Using Graph Neural Networks for Distributed Link Prediction

Konstantin Igin

375455

A thesis submitted to the

Faculty of Electrical Engineering and Computer Science

of the

Technical University of Berlin

in partial fulfillment of the requirements for the degree

Bachelor of Computer Science

Advisor: Morgan Geldenhuys

1st Examiner: Prof. Dr. habil. Odej Kao

2nd Examiner: Prof. Dr. Dr. h.c. Sahin Albayrak

Berlin, Germany

April, 2021

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin,

ACKNOWLEDGEMENTS

Throughout the creation of this thesis I have received a great deal of support.

I would like to thank my supervisor, Professor Odej Kao, for the opportunity to develop my idea in a friendly environment of the department of Distributed and Operating Systems (DOS).

I would also like to express my gratitude to the DOS team for assisting me in conducting my experiments on their computational cluster and give me hints and constructive critiques during my topic presentation. I would particularly like to single out my advisor, Morgan Geldenhuys, for providing me with the tools and tips that I needed to sharpen the topic of this thesis. I would also like to thank my previous seminar supervisor, Felix Lorenz, for introducing me to the subject of graph databases and link prediction.

In addition, I would like to thank my family and friends for always being there for me. My parents, as well as Tetiana Yakovenko and Anastasiia Huk were providing me a sympathetic ear during the whole work and have shown a great value of patience. Special acknowledgment belongs to Elisaveta Kulicheva, who read the thesis many times in many forms and provided helpful feedback. Thank you all for stimulating discussions and happy distractions.

ABSTRACT

Link prediction is one of the key problems in the domain of network science. The problem is often considered as classification - a given link either exists in a graph or not. That results in the appearance of many machine-learning-based approaches designed to solve it. Particularly interesting are graph neural networks - deep-learning-based methods that operate directly on graph-structured data. This thesis discusses the parallelization options for the recent state-of-the-art graph neural network-based framework for link prediction - SEAL. We show the possible ways of distributing the calculations required by the original method by integrating their logic in a popular big data processing system Apache Spark. In further steps, we also consider how existing data storage options in the form of graph databases can be included in our pipeline to increase computational performance. We provide implementations to the proposed parallelization methods and evaluate their performance on a cluster against the original SEAL framework.

ZUSAMMENFASSUNG

Die Link-Prädiktion ist eines der Schlüsselprobleme im Bereich der Network Science. Das Problem wird oft als Klassifizierung beschrieben - ein gegebener Link existiert entweder in einem Graphen oder nicht. Das führt dazu, dass es viele auf maschinellem Lernen basierende Ansätze gibt, die dieses Problem lösen sollen. Besonders interessant sind Graph Neural Networks - Deep-Learning-basierte Methoden, die direkt mit graph-strukturierten Daten arbeiten. In dieser Arbeit werden die Parallelisierungsmöglichkeiten für das aktuelle, auf Grapn Neural Networks basierende Framework zur Link-Prädiktion - SEAL - diskutiert. Es werden die Möglichkeiten gezeigt, die von der ursprünglichen Methode benötigten Berechnungen zu verteilen. Dazu wird ihre Logik in das Big-Data-Verarbeitungssystem Apache Spark integriert. In weiteren wird auch Schritten betrachtet, wie bestehende Datenspeicheroptionen in Form von Graphdatenbanken in unsere Pipeline einbezogen werden können, um die Rechenleistung zu erhöhen. Wir stellen Implementierungen zu den vorgeschlagenen Parallelisierungsmethoden zur Verfügung. Die Rechengeschwindigkeit der implementierten parallelizsiruntdmethoden wird bewertet, indem die Rechenleistung auf einem Cluster gegenüber dem ursprünglichen SEAL-Framework verglichen wird.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

APOC ”Awesome Procedures on Cypher”. 10, 15, 19, 20, 32

CNN Convolutional Neural Network. 4

DSL Domain-Specific Language. 10, 20

GNN Graph Neural Network. 2–5, 7, 8, 11, 12, 15, 17, 20, 35

GraphDB Graph Database. 3, 11

LP Link prediction. 1, 2, 5, 34

RDD Resilient Distributed Dataset. 8–10, 13, 14, 18, 19, 35

SECTION 1. INTRODUCTION

Link prediction (LP) is an actively studied problem in the domain of graph theory and network science. The problem consists of inferring the existence of a new edge or a still unknown connection between pairs in a given graph structure [1].

Link prediction technics have a number of practical appliances in various scientific domains. In biology, they can be applied to protein-protein interaction networks to predict previously unknown interactions between proteins [2]. It has also been shown how methods of link prediction can be used in criminology in order to identify missing connections in a criminal network [3]. Another interesting field of application of link prediction is marketing, where it is utilized to predict trends [4]. It is also being actively applied by big technology companies to build their recommendation engines, suggest their user topics of potential interest, make suggestions of new friendships etc. [1]. Any domain with entities that interact in a structured way is a potential application field for link prediction: it helps exclude unlikely connections and thereby decreases the costs of experiments' conduction.

Methods of link prediction are based on the extraction of implicit information present in the network as well as modeling of the network grow mechanisms. A basic solution for the problem is present in the form of **heuristic methods**. Heuristic methods assign target nodes similarity scores, notating the likelihood of link formation between those [5]. Similarity scores are calculated based on certain graph-structure features, e.g., the number of common neighbors of the target nodes, their ability to form connections (preferential attachment score) or their distance in the graph [6]. Calculated heuristic values are often combined to form a feature vector for a more advanced, machine-learning-based link prediction approach - **supervised heuristic learning**. In this case, machine learning models like logistic regression [7] or support vector machines [8] are trained on a selected heuristics combination. However, feature selection is a complicated step that often requires domain knowledge to succeed: e.g., the number of common neighbors may be relevant for link formation in social networks, but it fails in protein-protein interaction networks [9]. A feature vector is a compound of predefined heuristics set chosen during feature selection, which is why these methods cannot learn general graph structure features [5]. **Graph Neural Network (GNN)** is one of the particular solutions that address this problem, automating the process of feature selection and adapting learned features better for a particular graph [5]. Zhou et al. [10] define GNN in their recent overview as deep-learning-based methods that operate on graph structured data. Scarselli et al. [11] were the first to propose the concept of GNN. Works that followed showed that this method could be used to effectively solve graph-specific problems like node classification [12] or link prediction [5],[13].

Link prediction is particularly important in the context of knowledge graphs, where it is being applied to complete data segments and incorporate missing links [1]. According to Hogan [1], Knowledge Graphs tend to grow over time. Noy et al. [14] confirm this tendency by presenting huge corporate knowledge graphs consisting of millions or even billions of nodes and edges, highlighting the importance of scalable solutions in the domain. And the scientific world reacts to it. Fokoue et al. [15] proposed a system to collect information about drug-drug interactions from various sources and to predict possible interactions between drugs captured by the system. The proposed system uses a supervised heuristic method based on a linear regression model deployed in a popular distributed processing system Apache Spark. Mohan et al. [16] proposed a scalable method based on a parallel label propagation algorithm for community detection and a parallel community information-based Adamic–Adar measure for link prediction. A similar approach was shown by Wan et al. [17]: an algorithm that scales inside Apache Spark's graph processing framework, GraphX. All these methods apply supervised heuristic learning principles. As far as we know, there is currently no scientific evidence of distributed solutions for LP problem based on GNN. However, GNN are shown to have impressive results when applied for LP. Zhang and Chen [5] proved that reasonable heuristics could be learned from nearest surroundings of target nodes - *so called enclosing subgraphs* - in the graph structure and provided a way to learn a function mapping from neighboring subgraphs, using GNN. Resulting framework is called SEAL. It has been shown to outperform all heuristic methods, latent feature methods, and recent network embedding methods as well as previous state-of-the-art methods [5].

Some of the mentioned supervised heuristic learning-based solutions for link prediction can be efficiently distributed and scaled using big-data processing tools like Apache Spark. In this thesis, we propose several adjustments to a modern link prediction system based on GNN – SEAL [5]. We aim to effectively distribute the original approach by integrating it into the Apache Spark environment. That leads us to **the first hypothesis** of our work: *Using Apache Spark, we can effectively distribute calculations required by the SEAL system to perform a link prediction and, as such, reduce the amount of time required for it.*

Additionally, we consider how data storage solutions can be included in our pipeline to increase computational performance. Cohen and Cohen-Tzemach [18] have shown how heuristic methods can be implemented in databases' build-int query languages, like GraphQL, Cypher, or SQL. Feature extraction is computationally the most complex part of link prediction. Thus this advantage is critical for performance. We integrate a Graph Database (GRAPHDB) into our distributed prediction pipeline, *as a more suitable storage option for graph-modelled data.* GNN applied in SEAL [5] learns on subgraphs and those can be efficiently extracted using GRAPHDB built-in tools and query languages. This statement leads us to **the second hypothesis** of our work: *Link Prediction System based on* GNN *could benefit from applying* GRAPHDB *built-in tools and query languages by decreasing the amount of time required for feature extraction.*

SECTION 2. BACKGROUND AND THEORY

2.1 Graph Neural Networks

In the previous section we stated that heuristic-based solutions are constrained by their static nature. Meanwhile, GNNs can be efficiently applied to learn general graph features. That ability reduces the need for a resource-consuming feature selection process.

As shortly mentioned in the last section, Scarselli et al. [11] were the first ones to propose a concept of GNN. Since 2009, there have appeared many extensions and other methods currently associated with the term GNN. The approach developed by Scarselli et al. was centered around what they called an information diffusion mechanism: an input graph is processed by units, each of which is associated with a particular node of the input graph, connected according to their connectivity in the graph. These units update their states and exchange information as long as they reach a state of equilibrium [11]. The output is then computed locally at each node on the base of the unit state [11].

According to Zhou et al. [10], the concept of GNNs was largely impacted by Convolutional Neural Networks (CNNs) [19]. CNNs transformed the area of machine learning and provided a completely new way of processing images and text data while being limited by the Euclidean nature of these types of data [10]. However, the data structures of images as two-dimensional matrices and texts as one-dimensional sequences are particular cases of graphs. As stated by Zhou et al. [10] it is legit to consider GNNs as a generalization of CNNs to graphs. Another source of inspiration for GNNs are *graph embeddings*: these methods learn to represent the graph structure elements in low-dimensional vectors. *Graph embedding* methods achieved significant progress in efficiently representing the complex underlying structure of graphs [10]. Nevertheless, these methods do not generalize to new input graphs [5]. Zhou et al. [10] state that based on CNNs and *graph embedding*, GNNs are used to aggregate information from the graph structure.

2.2 Alternative GNN methods for Link Prediction

Recent GNN-based link prediction methods seem to outperform other approaches on a variety of problems. Gu, Gao and Lou [13] presented an approach called DeepLinker that improves the original concept of a Graph Attention Network (GAT) proposed by Velickovic et al. [12]. Compared to the original approach, it can solve link prediction tasks, avoiding the memory bottleneck problem associated with link prediction that is caused by much larger feature computation. Despite the fact that DeepLinker provides exceptional results in terms of accuracy of LP, it does not avoid scalability problems that come together with GNNs: usage of whole graphs as input compared to the supervised heuristic methods with calculated feature vectors leads to the memory bottleneck problem. Zhang and Chen [5] developed a theory that unified a variety of existing heuristics in a single framework and proposed an efficient method to learn adaptable heuristics from relatively small local subgraphs using GNN. A similar approach was proposed by Kipf and Welling [20]: their Graph Auto-Encoder also applies a GNN to learn local structures and features around target links. However, Zhang [21] noticed that SEAL still outperforms other methods due it its unique feature - *node labeling* - that will be discussed further on.

