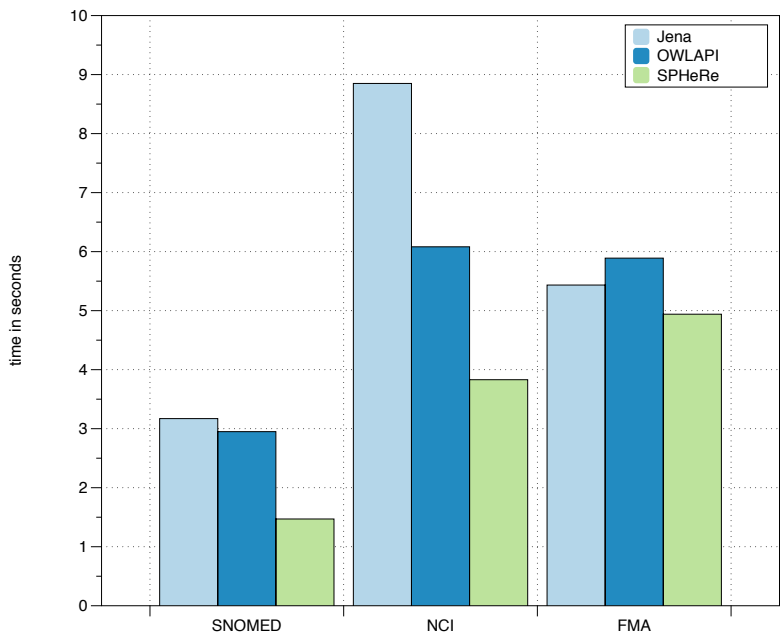


Figure 6.2: Ontology Load time comparison

following matching request, SPHeRe uses the cached candidate ontologies avoiding the re-parsing of OWL files. Results from Fig 6.2 validates that loading from cached serialized subsets take less

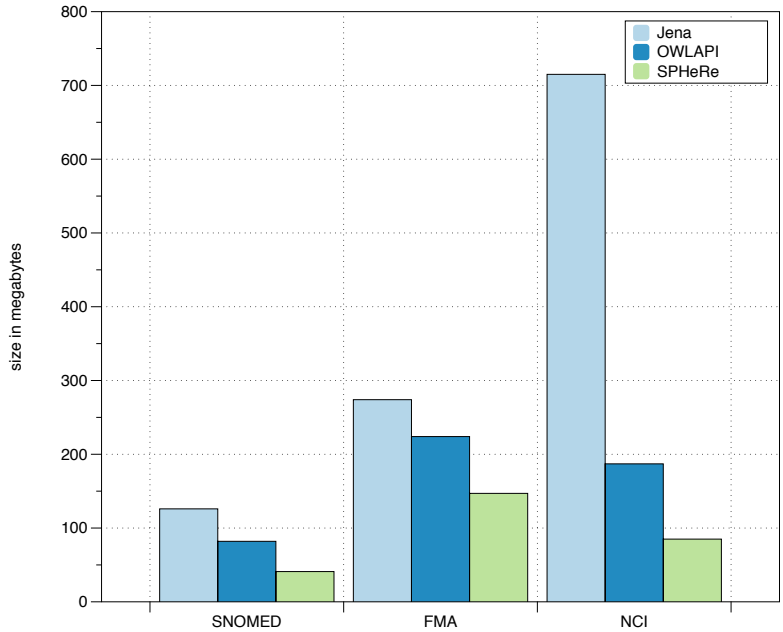


Figure 6.3: Memory Footprint comparison

time to load ontologies and retrieve classes. This technique ensures that following requests for matching of candidate ontologies will have better performance.

For memory footprint experiment on the same ontologies, results from SPHeRe, Jena, and OWL API are described in Fig 6.3. Memory footprint has been calculated by measuring the difference between the amounts of free memory available in the Java heap after the ontology load. Java's runtime library is used for this evaluation as described in [99], [100], and [101]. SPHeRe's cumulative memory footprint of all subsets is evaluated in contrast with memory footprint by Jena and OWL API. With the removal of redundancy of information and classification of ontology into subsets, SPHeRe produces up to 8 times smaller memory footprint than Jena and OWL API. This evaluation also complements to our remarks regarding the lightweight nature of SPHeRe, giving the system an edge over the existing systems to be a better option for ontology matching using commodity hardware and commodity hardware based distributed systems like cloud platforms.

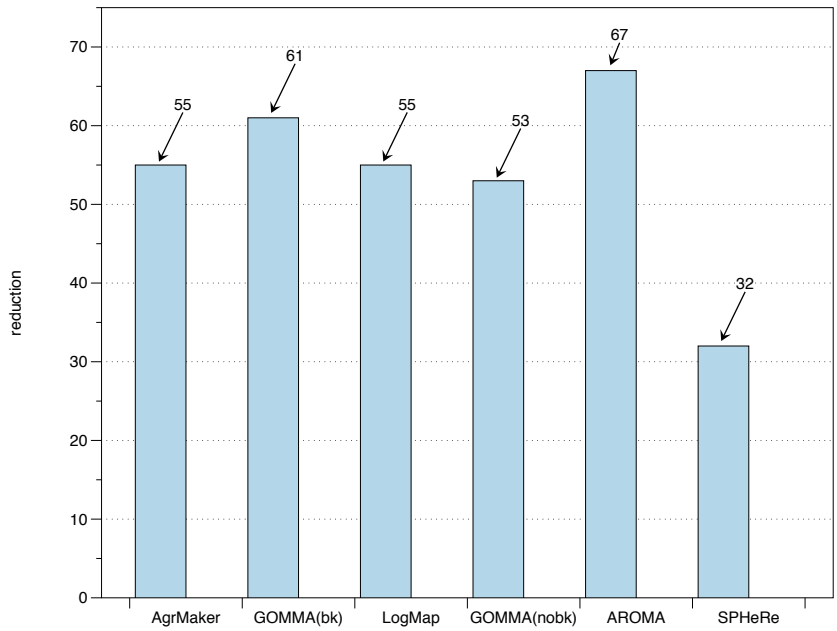


Figure 6.4: Scalability comparison

6.2 Scalability evaluation

In this subsection, we evaluate SPHeRe's scalability performance in contrast with existing scalable ontology matching systems from OAEI 2011.5 campaign. For measurement, as explained in [29], SPHeRe was executed over virtual instances with one, two, and four cores; however, each with lesser memory (4GB) in contrast with [29]. Fig 6.4 indicates the reduction rate achieved by SPHeRe when executing over 4-core environment. Reduction value is computed by dividing the execution time on four cores by execution time on one core. System with the best scalability will score a value around 25%. SPHeRe outperforms other scalable systems by scoring 32%, which is closest to the optimal value. SPHeRe outperforms the most scalable ontology matching system GOMMA by 40%. For precision, this evaluation has been repeatedly executed for 20 iterations.

Size-based partitioning in data parallel distributor of SPHeRe is a major contributor in achieving better scalability. This partitioning technique ensures that every matcher thread gets equal share of tasks by distribution. *Distributor* invokes matcher threads depending upon the runtime availability of the computing cores and assigns an equal number of matching tasks with matching algorithm instances among all the available matcher threads. Each matcher thread is responsible

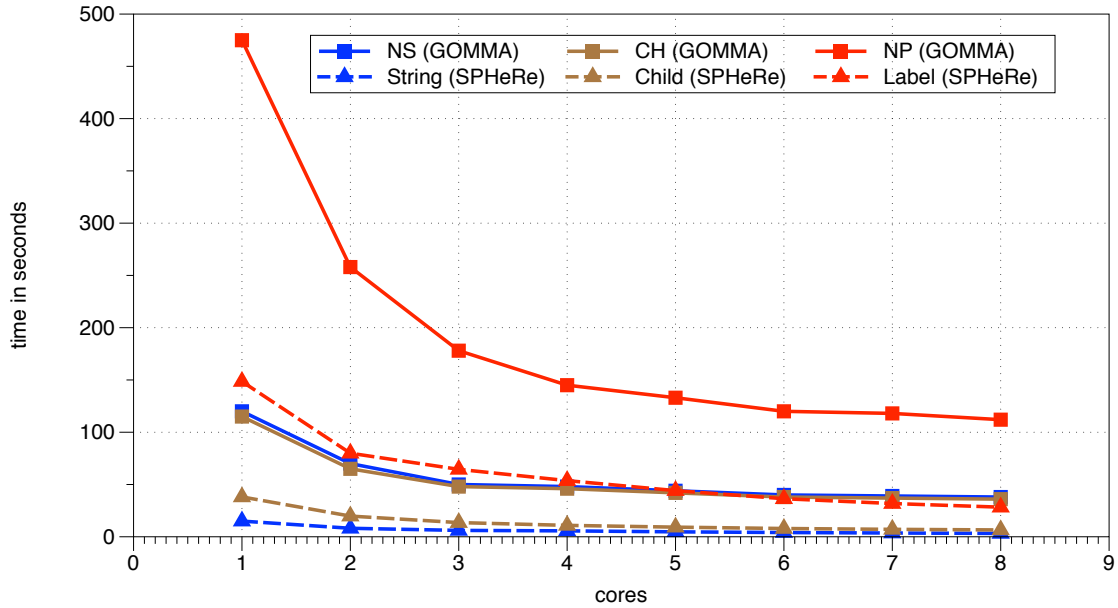


Figure 6.5: Performance comparison with GOMMA

for ensuring the execution of the matching algorithm on its own partition, preserving the overall accuracy of the system.

6.3 Performance comparison with GOMMA

In this subsection, we evaluate SPHeRe's performance against one of the most performance efficient ontology matching system GOMMA. Fig 6.5 describes SPHeRe's overall performance results in contrast with GOMMA's as presented in [20]. GOMMA's matching algorithms NameSynonym (NS), Children (CH), and NamePath (NP) are compared with similar (in complexity) matching algorithms of SPHeRe, i.e., $StringBased$ algorithm that calculates similarity by measuring edit distance [90] between concept names, $ChildBased$ algorithm that calculates similarity by comparing children of concepts, and a specialized $LabelBased$ algorithms that calculates similarity over tokenized and normalized labels of concepts. Algorithms of SPHeRe scaling from 1 up to 8 threads (= number of cores) outperform GOMMA by 4 times. For precision, this evaluation has been repeatedly executed for 100 iterations.

SPHeRe is able to achieve this performance by creating subsets of ontologies and distributing

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

6.4 Anatomy track

The anatomy track consists of mapping generation between the Adult Mouse Anatomy (2,744 concepts) [102] 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 SPHeRe for both multicore desktop and cloud scenario as a single-node execution (illustrated in Fig 6.6). 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 $(\frac{AdultMouseAnatomy_{classes} \times HumanAnatomy_{classes}}{8})$. As String and Label-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,

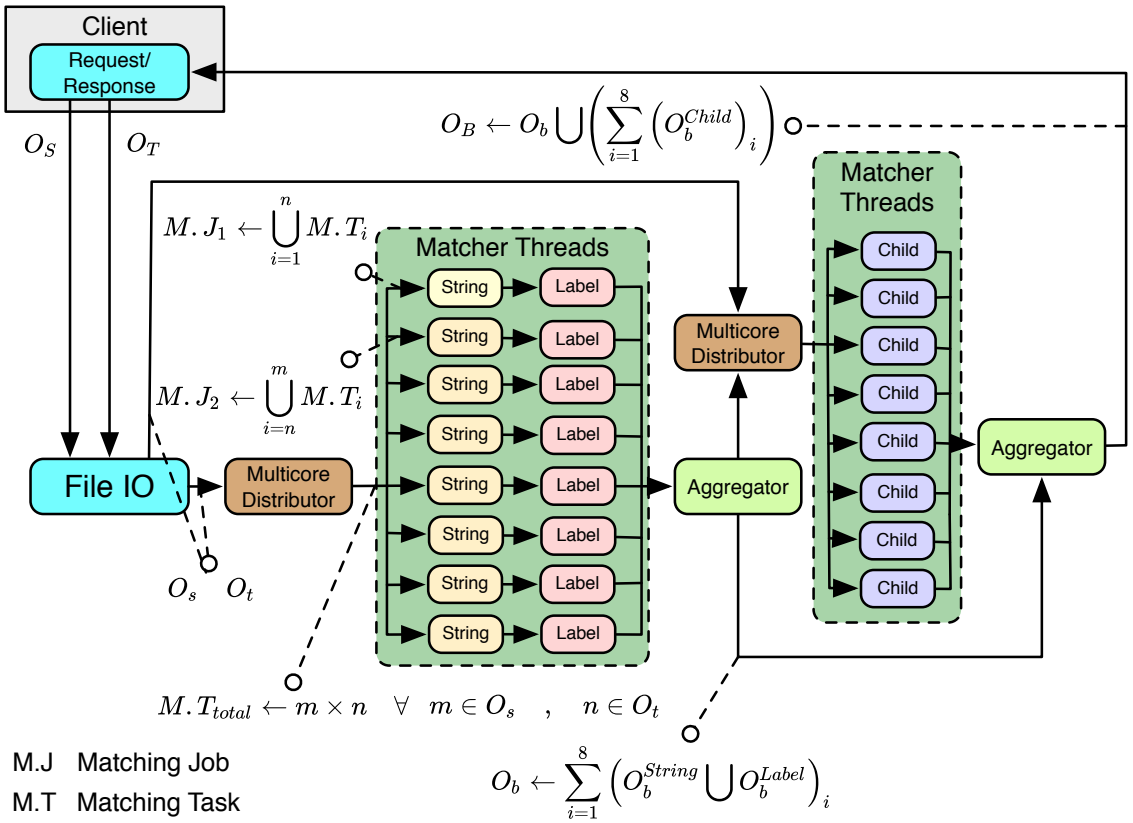
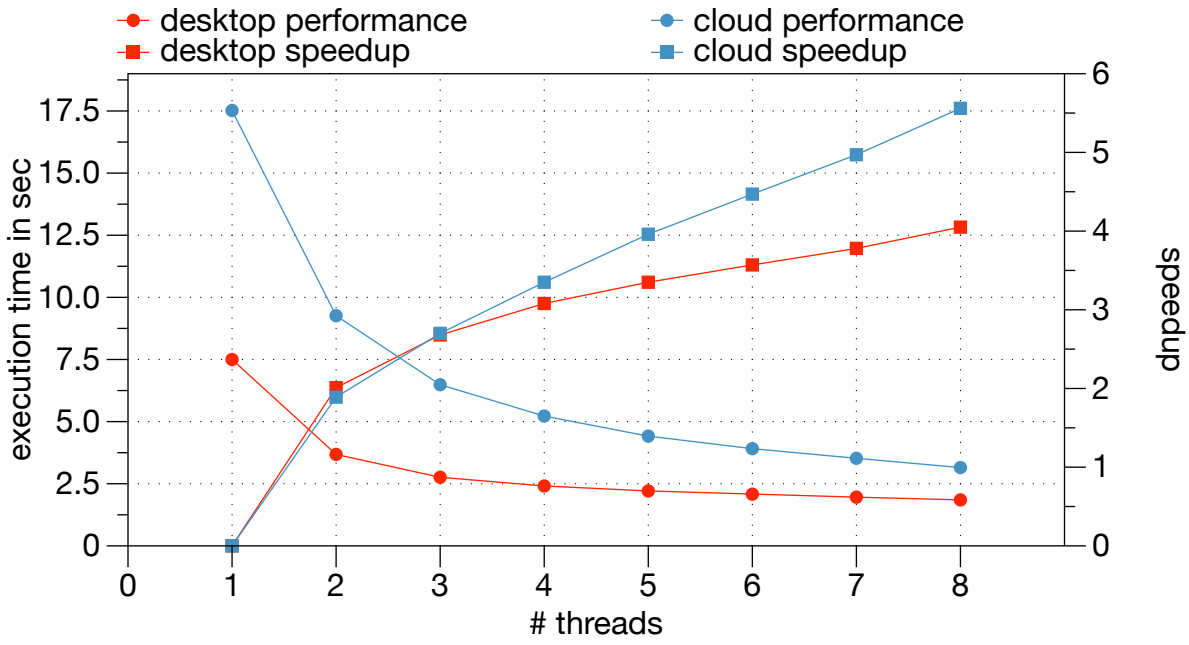


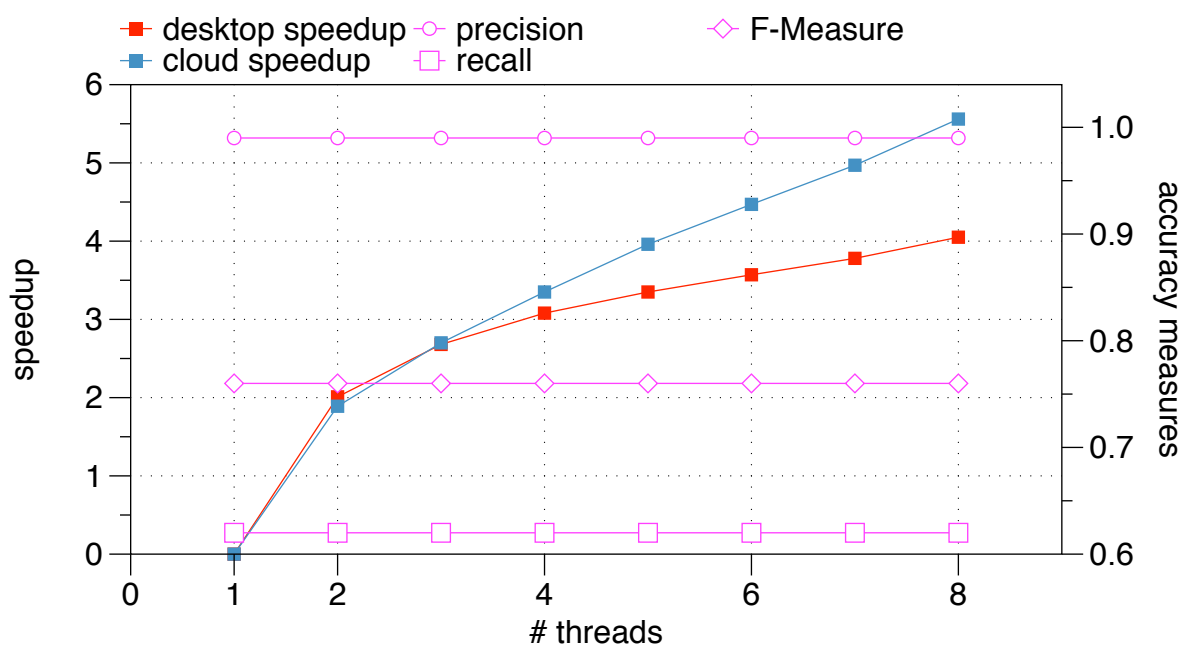
Figure 6.6: Parallel flow for Anatomy track over single-node

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 6.7. For the desktop scenario, the matching request executes over quad-core desktop and results are described in Fig 6.7(a). The sequential process (illustrated in Fig 6.8) takes 7.5 seconds to complete the matching request; however, with the use of our data parallelism enabled runtime 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 the performance speedup of 4 times. 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, SPHeRe completes the whole matching process over 8 threads within 3.1 seconds with an impressive speedup of 5.5 times.



