Eagle Framework: An Automatic Parallelism Tuning Architecture for Semantic Reasoners

Haifa Ali Al-Hebshi[1], Muhammad Ahtisham Aslam[2], Kawther Saeedi[3]
Information Systems Departmen-Faculty of Computing and Information Technology, King Abdulaziz University
Jeddah, 21589, Saudi Arabia[1,3]
Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, Berlin, 10589, Germany[2]

***Abstract*—Parallel semantic reasoners use parallel architectures to improve the efficiency of reasoning tasks. Studies in semantic reasoning rely on manual tuning to configure the degree of parallelism. However, manual tuning becomes increasingly challenging as ontologies become massive and complex. Studies in related fields have developed automatic tuning frameworks using optimization search methods. Although these methods offer performance gains, reducing search time and space size is still an open problem. This study aims to bridge the gap in semantic reasoning and the problem in existing search methods. To achieve these aims, we propose Eagle Framework (EF), an innovative automatic tuning framework designed to improve the performance of parallel semantic reasoners. EF automatically configures the degree of parallelism and calculates the performance data. It incorporates a novel search space and algorithm, inspired by the AVL tree, that efficiently identifies the optimal degree of parallelism. In a case study, EF completed the tuning processes in seconds to a few minutes, achieving performance gains up to 65 times faster than common search methods. The reliability findings, with ICC scores ranging from 0.90 to 0.99, confirmed the consistency of the performance data calculated by EF. The regression analysis revealed the effectiveness of EF in identifying the factors that affect reasoning scalability, with the conclusion that the size of the ontology is the dominant factor. The study underscores the need for adaptive approaches to tune the degree of parallelism based on the size of the ontology.**

***Keywords*—Automatic tuning; parallel semantic reasoning; performance optimization; ontology; high-performance computing**

I. Introduction

In today's digital landscape, machines use ontologies, structured knowledge representations, to process and infer information across domains, forming a larger knowledge base known as Linked Open Data (LOD) [1]. Ontologies are essential for applications such as the semantic web, artificial intelligence, and data-driven decision-making [2], enabling machines to derive insights from complex relationships [3]. However, as ontologies grow in size and complexity, especially with the rise of the Internet of Things (IoT) and social networks, reasoning over these vast knowledge graphs becomes challenging [4]. Traditional semantic reasoners, which use sequential processing, struggle to scale with the growing size of ontologies, leading to significant delays in driving inferences [4], [5].

Fortunately, parallel reasoning systems have emerged, leveraging multicore and distributed computing technologies to improve efficiency [6]. These systems divide reasoning tasks into smaller units, allowing concurrent processing between multiple computing cores or nodes, thus improving speed and performance [7]. Despite these improvements, the rapid growth of data from IoT systems and social networks continues to add complexity to the reasoning process, demanding more scalable and efficient solutions [8]. Therefore, optimizing parallel reasoning systems to successfully manage performance while addressing the increasing complexity of ontologies is a major challenge for researchers.

Numerous automatic tuning approaches have shown significant improvements in different application areas, such as hyperparameter tuning to optimize machine learning models [9], [10], big data analytics systems [11], [12], and parallel programs [13], [14]. These approaches varied in their strategies and techniques. Optimization search methods have shown accurate results in finding optimal solutions compared to other approaches. However, studies have reported a significant challenge in reducing search time as search space increases.

Linking these challenges in the areas of semantic reasoning and optimization, we present the Eagle Framework (EF), an advanced modular and extensible tool to optimize the performance of parallel reasoning systems. EF automatically generates parallelism configurations, recording performance data, and identifying the optimal degree of parallelism. Operating as a black-box solution on top of existing parallel semantic engines, EF relieves researchers and developers of the time-consuming process of manual tuning. A key innovation of EF is its novel search algorithm, which integrates an AVL tree with a priority queue, enabling efficient exploration for the optimal thread configuration. EF is implemented in Java, ensuring compatibility with a wide range of operating systems and server environments. In addition, EF saves performance data in CSV format and high-resolution line charts for data visualization. In general, EF significantly streamlines the optimization process, ensuring optimal performance and offering valuable time efficiency for researchers to develop scalable and advanced parallel reasoning systems.

In addition to proposing a tuning framework, this study includes a comprehensive case study, in which we validate the performance and reliability of our framework, as well as its effectiveness in optimizing parallel reasoners. We utilized multiple statistical methods through the assessment process, but a key method is introducing the Interclass Correlation Coefficient (ICC) for the reliability analysis. We will explain how we applied the Interclass Correlation Coefficient (ICC) to evaluate the reliability of tuning systems. To the best of our knowledge, ICC has not been used in evaluating automatic tuning.

The remainder of this study is organized as follows. Section II provides a comprehensive review, highlighting the gaps in the existing literature. Section III presents the conceptual and technical design of the EF architecture. Section IV details the case study that evaluates EF from the perspectives of performance, reliability, and effectiveness. Section V presents the findings of the study. Section VI discusses the findings and insights derived from this study. Finally, Section VII concludes the study.

II. Background and Related Work

In this section, we provide a brief review of existing work to highlight the gaps in the literature on semantic reasoning and optimization. This review begins with the challenges of manual tuning presented in the context of semantic reasoning. Then, we provide an overview of the automatic tuning approaches commonly applied in other domains. We review related works with existing optimization methods. For each section of this review, we grouped studies based on the tuning approach or optimization method.

A. Challenges in Tuning Parallel Semantic Reasoners

Semantic reasoning is critical in applications that require logical consistency and explainability, such as medical, bioinformatics, and law. However, the development of parallel semantic reasoning systems has been less active in the last ten years, and most of these systems have been abandoned and no longer maintained [15]. Research studies in semantic reasoning implemented manual tuning to adjust the degree of parallelism. Examples of these studies are [6], [16], [17]. Although these studies did not explicitly state the drawbacks of manual tuning approaches in their research results, recent studies highlighted the challenges of computational complexity and performance in automated reasoning [18] [19]. Reasoning tasks use algorithms that require extensive computations and resource allocation [20]. These scalability challenges are further complicated by the increase in ontology sizes and hierarchies that require intensive computation. Although parallel semantic reasoners leverage multicore and distributed architectures to tackle scalability issues, without automatic tuning strategies, reasoning scalability remains challenging. The Table I provides a comparison of various approaches based on different factors and parameters.

B. Overview of Automatic Tuning Approaches

This section reviews recent studies on optimizing parallel systems. It classifies them into six approaches, following the categorization by Herodotou et al. [21], and discusses the advantages, limitations, and methods for each approach.

1) Search-based approach: The search-based approach systematically searches for the optimal solution through experiments, improving performance and resource efficiency. Although this approach is reliable for identifying the optimal or near-optimal solution, it can be computationally expensive for large systems. Van Werkhoven introduced an automatic framework that integrated various optimization search, including simulated annealing and particle swarm optimization, to optimize GPU kernels in OpenCL and CUDA [22].

2) Rule-based approach: The rule-based approach uses predefined guidelines and domain expertise to guide tuning decisions, offering simplicity and fast execution. It works well in predictable environments where the behavior of the system follows established patterns. However, its lack of flexibility limits its effectiveness in complex or dynamic workloads. Schwarzrock et al. applied this approach to enhance performance and energy efficiency in NUMA systems. Their focus was on optimizing thread-to-core mapping, memory page mapping, and thread throttling [23].

3) Machine learning approach: The machine learning approach employs models, such as regression and neural networks, to predict optimal configurations by learning from historical data, capturing complex relationships for improved tuning accuracy. When trained on quality data, these models can achieve near-optimal configurations without exhaustive searches. However, they require substantial data and computational resources, and accuracy is dependent on data quality. Fan et al. used a random forest model to optimize query performance in databases by predicting optimal degrees of parallelism [24].

4) Adaptive approach: The adaptive approach dynamically tunes parameters in real-time, adjusting to workload changes, making it ideal for dynamic environments where static tuning fails. It can achieve near-optimal configurations quickly without exhaustive searches, though it may introduce overhead and struggle with stability in highly volatile conditions. Vogel et al. proposed reactive self-adaptive strategies to control parallelism in stream processing systems, allowing real-time adjustments without the need to restart applications [25].