2.3 SEAL

Zhang and Chen [5] proposed a way to learn adaptable heuristics for any given graph structure from local subgraphs instead of using pre-defined graph features. Due to the fact that our work is based on their method, in this section, we introduce the theoretical overview of their framework.

2.3.1 Enclosing Subgraph

Zhang and Chen prove that reasonable graph features can be learned from the surroundings of the target nodes - *the enclosing subgraphs* - this is a key concept of the SEAL method. Authors define the enclosing subgraph for a node pair *(x, y)* as the subgraph induced from the network by the union of x's and y's neighbors up to **h hops**. Figure 2.1 we copied from the original work, illustrates the

1-hop enclosing subgraphs extracted for node pairs *(A, B)* and *(C, D)*.

A formal definition for enclosing subgraph is: *For a graph G = (V, E), given two nodes $x, y \in V$, the h-hop enclosing subgraph for (x, y) is the subgraph $G^h_{x,y}$ induced from G by the set of nodes* $\{\, i \mid d(i,x) \leq h$ *or* $d(i,y) \leq h\}$.

One definite difference of enclosing subgraphs compared to the usual subgraphs, extracted by graph-processing frameworks using graph-traversal technics, is the union of edges of neighboring nodes: as we can see in Figure 2.1, the extracted subgraph for the pair *(C, D)* contains edges between neighbor-nodes of D and C and not only edges between D, C, and their neighbors. That would not have been achieved by graph-traversal of corresponding hop level. This difference is significant for our topic since it adds additional complexity to the task of enclosing subgraph extraction.

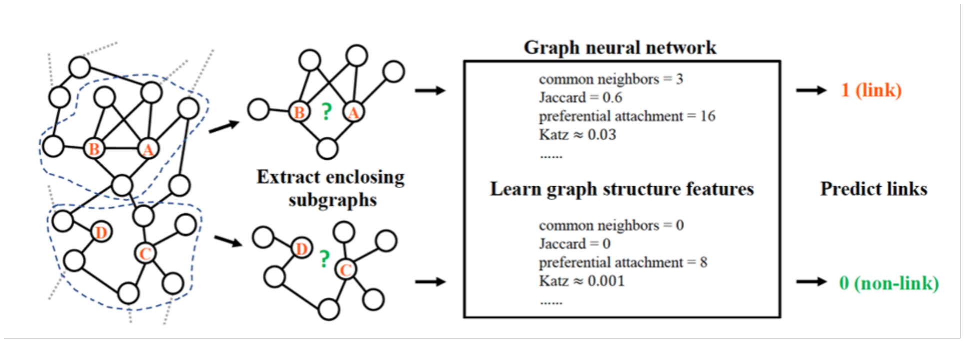

Figure 2.1. The SEAL framework, source: [5]

2.3.2 Theoretical justification

Zhang and Chen argue that existing heuristics can be classified by the maximum hop **h** of neighbors required to calculate the prediction score. They state that some heuristics require a small number of **h**, like Common Neighbors and preferential attachment, where required **h** is 1. At the same time, other methods are **high-ordered** and require knowledge about the entire network, like Katz, rooted PageRank, etc.

Authors state a theorem that any h-order heuristic for a pair of nodes *(x, y)* can be accurately calculated from the enclosing subgraph of level **h**. They then expand it by proving that learning high-order heuristics is also feasible with a small **h**. They are doing so by proposing a unifying theory for heuristics - $\gamma - decaying$ heuristic and by proving that a number of well-known high-level heuristics, *like Katz index, PageRank and SimRank*, can be generalized in this form. Then they show how to approximate the $\gamma - decaying$ heuristic from the h-hop enclosing subgraphs as such demonstrating that a number of high-order heuristics can be effectively approximated from an h-hop enclosing subgraph as well.

2.3.3 SEAL steps

SEAL framework is the practical implementation of Zhang and Chen's theoretical research, except that SEAL doesn't require the learned features to have the $\gamma - decaying$ form but learns general graph structures. The framework aims to learn a function mapping from enclosing subgraphs around target nodes to the likelihood of link existence. The process is depicted in Figure 2.1, however it should be noted that the heuristics listed inside the box serve only illustration purposes - the learned features may be completely different from existing heuristics. In order to learn such a function, authors apply a GNN over the enclosing subgraphs. When the function is learned, it can be applied in a link prediction stack with the same steps (*with* GNN *learning being constituted by trained* GNN *application*). SEAL comprises three steps to extract the features:

1. *Enclosing subgraph extraction*
2. *Node information matrix construction*
3. *GNN learning*

SEAL performs most of its graph operations using sparse matrices representation of graphs, making it highly efficient. In the Python version of the framework, the authors apply the SciPy-Sparse library for this purpose. The process of enclosing subgraph extraction is then performed by expanding the search area from target nodes to the required distance using various matrix-powered operations of the SciPy-Sparse library.

Node information matrix consists of three further components: 1) *structural node labels*, 2) *node embeddings*, 3) *node attributes*. Authors define node labeling as a function: $f_1 : V \to \mathbb{N}$ which assigns an integer label $f_1(i)$ to every node *i* in the enclosing subgraph. Zhang and Chen state that different labels provided to the nodes by the function mark node's different roles in an enclosing subgraph: center nodes are the target nodes between which link is located, other nodes have different relative positions to the target nodes. A proper node labeling function should mark these differences for the GNN to be able to recognize the target and surrounding nodes. The other two components of the node information matrix represent latent and explicit features and are shown to improve the prediction accuracy when included in the matrix but are optional for the whole process. In practice, authors do not attach structural node labels to the node information matrix but add them directly to the data structure processed by a GNN.

Enclosing subgraphs together with node information matrices are being used to train a GNN model. Authors apply the DGCNN framework previously proposed by Zheng et al. [22] as the default GNN engine of SEAL. DGCNN is a GNN framework for graph classification based on Torch that has been shown to have positive performance on a number of various benchmark datasets [22]. The configuration of the neural network model is not covered since it is applied as-is in our thesis.

2.4 Apache Spark

Apache Spark *(from now on also Spark for short)* is a unified engine for distributed data processing developed in 2009 by a group at the University of California, Berkeley [23]. Spark has a programming model similar to the well-known MapReduce[24], although it is extended through a powerful data-sharing abstraction - Resilient Distributed Datasets (RDDs) [23]. It plays a crucial role for our thesis and as such, we are going to look at it in more detail, using the recent publication of the group of its creators [23] together with the official documentation [25] of the engine as a basis for our discussion.

2.4.1 Architecture

Spark logic revolves around the concept of RDDs, which represent fault-tolerant collections of objects distributed across a cluster. RDDs provide the functionality to be processed simultaneously that is exposed via functional programming API in Scala, Java, Python, and R. As the official documentation [25] states: *"Every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster"*. RDDs can either be created through parallelization of existing collections in the driver program or read directly from an external data source. Recently Spark introduced a new abstraction level on top of RDDs that represents distributed relational structures - DataFrame API. Spark DataFrames support a number of SQL-inspired querying operations that make this level of abstraction popular among data scientists and engineers.

RDDs are being evaluated lazily and scheduled using a DAG-Scheduler. They can be operated on in two different ways: via *transformations* or *actions*. *Transformations* create a new dataset from an existing one and are being performed *lazily* - steps described through transformations are only being executed when an *action* is triggered. *Actions* return a value to the driver program after running a computation on the dataset. A prominent example of a transformation is the **map** method utilized to plan the execution of an input function on an RDD. The **collect** method is an example of an action used to trigger the execution of planned transformations and return the results to the driver machine. A central concept of Spark API is the passing of functions in the driver program to be executed on a cluster. Passed functions can be used in transformations as well as in actions, as long as all the required libraries and classes are available on all machines.

2.4.2 Graph Processing

Spark is an open-source project that is supported by more than 1000 contributors. Aside from the main engine, it is powered by a number of libraries for different arts of data analytics - SQL & DataFrames, MLib for Machine Learning, Graphx for Graph Processing, and Spark Streaming. Graph processing capabilities of Apache Spark are particularly interesting for our work. Graphx is a Spark component for parallel graph processing that extends the Spark concept of RDD with an additional Graph abstraction. However, this library is not quite adapted to the python version of Spark that we use in our work - PySpark. An alternative solution is present in the form of a recent library currently being not included in Spark - GraphFrames. GraphFrames provides similar functionality as Graphx, but it utilizies Spark DataFrames as its underlying data structure. GraphFrames extends its predecessor with additional features, like *motif-finding*, Data-Frame based serialization and powerful graph-queries.

GraphFrames official documentation [26] defines *Motif finding* as a way of searching for structural patterns in a graph, using a simple Domain-Specific Language (DSL) for expressing structural queries. ”$(a) - [e] \to (b)$” is an example of a motif. It expresses an edge e from vertex a to vertex b. Based on this motif, the GraphFrames engine returns all the nodes that are connected with each other in a pointed direction.

2.5 Graph databases and Neo4j

Graph data structures of knowledge graphs and social graphs represent a domain of high data variety – diversity of stored objects – and strong data interconnectedness – relational databases (SQL) are not suitable for such characteristics [1]. An alternative solution is present in the form of graph databases that employ graph-based abstraction of knowledge. Graph Database is a system developed for managing data present in a graph form according to the basic principles of databases [27].

The Neo4j [28] is a popular graph database solution. Its graph platform includes graph storage and supports transactional and analytical processing of graph data. A number of integrated tools come together with a common protocol, API, and query language (Cypher) to provide effective access for different uses. ”Awesome Procedures on Cypher” (APOC) library is supposedly the most prominent and most widely used extension library for Neo4j that contains a unified source of standard graph procedures, providing functionality for utilities, conversions, graph updates, etc.

SECTION 3. PROBLEM ANALYSIS

In the first section we stated two hypotheses that play a key role in this thesis. In this section, we will dive deeper into each hypothesis and discuss their problematic and associated complexity in more detail and define the limits and scope of this thesis. Let us repeat our stated hypotheses to maintain an overview:

1. *Using Apache Spark, we can effectively distribute calculations required by the SEAL system to perform a link prediction and, as such, reduce the amount of time required for it.*

2. *Link Prediction System based on* GNN *could benefit from applying* GRAPHDB *built-in tools and query languages by decreasing the amount of time required for feature extraction.*

3.1 Parallelization strategies

In the previous section, Apache Spark was introduced as a platform for spreading data and computations over a cluster of machines which is significant in the context of the first stated hypothesis. As such, Spark may be useful to increase the performance of any process that consists of a set of independent data computations that can be parallelized. The first problem we discuss is the parallelization strategy, i.e., which parts of the original SEAL process can be effectively parallelized and what steps can be transformed into independent data computations.

The SEAL system performs a link prediction in three basic steps that were already discussed in detail in the previous section: enclosing subgraph extraction, node information matrix construction, application of the trained GNN.