(a) Performance-speedup



(b) Speedup-matching effectiveness

Figure 6.7: Results from Anatomy track

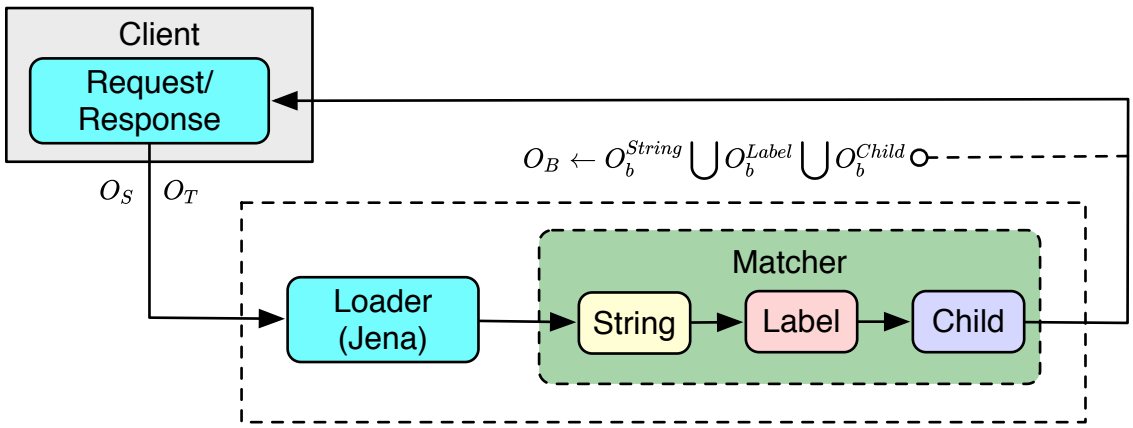


Figure 6.8: Sequential flow on a single-node

Overall performance of the matching process is slightly slower over the Azure VM due to the virtualization layer (Hyper-V).

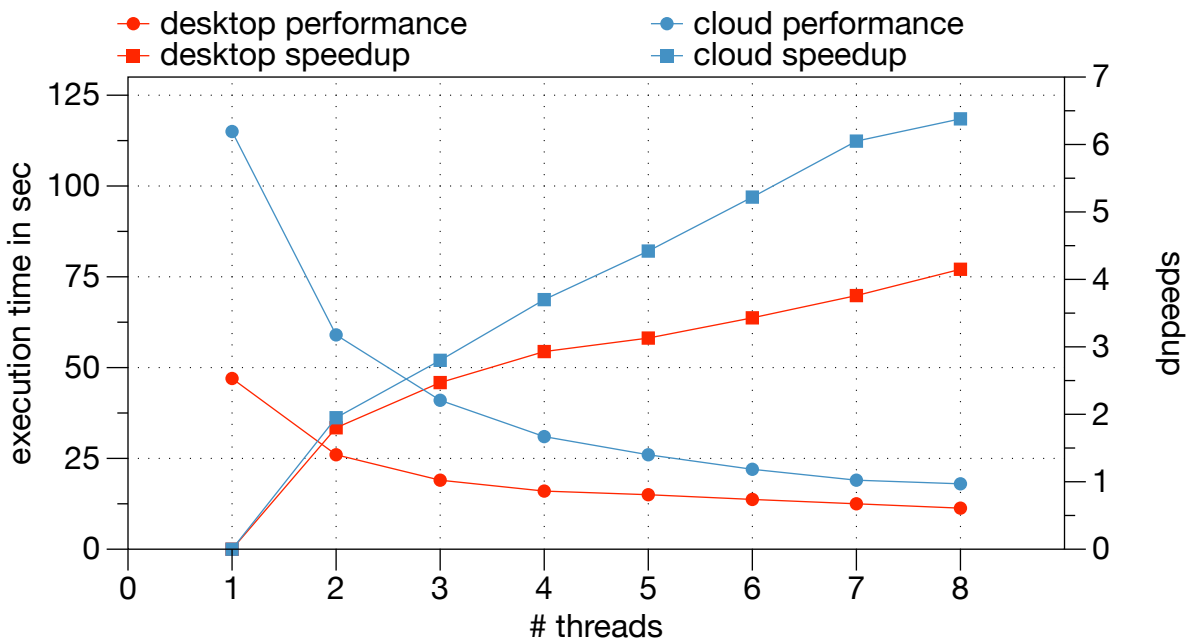
Accuracy preservation throughout the performance speedup is illustrated in Fig 6.7(b). As stated earlier, for effectiveness-independent performance-gain the performance is extracted from parallel threads over multiple cores, no changes in matching library have been made for performance reasons. Consequently, the matching effectiveness (e.g., precision, recall, F-Measure) stays the same throughout the performance speedup.

Same matching track was evaluated by [20] 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 [20]. 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 [20] 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)).

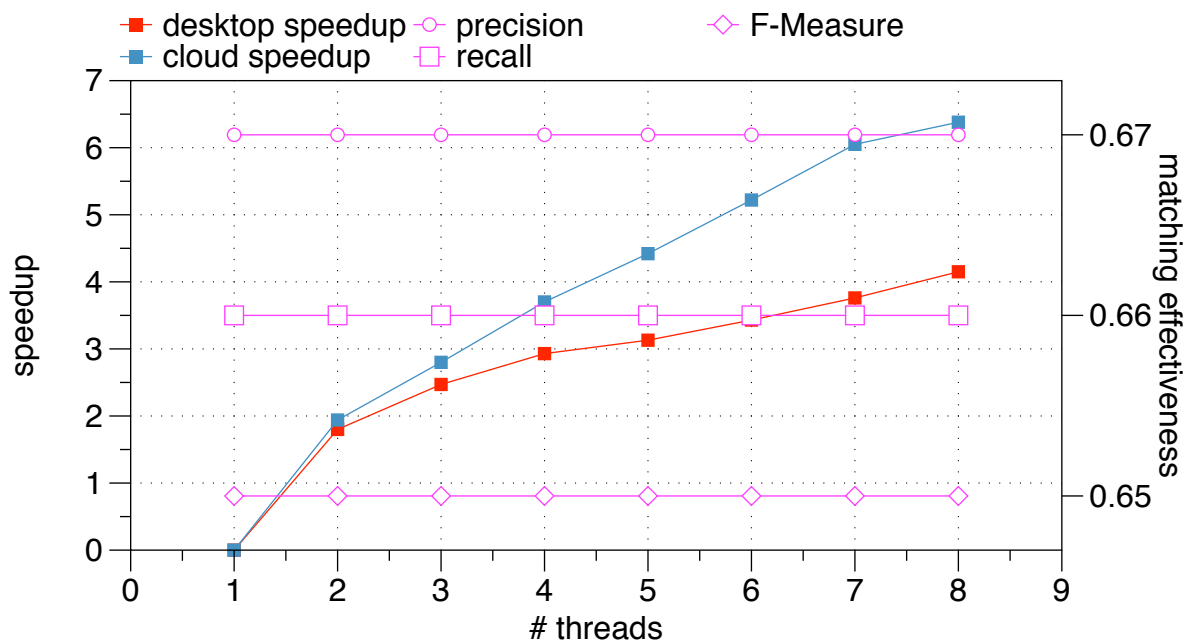
6.5 Library track

The library track consists of mapping generation between the STW [103] and the TheSoz thesaurus [104] 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 concepts. To generate the bridge ontology we have utilized the same matching library used earlier in anatomy track.

Similar to anatomy track, we have executed SPHeRe for both multicore desktop and cloud scenario as a single-node execution. Results from these scenarios are illustrated in Fig 6.9. For the single-node desktop scenario, the sequential process takes close to 47 seconds to complete the matching request; however, SPHeRe 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, SPHeRe 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 6.9(b)).



(a) Performance-speedup



(b) Speedup-matching effectiveness

Figure 6.9: Results from Library track

6.6 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 sub-sections.

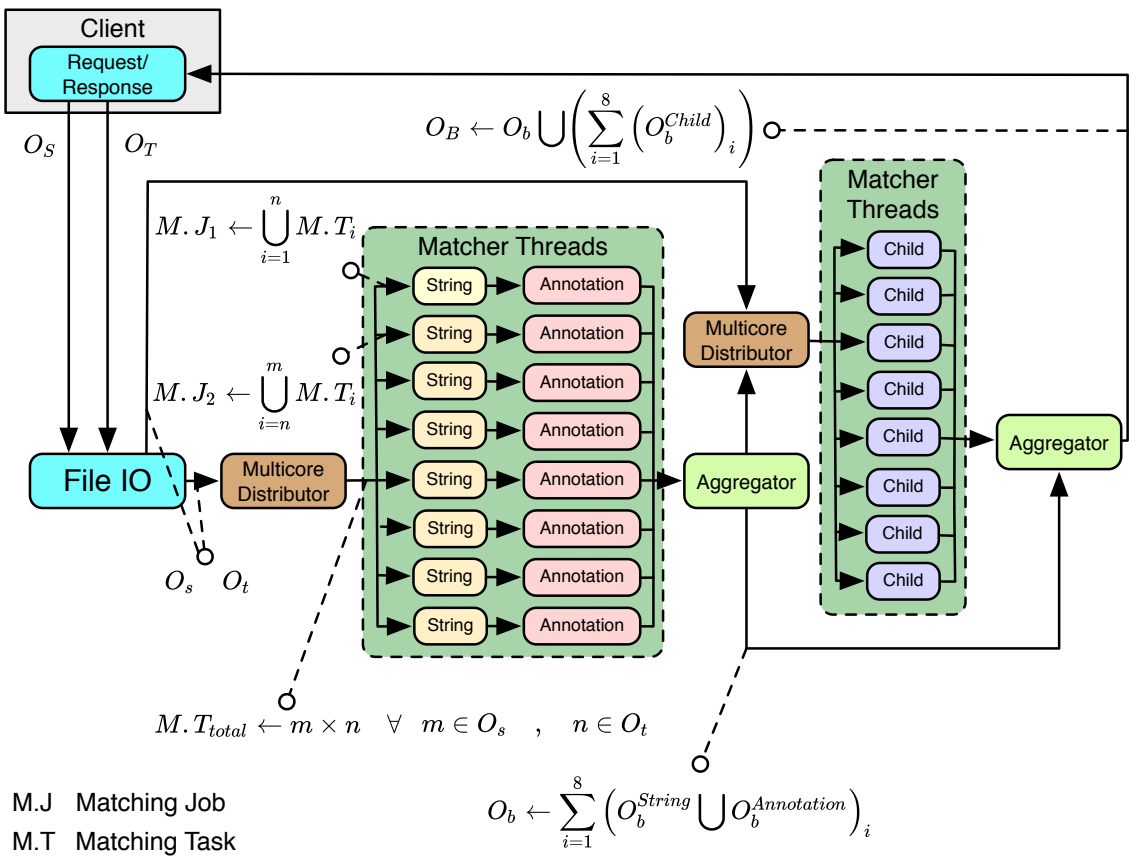


Figure 6.10: Parallel flow for Large Biomedical Ontologies track over single-node

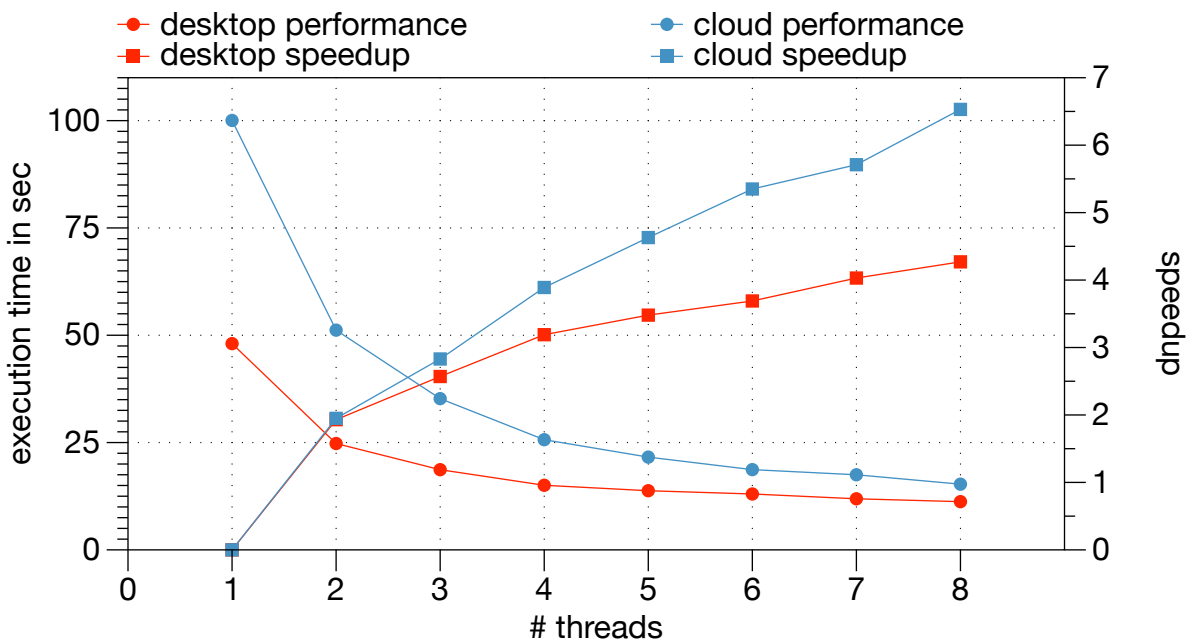
6.6.1 Task 1: FMA-NCI small fragments

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

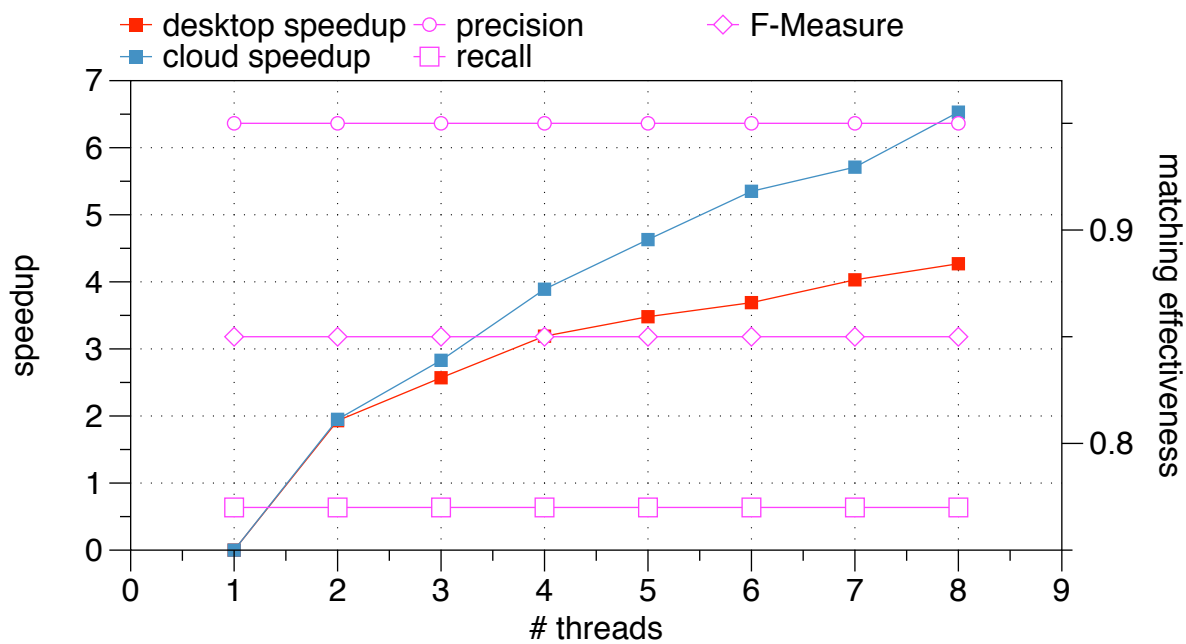
fragment consists upon 10% of the whole NCI ontology (6,488 concepts).

We have executed SPHeRe for both multicore desktop and cloud scenario illustrated in Fig 6.10 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 scenarios (desktop and cloud) are illustrated in Fig 6.11. For the desktop scenario, the matching request executes over quad-core desktop. The sequential process (similar to the illustration in Fig 6.8) takes 48 seconds to complete the matching request; however, SPHeRe 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, SPHeRe completes the whole matching process over 8 threads in slightly over 15 seconds with an impressive speedup of 6.5 times. Furthermore, similar to the previous tracks the accuracy of the matching process stays preserved with the same effectiveness throughout the performance speedup (illustrated in Fig 6.11(b)).