5) Cost modeling approach: Cost modeling estimates resource costs, such as memory and CPU usage, for different tuning configurations, helping to avoid costly trial runs. It provides fast and moderately accurate estimations; however, its limitations lie in the accuracy of the model as it may not capture all the dynamics and interactions in the real world. This limits its ability to find optimal configurations in complex environments. Siddiqui et al. introduced a machine learning-enhanced framework to improve the accuracy of cost modeling in big data systems [26].

6) Simulation-based approach: The simulation-based approach models the behavior of the system in a simulated environment to predict optimal settings without affecting live performance. This is especially useful for difficult-to-test scenarios, offering reliable approximations if the simulation models are accurate. However, accuracy depends on model fidelity and detailed simulations can be resource-intensive. Liu et al. developed HSim, a Hadoop simulator for modeling various performance parameters in cloud computing [27] .

C. Existing Optimization Search Methods

Among the six approaches previously discussed, we were particularly motivated by the reliability of search-based methods in finding optimal solutions. This section focuses on studies that implement these search-based methods. Currently, search methods are applied in system optimization in related domains, big data, machine learning, and high performance computing. These studies provide valuable insights that inform the design of our solution for optimizing parallel reasoning performance.

TABLE I. SUMMARY OF COMMON PARALLELISM TUNING APPROACHES

Approach	Search-based	Rule-based	Machine Learning	Adaptive	Cost Modeling	Simulation-based
Methods	Search algorithms.	Based on heuristics.	Data-driven predictions	Real-time dynamic adjustments.	Cost estimation models.	System behavior simulation.
Advantages	High-quality solutions.	Simple, fast decisions.	Adapts to changing conditions	Real-time tuning.	Guides resource decisions.	Tests without real-world impact.
Drawbacks	Expensive computing costs.	Expert knowledge required.	Large data needed, Slow training.	Struggles with unexpected changes.	May overestimate real-world factors.	Expensive computing costs.
Domain	Large, complex systems.	Simple, predictable systems.	Learning from history.	Workloads that change.	Resource-constrained systems.	Expensive or risky scenarios.
Data Size	Moderate to large.	Small to moderate.	Large.	Small to medium.	Small to medium.	Large.

1) Grid Search (GS): GS is an optimization method that systematically explores all possible parameter combinations to determine the optimal one. However, this exhaustive approach is computationally expensive, particularly for high-dimensional search spaces [28].

Recent studies have demonstrated the potential of GS in optimizing systems in various domains. For example, George and Sumathi applied GS to optimize a random forest classifier for sentiment analysis, leading to improved accuracy [29]. Similarly, Priyadarshini and Cotton used GS to tune a deep neural network model for sentiment analysis, achieving an accuracy above 96%, which outperformed several baseline models [30]. In the big data domain, Chen et al. incorporated GS into their system to optimize MapReduce performance on Hadoop clusters, identifying optimal configurations to minimize running times [31]. In a similar study, Sewal and Singh compared GS with other optimization methods such as Evolutionary Optimization and Random Search to fine-tune Apache Spark. They found that GS was effective in reducing execution times by 23.24%. [32]. These studies are examples among others that utilize determinism in GS to enhance performance in areas like machine learning and big data systems.

2) Hill Climbing (HC): HC is a simple and efficient local search algorithm that iteratively improves an initial solution by exploring neighboring options. However, it may get stuck in local optima, reducing the chances of finding the global.

Recent studies have highlighted the effectiveness of HC in various fields. For example, Sivakumar and Mangalam introduced a technique to improve adaptive cruise control systems in automated vehicles, using a combination of search methods, including HC, to optimize vehicle parameters, improving safety and fuel efficiency [33]. Zeng et al. employed a simple HC method with machine learning techniques to optimize parallelism in Parallel Nesting Transactional Memory (PN-TM) systems, achieving faster convergence and higher accuracy compared to other optimization methods [34]. Pradhan et al. applied HC to optimize a CNN model for classifying COVID-19 from chest X-ray images, improving its performance metrics and outperforming other hybrid techniques [35]. These studies are among several that demonstrate HC as a versatile and effective optimization method to improve system performance in diverse domains, from automated vehicles to machine learning and medical image classification.

3) Simulated Annealing (SA): SA is a probabilistic optimization algorithm that explores various solutions, including the worst ones to escape local optima. SA is widely used method for many optimization problems due to its adaptability and capability to navigate rugged search spaces [36]. However, it can be computationally intensive, sensitive to parameter choices, and slow to converge.

Recent studies demonstrate the adaptability and effectiveness of SA in finding optimal solutions for complex optimization problems. A study by Rasch et al. utilized SA within their Auto-Tuning Framework (ATF) to optimize interdependent parameters in parallel programs, using the chain of trees and coordinate search spaces to enhance multidimensional exploration and improve tuning efficiency [7]. Gülcü and Kuş introduced the multi-objective simulated annealing algorithm for optimizing hyperparameters in convolutional neural networks, balancing classification accuracy and computational complexity, and achieving superior results compared to traditional SA [37]. Similarly, Abdel-Basset et al. combined SA with Harris Hawks Optimization to enhance feature selection for classification tasks, using SA to escape local optima and explore better feature subsets effectively [38]. These studies highlight the effectiveness of SA, both as a standalone algorithm and within hybrid frameworks, in solving diverse optimization problems across fields.

4) Random Search (RS): Random Search (RS) is a simple algorithm that explores a search space by randomly sampling points and updating the best solution found. Although its simplicity and ability for global exploration, it lacks efficiency in high-dimensional spaces.

Recent studies underscore the role of RS as a simple and effective optimization method in different applications. A study by Willemsen et al. proposed a standardized benchmarking methodology for evaluating automatic tuning frameworks. This methodology integrates RS as a baseline for benchmarking optimization algorithms [39]. A similar study by Deligkaris used RS as a baseline method to benchmark evobpso, an algorithm based on particle swarm optimization, against 12 related methods and models. The benchmark results demonstrated the effectiveness of RS in the search for neural architectures [40]. Hosseini et al. employed RS in optimizing 10 hyperparameters of Long Short-Term Memory (LSTM) networks used in rainfall-runoff modeling, resulting in highly precise predictions for hourly stream-flow and water levels in Spain's Basque Country [41]. Despite the advancement of more sophisticated

methods, these studies highlight the effectiveness of RS as a baseline or complementary method in advanced optimization methods.

D. Limitations and Research Gaps

Section II-A presented studies on parallel reasoning, which currently rely on manual tuning for optimization and scalability analysis. As research labs, governmental sectors, and universities continue to develop ontologies, the need for scalable and efficient reasoners has increased. Furthermore, the substantial efficiency and accuracy of automatic tuning reported in studies applied in other domains, such as machine learning and big data, highlight that the lack of automatic tuning application in semantic reasoning is a considerable gap. To the best of our knowledge, the application of automatic tuning in semantic reasoning has not been explored. Therefore, addressing this gap is the main objective of this study.

The studies reviewed in Section II-C noted the advances gained from applying search optimization methods. However, they also noted significant limitations of these optimization methods (see Table II). For GS, though it is exhaustive and guaranteed to find the optimal solution, it is computationally expensive, especially in multidimensional spaces where the time increases exponentially with increase in the search space. On the other hand, HC, SA, and RS offer greater efficiency and scalability; however, they are limited by several drawbacks, such as risks of local optima, stochastic behavior, and incomplete search space exploration. These limitations underscore the trade-off between time complexity and the guarantee of identifying the optimal solution, and this trade-off is strongly correlated with search space size. As noted by Krestinskaya et al., optimizing search time for large search spaces remains a critical gap and an open research challenge in optimization [42]. Therefore, the second objective of this study is to address the complexity of search time by designing a deterministic algorithm and a tree-based search space that aims to guarantee finding the optimal solution in efficient time.

TABLE II. SUMMARY OF SEARCH OPTIMIZATION METHODS

Aspect	Grid Search	Hill Climbing	Simulated Annealing	Random Search
Exploration	Global	Local	Global (with refinement)	Global (with random sampling)
Efficiency	Very Low	High	Medium	Low
Dimensional Scalability	Poor	Poor	Good	Good
Risk of Local Optima	No	High	Low	No
Best Use Case	Small spaces	Small spaces	Large spaces	Large spaces
Time Complexity	$O(n^k)$	$O(k \cdot n)$	$O(k \cdot n)$	$O(k \cdot n)$

III. EAGLE FRAMEWORK (EF)

This section presents the architectural and algorithmic design of EF. Before diving into architectural design, we provide an overview of the EF multi-layered system, where the EF resides in the middle layer. This overview is essential for understanding the interoperability and portability of EF, which allows it to function with various parallel reasoning systems on different platforms. Following the system overview, we will explore the architectural details of EF from both mathematical and algorithmic perspectives.