3.1.1 Limits of the thesis

First of all, our efforts were concentrated purposefully only on the first and on the third steps. The experiments with SEAL ascertained that a significant amount of time required by the framework is utilized to extract the enclosing subgraphs for pairs in the test-set and is mandatory for further prediction. Figure 3.1 shows how much time the original system, which ran on a modern high-end laptop *(configuration given in the next section)* takes to execute this first step on different datasets and test data sizes of 10, 100, 500, and 1000 pairs, when hop-level is set to 2. The resulting enclosing subgraph extraction times for relatively big datasets facebook and arxiv are astonishing - with the arxiv dataset as a base and 500 test links, the original process requires more than an hour.

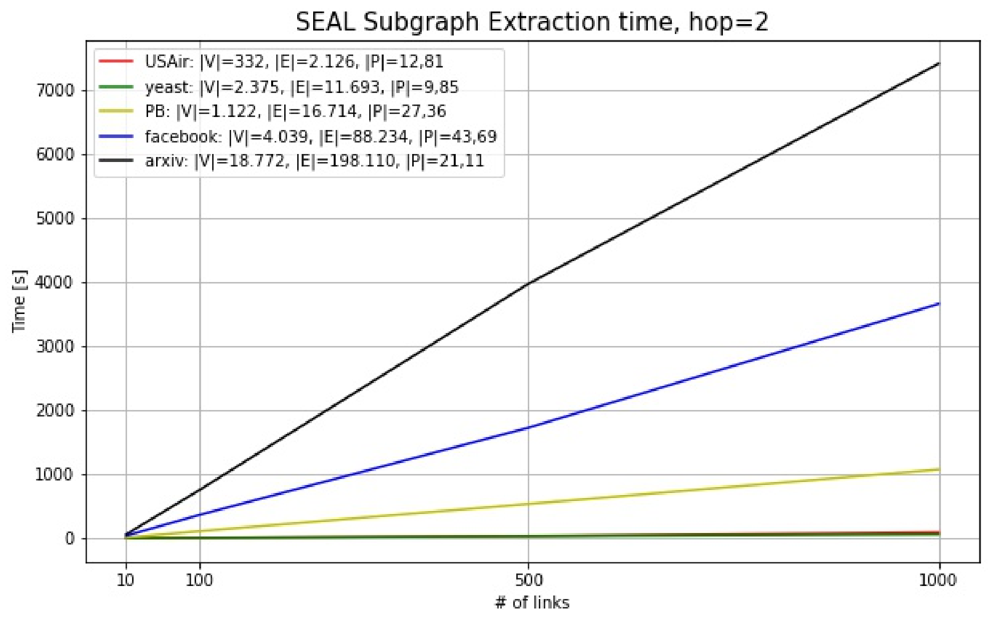

Figure 3.1. SEAL Subgraph Extraction time for hop of 2 w. different datasets, where |V| is number of nodes, |E| is number of edges, |P| is average node degree.

Meanwhile, the node information matrix construction attachment to the feature vector does improve the prediction accuracy but is an optional step for the whole process [5]. Nevertheless, we pay more attention to the scalability side of the solution rather than to the accuracy. Additionally, as mentioned in the previous section, the second step is rather complex itself (segregation into three other components). However, it is an exciting domain for future research in this field of study. Enclosing subgraph extraction together with the application of GNN is a sufficient amount of work for this thesis.

3.1.2 *Presentation of parallelization strategies*

Considering the Spark architecture, the two strategies to integrate the original approach into the Apache Spark environment were developed: A) to distribute operations on test-set or B) to distribute operations on the graph data itself. Figure 3.2 represents a diagram illustration of both parallelization options and serves as a basis for further discussion.

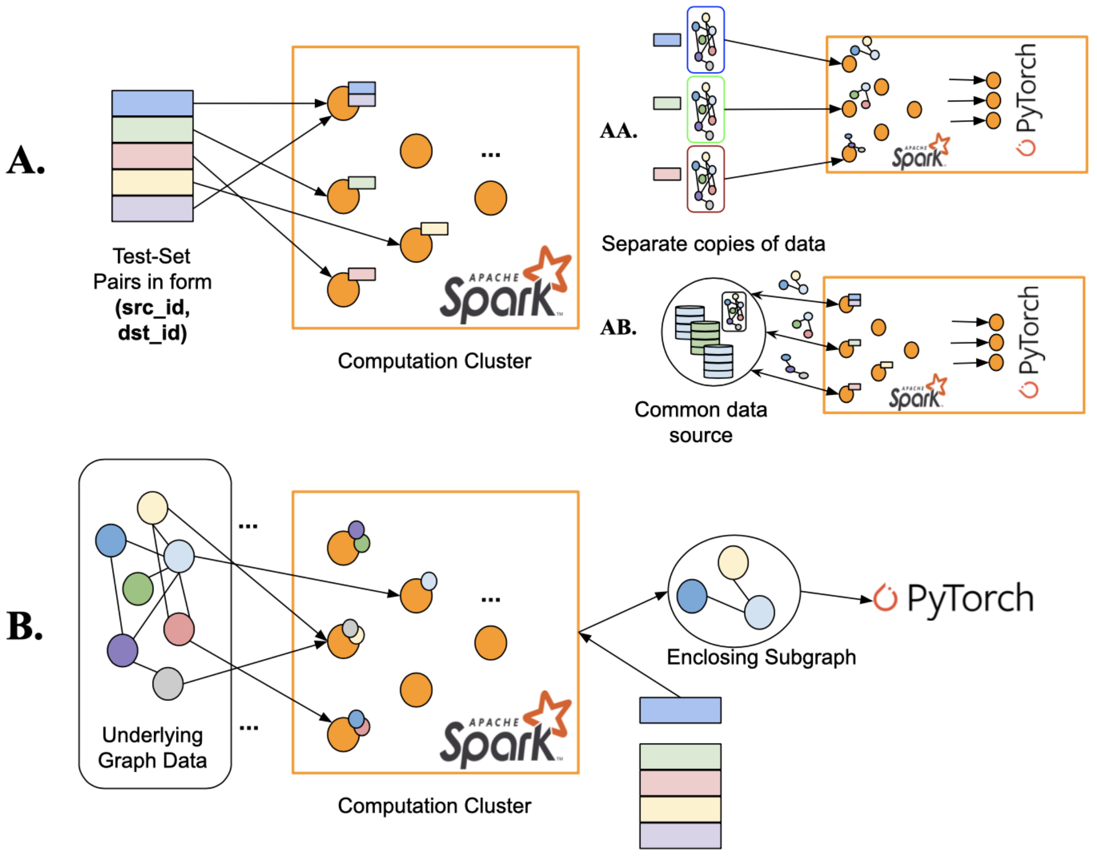

Figure 3.2. Proposed parallelization strategies

Let us discuss both approaches presented in figure 3.2 in detail, starting with strategy A.

- The first option **(A)** is to consider the test-set of pairs as separable and independent data. In other words, we parallelize the test set, creating an RDD based on it. Using this RDD, pairs from the test-set (blocks of different colors on the diagram) are distributed among executors (orange circles on the diagram) in the cluster (the orange box on the diagram) that proceed with SEAL logic themselves. In this case, the link prediction process of SEAL is represented as a set of independent computations on target pairs. In the following work, this approach is called **parallelization option A**. Further options are either sharing the graph data among all the executors with the logic required to extract enclosing subgraphs (sub-option AA on the diagram) or managing to create a shared data source (e.g., using a database) that performs the enclosing subgraph extraction itself (sub-option AB on the diagram). Either way, we get an RDD consisting of extracted enclosing subgraphs for pairs from the test-set that will be mapped to the prediction function of a corresponding pre-trained PyTorch model.

- The second option **(B)** is to make use of applying the graph processing libraries of Apache Spark – GraphX and Graphframes – in order to distribute the underlying graph data structure among executors in the cluster and to utilize tools provided by the framework to perform the subgraph extraction step. This approach is called **parallelization option B**. Executors in the cluster share the structure and complete the task of enclosing subgraph extraction on one pair altogether. After extraction, the enclosing subgraph is directly passed to the PyTorch model for prediction.

Initially another interesting approach is considered – to combine options **AA** and **B** in a common prediction stack. We would first create a distributed graph structure for the underlying data shared among nodes in the cluster (option B part). Then create an RDD from the test-data (option A part) and perform the extraction on parallelized test-set using the distributed graph. However, out of technical reasons, it is not possible to distribute operations on a distributed dataset – this problem is called RDD nesting in the domain, and the official version of Apache Spark does not support that type of operations yet. There are several workarounds and community-proposed improvements on the original Spark architecture to resolve this issue. Still, those are far too complex for our use case and lay out of the scope of this thesis.

3.2 Role of a Graph database

Parallelization strategy AB is based on the shared data source that performs the enclosing subgraph extraction step and provides executors in the Spark cluster with data required to perform prediction with a pre-trained GNN model. Under the data source for presenting this strategy was meant a graph database system from the very beginning.

In the last section, different graph processing approaches were discussed. A majority of existing graph database solutions also serve as graph processing platforms. Our idea is to use a graph query language of a database of our choice in combination with its integrated tools to extract enclosing subgraphs on request from our Spark cluster executors and return them back to Spark clusters in order to perform a prediction.

The Neo4j graph processing platform is one of the particular examples that includes graph schema storage, as well as transactional and analytical processing of graph data [29]. As already mentioned in the previous section, Neo4j provides a set of integrated tools for different use cases. The particularly useful attribute of Neo4j is that it is efficiently distributed (sharded or replicated) inside a cluster database for increasing this cluster performance through load balancing between entities. We use its graph query language Cypher combined with APOC library to perform the enclosing subgraph extraction on request from our Spark executors.

3.3 COST: Scalability price

The essence of our work is parallelization and, if possible, effective scaling of the SEAL method. For some time, there has been a debate in a scientific community whether graph processing problems should be scaled horizontally or vertically (or up/out). McSherry et al. [30] made an especially interesting impact with their proposed metric of performance measurement for big data processing systems called COST (”Configuration that outperforms a Single Thread”). The main advantage of COST is an effective way of evaluating a system’s scalability potential against the overhead the system itself introduces. The basic idea of COST is that a good-configured system that utilizes an optimized algorithm and efficient data structures can outperform existing horizontally scaled solutions. According to the authors, it is a method for *"measuring performance gains achieved through parallelization without rewarding systems that mask inefficiencies through it"*. They define a COST of a given platform for a given problem as *"the hardware configuration required before the platform outperforms a competent single-threaded implementation"*.

In the next section, we look at the COST of implementations of proposed parallelization strategies by comparing their performance with what McSherry et al. [30] call a competent single thread implementation – the original SEAL system executed on a high-end level laptop.

SECTION 4. METHODS

In the third section, possible parallelization strategies that can be applied to the SEAL method in order to increase its performance were described. In this part, we cover the implementation details of the discussed approaches.

We started our work by examining the functionality of the SEAL method. Zhang and Chen provided their implementation on a popular online git repository source GitHub in two variants - written in MATLAB and Python environments. Due to the fact that we were planning to integrate the original method in Apache Spark, we selected the version written in Python since Spark has an adaption for this language - PySpark. We made minor changes in the source code of the original method to evaluate its performance through time measurements. We copied its methods for enclosing subgraph extraction and node labeling for our parallelization strategy **AA**. Furthermore, the original method was applied to train and to save the PyTorch models and hyperparameters of GNN so as to utilize them both later in our Spark-based applications.

