Niu S, Chen S. TransGPerf: Exploiting transfer learning for modeling distributed graph computation performance. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(4): 778–791 July 2021. DOI 10.1007/s11390-021-1356-2

TransGPerf: Exploiting Transfer Learning for Modeling Distributed Graph Computation Performance

Songjie Niu, *Student Member*, *CCF*, and Shimin Chen*, *Senior Member*, *IEEE*

State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China

University of Chinese Academy of Sciences, Beijing 100049, China

E-mail: {niusongjie, chensm}@ict.ac.cn

Received February 2, 2021; accepted July 10, 2021.

Abstract It is challenging to model the performance of distributed graph computation. Explicit formulation cannot easily capture the diversified factors and complex interactions in the system. Statistical learning methods require a large number of training samples to generate an accurate prediction model. However, it is time-consuming to run the required graph computation tests to obtain the training samples. In this paper, we propose TransGPerf, a transfer learning based solution that can exploit prior knowledge from a source scenario and utilize a manageable amount of training data for modeling the performance of a target graph computation scenario. Experimental results show that our proposed method is capable of generating accurate models for a wide range of graph computation tasks on PowerGraph and GraphX. It outperforms transfer learning methods proposed for other applications in the literature.

Keywords performance modeling, distributed graph computation, deep learning, transfer learning

1 Introduction

The graph data model has been widely used to analyze a wide range of real-world datasets, including the web graph, social networks, and semantic webs. Distributed In-memory Graph Processing (DIGP) has become a promising solution to graph data analysis due to its performance advantage and the rapid increase in memory capacity. A growing number of DIGP platforms have been proposed in recent years [1–4]. Modeling performance for DIGP can help execution time prediction, resource planning, performance analysis, and computation optimization.

However, to our knowledge, previous work has not investigated performance modeling for DIGP. Several recent studies have focused on performance evaluation and benchmarking for graph computation [5, 6]. Ngai *et al.* [7] proposed a performance analysis system for graph processing to facilitate the monitoring and measurement of the computation stages as specified by users. There have been a number of studies on configuration tuning and performance prediction for general-purpose big data computation platforms. Starfish [8] profiles and tunes the performance of MapReduce jobs. Ernest [9] employs non-negative least squares to model Spark performance. However, it assumes that the job and the dataset are fixed, and predicts the job's performance while varying the number of machines. This method cannot be directly applied to build a single DIGP model to support various data graphs and graph algorithms.

It is challenging to model the performance of DIGP. Explicit formulation often requires strong, simplifying assumptions. It is difficult to accurately capture the diversified factors (e.g., input graph size and characteristics, graph algorithm properties, system parameters, and machine configurations) and the complex interactions (e.g., non-linear behaviors in computation and communication because of concurrency and CPU cache effects) in the system. Statistical learning methods, such as machine learning and deep learning, re-

Regular Paper
Special Section on AI4DB and DB4AI
*Corresponding Author

quire a large number of samples to train an accurate model. The more the factors in the design space, the more complex the model, and the more the samples required. However, it is time- and resource-consuming to run the required graph computation tests to obtain a large number of training samples.

We would like to reduce the sample size for building a good DIGP performance model. We investigate transfer learning as a promising solution. Transfer learning aims to exploit knowledge from a source domain to build an accurate model for a target domain using a limited number of samples [10, 11]. Suppose we already have a good model for a source DIGP scenario (e.g., a combination of a graph algorithm and a graph platform). Then the basic idea of transfer learning is to combine this existing model with a limited number of training samples collected for a target DIGP scenario to generate a model for the target scenario. Most existing transfer learning methods are domain invariance methods. In their applications, e.g., image classification [12–15], the source and the target domain often have the same or similar classification classes, and the methods focus on the commonality between the domains. In contrast, different DIGP domains (e.g., graph algorithms or platforms) often have different characteristics. Accurate performance prediction in DIGP requires capturing domain discrepancy.

In this paper, we propose TransGPerf, a transfer learning solution for modeling distributed graph computation performance. TransGPerf combines the base model and a manageable amount of training data to model a given target DIGP scenario. The workflow of TransGPerf is illustrated in Fig.1. The main components are as follows.

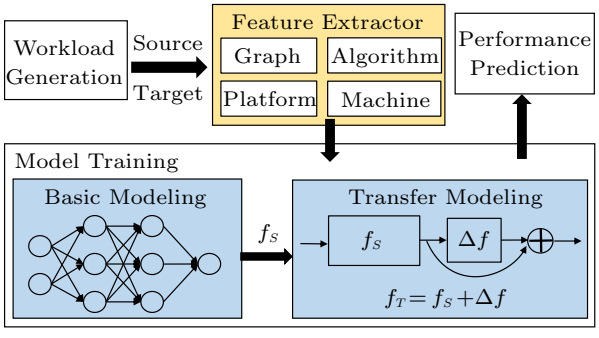

Fig.1. TransGPerf workflow.