A. System Overview

EF is a modular architecture that operates within a multilayered system, where EF occupies one layer alongside a parallel reasoner. The abstraction view of the EF system consists of five layers, as depicted in Fig. 1.

The first layer is the Command Line Interface (CLI), which accepts parameter values to set automation parameters and is responsible for displaying output results. The second layer represents the main contribution of this study, where EF and the parallel semantic reasoner are positioned. EF comprises one main algorithm and three auxiliary algorithms: *Thread Controller*, *Speedup Calculator*, and *Parallelism Optimizer*. The third layer is the java runtime environment, which serves as a bridge between the EF implementation and the operating system. This middle layer provides the resources needed to compile and execute EF classes on any machine. The fourth layer is the operating system, which abstracts physical hardware and manages system resources. The final layer is the hardware, which represents the physical components of the machine, such as memory and the CPU. The next section focuses on the second layer, detailing the EF architectural components and their interactions with the other layers.

B. EF Architecture

EF architecture consists of a main algorithm and auxiliary algorithms. The main algorithm represents the core architecture of the EF and the interface that manages the connection between the CLI layer, the parallel reasoner, and the auxiliary algorithms. The auxiliary algorithms help the main algorithm in the tuning process by adjusting the thread count, calculating the speedup factor, and identifying the optimal thread count. Fig. 2 presents a flow chart for the EF architecture. The following sections provide an in-depth explanation of the functionality of each algorithm within the EF architecture.

1) The Main algorithm: As shown in Fig. 2, the main algorithm comprises five phases: automation setup, sequential reasoning, parallelism tuning, optimization, and output formatting.

a) Phase 1: Automation setup: The main algorithm starts by accepting three parameter values from the CLI layer: the path to the ontology file (p), the scale difference (d), which defines the incremental scale used to calculate thread configurations, and the maximum thread count (m), which serves as a threshold to limit the generation of additional configurations. Based on these parameters, the number of thread configurations is proportional to the values of d and m. Additionally, it creates a tree (A) to use in the search for the optimal thread count (o), and a list (L) to store performance data. It also sets the thread count (n) and speedup factor (s) to the seed value of 1.

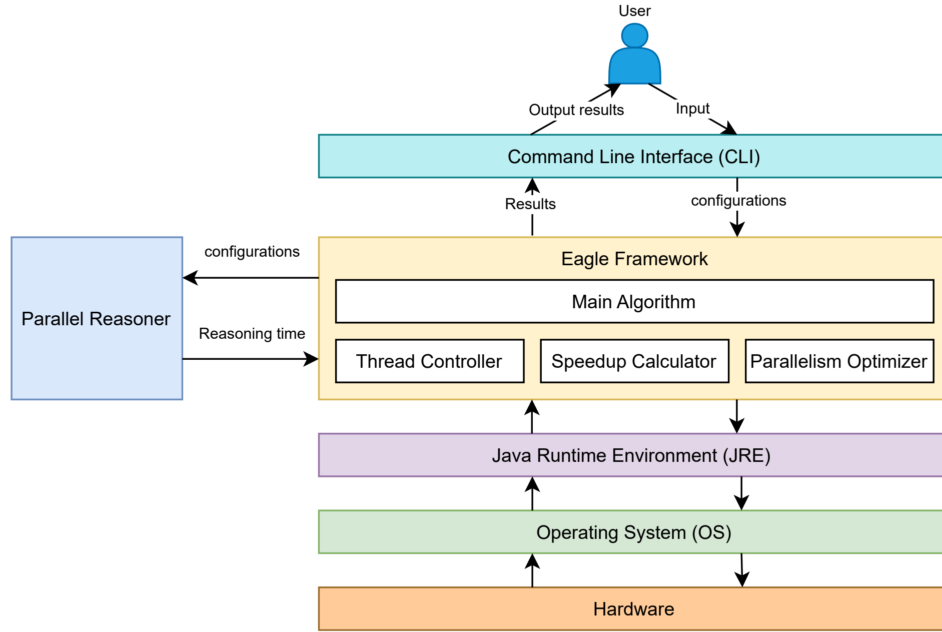

Fig. 1. EF System overview.

b) Phase 2: Sequential reasoning: In this phase, the main algorithm runs the parallel reasoner, `runReasoner(`p`, `n`)`, sequentially using one thread. Then, the main algorithm saves sequential reasoning time in a variable named T_{seq}. Since EF does not perform parallel reasoning at this phase, the speedup factor remains 1. Consequently, the main algorithm inserts the values of n and s in A and adds them with T_{seq} to L.

c) Phase 3: Parallelism tuning: This phase represents the core automation provided by EF. The main algorithm starts the automation by checking whether n does not exceed m. This step is essential to limit the EF from generating more configurations. If n is less than or equal to m, the main algorithm passes n and d in a call to the Thread Controller `ctrlThreads(`n`, `d`)`. After receiving the new n value from the Thread Controller, the main algorithm passes n with p in a call to the reasoner and stores the parallel reasoning time in a variable named T_{par}. Subsequently, the main algorithm sends T_{seq} and T_{par} in a call to the Speedup Calculator, `calcSpeedup(`T_{seq}`, `T_{par}`)`, to find the ratio and save it in s. Finally, the main algorithm performs the necessary operations to store performance data in A and L.

d) Phase 4: Optimization: In this phase, the main algorithm passes A in a call to the Parallelism Optimizer, `optParallelism(`A`)`, which searches for o in A. Once o is identified, the main algorithm prints o on the console. The parallelism tree and optimizer will be explained comprehensively in Section III-B4.

e) Phase 5: Output formatting: This phase is the final stage in EF, where performance data L are formatted into a comma separated value file (CSV). This format was chosen for its compatibility with most statistical analysis and database systems, which allows direct processing by analysis tools. In addition, a line chart is generated to illustrate the relationship between each thread count and the associated reasoning time.

2) Thread controller: This section explains the algorithm of Thread Controller, denoted as `ctrlThreads(`n`, `d`)`, designed to create thread configurations. Building on recommendation by Huang et al. [43], who recommended employing sampling strategies for generating configurations, this algorithm generates thread configurations systemically using a specified scale difference, d. To prevent incorrect input, we added a conditional statement that verifies the value of d entered by the user. If d is assigned a negative number or zero, the Thread Controller sets d to the seed value of 1 and recalculates n accordingly. Otherwise, the Thread Controller computes the new n based on the provided value of d.

Definition 1. *Let n be a thread configuration and d be the scale difference. The Thread Controller calculates is defined as:*

$$ctrlThreads(n, d) = \begin{cases} n+1 & \text{if } d \leq 0 \\ n+d & \text{if } d > 0 \end{cases} \quad (1)$$

3) Speedup calculator: Speedup is a common metric that is used to assess performance improvements in systems running on parallel computing architectures. To measure speedup, we designed Speedup Calculator, denoted as `calcSpeedup(`T_{seq}`, `T_{par}`)`, an algorithm that calculates the speedup factor as the ratio between the execution time for sequential reasoning and the time taken for parallel reasoning. Since most reasoners record reasoning times in milliseconds, we took account of scenarios where executing the reasoner on massively parallel computing resources results in reasoning times in fraction of a millisecond rounded to zero (i.e. in nanoseconds). To handle this, the Speedup Calculator first checks if the value of T_{par} is non-zero. If this condition is