4.1 Deployment

The first point of consideration was the deployment of our Apache Spark cluster. We selected the popular Kubernetes option to deploy the Apache Spark jobs on university machines. Kubernetes cluster and infrastructure were provided for the experiments by the DOS Department of TU Berlin. As stated on the official web page dedicated to the Kubernetes deployment[1], Spark-User can use Spark-Submit operation to deploy the Spark application. We prepared an application environment in the form of a docker image that can be deployed into containers within Kubernetes pods. That docker image includes all the libraries required by our application *(like DGCNN)*, pre-trained model files and hyperparameters, test-data files, etc. Dockerfile associated with the image describes how to initiate the correct installation of required dependencies. Operation parameters

[1]`https://spark.apache.org/docs/latest/running-on-kubernetes.html`

include cluster configuration, application image address on an image repository - docker hub in our case - as well as different application parameters. Figure 4.1 depicts further actions performed in the cluster when the Spark-Submit operation is executed: Spark creates a Spark driver inside a Kubernetes pod. The driver initiates the creation of Spark executors, which are associated with additional Kubernetes pods, and executes provided application code. The scheduling of pods is handled by Kubernetes.

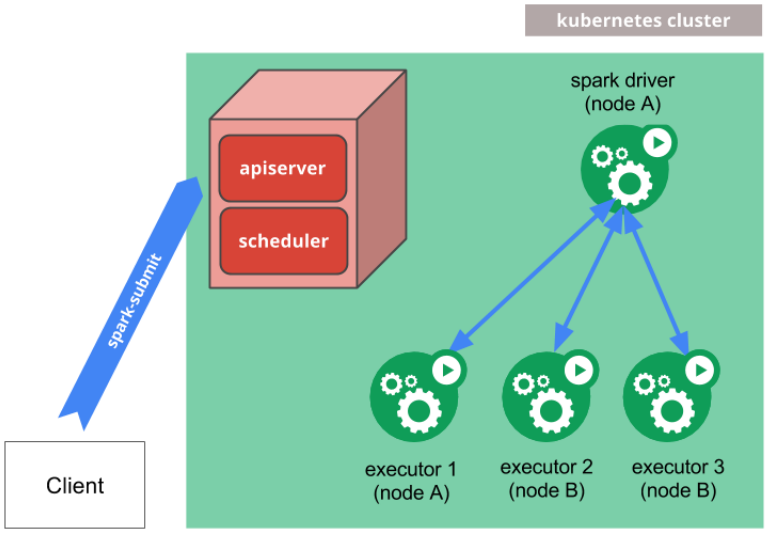

Figure 4.1. Spark kubernetes deployment, source: https://spark.apache.org/docs/latest/running-on-kubernetes.html

Proposed approaches were separated into three cases and three different action scripts, each available as an entry-point for the Spark Application inside the image. Let us discuss particular implementation details of previously discussed approaches.

4.2 Parallelization option AA

Parallelization option AA implies sharing of application data and logic among all the workers in the cluster. It is a parallelization of the original method on a distributed system. We parallelize the collection of the test pairs to an RDD. We copy the enclosing subgraph extraction function from the SEAL method and distribute this operation by passing the function to the map method of the previously created RDD. It forms a transformation on RDD that will result in the enclosing subgraphs after execution is triggered. The product of the mapping are subgraphs that are passed through a map method again to the distributed pre-trained DGCNN-function on all the executors to calculate the final prediction score.

4.3 Parallelization option AB

For option AB to succeed, an additional cluster of graph database nodes is required. Neo4j provides a quick way of cluster installation through a Kubernetes Helm manager. The cluster consisting of three Neo4j cores and additional three replicas are being installed. The Cypher language is applied to load required datasets to the database from file storage on GitHub.

After creating an RDD from the test pairs (exactly like in the option AA), each executor requests enclosing subgraphs from the database for its batch for pairs. Each executor performs a request through a fast-access protocol Bolt by sending a query for the subgraph extraction to the database. The query consists of two main parts: 1) collecting the nodes in the reachable distance of selected hop from both target nodes of the pair with APOC-library Subgraph Nodes method; 2) joining these nodes together with their interconnections using standard Cypher Match and Where statements. Since each Spark executor operates on a batch of pairs, query results are formed inside of an UNWIND Cypher block with an application of APOC RUN method to perform the query for each pair in separate.

Returned subgraphs are processed by each Spark executor, extended through the copied SEAL method for node labeling, and transformed to the enclosing subgraph structure consumable by a pre-trained DGCNN.

4.4 Parallelization option B

Graphframes library for Apache Spark-based graph parallelization and processing is being used since GraphX is not compatible with PySpark. At first, we load the dataset into a GraphFrame entity that creates dataframes of corresponding data and distributes them among the workers in the cluster. The test data is read on the driver and iteratively process pairs, distributing calculation steps for each pair.

Enclosing subgraph extraction is performed with the help of motif-finding methods written in DSL provided by the Graphframes engine. Our system is being supported by hop-levels 1 and 2 - every further level increases the complexity of the underlying motif-pattern. For example, motif for the hop-level 1 is "$(a) - [e] \rightarrow (b)$" with a filter for either *a* or *b* to be the target nodes, whereas motif for hop-level 2 consists of four further statements, including neighbors of neighbors in all the possible directions. However, it is not critical since Zhang and Chen [5] proved that these levels are sufficient in most cases. With the motif-finding method, we extract the nodes of the enclosing subgraphs and then filter edges inside of the Graphframe to contain only the edges that include extracted nodes. Analog to the method AB, the node labeling method of the original SEAL is applied on extracted structures, and the results are brought back to the iteration cycle. Prediction step - or application of the GNN - occurs when the number of iteratively processed pairs sums up to the batch-level, set in application parameters.

4.5 Subgraph extraction procedure drawback

Subgraph extraction is one of the standard operations on a graph. It is pre-implemented as a procedure or a method in many graph processing frameworks, including Neo4j APOC-library that we apply in our setup. Our intention was to use these pre-implemented options. The problem faced while integrating these methods and approaches in our system was the unification of subgraphs of both target nodes. Those should include all the reachable nodes and their interconnections, as mentioned back in the second section. We address the example illustrated in the Figure 4.2 below for the clarification of the difference: *if node **X** in the surrounding has a connection with the target node **A** and node **Y** has a connection with another target node **B**, a connection between **X** and **Y** should be included into the resulting enclosing subgraph.* That is the reason, why implementations of strategies **AB** and **B** include additional steps.

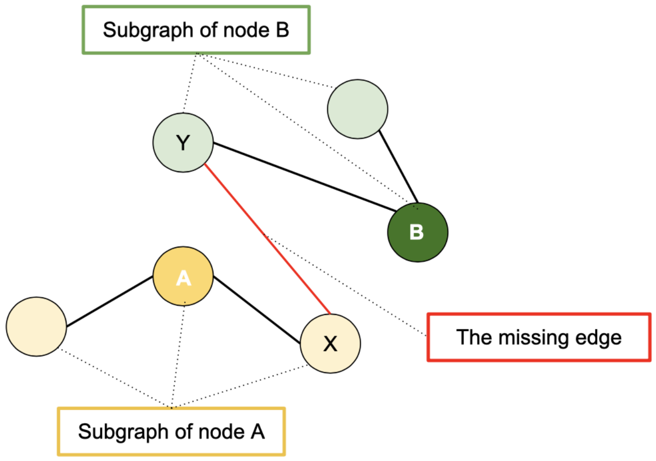

Figure 4.2. Illustration of a usual subgraph extraction drawback for the SEAL method.

4.6 Project information

All the project code and other practice-relevant information are available in the GitHub repository[2] associated with the thesis. We cover the project structure of the thesis code in Appendix B of this thesis.

[2]`https://github.com/kostjaigin/bachelor`

SECTION 5. EVALUATION

We conducted a series of experiments to evaluate the performance of all three parallelization options against an adequate single-thread implementation - original SEAL framework executed on a high-end level laptop configuration.

5.1 Data

We selected five datasets presented in Table 5.1 with the growing number of nodes and edges to evaluate system performance in different setups. The biggest of selected datasets - Arxiv - might seem relatively small to be processed within a big data processing system, but this decision was made on purpose - we have previously shown *(see Figure 3.1)* that the computationally heaviest task of enclosing subgraph extraction might take big amounts of time when executed with the original system on a single machine. It was essential for us to operate on data that can be processed by a single machine in order to estimate the COST metric of our parallelization methods.

USAir [31] is the network data of US Airlines, PB [32] is a network of US political blogs, Yeast [33] is an example of a protein-protein interaction network, Facebook is an anonymized network data from the corresponding social network and Arxiv is a dataset of scholar citations in papers and scientific articles [34].

Dataset	**Nodes**	**Edges**	**Avg. node degree**
USAir	332	2.126	12,81
PB	1.222	16.714	27,36
Yeast	2.375	11.693	9,85
Facebook	4.039	88.234	43,69
Arxiv	18.772	198.110	21,11

Table 5.1 Experiment datasets

5.2 System configurations

The aim is to compare the performance of a solution distributed on a cluster with a competent single machine implementation. We conducted our experiments with the original system on a high-end level laptop with 16 GB of 3733 MHz LPDDR4X RAM, 512 GB of SSD memory, 2 GHz Quad-Core Intel Core i5 CPU. In order to estimate the COST [30] of our solution, we selected different cluster sizes of our Apache Spark cluster with 4, 8, and 12 executors and a master machine. Each executor was set up to have a limit of 4 GB RAM, 4 CPU cores, and a default memory overhead factor of 0.1. Master pods had 6 GB RAM and 4 CPU cores in access.

For the parallelization strategy AB, we included six graph database pods, three of which are cores and the other three - replicas. We set a similar Requests-Limits policy to the graph database entities - each pod request was set to 2 GB of RAM with one core guaranteed to back up the process and limits of 4 GB of RAM with four cores.

Table 5.2 contains an overview of all the machine/cluster configurations that were applied in our experiments and assigns each configuration a unique identifier for later use. The first two rows represent the single-machine and database cluster configuration correspondingly. Further rows depict the Spark Cluster with the corresponding amount of executors (4, 8, or 12). The asterisk sign at Spark configurations indicates a particular parallelization strategy, e.g., configuration of the strategy AA with 4 Spark executors will be annotated as AA/4.

Id	**Configuration**	**CPU [cores]**	**RAM**
SEAL	Laptop	8	16GB
DB	Database	24	4*6GB
*/4	4 Spark executors	20	4*4+6GB
*/8	8 Spark executors	40	8*4+6GB
*/12	12 Spark executors	60	12*5+6GB

Table 5.2 Machine/Cluster configuration overview. The asterisk symbol is a placeholder for a particular parallelization strategy: * $\in \{AA, AB, B\}$

5.3 Experiments

Using the SEAL configuration, we acquired test-data for each dataset in sizes from range {10, 100, 500, 1000, 5000, 10000, 25000, 50000} pairs. Experiments were executed on both hop-levels 1 and 2 for each dataset, but not for all the parallelization strategies. We conducted performance measurements for all three parallelization options on cluster configurations from Table 5.2. Experiments with more than 1000 test links were conducted only on a cluster and only for the parallelization strategy AA. The intuition behind this decision is explained further on. Table 5.3 contains detailed information about experiments we conducted, including applied system configuration Ids from Table 5.2, used dataset from the Table 5.1, hop-level $h \in \{1, 2\}$ and test-sizes range, which is either 10, 100, 500 and 1000 links *([10:1000] in the table)* or 5000, 10000, 25000 and 50000 links *([5000:50000] in the table)*. We conducted 336 experiments in total.