(a) Performance-speedup



(b) Speedup-matching effectiveness

Figure 6.11: Results from Large Biomedical Ontologies track, task 1

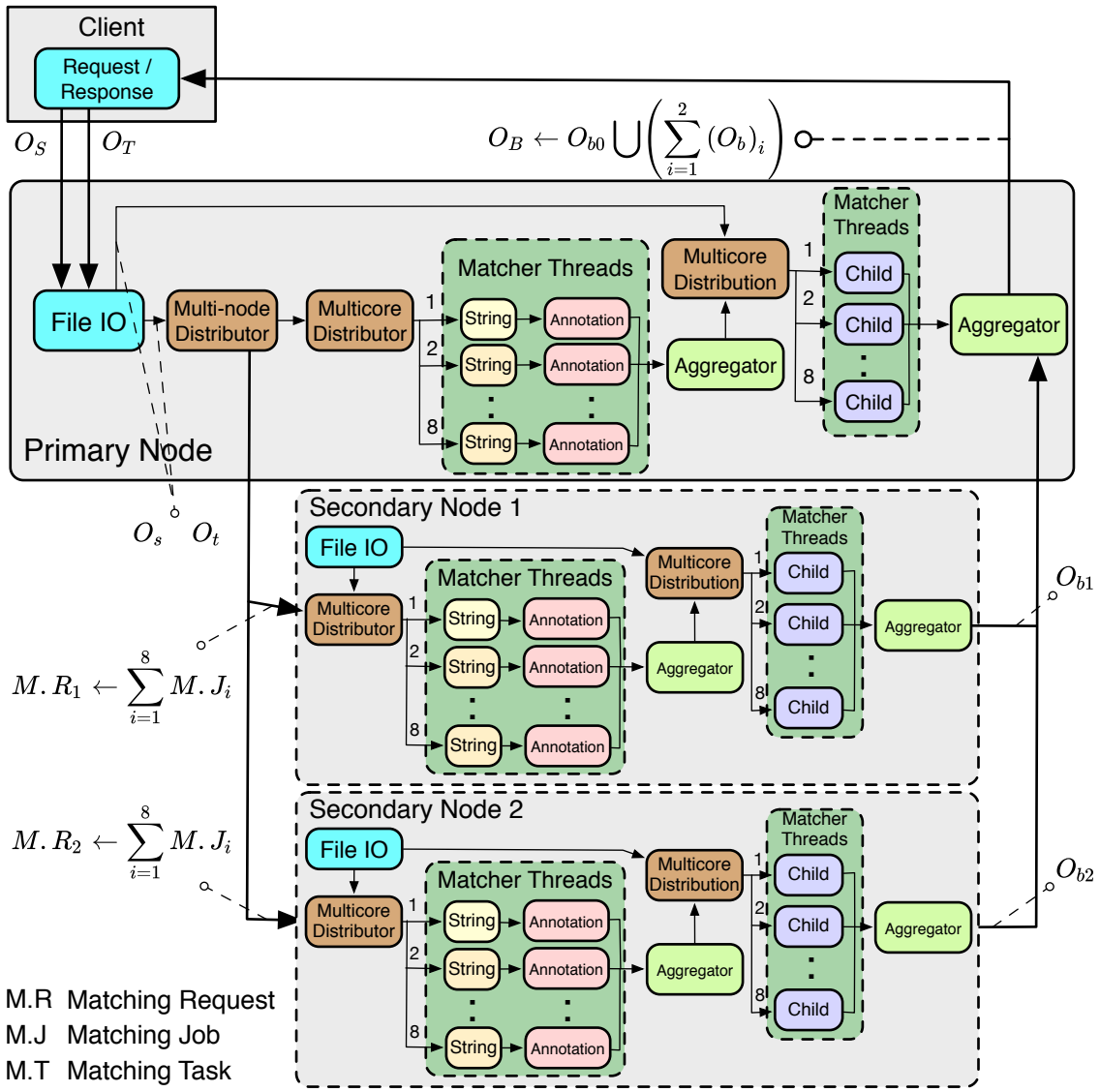


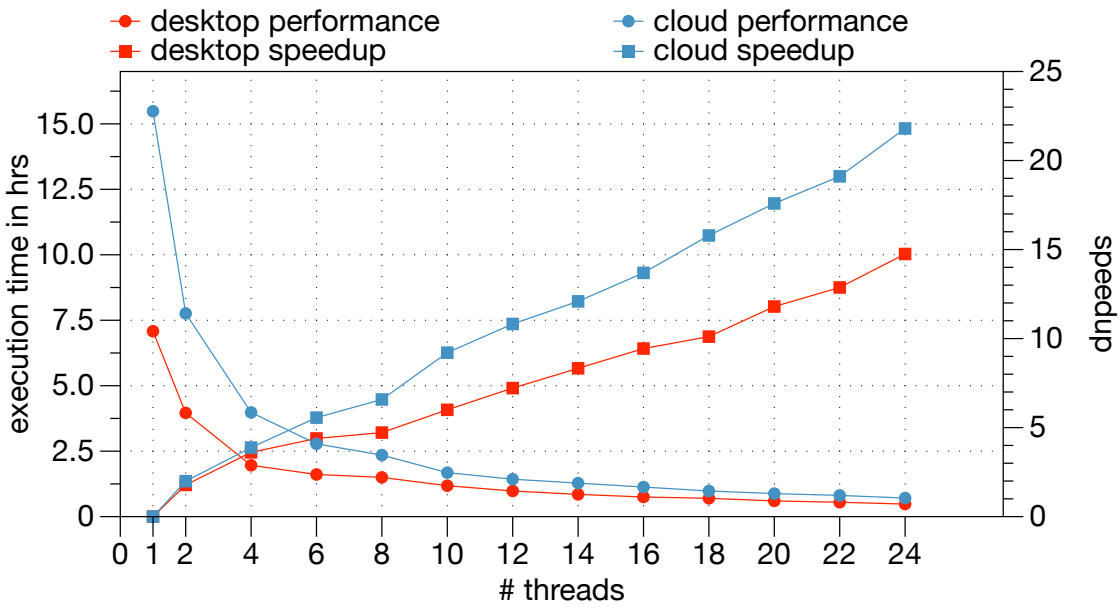
Figure 6.12: Parallel flow for Large Biomedical Ontologies track over multi-node

6.6.2 Task 2: FMA-NCI whole Ontologies

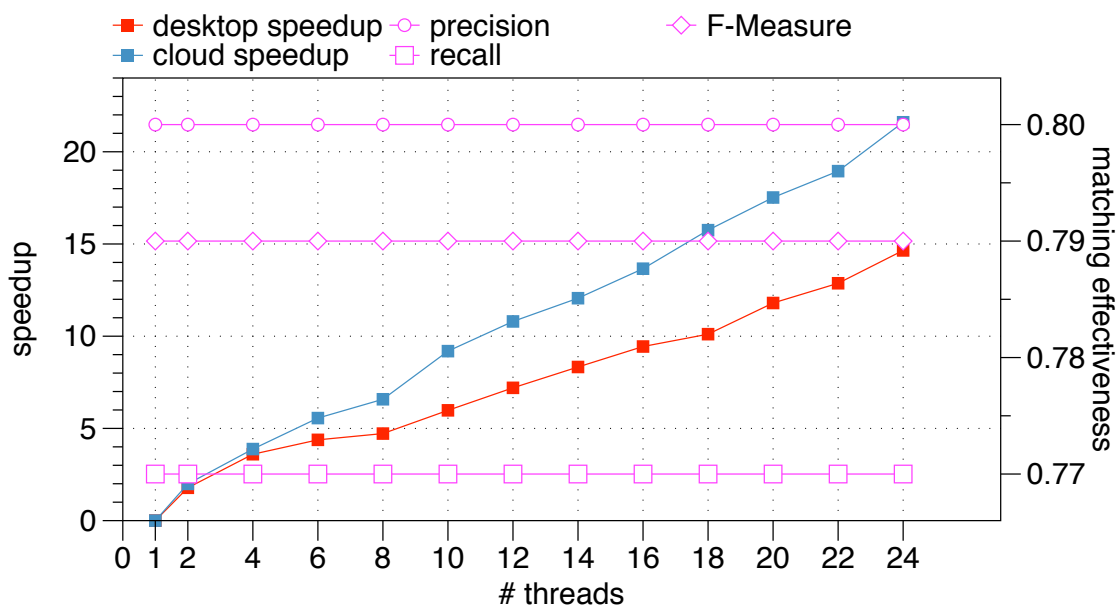
This task consists upon matching the whole FMA and NCI ontologies. The FMA ontology consists upon 78,989 concepts while the NCI ontology consists upon 66,724 concepts. Due to the very large size of the ontologies, the matching process is scaled over a multi-node environment, i.e., 3 desktops and Azure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

As illustrated in Fig 6.12, 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 ontology subsets (O_s, O_t) required by matching algorithms, multi-node distributor of primary node creates 3 independent matching requests of equal size. 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 with their respective matching requests, all 3 participating nodes load serialized subsets of 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}). Aggregator of secondary nodes sends their respective intermediate ontologies to the primary node. These bridge ontologies are accumulated by 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 scenarios (desktop and cloud) are illustrated in Fig 6.13. 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. Same matching request is executed for the second scenario over the multi-node Azure VM. The sequential process over the VM takes 15.5 hours to complete; however, SPHeRe 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 6.13(b)).



(a) Performance-speedup



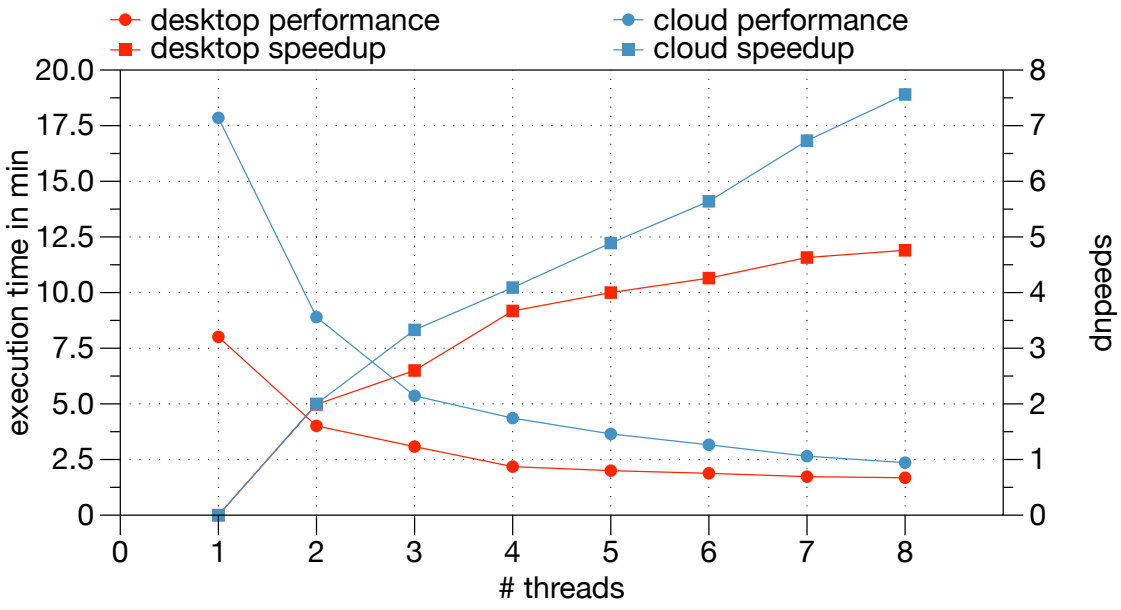
(b) Speedup-matching effectiveness

Figure 6.13: Results from Large Biomedical Ontologies track, task 2

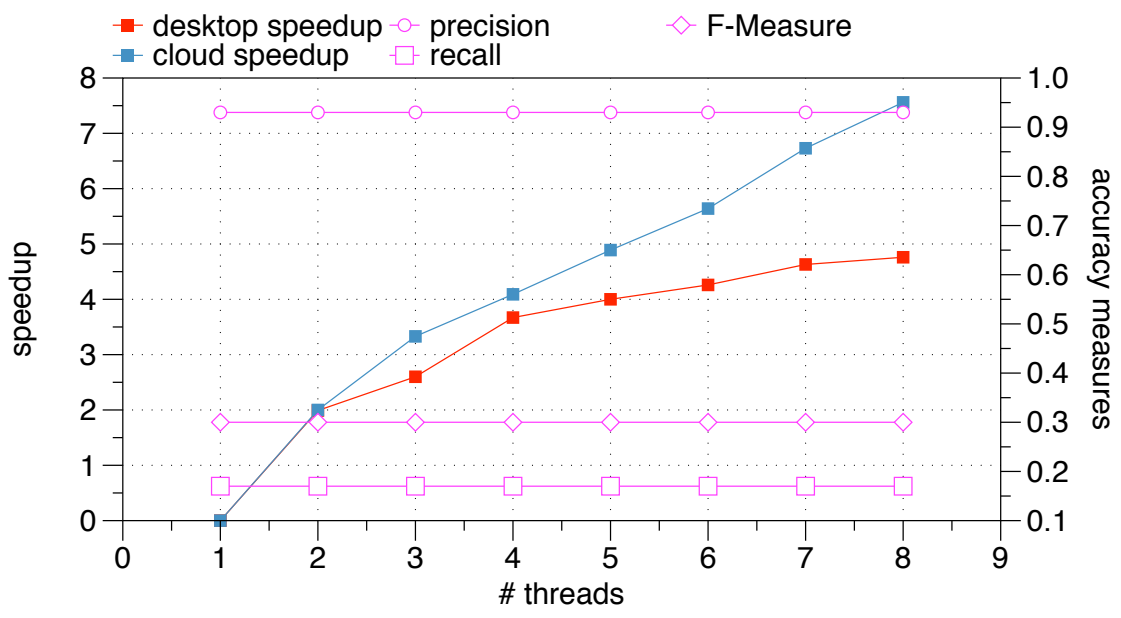
6.6.3 Task 3: FMA-SNOMED small fragments

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

Similar to task 1, we have executed SPHeRe in both multicore desktop and cloud scenario as a single-node execution. Results from these scenarios are illustrated in Fig 6.14. For the single-node desktop scenario, the sequential process takes around 8 minutes to complete the matching request; however, SPHeRe completes the matching process in slightly over one and half minute 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, SPHeRe 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 6.14(b)).



(a) Performance-speedup



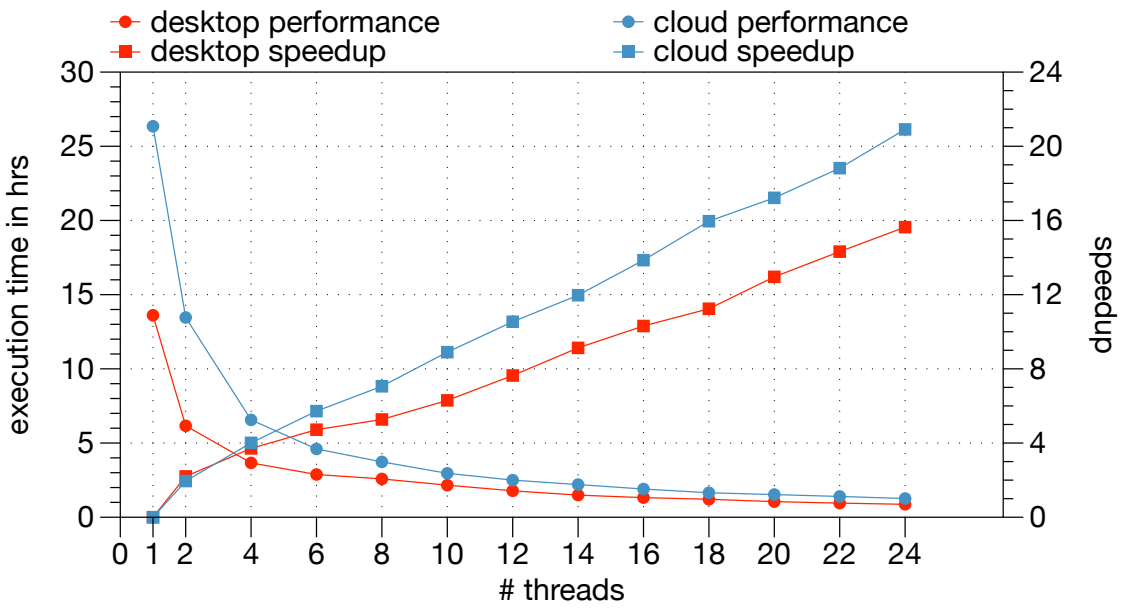
(b) Speedup-matching effectiveness

Figure 6.14: Results from Large Biomedical Ontologies track, task 3

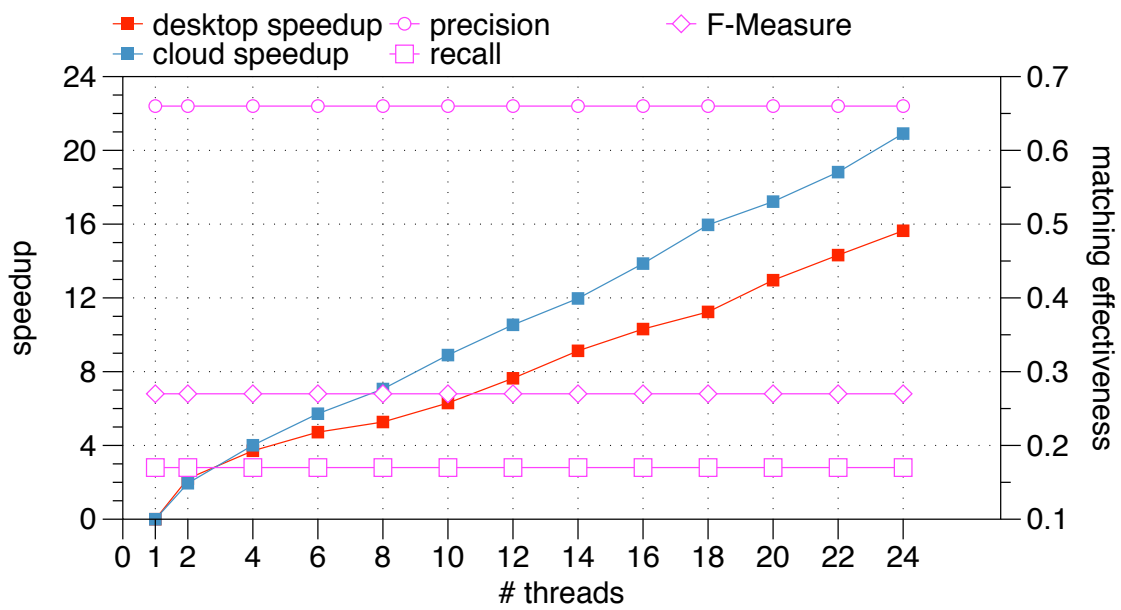
6.6.4 Task 4: FMA whole Ontology with SNOMED large fragment

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 the 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 6.15. 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. 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 6.15(b)).