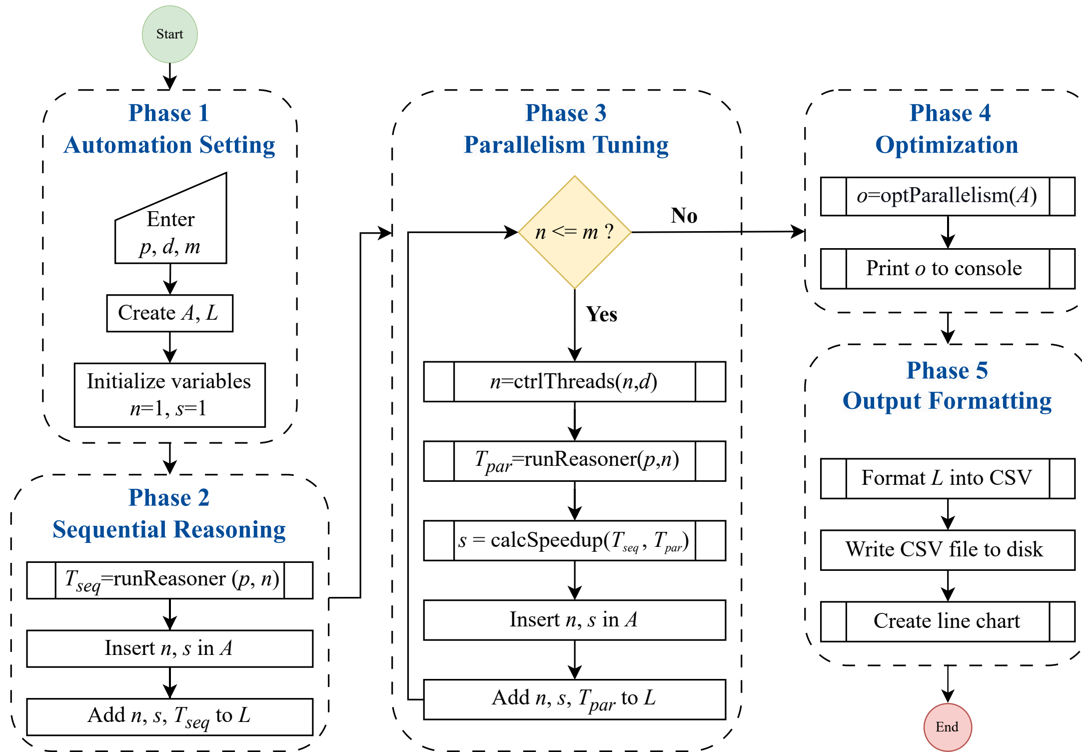

Fig. 2. EF Architecture.

met, the calculator proceeds with the division. However, if T_{par} equals zero, the calculator throws an arithmetic exception, which is treated as "undefined" in arithmetic, and the process safely halts.

Definition 2. *Let T_{seq} be the execution time of the sequential reasoning, and T_{par} be the execution time of the parallel reasoning. The Speedup Calculator is defined as:*

$$calcSpeedup(T_{seq}, T_{par}) = \begin{cases} Undefined & if\ T_{par} = 0 \\ \frac{T_{seq}}{T_{par}} & if\ T_{par} \neq 0 \end{cases} \tag{2}$$

4) Parallelism Tree (PT) and Parallelism Optimizer (PO): An AVL tree is a self-balanced binary search tree characterized by its speed in most operations, including insertion and searching [44] [45]. We chose the AVL tree to construct the search space in EF due to its time efficiency, with a worst-case time complexity of $O(\log n)$ for core operations, which outperforms other data structures to align with the objective of this study. For clarity, we used Parallelism Tree (PT) to refer to the tree-based search structure and Parallelism Optimizer (PO) to refer to the associated optimization algorithm.

PT is a modified version of the AVL tree, where each node a consists of performance data (s) and its associated thread configuration (n). In addition, each node has pointers to a left child (l) and a right child (r). Unlike the AVL tree, PT uses s as the key to determine the correct position to insert a new node. In PT, the sequential reasoning node is always the root, while the parallel reasoning nodes are placed based on their speedup factor. Algorithm 1 presents the insert procedure for PT.

Algorithm 1: insert(s, n)

```
1:  Input: s, n
2:  Output: rebalance(root)
3:  if root = null then
4:      return  new Node(s, n)
5:  else
6:      if root.s > s then
7:          root.l ← insert(s, n)
8:      else if root.s < s then
9:          root.r ← insert(s, n)
10:     else
11:         root.nQueue.add(n)
12:     end if
13: end if
14: return  rebalance(root)
```

In the initial experiments, we observed that different thread configurations produced identical speedup factors. Consequently, PT prevents the insertion of nodes with duplicate speedup. To resolve this issue, we integrated a priority queue within the PT node structure to store all thread configurations associated with the same speedup factor. This approach enables PT to encapsulate each speedup factor with its corresponding thread configurations in the same node. The priority queue orders the configurations from the smallest to the largest, enabling efficient exploration by neglecting the less effective configurations. In this study, we designed PO to select the

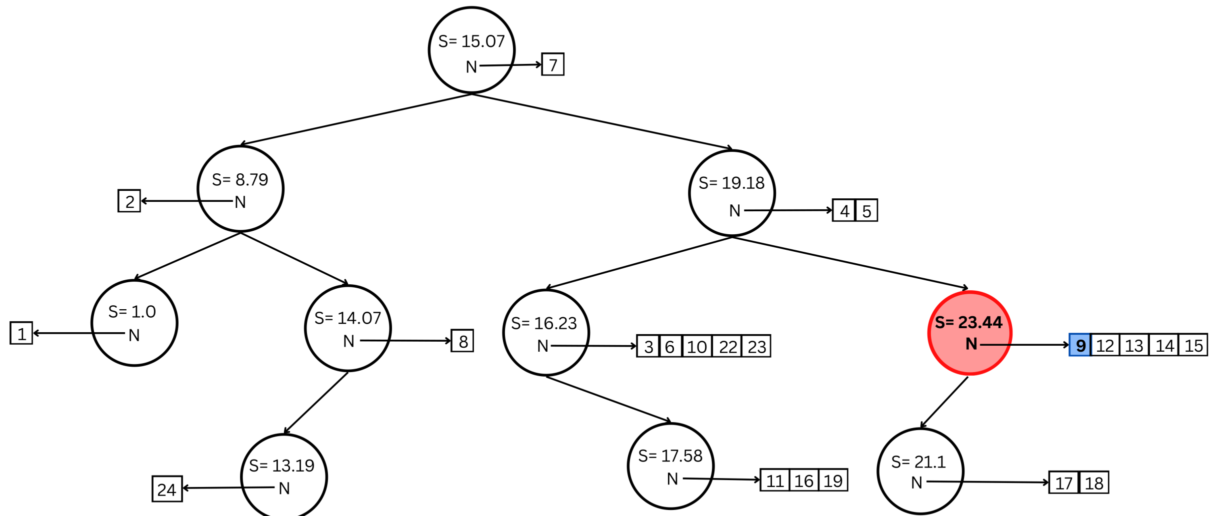

Fig. 3. Parallelism tree (PT) in EF where each node contains speedup factor (s) and the associated thread configuration queue (n).

smallest thread configuration as the optimal one, under the assumption that this configuration achieves the highest speedup factor and that performance will not improve beyond this point. Fig. 3 illustrates an example of PT resulting from one tuning experiment. The red circle denotes the node with the maximum speedup value, while the blue square indicates the optimal degree of parallelism PO selects from the thread queue.

Similarly to the AVL tree, PT is ordered and places the node with the highest speedup factor at the end of its rightmost path. Therefore, we designed the PO algorithm, denoted as `optParallelism(`A`)`, to search for the optimal configuration only in the rightmost path of the PT. Such an approach efficiently saves time compared to search in a multi-dimensional space structure. PO starts the optimization search by checking whether the root's right pointer points to NULL. If so, it returns the root node because it contains the optimal degree of parallelism. If not, it performs this check recursively until it finds the node whose right pointer refers to NULL as the node containing the optimal thread configuration. The PO algorithm is shown in Algorithm 2.

Algorithm 2: optParallelism($node$)

1: **Input:** $node$
2: **Output:** $node$
3: **if** $node.right == null$ **then**
4: **return** $node$
5: **else**
6: **return** optParallelism($node.right$)
7: **end if**

IV. Experimental Design and Setup

This study aims primarily to evaluate EF in terms of performance, reliability, and effectiveness in assessing the scalability of parallel reasoners. To achieve this, we conducted our experiments on a case study on the ELK reasoner, a reasoning engine specifically designed for OWL2 EL ontologies [16]. ELK is one of the few actively maintained parallel reasoners for OWL2, as many other reasoners have been discontinued [15]. Although ELK provides a variety of reasoning services to support ontology development and querying, our experiment scope is only on the classification reasoning service.