Datasets	Hop	Test-sizes	Applied Configurations
USAir, PB, yeast	{1,2}	[10:1000]	{SEAL,AA/4,AA/8,AA/12}
USAir, PB, yeast	1	[10:1000]	{AB/4,AB/8,AB/12}
USAir, PB, yeast	1	[10:1000]	{B/4,B/8,B/12}
facebook, arxiv	{1,2}	[10:1000]	{SEAL,AA/4,AA/8,AA/12}
facebook, arxiv	1	[10:1000]	{AB/4,AB/8,AB/12}
facebook, arxiv	1	[10:1000]	{B/4,B/8,B/12}
facebook, arxiv	{1,2}	[5000:50000]	{AA/4,AA/8,AA/12}
facebook, arxiv	1	[5000:50000]	{SEAL}

Table 5.3 Conducted experiments. Application of the configuration AB implies usage of DB configuration as well.

5.4 Results

We present the most significant and lucid results that help to comprehend the potential of all the three proposed parallelization strategies. Additional charts can be found in Appendix A of this thesis. Raw data is available in different formats in the associated GitHub repository[1].

5.4.1 Parallelization strategies AB and B

Both parallelization strategies AB and B have shown poor performance compared to the original system. Due to the limited access to the computational cluster of the university, we saved some time for additional experiments with strategy AA by conducting experiments with the strategies AB and B only on hop-level 1.

Considering strategy B, the conclusion was made that this method scales positively. Figure 5.1 illustrates method B gaining performance with the increasing number of Apache Spark executors *(and as such additional cores and memory).*

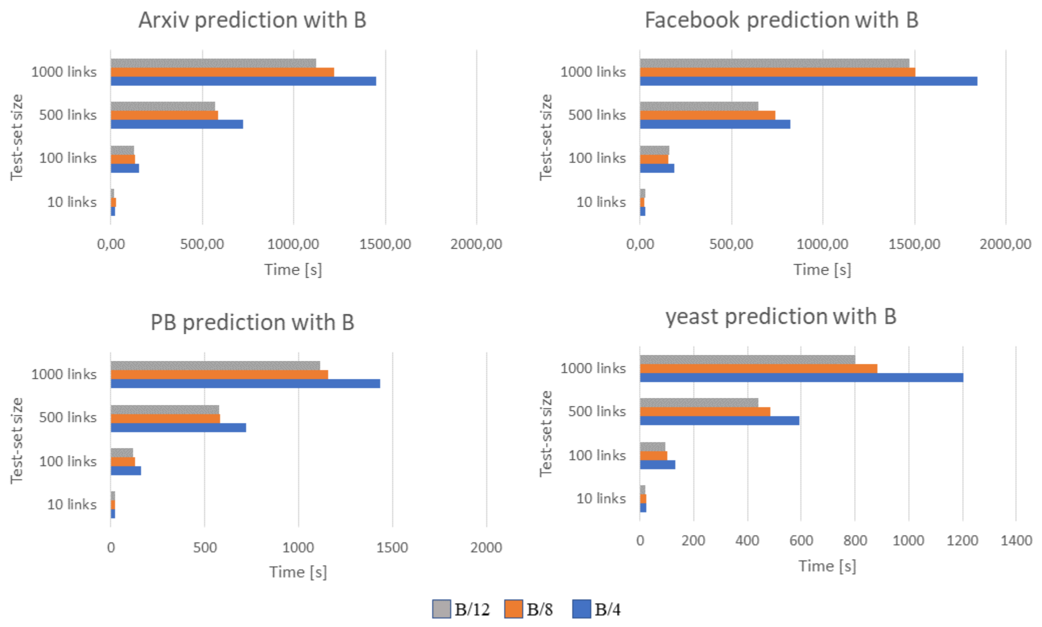

Figure 5.1. Prediction performance measurements with the parallelization strategy B on four different datasets.

It was stated that the increasing number of Apache Spark executors does not have any significant effect on the performance of the strategy AB (address Appendix A for the corresponding data

[1]GitHub Repository: `https://github.com/kostjaigin/bachelor/tree/master/results`

representation). However, that was not expected since the most computationally extensive task is being calculated on the graph database nodes and not on Apache Spark executors.

We were not able to estimate the COST of configurations AB and B since neither of them has outperformed the SEAL configuration. Figure 5.2 illustrates the vast difference in AB, B, and SEAL prediction time on different datasets with a hop level of one. For strategy B, the version with the available maximum of 12 Spark Executors was selected as presumably the most efficient one.

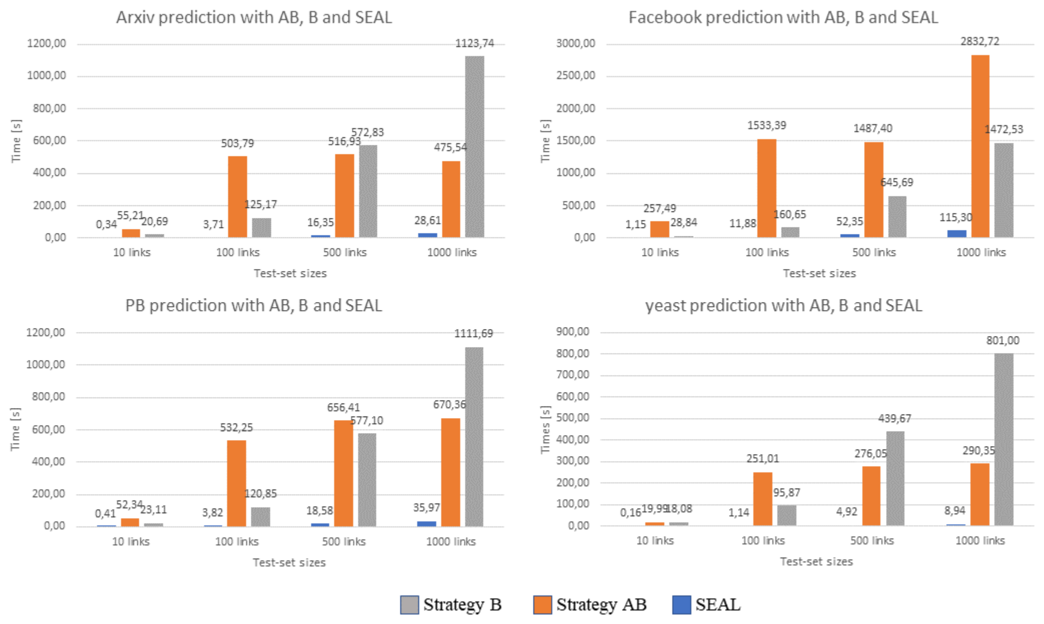

Figure 5.2. Prediction performance measurements with the parallelization strategies AB, B and single-thread implementation on four different datasets.

5.4.2 *Parallelization strategy AA*

Parallelization strategy AA was relatively easy in implementation and has outplayed the original system rapidly. It has proven to be highly scalable - an increasing number of machines mostly leads to faster predictions. This is demonstrated in Figure 5.3, where the results are being compared in four different setups on the largest available dataset - arxiv. The increase in performance with a growing number of executors is slightly visible even on small test sizes between 10 and 1000 links and becomes obvious on larger test sizes between 5000 and 50000 links. Similar results were achieved on other datasets as well, however, with less notable differences on smaller ones.

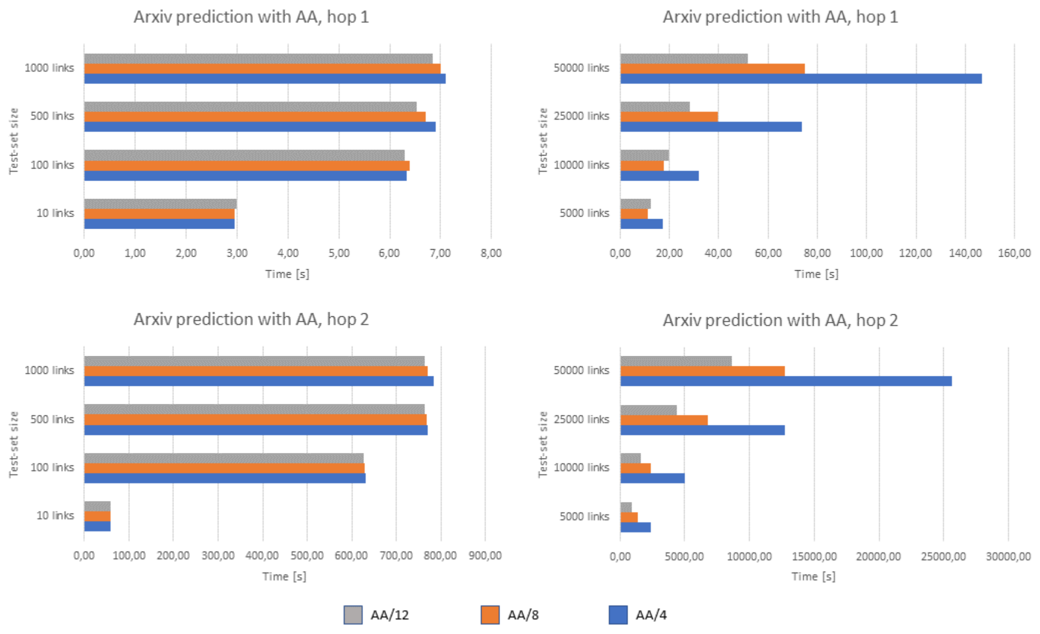

Figure 5.3. Arxiv dataset performance measurements with the parallelization strategy AA implementation on different test-set sizes with hop-levels of 1 and 2.

The minimal configuration of the solution AA that we implemented consisted of 4 Spark Executors, which was sufficient to outperform the SEAL system laptop-configuration in a majority of cases. This indicates that the COST of our implementation of strategy AA is at least less or equal to the configuration with 4 Spark executors. Configurations AA/8 and AA/12 were superior to the SEAL configuration as well. Figure 5.4 illustrates the performance of minimal configuration of AA compared with the SEAL configuration on relatively small datasets USAir and yeast with selected hop level of 2. Two approaches show similar performance at smaller test sizes of 10 and 100 links, and the difference in performance becomes evident at 500 links.

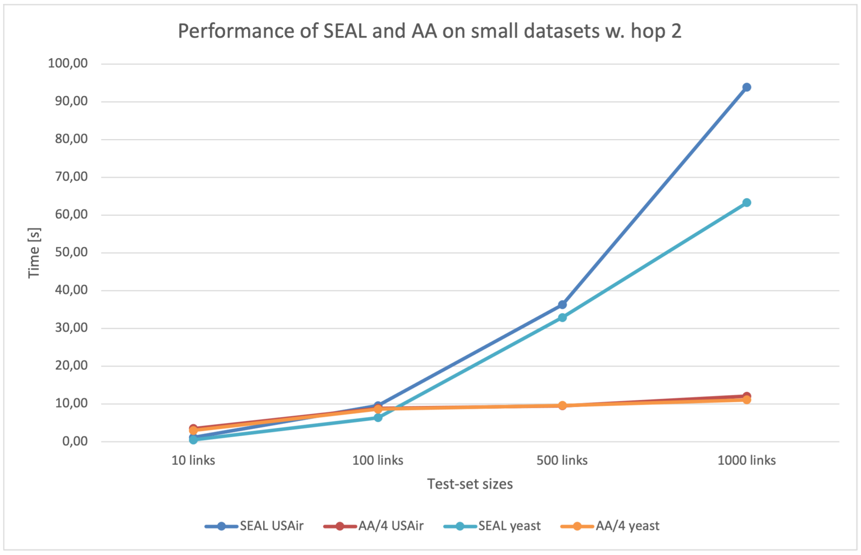

Figure 5.4. Prediction performance measurements with the parallelization strategy AA implementation on smaller datasets with different test-set sizes and hop 2.