(a) Performance-speedup



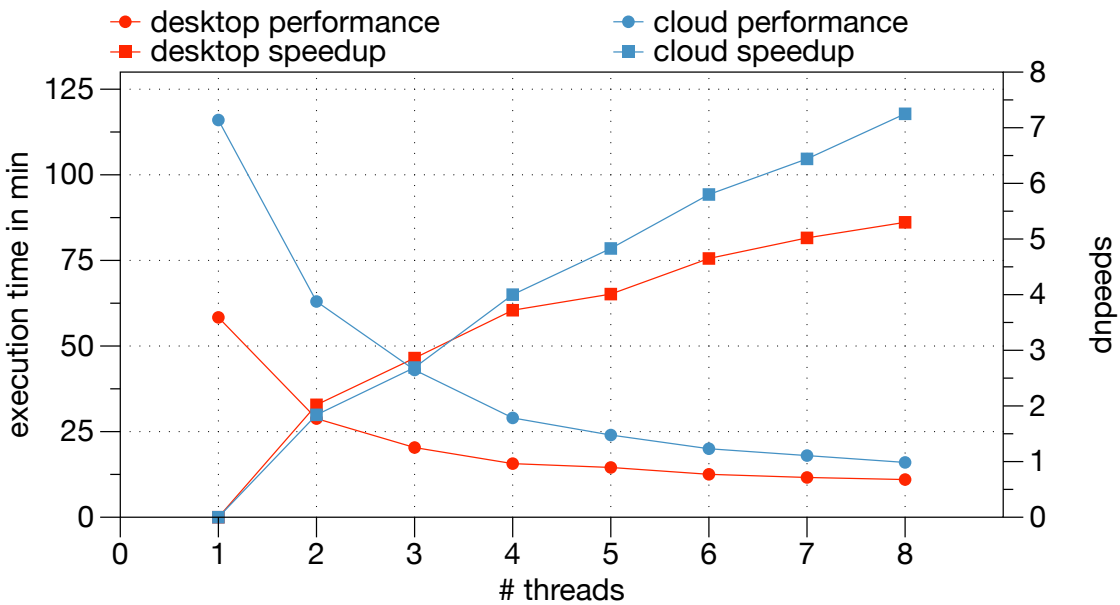
(b) Speedup-matching effectiveness

Figure 6.15: Results from Large Biomedical Ontologies track, task 4

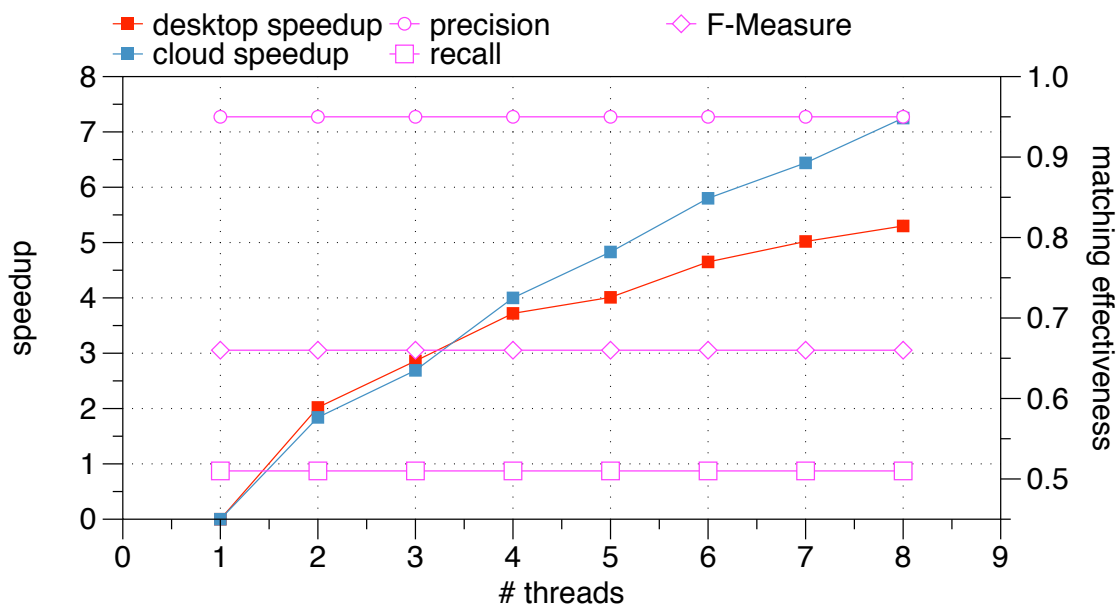
6.6.5 Task 5: SNOMED-NCI small fragments

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

Similar to task 1 and 3, we have executed SPHeRe in both multicore desktop and cloud scenario as a single-node execution. Results from these scenarios are illustrated in Fig 6.16. For the single-node desktop scenario, the sequential process takes around an hour to complete the matching request; however, SPHeRe 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, SPHeRe 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 6.16(b)).



(a) Performance-speedup



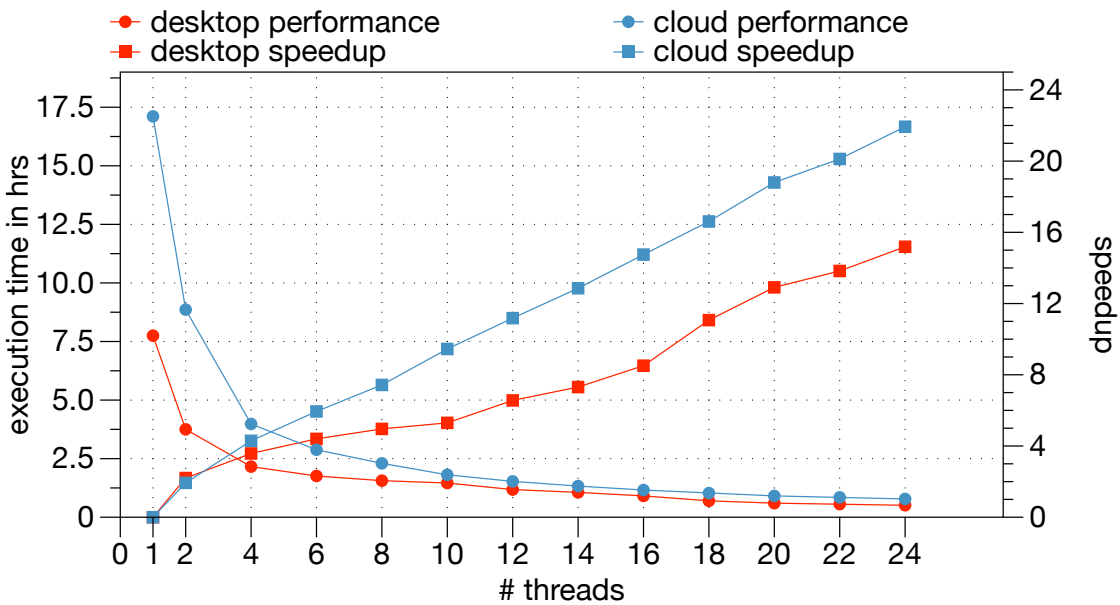
(b) Speedup-matching effectiveness

Figure 6.16: Results from Large Biomedical Ontologies track, task 5

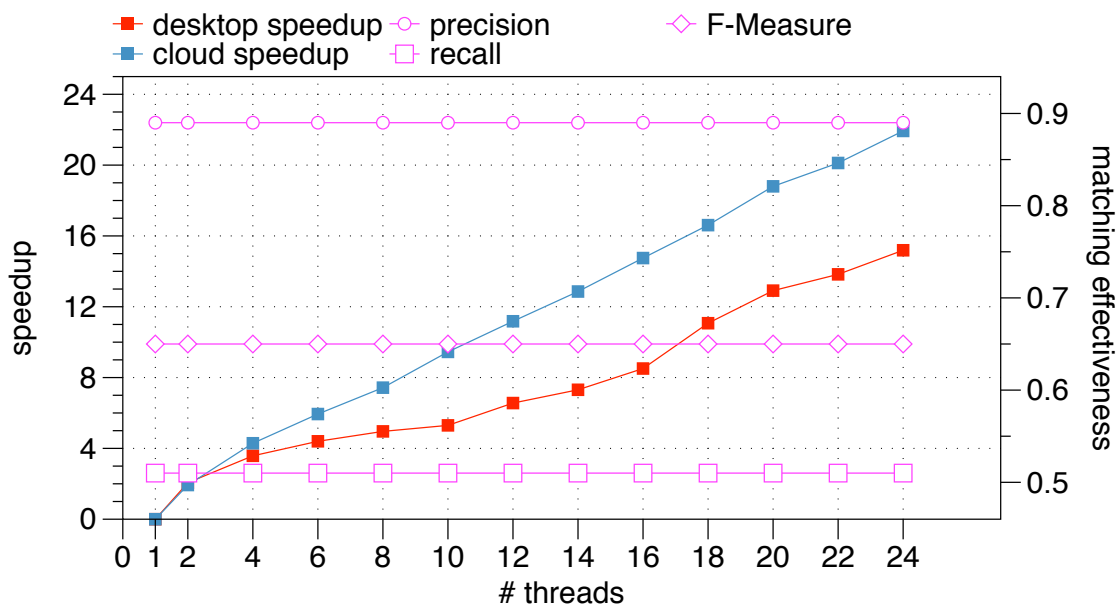
6.6.6 Task 6: NCI whole Ontology with SNOMED large fragment

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 the SNOMED ontology (122,464 concepts).

Similar to task 2 and 4, we have executed SPHeRe in both multicore desktop and cloud scenario as multi-node execution. Results from these scenarios are illustrated in Fig 6.17. For the multi-node desktop scenario, the sequential process takes close to 8 hours to complete the matching request; however, SPHeRe 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, SPHeRe 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 6.17(b)).



(a) Performance-speedup



(b) Speedup-matching effectiveness

Figure 6.17: Results from Large Biomedical Ontologies track, task 6

6.7 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 require 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 6.18). 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 following figures.

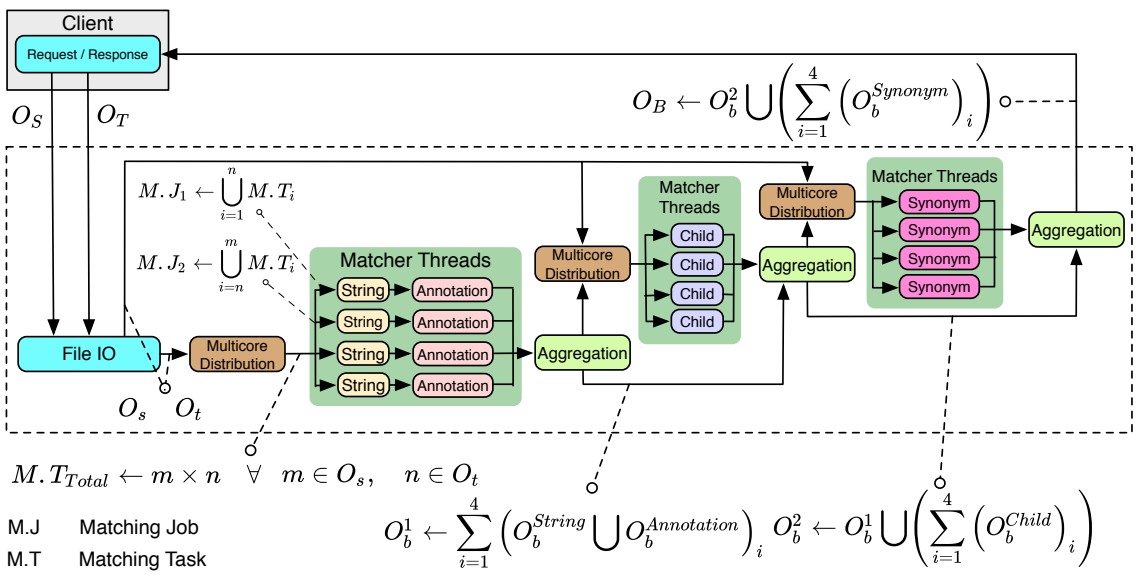


Figure 6.18: Parallel flow for Conference track over dual core single-node Azure VM