We conducted the experiments on the Aziz Supercomputer, where each node consists of two 12-core processors (Intel Xeon CPU E5-2695v2, 2.40 GHz) that support Hyper-Threading Technology, providing a total of 48 logical cores and a total memory of 256 GB (128 GB per processor). Since ELK was designed to operate exclusively on shared memory servers [16] [46], we performed all experiments on a single node. To maintain the integrity of our results, we secured exclusive access to server resources, thereby preventing the interleaving of jobs which could potentially compromise reasoning time and speedup factor. Furthermore, we configured the EF parameters with a scale difference of 1 and set the maximum thread count to 240 in all experiments. Section V-D1 will explain our choice of these parameter values.

Our research used a systematic sampling technique to select the ontologies. First, we downloaded ontologies from BioPortal [1] and OBO Foundry[2] that support the OWL2 EL profile. Then, we further classified the ontologies based on the number of TBox axioms into different sizes, ranging from tiny to medium. We based our categorization on the number of TBox axioms since the classification reasoning service is associated with only TBox axioms. We selected three ontologies from each size category, resulting in a total of nine ontologies. Each ontology was examined in a separate experiment, and each experiment was repeated three times to ensure reliability, resulting in a cumulative total of 27 experiments. This approach allowed us to ensure a diverse range of ontologies for our experiments while ensuring that the samples represented the entire population of OWL2 EL ontologies.

[1] https://bioportal.bioontology.org/
[2] https://obofoundry.org/

V. Case Study: Validating The Performance, Reliability, and Effectiveness of EF in Tuning the ELK Reasoner

ELK is a specialized OWL reasoner that classifies ontologies in the OWL2 EL profile. It is known for its high performance due to its parallel reasoning and robust optimization techniques [16]. ELK has expanded its capabilities to include incremental classification and proof tracing, with optimizations for handling role composition axioms and rewriting low-level inferences. These improvements simplify incremental reasoning, proof generation, and enable automated verification and ontology debugging.

This case study begins with an assessment of EF from performance and readability perspectives and ends with an evaluation of the effectiveness of EF in analyzing the scalability of the reasoning system.

A. Overall Framework Performance

The analysis starts by evaluating the performance of the EF for the overall tuning process, covering ontology loading, configuration generation, performance monitoring, and reasoning optimization. Table III presents the minimum, maximum, average and standard deviation of execution times (in seconds) required for EF to tune and optimize the ELK reasoner. Execution times range from a few seconds to just under five minutes.

For tiny ontologies, such as OLATDV, INO, and FBDV, the tuning process was completed in a few seconds, demonstrating the framework's efficiency in quickly exploring thread configurations. Small-sized ontologies, including PLANA and PDON, exhibited slightly longer tuning times, ranging from 17 to 28 seconds. In contrast, medium-sized ontologies, such as OBA, EMAPA, and ORDO, required significantly more time, with ORDO taking the longest at 4 minutes and 29 seconds. This variation in execution times reflects the influence of ontology size on EF performance, with larger ontologies requiring longer durations.

Narrowing the focus from overall framework performance, the next section presents a detailed comparative analysis of the EF's optimization algorithm against baseline search algorithms commonly used in optimization studies.

B. Parallelism Optimizer vs. Existing Optimization Methods

To evaluate EF's optimization performance, we compared its Parallelism Optimizer (PO) with existing search methods commonly used in optimization studies. Specifically, we compared PO with grid search (GS), hill climbing (HC), simulated annealing (SA) and random search (RS). To conduct a fair comparison, we separated PO from the EF architecture. We used one data set resulted from one tuning experiment for all the algorithms involved in the comparison. For SA, we set the initial temperature at 1000 and the cooling rate to 0.95, while for RS, we set the number of iterations to 100.

A summary of the comparative analysis is shown in Table IV. The results showed that PO significantly outperforms its competitors, achieving an average reasoning time of just 0.003 ms. In contrast, the average reasoning times for the other algorithms were 0.167 ms for GS, 0.090 ms for HC, 0.196 ms for SA, and 0.128 ms for RS. This high efficiency is further demonstrated by the success rate in identifying the optimal thread configuration. PO perfectly found the optimal thread configuration in all ten attempts, while both HC and SA failed in all attempts, and RS succeeded in only three trials.

C. Data Quality and Reliability

In data assessment, we focus on evaluating the quality of the EF measurements gathered during the tuning process and the consistency between these measurements. Listing 1 represents a sample console output of the type of variable data input, missing values, and duplicate rows in the data collected by the EF. As shown in Listing 1, the data underwent evaluation included the test ID for referencing purposes, thread count, reasoning time in milliseconds and in a format of days, hours, minutes and seconds to ease readability, and corresponding speedup factor. The assessment showed that each variable was correctly formatted in a suitable data type. In addition, it revealed that neither missing values nor duplicate rows were found in the data, reporting data integrity and quality.

```
=========================================================
File: ELK_ORDO_1_240_2.csv
=========================================================
| Variable                        | Data Types | Missing |
|---------------------------------|------------|---------|
| Test Number                     | object     | 0       |
| Number of Threads               | int64      | 0       |
| Total Reasoning Time (ms)       | int64      | 0       |
| Total Reasoning Time (d:h:m:s)  | object     | 0       |
| Speedup Factor                  | float64    | 0       |
|                                 |            |         |
| Duplicate Rows:                 |            | 0       |
```

Listing 1. Sample Console Output For EF's Data Quality Assessment.

To assess the reliability of the EF, we used the intraclass correlation coefficient (ICC). We selected ICC over other statistical methods because our study involves repeated exterminates conducted in the same environmental settings. ICC is a highly precise statistical method that is sensitive to variance [47]. In addition, ICC can assess both the consistency within and between configurations. It is widely used in other fields such as medicine, psychology, biology, and genetics, especially to evaluate the reliability of measurement tools such as medical instruments and computer-aided detection (CAD) systems [48]. To our knowledge, this is the first study to use ICC in assessing the reliability of automatic tuning methods.

Because the speedup factor is a ratio, we applied the ICC assessment exclusively to the reasoning time. The results of the reliability analysis for different ontology sizes are presented in Table V. Each assessment applied the ICC(3,k) model to a dataset of 720 measurements, calculated as each experiment repeated three times with 240 measurements per trial. As shown in Table V, the analysis of nine ontologies revealed high ICC scores, ranging from 0.789 to 0.992, indicating strong consistency between measurements. All ontologies showed statistically significant results ($p < 0.001$) with narrow confidence intervals, indicating precise measurements. The highest ICC was observed for INO (0.992), followed by OLATDV (0.990) and PLANA (0.962). ORDO and PDON also showed strong ICC scores of 0.951 and 0.932, respectively. However,

TABLE III. REASONING TIME CALCULATED BY EF IN TUNING ELK REASONER (IN SECONDS)

Ontology	TBox Axiom count	Ontology Size[a]	Min. Time	Max. Time	Avg. Time	Median Time	SD Time
OLATDV	88	Tiny	9.24	12.25	10.377	9.637	1.636
INO	384	Tiny	11.36	11.63	11.491	11.489	0.134
FBDV	646	Tiny	10.17	11.71	11.097	11.413	0.815
PDON	1252	Small	15.86	34.64	28.311	34.433	10.786
PLANA	2755	Small	15.89	18.21	17.071	17.113	1.157
WBPHENOTYPE	4026	Small	36.86	43.34	39.703	38.910	3.316
OBA	17811	Medium	137.06	295.03	194.653	151.877	87.241
EMAPA	23029	Medium	74.49	78.92	77.309	78.518	2.448
ORDO	53861	Medium	233.28	287.40	269.103	286.634	31.029

[a] Ontology sizes categorized by TBox axiom count: Tiny – fewer than 1,000 axioms; Small – 1,000 to 10,000 axioms; Medium – 10,000 to 100,000 axioms; Large – 100,000 axioms or more.