The difference in performance becomes even more visible with bigger datasets, such as facebook and arxiv, as demonstrated in Figure 5.5.

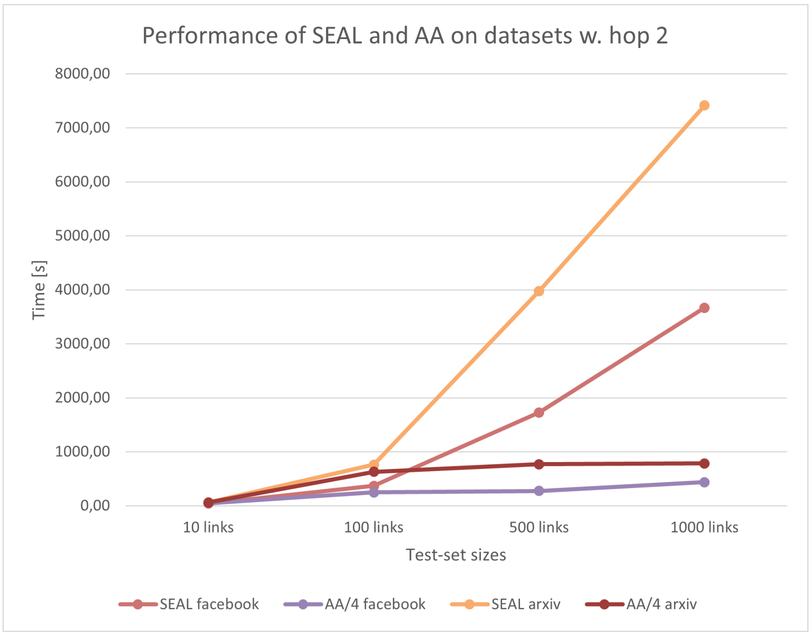

Figure 5.5. Prediction performance measurements with the parallelization strategy AA implementation with bigger datasets on different test-set sizes and hop 2.

Figure 5.6 composes the performance measurements on all three smaller datasets (USAir, yeast, PB). It is particularly interesting that the SEAL configuration firmly outperforms all the AA configurations on hop one on test sizes of 10 and 100 links. On such a small setup, AA configuration performance is drawn back by parallelization costs.

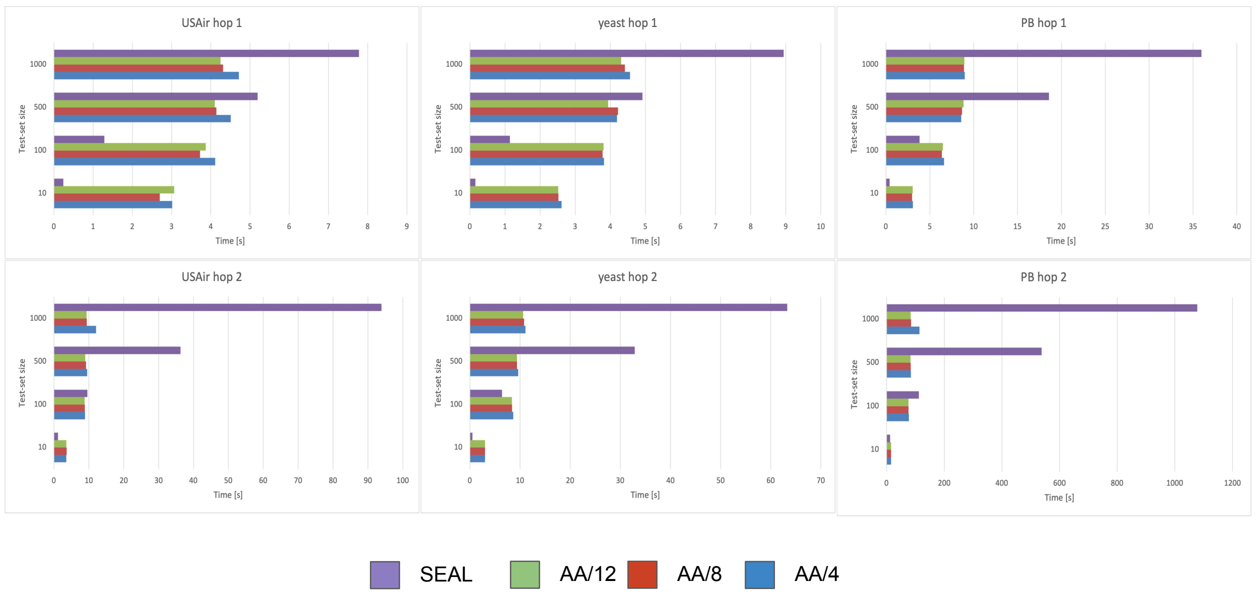

Figure 5.6. Prediction performance measurements with configurations SEAL, AA/4, AA/8 and AA/12 on smaller datasets

We were also interested in comparing the original system with the maximum-executors configuration of the strategy AA - 12 executors. Figures 5.7, 5.8, 5.9 illustrate the superiority of implementation of strategy AA with 12 executors over the original system on a laptop setup.

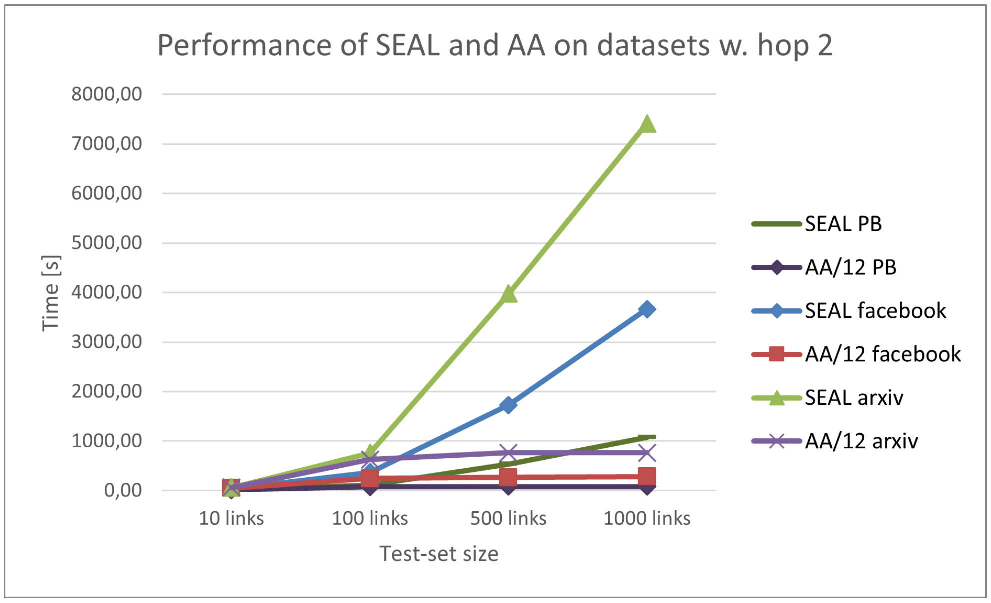

Figure 5.7. Prediction performance measurements with the parallelization strategy AA implementation with different datasets and on different test-set sizes, hop 2.

Figure 5.7 contains a comparison of three relatively complex datasets with the biggest numbers on connections: PB, facebook, and arxiv. Even on a small test range between 10 and 1000 links, our implementation of the strategy AA manages to outperform the original system immensely.

Increasing number of test-set for prediction on the SEAL configuration to tens thousands of links leads to unbearable amounts of the required time. For Figure 5.8, we calculated the mean time required to perform a single prediction from experiments on 1000 links. SEAL-data in Figure 5.8 represents the estimated time, while AA/12 data is based on conducted experiments. The strategy AA configuration with 12 Executors outperforms the SEAL configuration on a large scale.

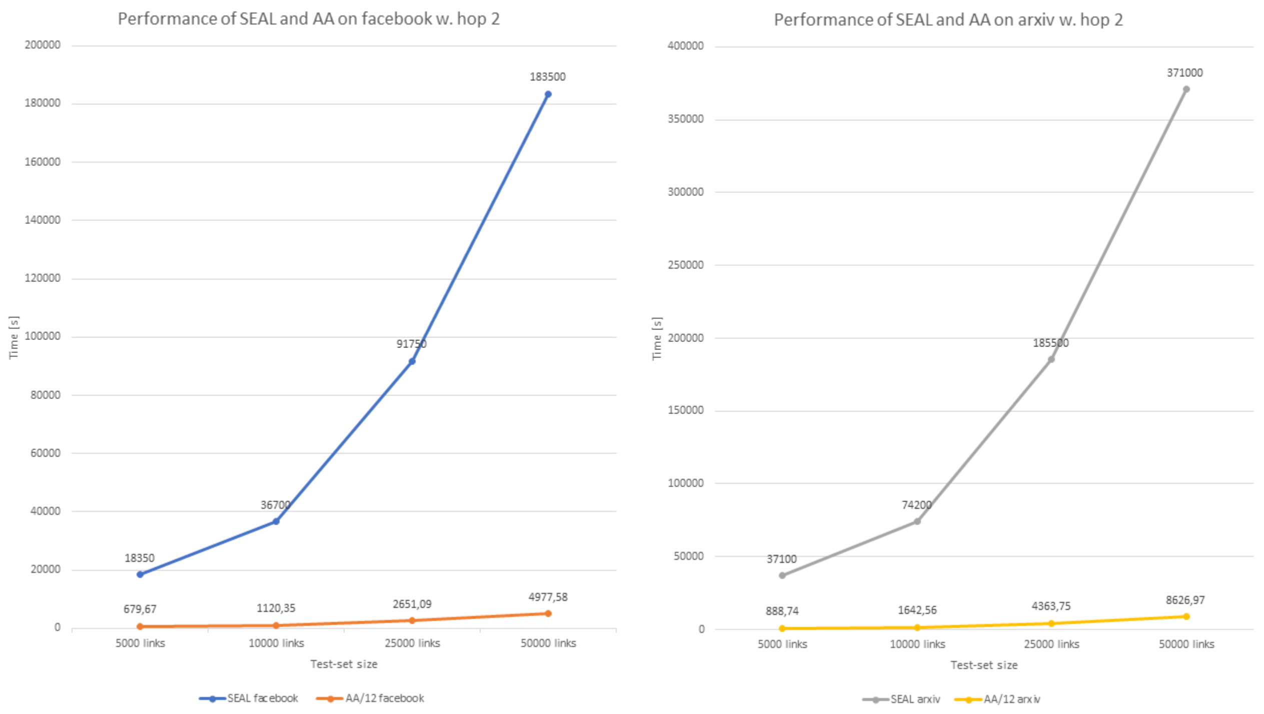

Figure 5.8. Facebook and Arxiv prediction performance measurements with the parallelization strategy AA and the SEAL framework on different test-set sizes with hop of 2.

The result presented in Figure 5.8 would not have been much different if we compared it with other AA configurations. However, the system scales even better in this case, as illustrated in Figure 5.9.