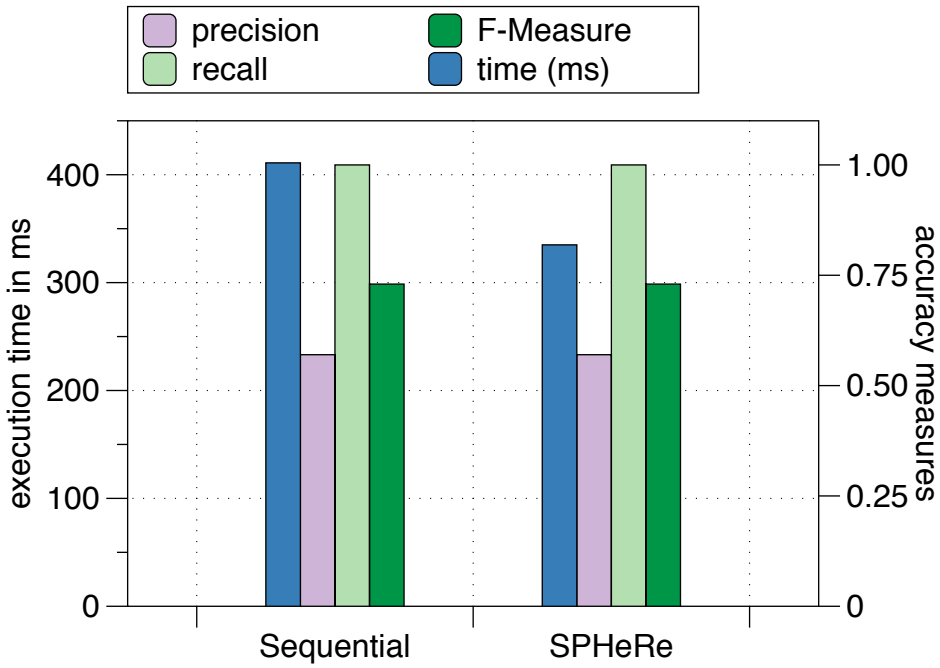


Figure 6.19: cmt-iasted

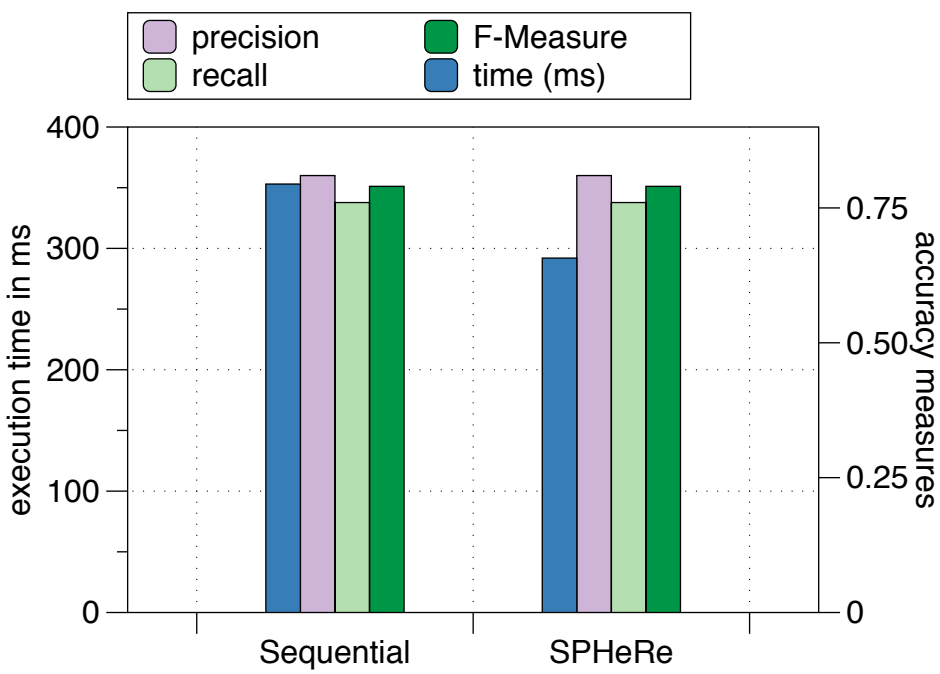


Figure 6.20: conf-edas

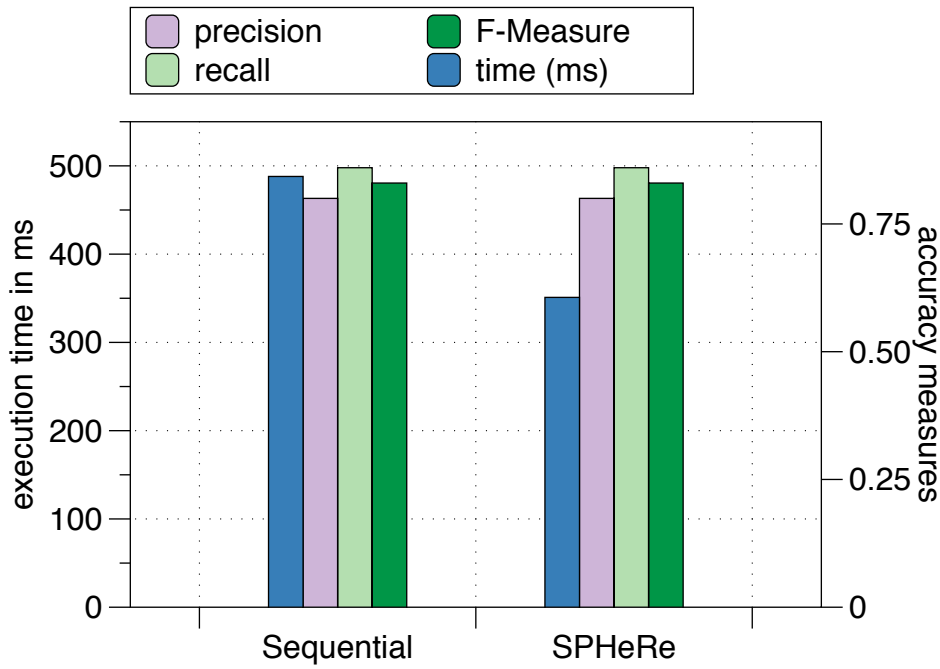


Figure 6.21: conference-iasted

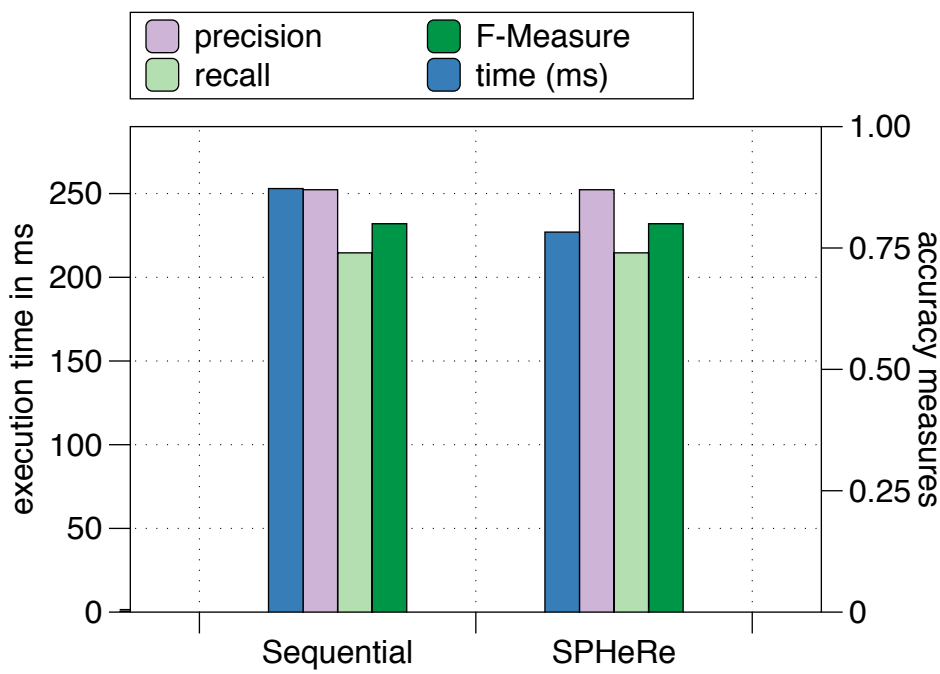


Figure 6.22: confOf-edas

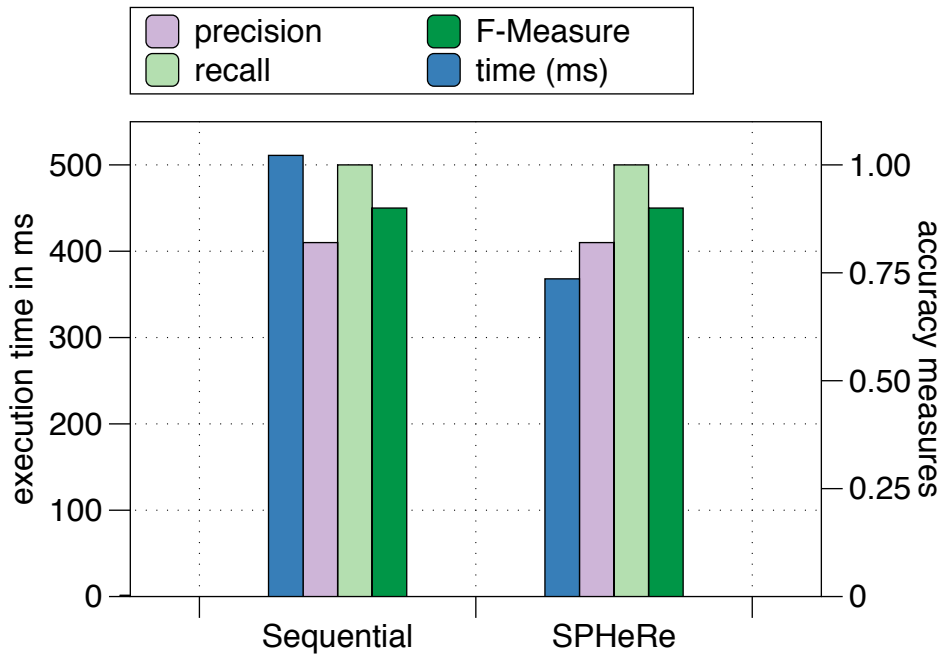


Figure 6.23: confOf-iasted

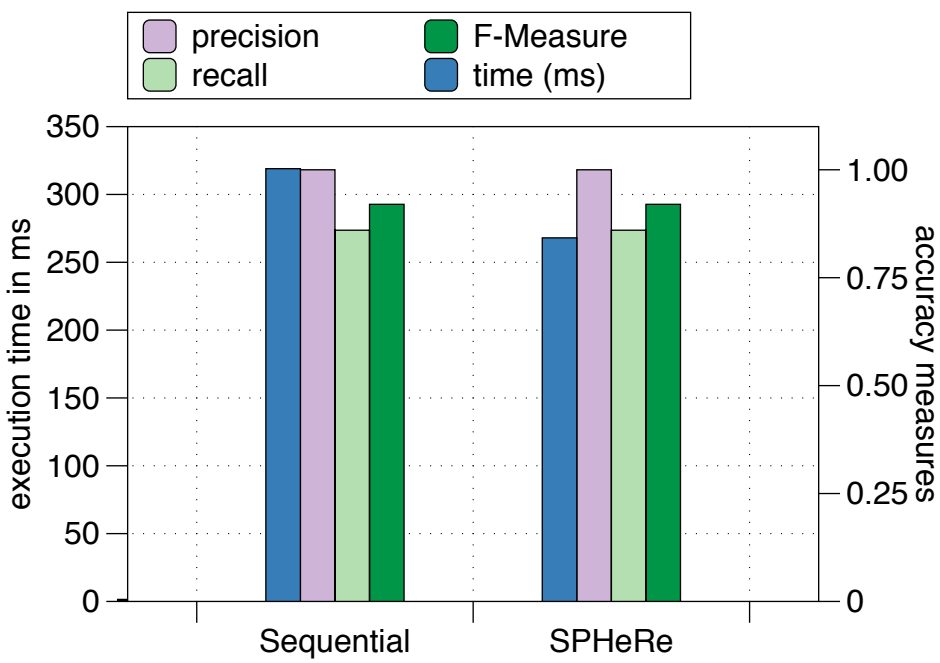


Figure 6.24: confOf-sigkdd

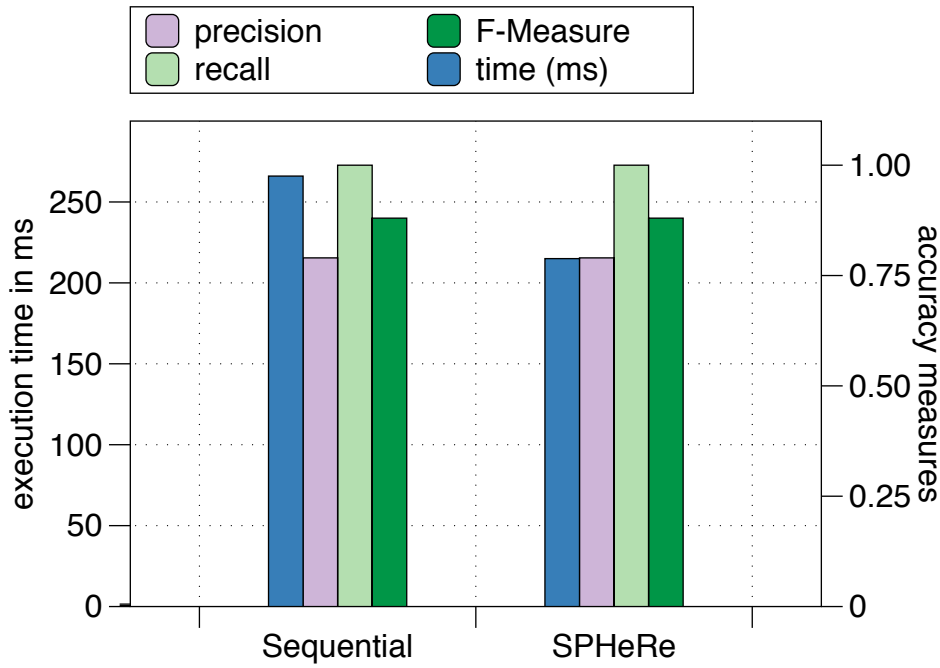


Figure 6.25: ekaw-sigkdd

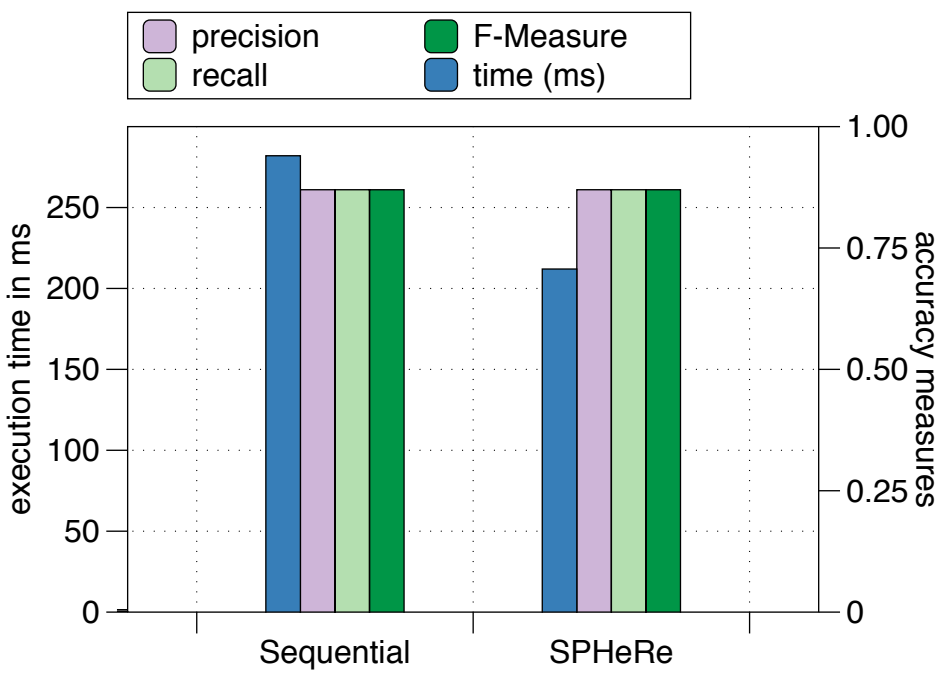


Figure 6.26: iasted-sigkdd

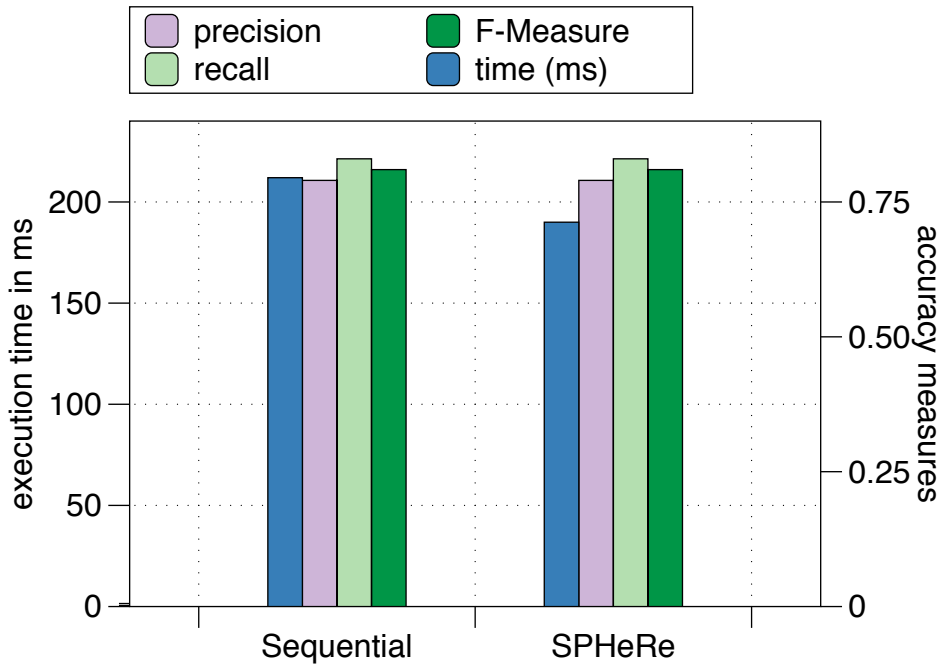


Figure 6.27: edas-ekaw

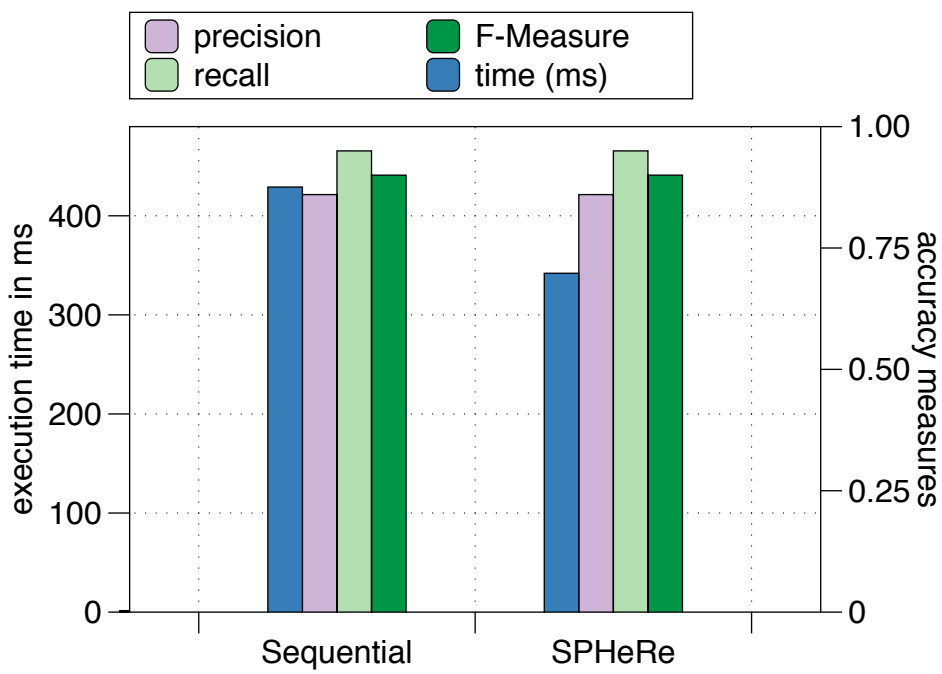


Figure 6.28: edas-iasted

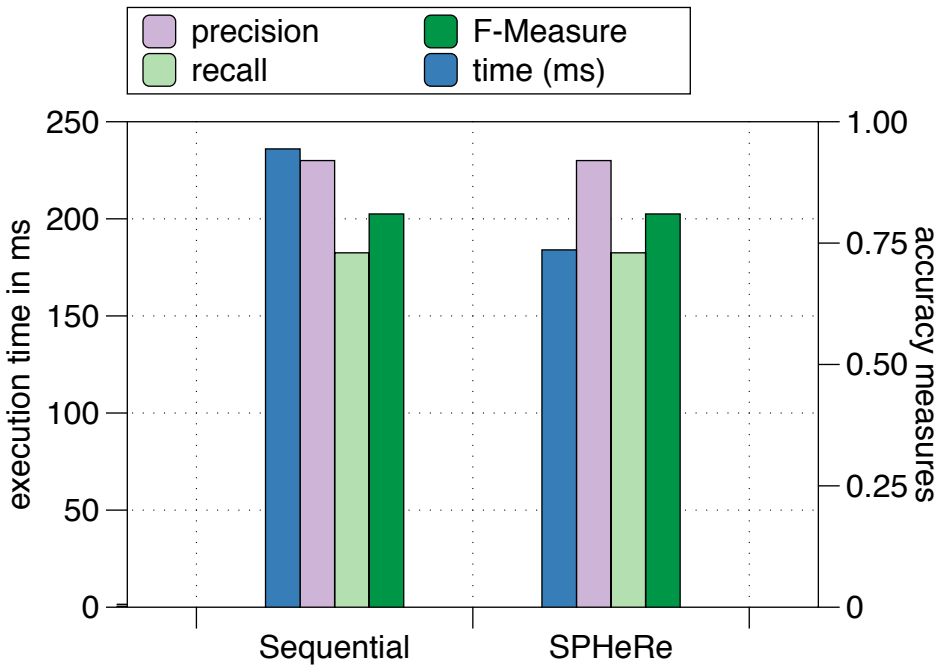


Figure 6.29: edas-sigkdd

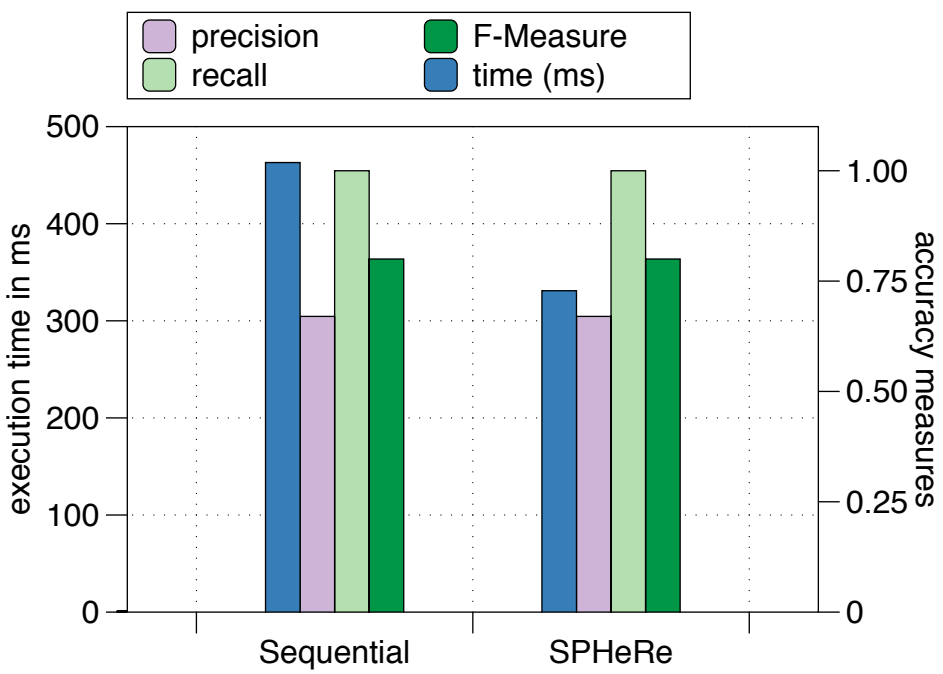


Figure 6.30: ekaw-iasted

Table 6.2: Evaluation 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

6.8 Evaluation Summary

In our evaluation, we have used the dataset of real-world ontologies provided by OAEI's 2012 and 2013 campaign. The key strength of this dataset is its comprehensiveness 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 6.2. These results provide evidence for four major characteristics of presented methodology, described in the following subsections.

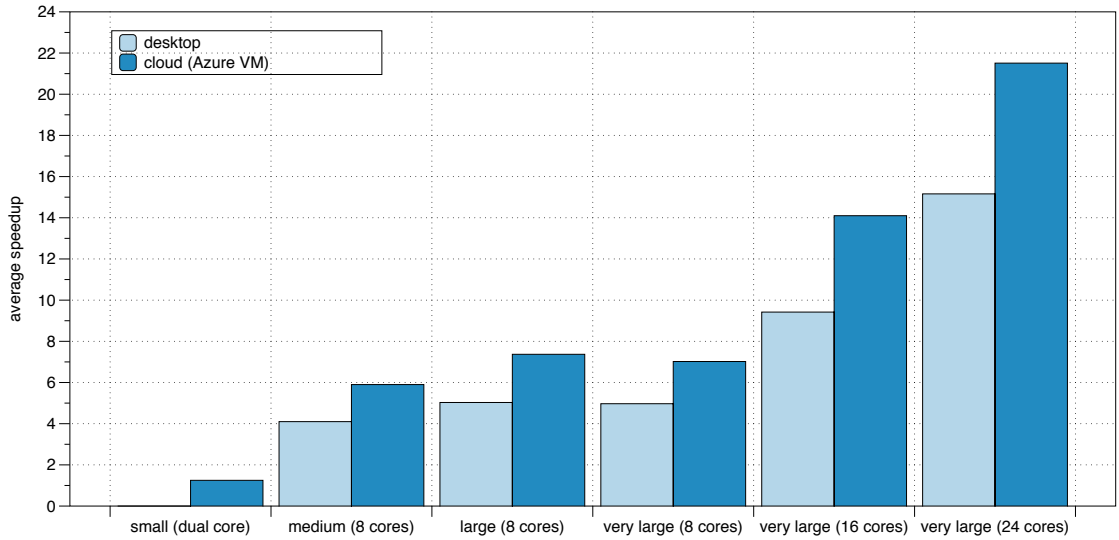


Figure 6.31: Results Summary

6.8.1 Independent of Ontology Domain

As stated in the related work chapter, some of the matching systems are built specific to ontology domains, particularly systems for matching biomedical ontologies. However, 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 candidate ontologies used in the matching problems evaluated of proposed methodology are of diverse domains. No change has been inflicted in the structure of the candidate ontologies, yet presented implementation scores an 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.

6.8.2 Performance-based Ontology Matching over various size of Matching Problems

As described in Table 6.2, 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 6.31. It is quite evident from the figure that presented methodology 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 presented implementation was able to improve the performance by an average speedup of 1.25 times ($\approx 20\%$ 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 on desktop and cloud respectively. Comparing these results to the average speedup of medium-scale problem of [20], even with an inferior hardware presented implementation 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 [20] on a single node, our implementation 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. Presented implementation outperforms the intra-matcher of [20] by 4.6% on desktop and 47.78% over cloud platform.

In case of multi-node platform with dual nodes (8 cores each), our implementation completes the matching process with an average speedup of 9.42 and 14.1 on desktop and cloud respectively. Comparing these results with Intra&Inter multi-node matcher of [20] 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 implementation completes the matching process with an average speedup of 15.16 times on a desktop and 21.51 times over cloud platform.