TABLE IV. BENCHMARKING THE PARALLELISM OPTIMIZER AGAINST EXISTING OPTIMIZATION SEARCH ALGORITHMS

Algorithm	Grid Search	Hill Climbing	Simulated Annealing	Random Search	Parallelism Optimizer
Average Reasoning Time (ms)	0.167	0.090	0.196	0.128	0.003
Success Rate for Optimal Thread Identifications (out of 10)	10/10	0/10	0/10	3/10	10/10
Time Complexity	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

TABLE V. RELIABILITY ANALYSIS FOR EF MEASUREMENTS AMONG DIFFERENT SIZES OF ONTOLOGIES

Ontology	ICC	F	P	CI95%
OLATDV	0.989824	98.267647	8.143461e-311	[0.99, 0.99]
INO	0.992448	132.417193	0.0	[0.99, 0.99]
FBDV	0.880456	8.365113	3.314461e-85	[0.85, 0.90]
PDON	0.932163	14.741265	1.651629e-130	[0.92, 0.95]
PLANA	0.962178	26.439423	9.941877e-183	[0.95, 0.97]
WBPHENOTYPE	0.789731	4.755818	9.179020e-48	[0.74, 0.83]
OBA	0.883609	8.591758	3.438310e-87	[0.86, 0.91]
EMAPA	0.889079	9.015433	8.344745e-91	[0.86, 0.91]
ORDO	0.951172	20.48018	2.327254e-159	[0.94, 0.96]

WBPHENOTYPE, with an ICC score of 0.790 and a 95% confidence interval of [0.74, 0.83], showed moderate consistency. Although the ICC score for WBPHENOTYPE was statistically significant, its lower ICC and broader confidence interval indicated weaker consistency compared to the other ontologies.

D. EF Effectiveness in Analyzing Reasoning Performance

This section explores the role of EF in helping researchers with scalability assessments to improve the performance of parallel semantic reasoners. To validate EF effectiveness, we performed exploratory and regression analysis.

Before conducting the evaluation, we combined all EF data resulted from all experiments. Additionally, we added characteristics information for each ontology, including the TBox axiom count and size. we performed the necessary pre-processing and normalization .

1) Exploratory analysis: This section explores the impact of varying thread configurations and ontology sizes on the performance of the ELK reasoner. It also examines the relationship between the optimal degree of parallelism and the total number of logical cores. To examine these relationships, we define three examination areas:

- Area 1: less than the total of logical cores.
- Area 2: equal to the total of logical cores.
- Area 3: greater than the total of logical cores.

As mentioned in Section IV , we set the scale difference to 1 and the maximum thread count to 240. These settings ensured the gradual increase in thread configurations with a threshold exceeding the total number of logical cores. In addition, these settings allowed us to cover all the examination areas in a single execution. Fig. 4a, 4b, and 4c demonstrate the scalability of the ELK reasoner with varying ontology sizes and thread configurations. The red dashed line presents a reference mark pointing to the 48 logical cores.

In Fig. 4a, processing tiny ontologies OLATDV, INO, and FBDV displayed optimal performance at around 48 threads, with OLATDV achieved a speedup factor of 30, followed by FBDV of 25 and INO of 16. For OLATDV and INO, the optimal thread configuration was found in Area 1, while for FBDV the optimal solution was found in the first portion of Area 3, after which the performance decreased significantly. Similarly to tiny ontologies, the small ontologies PDON,

(a) Tiny ontologies

(b) Small ontologies

(c) Medium ontologies

Fig. 4. ELK's Scalability with varying ontology sizes and thread configurations.

PLANA, and WBPHENOTYPE reached the speed factor of 16, 12, and 6, respectively, with optimal configuration found in Areas 1 and 3, as shown in Fig. 4b. However, the range of speedup factors for the small ontologies was narrower than that of the tiny one, as indicated by the reduced scale of the speedup axis in Fig. 4b compared to the axis in Fig. 4a.

In Fig. 4a, processing small ontologies like OLATDV, INO, and FBDV showed optimal performance at approximately 48 threads. Processing OLATDV achieved a speedup factor of 30, FBDV reached 25, and INO managed 16. The best thread configuration for OLATDV and INO was in Area 1, while FBDV's optimal performance was in the initial part of Area 3, followed by a significant drop. Similarly, small ontologies PDON, PLANA, and WBPHENOTYPE achieved speed factors of 16, 12, and 6, respectively, with optimal configurations in Areas 1 and 3, as shown in Fig. 4b. The speedup range for small ontologies was narrower compared to tiny ones, reflected by the smaller scale of the speedup axis in Fig. 4b compared to Fig. 4a.

Fig. 4c presents notable performance gains for medium-sized ontologies compared to the small ones, where the average speedup for processing ORDO and EMAPA achieved factors between 4 and 5, while performance in processing OBA stabilized at a less speedup factor. Key observations were deduced from this figure. First, there is a notable decrease in the range of speedup factors compared to Fig. 4a and 4b. Second, the optimal thread configurations were identified in Area 1, indicating that adding more threads did not lead to further enhancements. Third, the trend line for this category shows a performance stabilization, suggesting that ELK reasoners benefits from parallelization in reasoning large ontologies more than small ones.

2) Regression analysis: In this analysis, we used an Ordinary Least Squares (OLS) regression model to quantify the impact of thread configurations and ontology size, measured in terms of the TBox axiom count, on the reasoning time. This model, with standardized predictors, explained 78.4% of the variance in reasoning time ($R^2 = 0.784$), highlighting its effectiveness in capturing the relationship between predictors and reasoning time. The results, presented in a 3D scatter plot shown in Fig. 5, revealed that while the TBox axiom count significantly affected the time of reasoning (β_1 = 0.8855, $p < 0.001$), the thread count has a negligible impact (β_2 = 0.0066, p = 0.512). Furthermore, the model showed a notable predictive accuracy, with an average Mean Squared Error (MSE) of 0.2165 and a Root Mean Squared Error (RMSE) of 0.4653.

VI. Discussion

The rapid expansion of ontologies and the lack of automatic tuning approaches have poses challenges on advancing parallel semantic reasoners. In related domains, several sophisticated tuning frameworks have been developed, applying existing search methods for optimization. Although existing search methods presented significant improvements, reducing their search time is still an active research field. This study addressed these gaps by proposing an automatic tuning methodology with an innovative tree-based search algorithm.

Our case study presented in Section V validated the performance gains, reliability, and effectiveness of automatic tuning in optimizing parallel semantic reasoners. Using ELK reasoner as a case study, our methodology, Eagle Framework (EF), efficiently completed the entire tuning process, from ontology loading to final optimization results, in less than five minutes across 240 thread configurations. Practically, such efficiency cannot be achieved in manual tuning approaches, underscoring the importance of applying automatic tuning methods to optimize the performance of semantic reasoner. These findings align with the conclusion of Mustafa's study,

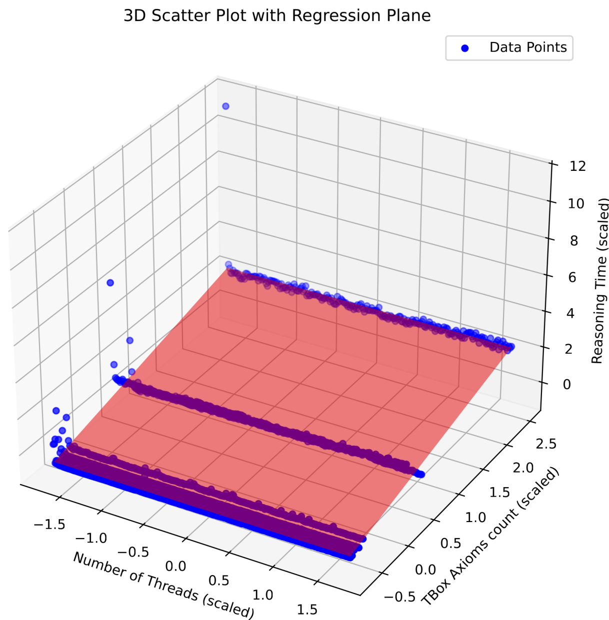

Fig. 5. A 3D Scatter plot with regression plane illustrating the relationship between TBox axiom count, number of threads, and reasoning time.

who also found through his survey study that automatic tuning ultimately outperforms manual tuning and becomes a crucial demand for optimizing parallel architectures [49].