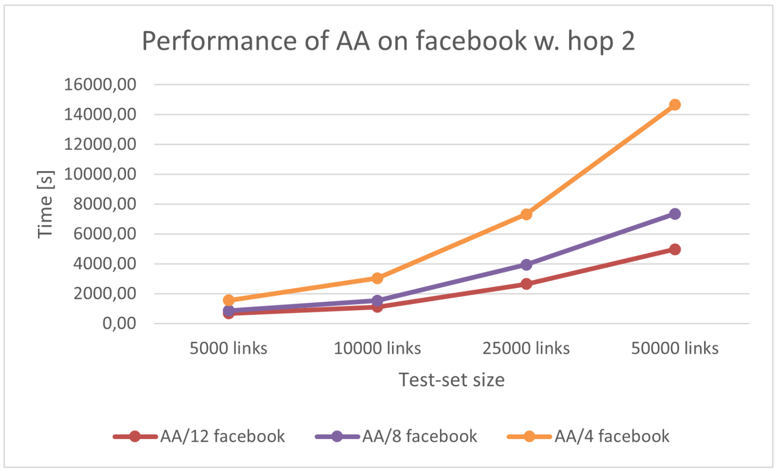

Figure 5.9. Facebook prediction performance measurements with the parallelization strategy AA configurations on different test-set sizes with hop of 2.

5.5 Discussion

We can state that the proposed parallelization strategy AB failed to prove the second hypothesis of our thesis. From our knowledge, there might be several reasons for that.

- The original system performs the most complicated step of the prediction - *the subgraph extraction* - using fast matrix transformations *(as stated in the second section).* Our implementation of the parallelization strategy AB is based on the built-in APOC-library method for subgraph extraction that performs the graph traversal operations, which are significantly slower than matrix transformations.

- Inefficient cluster configuration might have also played a role in the poor performance of our implementation. It might be related to the pod configuration on the Kubernetes cluster, e.g., not enough resources for given calculations as well as to the interconnection of the Apache Spark cluster with the database inside of the cluster via the bolt-protocol.

- Last possible explanation for the failure of our implementation might be the lack of experience in querying, using cypher-query language of the Neo4j. Graph databases are a complex topic for themselves. One should not exclude the possibility of own incompetence.

The parallelization strategy B was an interesting experiment. From one side, our implementation of the strategy failed to provide a confirmation to the first hypothesis. Considering the fact that it is based on the GraphFrames engine, which itself is based on the GraphX Spark engine, we can correlate our results with the results of COST-authors McSherry et al. [30] - they have also shown the poor performance of the GraphX engine compared to the adequate single-thread implementations of considered problems. However, the system has shown potential for scaling: the increasing number of Spark Executors has made the application to calculate the predictions faster. Currently, we state that the parallelization strategy B is not competitive against the original system. But with the technological advance of graph processing frameworks, it might change over time.

The parallelization strategy AA scales efficiently, outperforms the original system in almost all the given configurations and even on smaller datasets and lower hop-level of 1. With the increasing complexity of underlying tasks, system performs the prediction the faster the more executors are involved. We state that our implementation of the proposed parallelization strategy AA is capable of confirming the first hypothesis of the thesis. The reason behind that is quite simple: our solution benefits from the fast matrix-transformation-based methods of subgraph extraction adapted from the original system while effectively distributing calculation tasks among the available executors.

SECTION 6. CONCLUSION

After evaluating the results of the proposed methods, there remains room for further discussion. At the start of the thesis, we stated two hypotheses: the first one persuades the application of big data processing tools in the considered method of LP, while the second one pushes the idea of applying graph databases included analytical and processing tools for the benefit of the considered method. Experiment data presented in the previous section has shown that at least one of our implementations of the parallelization strategies proposed in section 3 is capable of efficiently utilizing the Apache Spark controlled distributed resources to boost the performance of prediction. Apache Spark-based application AA was capable of outrunning the original system on its minimal tested configuration, which confirms the first hypothesis.

The second hypothesis, associated with graph databases, was not confirmed. However, that does not imply the failure of it. Due to technical reasons, our experiments included only one configuration of a graph database, consisting of three cores and three replicas with hard-set limits on memory and CPU. It would be particularly interesting to provide a database cluster configuration with more resources and to repeat the conducted experiments. Additionally, we should mention that we applied only one of many present graph database technologies - Neo4j. Meanwhile, there are plenty of other solutions and, as such, possible extensions to our work: the implementation of the strategy AB could be replaced by one of the concurrent technologies, like Dgraph[1], Amazon Neptune[2], etc. Another graph database-related issue was first mentioned in the last section - one could apply a different subgraph extraction query or optimize the existing one in order to speed up this computationally extensive task.

We did not manage to calculate the COST of our implementation of the parallelization strategy B. Despite its poor performance, the system showed signs of good scalability. In section 3 we

[1]`https://dgraph.io`
[2]`https://aws.amazon.com/neptune/`

mentioned the problem of RDD nesting - passing of RDD reference into transformation of another RDD is not supported by the official Apache Spark version. A possible bypass of the problem is to separate the system into two services. In this case, the structure of the application would be more similar to the one of the strategy AB, with one part of the system holding responsibility for the subgraph extraction and data serving, while the other one manages the process.

Application of Apache Spark on relatively small amounts of data, with the biggest dataset consisting of only several hundred thousand edges, was justified by the requirement to compare the performance of the system with an adequate single-machine implementation. Extraction time simulation in Figure 5.7 shows that subgraph extraction on tens of thousands of links with the SEAL configuration and illustrates the necessity of limitation of underlying data sizes. However, it would still be interesting to conduct experiments on production-level graphs with millions of nodes and edges. Such experiments would require another level of resources, though. Probably that would lead to immense complications in our most efficient strategy AA due to the creation of copies of the underlying data on each node for each step and give a completely different perspective for the strategies AB and B.

While our implementation of the strategies AB and B lack in performance, the solution for the AA could be directly applied to boost the training process of GNN models for the SEAL-based link prediction. Neural networks consume the same input parameters during the training phase as they do during the application of a trained model. This implies that the system requires enclosing subgraphs data for each training set. Our implementation provides a way to speed up their extraction. Enclosing subgraphs required for further process of training need to be collected in one place - *a master node inside of the Spark cluster* - in order to continue with the training phase. That step is more challenging the bigger the underlying data size is. Extracted data might be huge in some cases and unsuitable for collecting it on a master node alone. This creates an opportunity for involving the concept of distributed training of machine learning models: instead of performing the complex collect action on an RDD of enclosing subgraphs, it could be further used in a distributed form for the training. The topic of distributed training of machine learning models is actively studied. Particularly interesting impacts were made by Li et al. [35] and Niu et al. [36]. The theoretical foundations were published by Arnold [37] in the form of a tutorial paper on the issue. The topic is particularly interesting as a future extension to our work, e.g., with applying a training of DGCNN model with the recent SparkTorch[3] framework for a distributed training of Torch models on the Apache Spark basis.

Finally, many different frameworks apply Graph Neural Networks for link prediction. Our work can be extended without the end. The research area is vast, and the number of application scenarios is constantly growing. With the growing interest in graph data structures, it bares excellent research potential.

[3] `https://pypi.org/project/sparktorch/`

Appendices

APPENDIX A. ADDITIONAL REPRESENTATIONS.

Table A.1 contains experiment results data for the configurations AB/4, AB/8 and AB/12 *(for the configuration convention address the table 5.2)* and illustrates that increasing number of Apache Spark executors does not have any effect on performance of the implementation.

Dataset	**Executors**	**10 links [s]**	**100 links [s]**	**500 links [s]**	**1000 links [s]**
arxiv	4	55,21	503,79	516,93	475,54
arxiv	8	52,07	461,42	458,08	462,63
arxiv	12	51,39	462,42	469,03	438,11
facebook	4	257,49	1533,39	1487,4	2832,72
facebook	8	255,3	1512,44	1549,87	1616,68
facebook	12	255,41	1532,04	1525,66	1692,59
USAir	4	37,96	233,71	252,43	225,18
USAir	8	37,45	229,12	245,38	232,92
USAir	12	37,54	231,11	260,36	267,42

Table A.1 Experiment data for the configurations of the parallelization strategy AB for three datasets

Following figures A.1, A.2, A.3 contain a data representation of all the AA configurations compared with SEAL setup. Figure A.1 illustrates the performance measurements on configurations AA/4, AA/8, AA/12 and SEAL on arxiv dataset and figure A.2 demonstrates equivalent data for the facebook dataset. Low left sectors of each figure does not include the data for SEAL configuration, since as mentioned in the fifth section we did not conduct experiments on SEAL configuration with hop 2 on bigger test-sizes.