6.8.3 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 methodology extracts performance-gain without inflicting any changes in the accuracy of the matching algorithm. From the results, it is clear that the accuracy 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.

6.8.4 Matching Library Interface

To implement effectiveness-independent performance-gain, distributor components of the implementation, decouples the matching library from the performance runtime with the help of a matching library interface. This approach offers 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.

Chapter 7

Conclusion and Future Directions

7.1 Conclusion

In this thesis, we presented our multi-dimensional performance-based ontology matching methodology over parallel platforms. Ontology matching is a widely used technique for heterogeneity resolution among information and knowledge-based systems; however, size, complexity, and availability of these ontologies requires solutions that are built from a performance aspect. With the availability of affordable parallelism-enabled multicore platforms like desktop and cloud, our methodology exploits their performance benefits by data parallelism for ontology matching.

Presented methodology caters performance aspects of ontology matching with effectiveness-independence from four dimensions, i.e., memory space, over-all execution time, performance-based ontology matching runtime, and non-monolithic ontology matching resolution. For memory space, our methodology converts the candidate ontologies into smaller, simpler, and scalable resource-based ontology subsets, based on the requirements of matching algorithms. This approach provides the resolution to the scalability challenge of ontology matching by providing ontology subsets that are distribution friendly. Furthermore, due to the smaller size, independent 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.

To further contribute in memory space reduction of the ontology matching process, our methodology also aligns 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 the redundant matching

operations.

For over-all execution time, presented methodology provides resolution by parallel and distributed ontology matching. For its implementation over the matching process, three-layer distribution abstraction is defined by our methodology. 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 approach of distributing matching process from grainer level matching request to finer level matching tasks provides the resolution to the resource utilization challenge of the performance aspect 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 the 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 parallel and distributed matching. Matched results from matched tasks distributed over computing resources are aggregated to generate the required mappings as mediation bridge ontology by post-matching stage.

Presented approaches are implemented as performance-based ontology matching runtime, deployed and evaluated over parallel platforms like multicore desktops and multi-node cloud platforms. This runtime is developed using the Java programming language and has been incorporated in our ontology matching system called SPHeRe as a core component. Presented runtime utilizes the subset generation and matching space reduction approaches for efficient memory space utilization, and parallel and distribution abstractions with explicit thread-level parallelism for performance-gain during ontology matching process. Furthermore, this performance-based runtime is extended with interfaces to service and platform level, and deployed over public cloud platform (MS. Azure) for availability. At service level, it is presented as Ontology Matching as a Service where heterogeneity resolution clients can submit their ontologies for bridge ontology

generation. Runtime is also presented as Platform as a Service where semantic-web experts can plug-in their matching libraries and algorithms for execution and evaluation. Due to the decoupled interface between performance components and matching library, semantic-web experts need not to write any parallelism-based code within their matching algorithms, parallel and distributed matching is automatically taken care by the distributor components of the ontology matching runtime.

For benchmarking of the presented methodology, 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 assess 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 methodology, 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 approach provides a small performance speedup (1.25 times) on smaller ontologies; however, it scores 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.

7.2 Future Work

From the recorded results drawn by the presented methodology working with real-world ontologies, it is apparent that our approach offers a comprehensive resolution to the performance challenges of ontology matching problems. Moreover, our methodology 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 methodology performs impressively well on medium to very-

large scale ontology matching problems. Thus it has the required longevity for the future ontology matching problems. We have on-going research in the area of performance-based ontology matching, with the proposed methodology as a research outcome. Although our implementation scores impressively during evaluation; however, there are few limitations of current approach, which will be overcome, in future work. Apparently our implementation 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 can be identified automatically.

From the application and usability perspective of our methodology, 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. Presented methodology provides the resolution to these clients by performing performance-based ontology matching in parallel over affordable platforms for fast results. This approach is built to scale from multicore 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 to complement. Above of all, due to the effectiveness-independent approach of our methodology, these experts do not have to worry about any accuracy loss with performance speedup.

Bibliography

[1] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*, 1st ed. Morgan Kaufmann Publishers Inc., Jul. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2401764

[2] F. Hakimpour and A. Geppert, "Resolving semantic heterogeneity in schema integration." in *FOIS*, 2001, pp. 297–308. [Online]. Available: http://dblp.uni-trier.de/db/conf/fois/fois2001.html#HakimpourG01

[3] Microsoft, "Microsoft BizTalk Server," http://www.microsoft.com/biztalk/en/us/default.aspx, 2012.

[4] ——, "Applying Microsoft Patterns to Solve EAI Problems," http://msdn.microsoft.com/en-us/library/ee265635(v=bts.10).aspx, 2004.

[5] Z. Dilong, "Analysis of XML and COIN as Solutions for Data Heterogeneity in Insurance System Integration," Master's thesis, Sloan School of Managment, Massachusetts Institute of Technology (MIT), Cambridge, MA, 2001.

[6] J. Gracia and E. Mena, "Semantic heterogeneity issues on the web." *IEEE Internet Computing*, vol. 16, no. 5, pp. 60–67, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/internet/internet16.html#GraciaM12

[7] J. Euzenat and P. Shvaiko, *Ontology Matching*, 2nd ed. Berlin: Springer, 2013.

[8] A. K. M. A. S. L. Wajahat Ali Khan, Muhammad Bilal Amin and E. S. Kim, "Object oriented and ontology alignment patterns based expressive mediation bridge ontology (mbo)." *Journal of Information Science*, pp. 1–22, 2014.

[9] D. Isern, S. David, and A. Moreno, "Ontology-driven execution of clinical guidelines." *Computer Methods and Programs in Biomedicine*, vol. 107, no. 2, pp. 122–139, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/cmpb/cmpb107.html#IsernSM12

[10] J. Cimino and X. Zhu, "The practical impact of ontologies on biomedical informatics," *IMIA Yearbook of Medical Informatics*, vol. 1, no. 1, pp. 124–135, 2006. [Online]. Available: /brokenurl#http://publication.wilsonwong.me/load.php?id=233282059

[11] P. D. Potter, H. Cools, K. Depraetere, G. Mels, P. Debevere, J. D. Roo, C. Huszka, D. Colaert, E. Mannens, and R. V. de Walle, "Semantic patient information aggregation and medicinal decision support." *Computer Methods and Programs in Biomedicine*, vol. 108, no. 2, pp. 724–735, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/cmpb/cmpb108.html#PotterCDMDRHCMW12

[12] "Gene ontology consortium: The gene ontology (GO) database and informatics resource," *Nucleic Acids Research*, vol. 32, no. Database-Issue, pp. 258–261, 2004. [Online]. Available: http://dx.doi.org/10.1093/nar/gkh036

[13] J. Golbeck, G. Fragoso, F. Hartel, J. Hendler, J. Oberthaler, and B. Parsia, "The national cancer institute's thesaurus and ontology," *Journal of web semantics*, vol. 1, no. 1, pp. 75–80, 2003.

[14] C. Rosse and J. L. V. M. Jr., "A reference ontology for biomedical informatics: the foundational model of anatomy." *Journal of Biomedical Informatics*, vol. 36, no. 6, pp. 478–500, 2003. [Online]. Available: http://dblp.uni-trier.de/db/journals/jbi/jbi36.html#RosseM03

[15] S. Schulz, R. Cornet, and K. A. Spackman, "Consolidating snomed ct's ontological commitment." *Applied Ontology*, vol. 6, no. 1, pp. 1–11, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/ao/ao6.html#SchulzCS11

[16] B. Smith, M. Ashburner, C. Rosse, J. Bard, W. Bug, W. Ceusters, L. Goldberg, K. Eilbeck, A. Ireland, and C. Mungall, "The obo foundry: coordinated evolution of ontologies to

support biomedical data integration," *Nature biotechnology*, vol. 25, no. 11, pp. 1251–1255, 2007.

[17] P. L. Whetzel, N. F. Noy, N. H. Shah, P. R. Alexander, C. Nyulas, T. Tudorache, and M. A. Musen, "Bioportal: enhanced functionality via new web services from the national center for biomedical ontology to access and use ontologies in software applications," *Nucleic acids research*, vol. 39, no. suppl 2, pp. W541–W545, 2011.

[18] A. Algergawy, R. Nayak, N. Siegmund, V. Kppen, and G. Saake, "Combining schema and level-based matching for web service discovery." in *ICWE*, ser. Lecture Notes in Computer Science, B. Benatallah, F. Casati, G. Kappel, and G. Rossi, Eds., vol. 6189. Springer, 2010, pp. 114–128. [Online]. Available: http://dblp.uni-trier.de/db/conf/icwe/icwe2010.html#AlgergawyNSKS10

[19] D. Fensel, H. Lausen, A. Polleres, J. D. Bruijn, M. Stollberg, D. Roman, and J. Domingue, Eds., *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Heidelberg: Springer-Verlag, 2006.

[20] A. Gross, M. Hartung, T. Kirsten, and E. Rahm, "On matching large life science ontologies in parallel." in *DILS*, ser. Lecture Notes in Computer Science, P. Lambrix and G. J. L. Kemp, Eds., vol. 6254. Springer, 2010, pp. 35–49. [Online]. Available: http://dblp.uni-trier.de/db/conf/dils/dils2010.html#GrossHKR10

[21] P. Shvaiko and J. Euzenat, "Ontology matching: State of the art and future challenges." *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 1, pp. 158–176, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/tkde/tkde25.html#ShvaikoE13

[22] T. J. LeBlanc and S. A. Friedberg, "Hpc: A model of structure and change in distributed systems." *IEEE Trans. Computers*, vol. 34, no. 12, pp. 1114–1129, 1985. [Online]. Available: http://dblp.uni-trier.de/db/journals/tc/tc34.html#LeBlancF85

[23] S. Han and H. G. Choi, "Investigation of the parallel efficiency of a pc cluster for the simulation of a cfd problem," *Personal Ubiquitous Comput.*, vol. 18, no. 6, pp. 1303–1314, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1007/s00779-013-0733-4

[24] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[25] M. Amin, A. Shafi, S. Hussain, W. Khan, and S. Lee, "High performance java sockets (hpjs) for scientific health clouds," in *e-Health Networking, Applications and Services (Healthcom), 2012 IEEE 14th International Conference on*, oct. 2012, pp. 477 –480.

[26] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility." *Future Generation Comp. Syst.*, vol. 25, no. 6, pp. 599–616, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/fgcs/fgcs25.html#BuyyaYVBB09

[27] T. Hayamizu, M. Mangan, J. Corradi, J. Kadin, and M. Ringwald, "The adult mouse anatomical dictionary: a tool for annotating and integrating data," *Genome Biology*, vol. 6, pp. 1–8, 2005. [Online]. Available: http://dx.doi.org/10.1186/gb-2005-6-3-r29

[28] G. Stoilos, G. Stamou, and S. Kollias, "A string metric for ontology alignment," in *Proceedings of the 4th international conference on The Semantic Web*, ser. ISWC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 624–637. [Online]. Available: http://dx.doi.org/10.1007/11574620_45

[29] E. J. Ruiz, B. C. Grau, and I. Horrocks, "Evaluating Ontology Matching Systems on Large, Multilingual and Real-world Test Cases," http://www.cs.ox.ac.uk/isg/projects/SEALS/oaei/, 2012.

[30] M. B. Amin, R. Batool, W. A. Khan, S. Lee, and E.-N. Huh, "Sphere," *The Journal of Supercomputing*, vol. 68, no. 1, pp. 274–301, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11227-013-1037-1

[31] W. Hu, "Falcon-AO," http://ws.nju.edu.cn/falcon-ao/, 2010.

[32] H. Seddiqui and M. Aono, "An efficient and scalable algorithm for segmented alignment of ontologies of arbitrary size," *Web Semantics: Science, Services*

and Agents on the World Wide Web, vol. 7, no. 4, 2009. [Online]. Available: http://www.websemanticsjournal.org/index.php/ps/article/view/272

[33] E. Jimnez-Ruiz and B. C. Grau, "Logmap: Logic-based and scalable ontology matching." vol. 7031, pp. 273–288, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/semweb/iswc2011-1.html#Jimenez-RuizG11

[34] J. David, F. Guillet, and H. Briand, "Matching directories and owl ontologies with aroma," in *Proceedings of the 15th ACM international conference on Information and knowledge management*, ser. CIKM '06. New York, NY, USA: ACM, 2006, pp. 830–831. [Online]. Available: http://doi.acm.org/10.1145/1183614.1183752

[35] I. F. Cruz, F. P. Antonelli, and C. Stroe, "Agreementmaker: Efficient matching for large real-world schemas and ontologies." *PVLDB*, vol. 2, no. 2, pp. 1586–1589, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/pvldb/pvldb2.html#CruzAS09

[36] M. B. Amin, W. A. Khan, S. Lee, and B. H. Kang, "Performance-based ontology matching," *Applied Intelligence*, pp. 1–30, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10489-015-0648-z

[37] A. C. G. G. E. D. V. A. Tenschert, M. Assel and I. Celino, "Parallelization and distribution techniques for ontology matching in urban computing environments," 2009.

[38] D. Andrade, B. B. Fraguela, J. C. Brodman, and D. A. Padua, "Task-parallel versus data-parallel library-based programming in multicore systems." in *PDP*, D. E. Baz, F. Spies, and T. Gross, Eds. IEEE Computer Society, 2009, pp. 101–110. [Online]. Available: http://dblp.uni-trier.de/db/conf/pdp/pdp2009.html#AndradeFBP09

[39] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering in distributed systems." *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 3, pp. 568–586, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/pami/pami33.html#ChenSBLC11

[40] I. D. Zone, "Choose the Right Threading Model (Task-Parallel or Data-Parallel Threading)," https://software.intel.com/en-us/articles/choose-the-right-threading-model-task-parallel-or-data-parallel-threading, 2011.

[41] "Apache Hadoop," https://hadoop.apache.org.

[42] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[43] "HDFS Architecture Guide," http://hadoop.apache.org/docs/hdfs/current/hdfs_design.html.

[44] T. Kirsten, A. Gross, M. Hartung, and E. Rahm, "Gomma: a component-based infrastructure for managing and analyzing life science ontologies and their evolution," *J. Biomedical Semantics*, vol. 2, p. 6, 2011.

[45] ——, "Gomma: a component-based infrastructure for managing and analyzing life science ontologies and their evolution," *Journal of Biomedical Semantics*, vol. 2, no. 1, 2011. [Online]. Available: http://dx.doi.org/10.1186/2041-1480-2-6

[46] Y. R. Jean-Mary, E. P. Shironoshita, and M. R. Kabuka, "Ontology Matching with Semantic Verification," *Web Semantics*, vol. 7, no. 3, pp. 235–251, September 2009.

[47] W. Hu and Y. Qu, "Falcon-AO: a practical ontology matching system," *Web Semantics*, vol. 6, no. 3, pp. 237–239, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1412999&jmp=cit&coll=&dl=GUIDE

[48] W. Hu, Y. Qu, and G. Cheng, "Matching large ontologies: A divide-and-conquer approach." *Data Knowl. Eng.*, vol. 67, no. 1, pp. 140–160, 2008. [Online]. Available: http://dblp.uni-trier.de/db/journals/dke/dke67.html#HuQC08

[49] M. S. Hanif and M. Aono, "An efficient and scalable algorithm for segmented alignment of ontologies of arbitrary size." *J. Web Sem.*, vol. 7, no. 4, pp. 344–356, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/ws/ws7.html#HanifA09

[50] S. Garruzzo and D. Rosaci, "Agent clustering based on semantic negotiation." *TAAS*, vol. 3, no. 2, 2008. [Online]. Available: http://dblp.uni-trier.de/db/journals/taas/taas3.html#GarruzzoR08

[51] P. D. Meo, G. Quattrone, D. Rosaci, and D. Ursino, "Bilateral semantic negotiation: a decentralised approach to ontology enrichment in open multi-agent systems." *IJDMMM*, vol. 4, no. 1, pp. 1–38, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijdmmm/ijdmmm4.html#MeoQRU12

[52] S. Garruzzo and D. Rosaci, "Information agents that learn to understand each other via semantic negotiation." in *DAIS*, ser. Lecture Notes in Computer Science, F. Eliassen and A. Montresor, Eds., vol. 4025. Springer, 2006, pp. 99–112. [Online]. Available: http://dblp.uni-trier.de/db/conf/dais/dais2006.html#GarruzzoR06

[53] ——, "Information agents that learn to understand each other via semantic negotiation." in *DAIS*, ser. Lecture Notes in Computer Science, F. Eliassen and A. Montresor, Eds., vol. 4025. Springer, 2006, pp. 99–112. [Online]. Available: http://dblp.uni-trier.de/db/conf/dais/dais2006.html#GarruzzoR06

[54] "JADE TUTORIAL, JADE PROGRAMMING FOR BEGINNERS Giovanni Caire (TILAB, formerly CSELT)," http://www.cs.uu.nl/docs/vakken/map/JADEProgramming-Tutorial-for-beginners.pdf.

[55] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds., *Advances in knowledge discovery and data mining*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1996.

[56] W. J. Frawley, G. Piatetsky-shapiro, and C. J. Matheus, "Knowledge discovery in databases: an overview," 1992.

[57] T. Kirsten, L. Kolb, M. Hartung, A. Gross, H. Kpcke, and E. Rahm, "Data partitioning for parallel entity matching," *CoRR*, vol. abs/1006.5309, 2010, informal publication. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1006.html#abs-1006-5309

[58] E. Jimnez-Ruiz, C. Meilicke, B. C. Grau, and I. Horrocks, "Evaluating mapping repair systems with large biomedical ontologies." vol. 1014, pp. 246–257, 2013. [Online]. Available: http://dblp.uni-trier.de/db/conf/dlog/dlog2013.html#Jimenez-RuizMGH13

[59] P. Lambrix and H. Tan, "Sambo - a system for aligning and merging biomedical ontologies." *J. Web Sem.*, vol. 4, no. 3, pp. 196–206, 2006. [Online]. Available: http://dblp.uni-trier.de/db/journals/ws/ws4.html#LambrixT06

[60] "What is WordNet?, Princeton University 2013," https://wordnet.princeton.edu.

[61] "National Center for Biotechnology Information, U.S. National Library of Medicine, PubMed, 2013," http://www.ncbi.nlm.nih.gov/pubmed.

[62] T. Takai-Igarashi and T. Takagi, "Signal-ontology: Ontology for cell signaling," vol. 11, 2000.

[63] "U.S. National Library of Medicine, National Institute of Health, Medical Subject Headings, 2013." http://www.nlm.nih.gov/mesh/MBrowser.html.