We benchmarked our search algorithm, Parallelism Optimizer (PO), against the methods reviewed in Section II-C, namely: Grid search (GS), hill climb (HC), simulation annealing (SA), and random search (RS). We ensured a fair comparison by isolating the implementation of PO from other components in EF, similar to the isolation strategy employed in [40]. The results demonstrated the superiority of PO over its counterparts. PO exhibited logarithmic growth in search time and achieved a perfect success rate in identifying the optimal degree of parallelism. In contrast, other algorithms showed linear growth in search time and varying success rates. This superiority is gained from the deterministic characteristics of PO combined with the structural design of the Parallelism Tree (PT) search space. Specifically, the ordering nodes in PT based on a performance metric (i.e. the speedup factor for this study) and the queuing mechanism for storing parallelism configurations significantly decreased the search time and reduced tree size. Compared to the methodology proposed in [7], where a long chain of trees with a distinct node was used only for storage purposes, our methodology leveraged the efficiency of an integration between the tree-based and priority queue in storage and exploration purposes. However, our methodology provides efficiency for one-parameter optimization, and multi-dimensionality is not supported yet.

The analysis in Section V-D investigated the influence of increasing thread count and ontology size on ELK's scalability. For small ontologies, the results showed severe performance degradation as the number of threads increases. On the other hand, larger ontologies exhibited a lower speedup factor but maintained stable performance with the increase in thread count. Our findings demonstrate that the ELK reasoner scales efficiently with larger ontologies using a high degree of parallelism, while for small ontologies it performs poorly due to over-utilization of processing units. These findings emphasize those in [16], where the authors stated that their ELK reasoner benefited more from increased parallelism when processing larger ontologies than smaller ones. In summary, this study highlights the need for adaptive tuning approaches to develop efficient and scalable reasoning systems.

This study effectively applied the intraclass correlation coefficient (ICC) method to analyze the reliability of EF. This effectiveness was the result of the following conditions. First, the sample size used in each ICC assessment were relatively large. Second, the massively parallel resources in HPC environment led to precise variance in the reasoning time measurements. Third, the experimental setup ensured exclusive access to HPC resources, leading to clean results, demonstrating the robustness of EF. These conditions enabled the ICC to effectively detect variance in time measurements both within and between parallelism configurations, contributing to a narrower confidence interval range. Based on these findings, we recommend future investigations to explore the viability of ICC in evaluating tuning results implemented under the same conditions.

This study offers significant contributions to revitalizing the domain of semantic reasoning and expanding existing research on optimization approaches. However, it was constrained by the limitations of the ELK reasoner, which operates only on a shared memory system. Furthermore, the hardware resources of the experimental environment restricted us from using massive-size ontologies. Future experimentation is required to validate the effectiveness of our methodology in optimizing different semantic reasoners on different computing architectures.

VII. Conclusion

As the size and complexity of ontologies expand, particularly with the advent of the Internet of Things and other data-driven systems, optimizing parallel semantic reasoners has become a significant challenge. This study proposed the Eagle Framework (EF), an innovative automatic tuning framework aimed at helping researchers optimize the performance of semantic reasoners. EF automatically generates thread configurations and effectively records performance data. EF differentiates itself through its modular design and adaptability, operating as a black-box solution that integrates seamlessly with various parallel reasoning engines. By designing a novel tree-based search algorithm, EF efficiently identifies the optimal number of threads. EF's methodology significantly reduces the manual effort required for tuning parallelism, saving researchers time and enabling them to focus on higher-level tasks. EF's ability lies in writing the performance measurements in CSV files, making them ready for data analysis. In addition, EF represents performance data in high-resolution visualization, offering researchers a comprehensive understanding of how different configurations impact reasoning efficiency.

Through a case study, this research validated the efficiency of EF in tuning thread configurations for the ELK reasoner across varied ontology sizes. Comparative analysis shows that

EF efficiently identifies optimal parallelism, outperforming existing search algorithms applied in optimization studies. Furthermore, this study validated the effectiveness of EF in addressing key research questions commonly discussed in the literature, such as the relationship between optimal performance and the full utilization of logical cores and the scalability of parallel reasoners to increase both processing resources and ontology size. In addition, this study introduced the application of the intraclass correlation coefficient (ICC) in assessing the reliability of performance tuning tools. The findings validated the consistency of the EF tuning measurements within and between configurations, suggesting the accuracy of ICC in assessing the reliability of tuning systems executed on a high-performance computing architecture.

REFERENCES

[1] M. Lnenicka and J. Komarkova, "Big and open linked data analytics ecosystem: Theoretical background and essential elements," *Government Information Quarterly*, vol. 36, pp. 129–144, 1 2019.

[2] M.-C. Valiente and J. Pavón, "Web3-dao: An ontology for decentralized autonomous organizations," *Journal of Web Semantics*, vol. 82, p. 100830, 10 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1570826824000167

[3] C. Yang, Y. Zheng, X. Tu, R. Ala-Laurinaho, J. Autiosalo, O. Seppänen, and K. Tammi, "Ontology-based knowledge representation of industrial production workflow," *Advanced Engineering Informatics*, vol. 58, p. 102185, 10 2023.

[4] P. Bonte, F. D. Turck, and F. Ongenae, "Bridging the gap between expressivity and efficiency in stream reasoning: a structural caching approach for iot streams," *Knowledge and Information Systems*, vol. 64, pp. 1781–1815, 7 2022.

[5] S. Arslan and O. Ünsal, "Efficient thread-to-core mapping alternatives for application-level redundant multithreading," *Concurrency and Computation: Practice and Experience*, vol. 35, 11 2023.

[6] Z. Quan and V. Haarslev, "A parallel computing architecture for high-performance owl reasoning," *Parallel Computing*, vol. 83, pp. 34–46, 4 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016781911830142X

[7] A. Rasch, R. Schulze, M. Steuwer, and S. Gorlatch, "Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (atf)," *ACM Transactions on Architecture and Code Optimization*, vol. 18, pp. 1–26, 3 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3427093

[8] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in internet of things: Taxonomies and open challenges," *Mobile Networks and Applications*, vol. 24, pp. 796–809, 6 2019.

[9] E. P. Cynthia, S. B. M. Samuri, W. S. Li, E. Ismanto, L. Afriyanti, and M. I. Arifandy, "Improved machine learning algorithm for heart disease prediction based on hyperparameter tuning," in *2023 IEEE International Conference on Artificial Intelligence in Engineering and Technology (IICAIET)*. IEEE, 9 2023, pp. 176–181.

[10] M. A. Ramadhani, Y. Azhar, and G. W. Wicaksono, "A study on the implementation of yolov4 algorithm with hyperparameter tuning for car detection in unmanned aerial vehicle images," in *2023 11th International Conference on Information and Communication Technology (ICoICT)*. IEEE, 8 2023, pp. 639–644.

[11] G. Cheng, S. Ying, and B. Wang, "Tuning configuration of apache spark on public clouds by combining multi-objective optimization and performance prediction model," *Journal of Systems and Software*, vol. 180, p. 111028, 10 2021.

[12] D. Nikitopoulou, D. Masouros, S. Xydis, and D. Soudris, "Performance analysis and auto-tuning for spark in-memory analytics," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2 2021, pp. 76–81.

[13] J. J. Durillo, P. Gschwandtner, K. Kofler, and T. Fahringer, "Multi-objective region-aware optimization of parallel programs," *Parallel Computing*, vol. 83, pp. 3–21, 4 2019.

[14] J. B. Fernandes, F. H. S. da Silva, T. Barros, I. A. Assis, and S. X. de Souza, "Patsma: Parameter auto-tuning for shared memory algorithms," *SoftwareX*, vol. 27, p. 101789, 9 2024.

[15] A. N. Lam, B. Elvesaeter, and F. Martin-Recuerda, "A performance evaluation of owl 2 dl reasoners using ore 2015 and very large bio ontologies," in *DMKG2023: 1st International Workshop on Data Management for Knowledge Graphs*, vol. 3443. Technical University of Aachen, 5 2023, p. 13. [Online]. Available: https://dmkg-workshop.github.io/papers/paper2861.pdf

[16] Y. Kazakov, M. Krötzsch, and F. Simančík, "The incredible elk," *Journal of Automated Reasoning*, vol. 53, pp. 1–61, 6 2014. [Online]. Available: http://link.springer.com/10.1007/s10817-013-9296-3

[17] G. Santipantakis and G. A. Vouros, "Distributed reasoning with coupled ontologies: the e-shiq representation framework," *Knowledge and Information Systems*, vol. 45, no. 2, pp. 491–534, November 2015. [Online]. Available: http://link.springer.com/10.1007/s10115-014-0807-2

[18] T. Wang, Y. Zhu, P. Ye, W. Gong, H. Lu, H. Mo, and F.-Y. Wang, "A new perspective for computational social systems: Fuzzy modeling and reasoning for social computing in cpss," *IEEE Transactions on Computational Social Systems*, vol. 11, pp. 101–116, 2 2024.