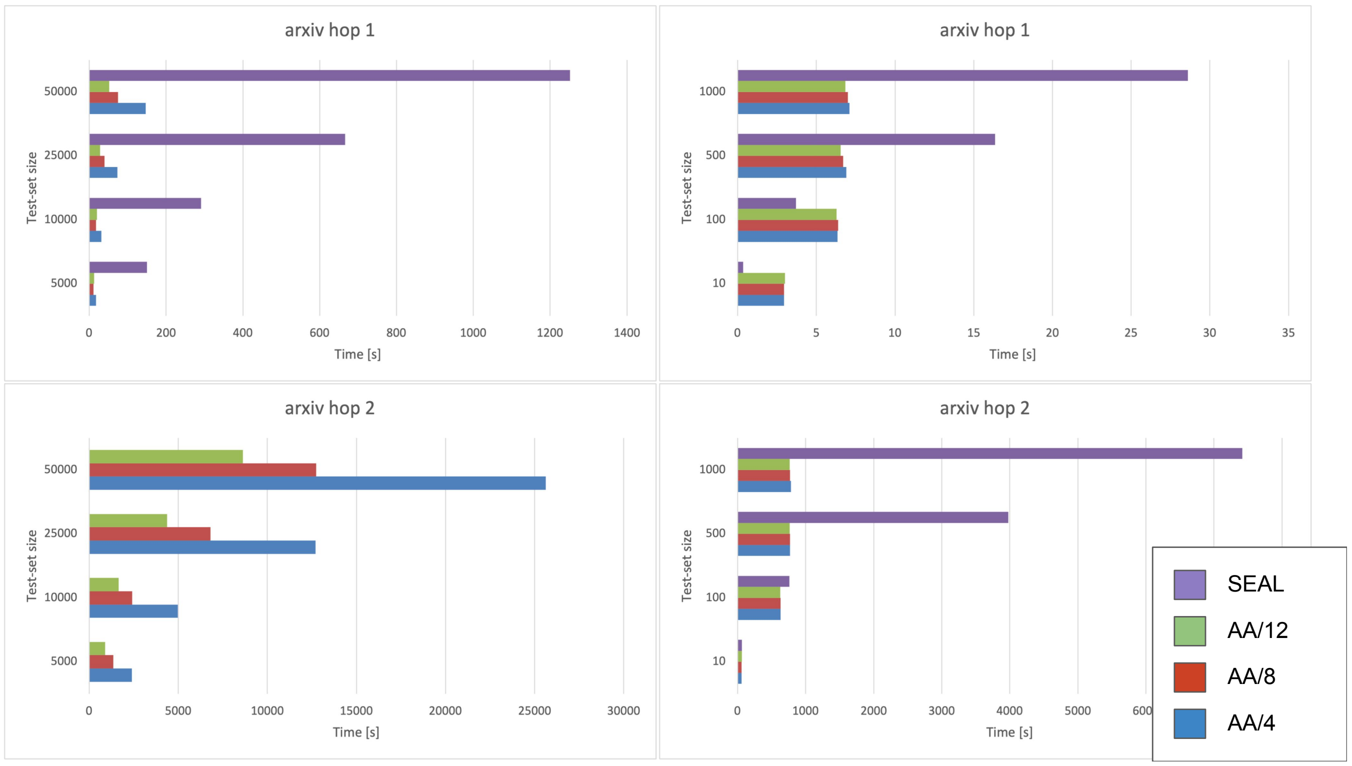

Figure A.1. Arxiv experiments data configurations AA and SEAL

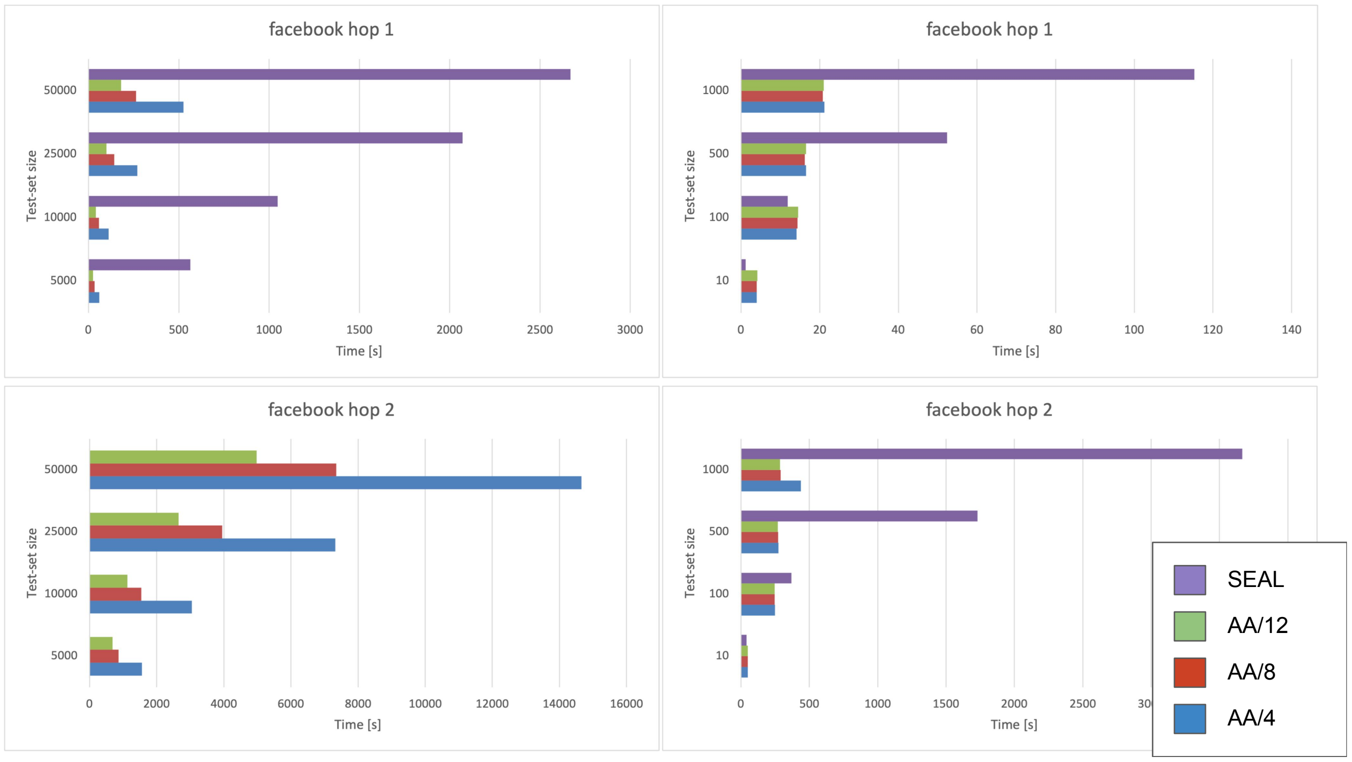

Figure A.2. Facebook experiments data configurations AA and SEAL

APPENDIX B. STRUCTURE OF ASSOCIATED REPOSITORY.

As mentioned earlier, all the project code and other practice-relevant information are available in the GitHub repository[1] associated with the thesis. The repository is built on top of a distribution of the latest version of Apache Spark - 3.0.1 *(during the composition of this thesis in April 2021).* The data folder inside of the repository contains all the Spark applications applied in our experiments:

- **App.py** contains the application logic for the parallelization strategy AA.
- **AppDB.py** contains the application logic for the parallelization strategy AB.
- **AppFrames.py** contains the application logic for the parallelization strategy B.

The ”dependencies” folder inside of the data folder contains the dependency files, prediction data for test sets is available under the same-named folder. The ”results” folder contains the results of our experiments in excel and .csv formats.

We conducted 336 experiments in total. The execution on the cluster was not performed manually. The ”experiments” shell-script available in the root folder and composes submission of required application with experiment parameters. For all three strategies, it executes a call on a corresponding Spark-Submit[2] Shell-Script:

- **”exe.sh”** performs Spark-Submit operation for the strategy AA.
- **”exeDB.sh”** performs Spark-Submit operation for the strategy AB.
- **”exeFrames.sh”** performs Spark-Submit operation for the strategy B.

”Experiments” script: 1) loads the required dataset into the database (*if required*), 2) starts the experiment with a given configuration, 3) removes the completed pods to keep the cluster clean, 4) saves the experiments results from the HDFS-storage.

[1]`https://github.com/kostjaigin/bachelor`
[2]`https://spark.apache.org/docs/latest/submitting-applications.html`

REFERENCES

[1] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, R. Navigli, A.-C. N. Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann, *Knowledge graphs*, 2021. arXiv: `2003.02320 [cs.AI]`.

[2] Y. Qi, Z. Bar-Joseph, and J. Klein-Seetharaman, "Evaluation of different biological data and computational classification methods for use in protein interaction prediction," *Proteins: Structure, Function, and Bioinformatics*, vol. 63, no. 3, pp. 490–500, 2006. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/prot.20865`.

[3] G. Berlusconi1, F. Calderoni1, N. Parolini, M. Verani, and C. Piccardi, "Link prediction in criminal networks: A tool for criminal intelligence analysis," *PLOS ONE*, vol. 11(4), 2016.

[4] M. Richardson and P. Domingos, "Mining knowledge-sharing sites for viral marketing," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '02, Edmonton, Alberta, Canada: Association for Computing Machinery, 2002, pp. 61–70.

[5] M. Zhang and Y. Chen, *Link prediction based on graph neural networks*, 2018. arXiv: `1802.09691 [cs.LG]`.

[6] D. Liben-Nowell and J. Kleinberg, "The link prediction problem for social networks," ser. CIKM '03, New Orleans, LA, USA: Association for Computing Machinery, 2003, pp. 556–559.

[7] N. Lao and W. W. Cohen, "Relational retrieval using a combination of path-constrained random walks," *Mach. Learn.*, vol. 81, no. 1, pp. 53–67, Oct. 2010.

[8] M. Hasan, V. Chaoji, S. Salem, and M. Zaki, "Link prediction using supervised learning," Jan. 2006.

[9] I. Kovacs, K. Luck, K. Spirohn, Y. Wang, C. Pollis, S. Schlabach, W. Bian, D.-K. Kim, N. Kishore, T. Hao, M. Calderwood, M. Vidal, and A.-L. Barabasi, "Network-based prediction of protein interactions," *Nature Communications*, vol. 10, Mar. 2019.

[10] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, *Graph neural networks: A review of methods and applications*, 2019. arXiv: `1812.08434 [cs.LG]`.

[11] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[12] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, *Graph attention networks*, 2018. arXiv: `1710.10903 [stat.ML]`.

[13] W. Gu, F. Gao, X. Lou, and J. Zhang, *Link prediction via graph attention network*, 2019. arXiv: `1910.04807 [cs.SI]`.

[14] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, and J. Taylor, "Industry-scale knowledge graphs: Lessons and challenges," *Commun. ACM*, vol. 62, no. 8, pp. 36–43, Jul. 2019.

[15] A. Fokoue, O. Hassanzadeh, M. Sadoghi, and P. Zhang, "Predicting Drug-Drug Interactions Through Similarity-Based Link Prediction Over Web Data," in *Proceedings of the 25th International Conference Companion on World Wide Web*, ser. WWW '16 Companion, Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, pp. 175–178.

[16] A. Mohan, R. Venkatesan, and K. Pramod, "A scalable method for link prediction in large real world networks," *Journal of Parallel and Distributed Computing*, vol. 109, pp. 89–101, 2017.

[17] J. Wang, Y. Ma, M. Liu, and W. Shen, "Link prediction based on community information and its parallelization," *IEEE Access*, vol. 7, pp. 62 633–62 645, 2019.

[18] S. Cohen and N. Cohen-Tzemach, "Implementing link-prediction for social networks in a database system," Jun. 2013, pp. 37–42.

[19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[20] T. N. Kipf and M. Welling, *Variational graph auto-encoders*, 2016. arXiv: `1611.07308 [stat.ML]`.

[21] M. Zhang, P. Li, Y. Xia, K. Wang, and L. Jin, *Revisiting graph neural networks for link prediction*, 2021. arXiv: `2010.16103 [cs.LG]`.

[22] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, *An end-to-end deep learning architecture for graph classification*, 2018.

[23] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[24] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[25] *Apache spark official documentation*, https://spark.apache.org/documentation.html, visited April 26, 2021.

[26] *Graphframes official documentation*, https://graphframes.github.io/graphframes/docs/_site/user-guide.html, visited April 26.04.2021.

[27] R. Angles and C. Gutierrez, “An introduction to graph data management,” *Graph Data Management*, pp. 1–32, 2018.

[28] *Neo4j official web-site*, neo4j.com, , visited April 26.04.2021.

[29] M. Needham and A. Hodler, *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O’Reilly Media, 2019.

[30] F. McSherry, M. Isard, and D. G. Murray, “Scalability! but at what cost?” In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland: USENIX Association, May 2015.

[31] V. Batagelj and A. Mrvar, http://vlado.fmf.uni-lj.si/pub/networks/data/, visited April 26.04.2021, 2006.

[32] R. Ackland, “Mapping the u.s. political blogosphere: Are conservative bloggers more prominent?,” Jan. 2005.

[33] C. von Mering, R. Krause, B. Snel, M. Cornell, S. Oliver, S. Fields, and P. Bork, “Comparative assessment of large-scale data sets of protein-protein interactions,” *Nature*, vol. 417, pp. 399–403, Jun. 2002.

[34] J. Leskovec and A. Krevl, *Stanford large network dataset collection*, https://snap.stanford.edu, visited April 26.04.2021, 2015.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, “Scaling distributed machine learning with the parameter server,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598.

[36] F. Niu, B. Recht, C. Re, and S. J. Wright, *Hogwild!: A lock-free approach to parallelizing stochastic gradient descent*, 2011. arXiv: 1106.5730 [math.OC].

[37] S. Arnold, *An introduction to distributed deep learning*, https://seba1511.com/dist_blog/, visited April 26.04.2021, 2016.