[64] P. Lambrix, H. Tan, and Q. L. 0002, "Sambo and sambodtf results for the ontology alignment evaluation initiative 2008." vol. 431, 2008. [Online]. Available: http://dblp.uni-trier.de/db/conf/semweb/om2008.html#LambrixTL08

[65] S. Zhang and O. Bodenreider, "Hybrid alignment strategy for anatomical ontologies: Results of the 2007 ontology alignment contest." vol. 304, 2007. [Online]. Available: http://dblp.uni-trier.de/db/conf/semweb/om2007.html#ZhangB07

[66] M. Ba and G. Diallo, "Large-scale biomedical ontology matching with ServOMap," *IRBM*, vol. 34, no. 1, pp. 56–59, 2013.

[67] "HDFS Architecture Guide," http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[68] "To Hadoop or Not to Hadoop?, Anand Krishnaswamy, 2013," http://www.thoughtworks.com/insights/blog/hadoop-or-not-hadoop.

[69] A. Matsunaga, M. Tsugawa, and J. Fortes, "Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, Dec 2008, pp. 222–229.

[70] "Reasoning-Hadoop," http://www.jacopourbani.it/reasoning-hadoop.html.

[71] "Heart Project," http://rdf-proj.blogspot.kr/.

[72] "Hadoop Distributed RDF Store," https://code.google.com/p/hdrs/.

[73] "Flynnś Taxonomy." http://en.wikipedia.org/wiki/Flynn\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{s\global\mathchardef\accent@spacefactor\spacefactor}\accent19s\egroup\spacefactor\accent@spacefactor_taxonomy.

[74] M.-J. Park, J. Lee, C.-H. Lee, J. Lin, O. Serres, and C.-W. Chung, "An efficient and scalable management of ontology." in *DASFAA*, ser. Lecture Notes in Computer Science, K. Ramamohanarao, P. R. Krishna, M. K. Mohania, and E. Nantajeewarawat, Eds., vol. 4443. Springer, 2007, pp. 975–980. [Online]. Available: http://dblp.uni-trier.de/db/conf/dasfaa/dasfaa2007.html#ParkLLLSC07

[75] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan, "Minerva: A scalable owl ontology storage and inference system," in *ASWC*, 2006, pp. 429–443.

[76] G. Zhao and R. Meersman, "Architecting ontology for scalability and versatility," in *OTM Conferences (2)*, ser. Lecture Notes in Computer Science, R. Meersman, Z. Tari, M.-S. Hacid, J. Mylopoulos, B. Pernici, Ö. Babaoglu, H.-A. Jacobsen, J. P. Loyall, M. Kifer, and S. Spaccapietra, Eds., vol. 3761. Springer, 2005, pp. 1605–1614.

[77] J. Bloch, *Effective Java (2nd Edition)*. Addison-Wesley, 2008.

[78] R. Rivest, "The MD5 Message-Digest Algorithm," Internet Requests for Comment, RFC Editor, Fremont, CA, USA, Tech. Rep. 1321, Apr. 1992. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1321.txt

[79] A. M. Khattak, K. Latif, and S. Lee, "Change management in evolving web ontologies," *Knowledge-Based Systems*, vol. 37, no. 0, pp. 1 – 18, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705112001323

[80] A. M. Khattak, Z. Pervez, K. Latif, and S. Lee, "Time efficient reconciliation of mappings in dynamic web ontologies." *Knowl.-Based Syst.*, vol. 35, pp. 369–374, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/kbs/kbs35.html#KhattakPLL12

[81] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[82] H. Lpez-Fernndez, M. Reboiro-Jato, D. Glez-Pea, F. Aparicio, D. Gachet, M. Buenaga, and F. Fdez-Riverola, "Bioannote: A software platform for annotating biomedical documents with application in medical learning environments." *Computer Methods and Programs in Biomedicine*, vol. 111, no. 1, pp. 139–147, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/cmpb/cmpb111.html#Lopez-FernandezRGAGBF13

[83] X. Sun and J. Li, "pairheatmap: Comparing expression profiles of gene groups in heatmaps." *Computer Methods and Programs in Biomedicine*, vol. 112, no. 3, pp. 599–606, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/cmpb/cmpb112.html#SunL13

[84] J. H. Gennari and A. Silberfein, "Leveraging an alignment between two large ontologies: Fma and go," 2004.

[85] M. A. M. B. A. M. A. S. Wajahat Ali Khan, Maqbool Hussain and S. Lee, "Personalized-detailed clinical model for data interoperability among clinical standards," *Telemedicine and e-Health*, vol. 19, no. 3, pp. 632–642, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/cmpb/cmpb112.html#SunL13

[86] L. Magee, "Somet: Shared ontology matching environment." in *Ontology Matching*, ser. CEUR Workshop Proceedings, P. Shvaiko, J. Euzenat, N. F. Noy, H. Stuckenschmidt, V. R.

Benjamins, and M. Uschold, Eds., vol. 225. CEUR-WS.org, 2006. [Online]. Available: http://dblp.uni-trier.de/db/conf/semweb/om2006.html#Magee06

[87] G. Correndo and H. Alani, "Collaborative ontology mapping and data sharing," 2008.

[88] "RESTful Web services: The basics," http://www.ibm.com/developerworks/library/ws-restful//.

[89] M. S. T. P L Schuyler, W T Hole and D. D. Sherertz, "The umls metathesaurus: representing different views of biomedical concepts," *Bull Med Libr Assoc*, vol. 81, no. 3, pp. 217–222, 1993.

[90] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001. [Online]. Available: http://doi.acm.org/10.1145/375360.375365

[91] K. E. J. . E. A. F. R. G. V. I. E. J. .-R. A. O. K. P. L. A. N. H. P. D. R. F. . S. P. S. C. . T. Bernardo Cuenca Grau, Zlatan Dragisic and O. Zamazal, "Results of the ontology alignment evaluation initiative 2013." ser. CEUR Workshop Proceedings. CEUR-WS.org.

[92] Z. Dragisic, K. Eckert, J. Euzenat, D. Faria, A. Ferrara, R. Granada, V. Ivanova, E. Jiménez-Ruiz, A. O. Kempf, P. Lambrix, S. Montanelli, H. Paulheim, D. Ritze, P. Shvaiko, A. Solimando, C. T. dos Santos, O. Zamazal, and B. C. Grau, "Results of the ontology alignment evaluation initiative 2014," in *Proceedings of the 9th International Workshop on Ontology Matching collocated with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Trentino, Italy, October 20, 2014.*, 2014, pp. 61–104. [Online]. Available: http://ceur-ws.org/Vol-1317/oaei14_paper0.pdf

[93] "Intel Hyper-Threading Technology, Intel Corporation 2013," http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[94] "Oracle - Java 8," http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html.

[95] "Concurrent access to Models," http://jena.apache.org/documentation/notes/concurrency-howto.html.

[96] "Jena, a Framework for developing Semantic Web Applications," http://semanticwebbuzz.blogspot.kr/2009/10/jena-framework-for-developing-semantic.html.

[97] M. Horridge and S. Bechhofer, "The owl api: A java api for owl ontologies," *Semant. web*, vol. 2, no. 1, pp. 11–21, Jan. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2019470.2019471

[98] The owl api - a java api for working with owl 2 ontologies. [Online]. Available: http://ontolog.cim3.net/file/work/OWL2/OWL-2_Tools-n-Applications/owl-api-presentation--MatthewHorridge_20100805.pdf

[99] "Do you know your data size?, Vladimir Roubtsov, JavaWorld," http://www.javaworld.com/javatips/jw-javatip130.html.

[100] "How to get max memory, free memory and total memory in Java, Javin Paul, Javarevisited," http://javarevisited.blogspot.kr/2012/01/find-max-free-total-memory-in-java.html.

[101] "Java Performance - Memory and Runtime Analysis - Tutorial, Lars Vogel, vogella.com," http://www.vogella.com/articles/JavaPerformance/article.html.

[102] "Adult Mouse Anatomy," http://www.informatics.jax.org/searches/AMA_form.shtml.

[103] "STW Thesaurus of Economics Ontology," http://zbw.eu/stw/versions/8.10/descriptor/29234-2/about.en.html.

[104] "Thesaurus for the Social Sciences," http://www.gesis.org/en/services/research/thesauri-und-klassifikationen/social-science-thesaurus/.

Appendix A

List of Publications

International Journal Papers:

[1] **Muhammad Bilal Amin**, Wajahat Ali Khan, Byeong Ho Kang, and Sungyoung Lee, "Performance-based Ontology Matching, A Data-parallel approach for an Effectiveness-independent Performance-gain in Ontology Matching", Applied Intelligence (SCI, IF:1.83), 2015 : 1-30

[2] **Muhammad Bilal Amin**, Rabia Batool, Wajahat Ali Khan, Sungyoung Lee, and Eui-Nam Huh, "SPHeRe:A Performance Initiative towards Ontology Matching by implementing Parallelism over Cloud Platform." The Journal of Supercomputing (SCI, IF:0.917), 68, no. 1 (2014): 274-301.

[3] Dinh-Mao Bui, Huu-Quoc Nguyen, YongIk Yoon, SungIk Jun, **Muhammad Bilal Amin**, and Sungyoung Lee, "Gaussian process for predicting CPU utilization and its application to Energy Efficiency." Applied Intelligence (SCI, IF:1.83), 2015

[4] Shujaat Hussain, Jae Hun Bang, Manhyung Han, Muhammad Idris Ahmed, **Muhammad Bilal Amin**, Chris Nugent, Sally McClean, Bryan Scotney, Gerard Parr and Sungyoung Lee. "Behavior Life Style analysis for mobile sensory data in cloud computing through MapReduce." Sensors (SCIE, IF:2.04), (2014): 14(11), 22001-22020.

[5] Wajahat Ali Khan, **Muhammad Bilal Amin**, Asad Masood Khattak, Maqbool Hussain, Muhammad Afzal, Sungyoung Lee and Eun Soo Kim,"Object Oriented and Ontology Alignment Patterns based Expressive Mediation Bridge Ontology (MBO)", Journal of Information Science, (SCIE, IF:1.08), 2014.

[6] Wajahat Ali Khan, Asad Masood Khattak, Maqbool Hussain, **Muhammad Bilal Amin**, Muhammad Afzal, Christopher Nugent and Sungyoung Lee, “An Adaptive Semantic based Mediation System for Data Interoperability among Health Information Systems”, Journal of Medical Systems (SCIE, IF 1.372), 38(8):28, 2014.

[7] Wajahat Ali Khan, Maqbool Hussain, Muhammad Afzal, **Muhammad Bilal Amin**, Muhammad Aamir Saleem and Sungyoung Lee, ”Personalized-Detailed Clinical Model for Data Interoperability among Clinical Standards”, Telemedicine and EHealth (SCI, IF:1.544), Vol. 19 Issue 8, pp.632-642, 2013.

[8] Maqbool Hussain, Asad Masood Khattak, Wajahat Ali Khan, Iram Fatima, **Muhammad Bilal Amin**, Zeeshan Pervez, Rabia Batool, Muhammad Amir Saleem, Muhammad Afzal, Muhammad Fahim, Muhammad Hameed Saddiqi, Sungyoung Lee, and Khalid Latif, “Cloud-based Smart CDSS for Chronic Diseases”, In Journal of Health and Technology - 3, no. 2 (2013): 153-175.

International Conference Papers:

[9] **Muhammad Bilal Amin**, Shujaat Hussain, Manhyung Han, Byeong Ho Kang, Yoon Yong Ik, SungIk Jun, Sungyoung Lee, "Profiling-Based Energy-Aware Recommendation System for Cloud Platforms", Computer Science and its Applications Lecture Notes in Electrical Engineering Volume 330, 2015, pp 851-859.

[10] **Muhammad Bilal Amin**, Mahmood Ahmad, Wajahat Ali Khan, and Sungyoung Lee, "Biomedical Ontology Matching as a Service", Smart Homes and Health Telematics Lecture Notes in Computer Science 2015, pp 195-203.

[11] **Muhammad Bilal Amin**, Aamir Shafi, Shujaat Hussain, Wajahat Ali Khan, and Sungyoung Lee. "High performance Java sockets (HPJS) for scientific health clouds." In e-Health Networking, Applications and Services (Healthcom), 2012 IEEE 14th International Conference on, pp. 477-480. IEEE, 2012.

[12] **Muhammad Bilal Amin**, Wajahat Ali Khan, Ammar Ahmad Awan, and Sungyoung Lee. "Intercloud message exchange middleware." In Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, p. 79. ACM, 2012.

[13] Banos, O., **Muhammad Bilal Amin**, Ali Khan, W., Afzal, M., Ali, T., Kang, B. H., Lee, S. The Mining Minds Platform: a Novel Person-Centered Digital Health and Wellness Framework. Proceedings of the 9th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth 2015), Istanbul, Turkey, May 20-23, (2015)

[14] Oresti Banos, **Muhammad Bilal Amin**, Wajahat Ali Khan, Muhammad Afzal, Mahmood Ahmad, Maqbool Ali, Taqdir Ali, Rahman Ali, Muhammad Bilal, Manhyung Han, Jamil Hussain, Maqbool Hussain, Shujaat Hussain, Tae Ho Hur, Jae Hun Bang, Thien Huynh-The, Muhammad Idris, Dong Wook Kang, Sang Beom Park, Hameed Siddiqui, Le-Va Vui, Muhammad Fahim, Asad Masood Khattak, Byeong Ho Kang and Sungyoung Lee. "The Mining Minds Platform: a Novel Person-Centered Digital Health and Wellness Framework." IWBBIO 2015 (3rd International Work-Conference on Bioinformatics and Biomedical Engineering).

[15] Wajahat Ali Khan, Maqbool Hussain, **Muhammad Bilal Amin**, Asad Masood Khattak, Muhammad Afzal, and Sungyoung Lee. "AdapteR Interoperability ENgine (ARIEN): An approach of Interoperable CDSS for Ubiquitous Healthcare." In Ubiquitous Computing and Ambient Intelligence. Context-Awareness and Context-Driven Interaction, pp. 247-253. Springer International Publishing, 2013.

[16] Wajahat Ali Khan, **Muhammad Bilal Amin**, Asad Masood Khattak, Maqbool Hussain, and Sungyoung Lee. "System for Parallel Heterogeneity Resolution (SPHeRe) results for OAEI 2013." In OM, pp. 184-189. 2013.

[17] Shujaat Hussain, **Muhammad Bilal Amin**, Jae Hun Bang, Manhyung Han, Sungyoung Lee, Chris Nugent, Sally McClean, Brian Scotney, Gerald Parr, "Activity recognition and resource optimization in mobile cloud through MapReduce 2013 IEEE 15th International Conference on e-Health Networking, Applications and Services (Healthcom 2013) (pp. 471475). doi:10.1109/HealthCom.2013.6720722

[18] Ammar Ahmad Awan, **Muhammad Bilal Amin**, Shujaat Hussain, Aamir Shafi and Sungyoung Lee, "An MPI-IO Compliant Java based Parallel I/O Library." 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid2013), Delft, Netherlands, May 13-16, 2013.

[19] Wajahat Ali Khan, Maqbool Hussain, Muhammad Afzal, **Muhammad Bilal Amin**, and Sungyoung Lee, "Healthcare standards based sensory data exchange for Home Healthcare Monitoring System." In Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE, pp. 1274-1277. IEEE, 2012.

[20] Wajahat Ali Khan, Maqbool Hussain, Asad Masood Khattak, Muhammad Afzal, **Muhammad Bilal Amin**, and Sungyoung Lee, "Integration of HL7 Compliant Smart Home Healthcare System and HMIS", 10th International Conference On Smart Homes and Health Telematics (ICOST 2012), Artimino, Italy, June 15-18, 2012

[21] Shujaat Hussain, **Muhammad Bilal Amin**, Zeeshan Pervez, Ammar Ahmad Awan, and Sungyoung Lee, "A Hybrid Cloud Based Smart Home, Ambient Assisted Living 2012 (AAL)

27 - 29th June 2012, Euskalduna Conference Center, Bilbao

[22] Wajahat Ali Khan, Asad Masood Khattak, Sungyoung Lee, Maqbool Hussain, **Muhammad Bilal Amin**, and Khalid Latif, "Achieving interoperability among healthcare standards: building semantic mappings at models level." In Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, p. 101. ACM, 2012.

[23] Wajahat Ali Khan, Maqbool Hussain, Asad Masood Khattak, **Muhammad Bilal Amin**, and Sungyoung Lee, "SaaS based interoperability service for semantic mappings among healthcare standards." In 8th International Conference on Innovations in Information Technology. 2012.

Domestic Conference Papers:

[24] **Muhammad Bilal Amin**, Shujaat Hussain, Manhyung Han, Sungyoung Lee, and Yong Ik Yoon, Prediction, Provisioning, Elastic (PPE) Energy-aware recommendations model for Cloud Platforms, 2013 Korea Information Science 40th Annual General Meeting , 2013.11, 72-74

[25] **Muhammad Bilal Amin**, Sungyoung Lee, and Young-Koo Lee ” cMac : A Context-aware Mobile Apps-on-a-Cloud Architecture”, Korea Information Processing Society 2011 Spring Conference of the 35th / 2011 Apr. 30 , 2011, pp.40-42

[26] **Muhammad Bilal Amin**, Wajahat Ali Khan, Sungyoung Lee, Young-Koo Lee ” Cloud Computing for Healthcare IT Infrastructure- Utilization models for Hybrid and Community Clouds” Korea Information Science 2011 Korea Society of Computer General Conference Article 38 No. 1 (A) 2011.6, 112-115

Appendix B

MSRA 2013, Accepted Proposal

Title: **Semantic Heterogeneity Resolution by Implementing Parallelism over Multicore cloud platform**

Funded by: **Microsoft Research Asia, Beijing, China (MSRA. 2013)**

Duration: **June 2013 - June 2014**

B.1 Abstract

The abundance of semantically related information over the web has resulted in semantic heterogeneity. For resolution, ontology matching tools and techniques are utilized by semantic web systems. However, effective ontology matching is computationally intensive i.e., a time consuming process. Medium to large-scale ontologies take from hours up to days of computation time, depending upon the provided computational resources and the complexity of the utilized matching algorithms. This delay in producing required results makes ontology matching unsuitable for semantic web-based interactive and semi-real time systems. So far the evolution in ontology matching from the perspective of performance is either by improvement of ontology matching algorithms or by partitioning of ontologies for matching algorithms. However, none of the techniques and tools implemented takes the explicit benefit of better computational resources available today as commodity hardware. Commodity desktop, laptops, and even smartphones are equipped with hardware to perform parallel tasks. Multicore nature of todays microprocessor is equipped to handle multiple requests over multiple processing cores, treating each core as a virtual microprocessor. Instead of confining to a desktop, a laptop, or a smartphone, a huge resource of multicore processors is available in the form of todays cloud platform. With clouds reliability, cost, and ubiquity in focus, a technique needs to be developed that can implement ontology matching over mul-

ticore cloud platform. This proposal presents one such technique that implements data-parallelism over multicore cloud platform, by exploiting the individual cores for the benefit of ontology matching. Consequently, a system of parallel ontology matching will be developed. This system will be deployed over Microsofts Azure platform and will utilize dot nets parallel programming and concurrency libraries for multicore exploitation, resulting in high performance ontology matching solution. Technique described in this proposal bridges the gap between semantic heterogeneity resolution and cloud computing.