[19] Y.-B. Kang, S. Krishnaswamy, W. Sawangphol, L. Gao, and Y.-F. Li, "Understanding and improving ontology reasoning efficiency through learning and ranking," *Information Systems*, vol. 87, p. 101412, 1 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0306437917306476

[20] S. Borgwardt and R. Peñaloza, "Algorithms for reasoning in very expressive description logics under infinitely valued gödel semantics," *International Journal of Approximate Reasoning*, vol. 83, pp. 60–101, 4 2017.

[21] H. Herodotou, Y. Chen, and J. Lu, "A survey on automatic parameter tuning for big data processing systems," *ACM Computing Surveys*, vol. 53, pp. 1–37, 3 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3381027

[22] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 1 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X18313359

[23] J. Schwarzrock, H. M. G. de A. Rocha, A. C. S. Beck, and A. F. Lorenzon, "Effective exploration of thread throttling and thread/page mapping on numa systems," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 12 2020, pp. 239–246. [Online]. Available: https://ieeexplore.ieee.org/document/9408014/

[24] Z. Fan, R. Sen, P. Koutris, and A. Albarghouthi, "Automated tuning of query degree of parallelism via machine learning," in *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, 6 2020, pp. 1–4. [Online]. Available: https://dl.acm.org/doi/10.1145/3401071.3401656

[25] A. Vogel, D. Griebler, and L. G. Fernandes, "Providing high-level self-adaptive abstractions for stream parallelism on multicores," *Software: Practice and Experience*, vol. 51, pp. 1194–1217, 6 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.2948

[26] T. Siddiqui, A. Jindal, S. Qiao, H. Patel, and W. Le, "Cost models for big data query processing: Learning, retrofitting, and our findings," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 6 2020, pp. 99–113. [Online]. Available: https://dl.acm.org/doi/10.1145/3318464.3380584

[27] Y. Liu, M. Li, N. K. Alham, and S. Hammoud, "Hsim: A mapreduce simulator in enabling cloud computing," *Future Generation Computer Systems*, vol. 29, pp. 300–308, 1 2013. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X11000884

[28] R. Andonie, "Hyperparameter optimization in learning systems," *Journal of Membrane Computing*, vol. 1, pp. 279–291, 12 2019. [Online]. Available: http://link.springer.com/10.1007/s41965-019-00023-0

[29] S. G. C. G and B.Sumathi, "Grid search tuning of hyperparameters in random forest classifier for customer feedback sentiment prediction," *International Journal of Advanced Computer Science and Applications*, vol. 11, 2020. [Online]. Available: http://thesai.org/Publications/ViewPaper?Volume=11\&Issue=9\&Code=IJACSA\&SerialNo=20

[30] I. Priyadarshini and C. Cotton, "A novel lstm–cnn–grid search-based deep neural network for sentiment analysis," *The Journal of Supercomputing*, vol. 77, pp. 13 911–13 932, 12 2021. [Online]. Available: https://link.springer.com/10.1007/s11227-021-03838-w

[31] D. Chen, R. Zhang, and R. G. Qiu, "Noninvasive mapreduce performance tuning using multiple tuning methods on hadoop," *IEEE Systems Journal*, vol. 15, pp. 2906–2917, 6 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9205847/

[32] P. Sewal and H. Singh, "Algorithmic proficiency in spark configuration tuning: An empirical study using execution time metrics across varied workloads," *Procedia Computer Science*, vol. 235, pp. 2307–2317, 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050924008950

[33] R. Sivakumar and H. Mangalam, "Ensemble hill climbing optimization in adaptive cruise control for safe automated vehicle transportation," *Journal of Supercomputing*, vol. 76, pp. 5780–5800, 8 2020.

[34] J. Zeng, P. Romano, J. Barreto, L. Rodrigues, and S. Haridi, "Online tuning of parallelism degree in parallel nesting transactional memory," in *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium, IPDPS 2018*. Institute of Electrical and Electronics Engineers Inc., 8 2018, pp. 474–483.

[35] A. K. Pradhan, D. Mishra, K. Das, M. S. Obaidat, and M. Kumar, "A covid-19 x-ray image classification model based on an enhanced convolutional neural network and hill climbing algorithms," *Multimedia Tools and Applications*, vol. 82, pp. 14 219–14 237, 4 2023. [Online]. Available: https://link.springer.com/10.1007/s11042-022-13826-8

[36] A. Kuznetsov, M. Karpinski, R. Ziubina, S. Kandiy, E. Frontoni, O. Peliukh, O. Veselska, and R. Kozak, "Generation of nonlinear substitutions by simulated annealing algorithm," *Information (Switzerland)*, vol. 14, 5 2023.

[37] A. Gülcü and Z. Kuş, "Multi-objective simulated annealing for hyper-parameter optimization in convolutional neural networks," *PeerJ Computer Science*, vol. 7, p. e338, 1 2021. [Online]. Available: https://peerj.com/articles/cs-338

[38] M. Abdel-Basset, W. Ding, and D. El-Shahat, "A hybrid harris hawks optimization algorithm with simulated annealing for feature selection," *Artificial Intelligence Review*, vol. 54, pp. 593–637, 1 2021. [Online]. Available: https://link.springer.com/10.1007/s10462-020-09860-3

[39] F.-J. Willemsen, R. Schoonhoven, J. Filipovič, J. O. Tørring, R. van Nieuwpoort, and B. van Werkhoven, "A methodology for comparing optimization algorithms for auto-tuning," *Future Generation Computer Systems*, vol. 159, pp. 489–504, 10 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X24002498

[40] K. Deligkaris, "Particle swarm optimization and random search for convolutional neural architecture search," *IEEE Access*, vol. 12, pp. 91 229–91 241, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10577981/

[41] F. Hosseini, C. Prieto, and C. Álvarez, "Hyperparameter optimization of regional hydrological lstms by random search: A case study from basque country, spain," *Journal of Hydrology*, vol. 643, p. 132003, 11 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0022169424013994

[42] O. Krestinskaya, M. E. Fouda, H. Benmeziane, K. E. Maghraoui, A. Sebastian, W. D. Lu, M. Lanza, H. Li, F. Kurdahi, S. A. Fahmy, A. Eltawil, and K. N. Salama, "Neural architecture search for in-memory computing-based deep learning accelerators," *Nature Reviews Electrical Engineering*, vol. 1, pp. 374–390, 5 2024. [Online]. Available: https://www.nature.com/articles/s44287-024-00052-7

[43] C. Huang, Y. Li, and X. Yao, "A survey of automatic parameter tuning methods for metaheuristics," *IEEE Transactions on Evolutionary Computation*, vol. 24, pp. 201–216, 4 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8733017/

[44] C. C. Foster, "A generalization of avl trees," *Communications of the ACM*, vol. 16, pp. 513–517, 8 1973.

[45] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," *ACM SIGPLAN Notices*, vol. 45, pp. 257–268, 5 2010.

[46] G. Antoniou, S. Batsakis, R. Mutharaju, J. Z. Pan, G. Qi, I. Tachmazidis, J. Urbani, and Z. Zhou, "A survey of large-scale reasoning on the web of data," *The Knowledge Engineering Review*, vol. 33, p. e21, 12 2018.

[47] D. Liljequist, B. Elfving, and K. S. Roaldsen, "Intraclass correlation – a discussion and demonstration of basic features," *PLOS ONE*, vol. 14, p. e0219854, 7 2019.

[48] H. Kim, C. M. Park, and J. M. Goo, "Test-retest reproducibility of a deep learning–based automatic detection algorithm for the chest radiograph," *European Radiology*, vol. 30, pp. 2346–2355, 4 2020.

[49] D. Mustafa, "A survey of performance tuning techniques and tools for parallel applications," *IEEE Access*, vol. 10, pp. 15 036–15 055, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9698048/