B.2 Problem Statment

Effective ontology matching is a computationally intensive operation, requiring ontologys resource-based matching algorithms like Name-based, Hierarchy-based, Annotation-based, and Property-based matchers to be executed over candidate ontologies. The process of ontology matching between two ontologies is a Cartesian product of all the concepts and their relationships leading to quadratic complexity with respect to ontology size. Several ontology-matching systems have emerged over the years; however, the techniques for achieving better performance during ontology matching are either focus on the optimization of matching algorithms or the fragmentation of ontologies. The performance improvement based on exploitation of newer hardware has largely been missing. Among these technologies, is the exploitation of parallelism enabled multicore processors, largely available over distributed cloud platform as commodity hardware.

A performance driven initiative is required that avails the opportunity of affordable commodity infrastructure in the form of cloud, to be utilized for parallel ontology matching and execution; consequently, providing improved performance.

B.3 Technical Importance

In the era of automation, integration of information has become a key tool for providing knowledge driven services. With the excess of information over the web, problems regarding information heterogeneity have emerged. Data heterogeneity has solutions based on data definitions, types, formats, and precision. Semantic heterogeneity; however, involves data's intend, making it a

challenging opportunity for integration. The volume of data makes manual annotation of concepts unrealistic; consequently, software agents use automated solutions based on ontologies. The most prominent solution for semantic heterogeneity resolution is ontology matching, which determines conformity among semantically related ontologies. Mappings drawn from ontology matching can be further utilized in information systems and database integration, e-commerce systems, semantic web services, and social networks. Ontology matching being a computationally intensive problem requires performance efficient resolution, so that it can be suitable for interactive and semi-real-time systems. A possible solution to improve performance during ontology matching is the exploitation of parallelism enabled multicore hardware, readily available over cloud platform. This proposal presents one such solution.

B.4 Objectives

- Improve over all ontology matching performance without trading off accuracy by implementing parallelism over multicore cloud platform;
- Lower memory strains by having smaller memory footprint of ontologies utilized during matching in parallel;
- Thread-safe object model (ontology model) to enable parallelism;
- Improved scalability by utilizing available computation resources (cores) to their fullest.

B.5 Methodology

In order to improve ontology matchings performance we are proposing an end-to-end parallel system that exploits multicore hardware for its benefit from ontology loading till result delivery. Figure above illustrates our proposed system. Following are the details of components from the figure and their inter-component communication.

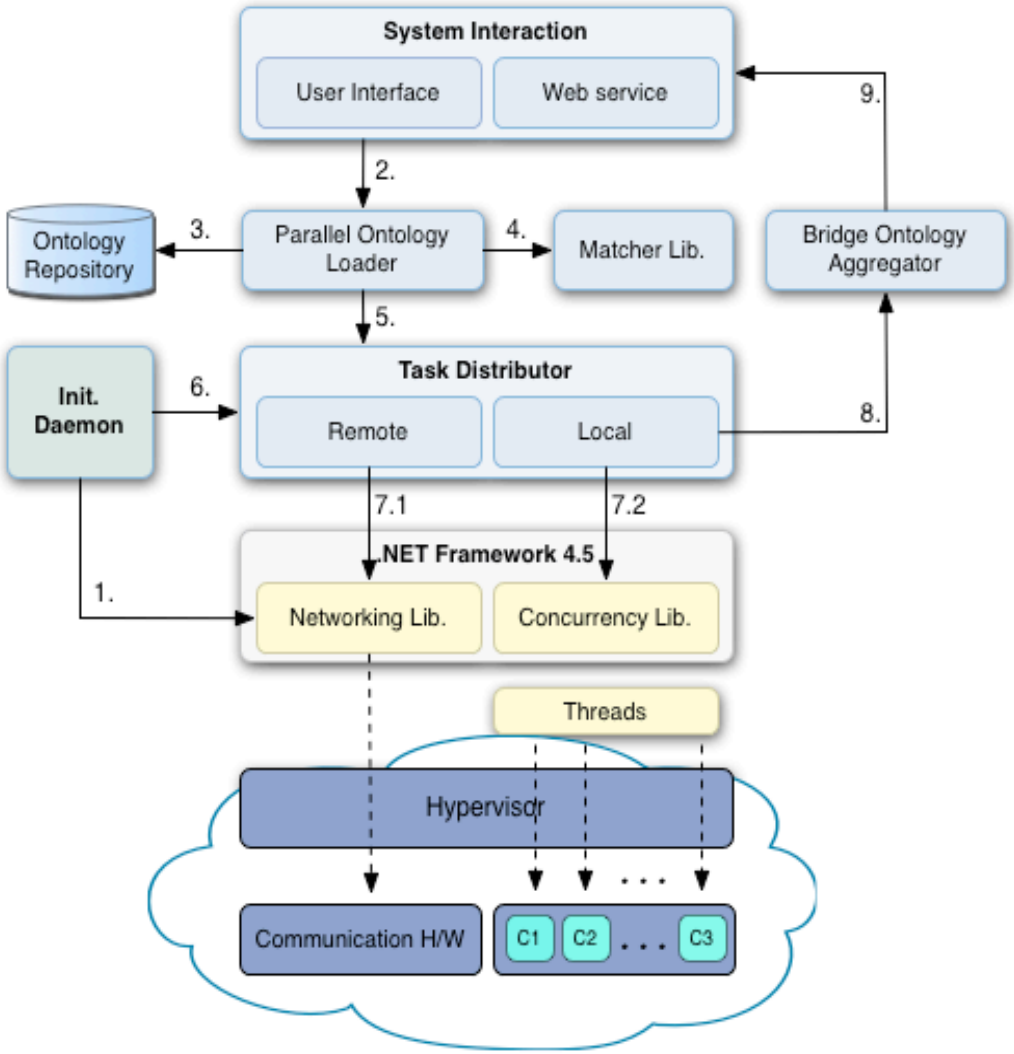


Figure B.1: Architecture

Init daemon

Pre-condition: Our parallel ontology matching system is deployed as instances on all the cloud nodes, participating in ontology matching process.

Process: Init. Daemon executes first to make system aware of over all computational resources (local and remote) contributing. For remote, all Init. Daemons from their respective instances

share their socket objects with others; consequently, a socket table of contributing nodes is created at every system instance.

Post-condition: Overall knowledge of available computational resources.

Collaborator: (1.) Networking libraries from .NET Framework 4.5, used for socket communication (6.) Socket table containing socket objects of participating nodes.

System interaction

Pre-condition: Our parallel ontology matching system is deployed and overall knowledge of available computational resources is known to all instances.

Process: System Interaction component provides a User Interface for clients to interact by providing the candidate ontologies to be matched. Matched results are returned to the client as bridge ontology. For third-party systems, a web-service is also available for interaction. Participating node that invokes matching request is treated as primary node. Furthermore, this node is responsible for ontology partitioning and matching task assignment in parallel environment.

Post-condition: Candidate ontologies are fed to the system for matching.

Collaborator: (2.) Parallel ontology loader is fed with candidate ontologies to be serialized.

Parallel ontology loader

Pre-condition: Candidate ontologies for matching are known to the system.

Process: Candidate ontologies are parsed in parallel. Parsed ontology resources are populated in multiple thread safe ontology model objects. Each object encapsulates the information required by a single matching algorithm during runtime. Furthermore, redundancy like URI based names of concepts etc., is removed during this process. This keeps system to load un-necessary and redundant information in main memory during execution, preventing memory strains at runtime. Ontology model objects are persisted as serialized objects in Ontology Repository for future requests. If request for same ontologies is submitted in future, Candidate ontology loader skips the parsing process and loads the required serialized objects from ontology repository.

Post-condition: Candidate ontologies are persisted as serialized objects in ontology repository and loaded as ontology model for Task Distributor.

Collaborator: (3.) Ontology repository to persist serialized ontology objects. (4.) Matcher library for assignment of matcher algorithms to be executed over ontology objects. (5.) Task distributor component to implement partitioning scheme on ontology objects for parallel matching.

Ontology process

Process: Persists the serialized objects of object models for candidate ontologies. Ontology repository is synchronized over system instances by cloud data synchronization service.

Matching library

Pre-condition: Candidate ontologies are available as serialized objects in ontology repository.
Process: Matcher component provides a library of ontology matching algorithm. These algorithms are classified into primary, secondary, and complementary type. Primary algorithms execute for every matching request, secondary algorithms execute for higher accuracy, and complimentary algorithms execute with respect of ontology scope.
Post-condition: Instance of matching algorithms to be executed over candidate ontologies is assigned.

Task distributor

Pre-condition: Candidate ontologies are loaded for matchers.
Process: Task distributor partitions the candidate ontologies as subsets and assign over to the computing cores available. Several partitioning schemes including size-based and complexity-based partitioning can be used. For local resources, threads are assigned to perform parallel matching invoking available cores. For remote resources, control message is generated for participating nodes regarding their chunk of partition to work and matching algorithm to execute. Each node after receiving the control message loads its own partition of candidate ontology from its local ontology repository for parallel matching over available computing cores.
Post-condition: All parallel matchers complete their assigned matching task over their partition of candidate ontologies.
Collaborator: (7.1) Networking libraries from .NET Framework 4.5, used for sending control

message and receive computed results. (7.2) Concurrency libraries from .NET Framework 4.5, used for invoking multicore platform via threads. (8.) Bridge ontology aggregator for results accumulation.

Bridge Ontology Aggregator

Pre-condition: All parallel matchers have completed their assigned matching tasks and transmitted their results to the primary node.
Process: Every participating node generates their respective matched results. Bridge ontology aggregator, accumulates these results and generate a bridge ontology file. Bridge ontology aggregator on primary node has the extra responsibility of aggregating the local and remote bridge ontology files to compile as one. URI to this aggregated bridge ontology is generated and provided back to the client. This ontology, if required, can also be persisted in ontology repository for future use.
Post-condition: Aggregated bridge ontology is created, persisted in ontology repository and delivered to the client.
Collaborator: (9.) Delivery of aggregated bridge ontology file to the client via system interaction component.

B.6 Social Impact

- Efficient access of medical records at the point of care;
- Reduced medication errors, and unnecessary medical procedures;
- Improved patient care coordination among health-care professionalsThread-safe object model (ontology model) to enable parallelism;
- Ubiquitous availability of information through services for communication between health-care systems;
- No technological infrastructure burden on organization as cloud based services is provided.

Benefit to Talent and Research tool

A prototype implementation for a candidate standard with public release as open-source tool for other researchers in the field of health-care standardization to benefit.

B.7 Adoption of Technologies

- Our Solution will be built using Microsoft Technologies and Eclipse for required Java based Components;
- Platform : Microsoft.NET 4.0 and JDK 1.5;
- Language: C# and Java;
- IDE : Visual Studio 2010, Eclipse for Java EE;
- Deployment Environment: Microsoft Azure Platform;
- Application Server: Windows Azure and Tomcat;
- Database Server: SQL Azure;
- Database Server for Development and in-house Testing: SQL Server 2008 R2;
- Supporting Technologies : Windows 7 x64 Ultimate for Development, Microsoft Office 2010 Professional, Microsoft Visio, Microsoft Azure SDK, WindowsAzure4j;
- Research Technologies Tools and Languages: Protege 4.1, Falcon, Agreement Maker, Microsoft Biztalk 2010, OWL.

Appendix C

Azure4Research 2014, Accepted Proposal

Title: **Enabling Data Parallelism for large-scale Biomedical Ontology Matching over Multicore Cloud Instances**

Funded by: **Microsoft Research, Redmond, USA. (Azure4Research Award. 2014)**

Duration: **January 2014 - January 2015**

C.1 Abstract

Ontology matching is among the core techniques used for integration and interoperability resolution between biomedical systems. However, due to the ever-evolving nature of biomedical data, ontologies are becoming large-scale and complex; consequently, leading to performance bottlenecks during matching. In this proposal, we present a parallel ontology matching system for large-scale biomedical ontologies that implements data parallelism over multicore cloud platform for performance benefits. Our system decomposes these complex ontologies into smaller and simpler subsets depending on 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 by thread-level parallelism over multicore cloud instance. Matched results from these abstractions are aggregated to generate mediation bridge ontology. We evaluate our system by integrating it with the interoperability engine of a clinical decision support system (CDSS), which generates mapping requests for large-scale NCI, FMA, and SNOMED-CT biomedical ontologies.

C.2 Problem Statment

Large-scale biomedical ontologies are complex in nature, leading to performance hindrance during their usage among biomedical systems. Ontology matching systems developed over the years have taken large-scale biomedical ontologies into consideration and have implemented possible resolutions. However, These resolutions are more focused on optimization of the matching algorithms and partitioning of larger ontologies into smaller chunks for performance benefits. Nevertheless, ontology matching being a quadratic complexity problem can go to a certain extent in gaining performance by optimizing only the algorithms. Furthermore, the performance improvement based-on exploitation of parallel and distributed techniques has largely been missed. Among these technologies are parallel multicore cloud instances, which can be exploited by thread-level parallelism over virtual cores for large-scale biomedical ontology matching in parallel.

C.3 Importance

Large-scale biomedical ontologies contain overlapping information, utilization of which is necessary for the integration, aggregation, and interoperability; for example, the plethora of web-based medical information resources provide related information over the Internet. If these resources are annotated by ontologies, software agents can automatically aggregate information for biomedical researchers and other biomedical querying systems. For example, NCI ontology defines the concept of Myocardium related to the concept Cardiac Muscle Tissue, which describes the muscles surrounding human heart. Concept Cardiac Muscle Tissue is defined in FMA ontology; therefore, a biomedical system integrating knowledge regarding human heart requires correspondence between candidate ontologies FMA and NCI. Likewise, GO is a highly organized structure of medical knowledge facilitating medical genetics. It is widely used by biomedical researchers in numerous genetics research fields including gene group-based analysis for discovering the hidden links overlooked by the single-gene analysis. Finding matching between GO ontology and FMA ontology can be utilized by molecular biologist in understanding the outcome of proteomics and genomics in a large-scale anatomic view. Moreover, correspondence between ontologies has also been used for heterogeneity resolution among various health standards; however, ontology

matching over large-scale biomedical ontologies is a computationally intensive task with computational complexity $O(n^2)$. Ontology matching is a Cartesian product of two candidate ontologies, which requires Resource-based matching algorithms (Name-based, Hierarchy-based, Annotation-based, and Property-based) to be executed over candidate ontologies for the generation of required mappings. In our experiments, executing these matching algorithms over large-scale biomedical ontologies, whole FMA with whole NCI has taken 3 days to produce desirable results. This delay in mapping results makes ontology matching of large-scale biomedical ontologies ineffective for biomedical systems with in-time processing demands. Our proposed system will provide resolution to large-scale biomedical ontology matching performance bottleneck by providing a parallel matching implementation over parallelism-enable platform i.e., usage of multicore cloud instances that are affordable and ubiquitous.

C.4 Implementation Overview

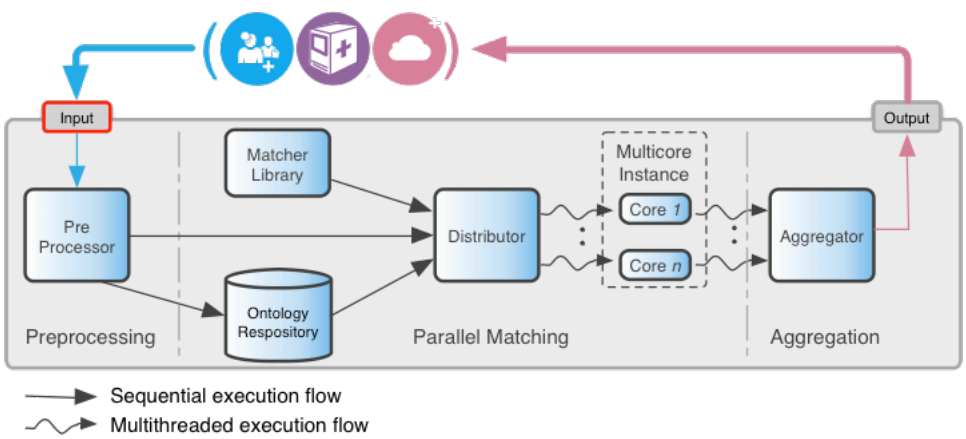


Figure C.1: Proposed Methodology

As illustrated in Fig. C.1, our proposed system has a three-stage execution flow for parallel ontology matching.

Requests for matching large-scale biomedical ontologies can be generated from several sources including, biomedical professionals and researchers, biomedical and bioinformatics sys-

tems, and even third-party healthcare information services running over cloud platforms. Ontology matching request is submitted to our system by either providing the actual candidate ontologies O_{source} and O_{target} or their Uniform Resource Identifier (URI).

Preprocessing

Decomposition of the complex large-scale biomedical ontologies into smaller and simpler (reduced computational complexity) Resource-based ontology subsets depending upon the needs of matching algorithms. This mechanism contributes to our systems performance by only loading the ontology Resources required by matching algorithms, executing in parallel by matching threads. These subsets are also preserved in ontology repository by serialization to reduce the matching effort for future matching requests of same ontologies.

Parallel Matching

To enable data parallelism, distribution of matching process over ontology subsets from preprocessing stage into finer levels of abstractions (independent Matching Jobs and Matching Tasks) depending upon the available cores. Matching Requests are assigned to every cloud node, matching jobs are the division of one matching request over available computing cores, and each core is assigned by thread-level parallelism with a set of equal number of matching tasks to complete the whole matching process. This mechanism contributes to our systems performance by distributing matching tasks over participating computing cores and executing them in parallel at finer level with optimal computing resource utilization.

$$MatchingRequest \leftarrow \sum_{i=1}^{cores} MatchingJob_i \tag{C.1}$$

$$MatchingJob_i \leftarrow \left\{ \bigcup_{i=1}^{MatchableConcepts} MatchingTask_i \right\} \tag{C.2}$$

Aggregation

After parallel matching, matched results are aggregated from all matching threads as mediation bridge ontology O_{bridge}. This mechanism contributes in standardized delivery of mappings as matched results by our system to biomedical clients.

$$O_b^{job} \leftarrow \bigcup_{i=1}^{MatchingTasks} (m \times n \neq \emptyset)_i \tag{C.3}$$

$$O_b \leftarrow \sum_{i=1}^{MatchingJobs} O_b^i \tag{C.4}$$

C.5 Contributions

- Improved overall performance of ontology matching over large-scale biomedical ontologies by implementation data parallelism;
- No performance-accuracy tradeoff, as exploitation of performance from cores instead of matching algorithms;
- Reduced memory strain while execution, as subsets of ontologies with required information are loaded only.

C.6 Utilization of Microsoft Azure Platform

We will be utilizing Microsoft Azures multicore cloud instance for our systems implementation. We require a Large (A3) Compute Instance with 4 Virtual Cores and 7 Gb RAM with 500Gb of space for applications, large-scale biomedical ontologies with multiple versions, and ontology repository. Evaluating the performance speedup from 1 to 4 cores for parallel matching will benchmark our system.