• *Basic Modeling.* For a source DIGP scenario, we collect a large number of samples by considering combinations of factors that impact DIGP performance (e.g., graph datasets, graph algorithms, DIGP platforms, and machine hardware). Then, we build an MLP (multilayer perceptrons) model based on the training samples.

• *Transfer Modeling.* We propose a novel neural network based transfer learning model for DIGP. Inspired by ResNet [16], the transfer network structure adds residual layers after the source MLP model to capture the discrepancy of the predictive function of the target scenario compared with the source scenario.

• *Feature Extractor.* We propose a set of representative features that capture the graph, algorithm, DIGP platform, and machine characteristics. Some features are obtained directly by examining configurations or by invoking various tools. Others are derived with computation or approximated with black-box models.

Outline. Section 2 provides the background of our study. Section 3 presents the principles and the network structures of TransGPerf. Section 4 describes the feature selection. Section 5 shows the evaluation results. Section 6 discusses the significance of our study. Finally, Section 7 concludes this paper.

2 Background

In this section, we provide the background on DIGP and transfer learning.

2.1 Distributed In-Memory Graph Processing (DIGP)

In DIGP, there are typically a master and a number of worker machines connected through a data center network [1, 2] as depicted in Fig.2. The input graph is partitioned across workers. Graph computation consists of a series of supersteps. In every superstep, a worker performs computation on every vertex in its assigned graph partition and sends vertex-to-vertex messages. The computation follows the Bulk Synchronous Parallel (BSP) model or Asynchronous model. Fig.2 illustrates the BSP model.

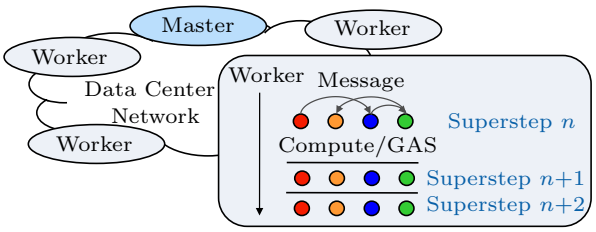

Fig.2. DIGP overview.

There are two main computation paradigms in DIGP: Pregel [1] and GAS [2]. In Pregel, graph computation is expressed as a `compute` function on every

vertex. In GAS, `compute` is replaced with three functions, i.e., `gather`, `apply`, and `scatter`, to reduce communication cost for high-degree vertices in power-law graphs.

Distributed graph computation scenarios are full of diversity. First, graph datasets range widely in their sizes in terms of the numbers of vertices and edges, and intrinsic properties, such as degree distributions and structure characteristics. Second, graph algorithms vary significantly in their computation loads and communication patterns. For example, PageRank essentially conducts random walks, while SSSP (single-source shortest path) performs BFS traversals on the graphs. The number of vertex-to-vertex messages stays roughly the same across supersteps in PageRank, while the number of messages changes drastically in SSSP. Third, the computation models and execution mechanisms can be quite different in different DIGP platforms. For example, Pregel-like systems and GAS-like systems differ in design and implementation strategies. Spark GraphX takes advantage of the Spark RDD framework for graph computation. Finally, machine configurations (e.g., CPU frequency and network bandwidth) have significant impact on the computation and communication timings of DIGP. The number of cross-partition edges is also affected by the number of workers in the system.

2.2 Problems of Traditional Modeling Methods for DIGP

There are generally two types of modeling methods: white-box and black-box. A white-box [17–19] model has explicit arithmetic formulations. It usually makes strong simplifying assumptions (e.g., linear combinations, independent activities). However, it is difficult to capture many diversified influential factors and their complex interactions in DIGP using white-box models. For example, the vertex computation and communication costs in DIGP cannot be captured simply with average values because there are a wide varieties of vertex degrees, computation logics, and communication patterns. The cost of different supersteps in a graph algorithm (e.g., SSSP) can vary significantly. To make matters worse, DIGP is a highly dynamic and parallel process with many concurrent and correlated activities.

A black-box [20–22] model is typically based on statistical machine learning methods. Given a task, a black-box method collects training samples and trains the prediction model. The number of training samples is correlated to the complexity of the model, which is determined by the design space of the target system. The more the influential factors in the target system, the more the training samples required to obtain an accurate model. As discussed in Subsection 2.1, there are many factors that impact the performance of DIGP. As a result, we need to collect a large number of samples for a black-box model. Note that every sample is a DIGP run. It is time- and resource-consuming to perform many sample runs. Moreover, machine learning methods assume that the training and the test data are drawn from the same feature distribution. When the distribution changes, the model needs to be rebuilt from scratch using newly-collected training samples. Therefore, we cannot reuse a model in different DIGP scenarios (e.g., different DIGP platforms).

2.3 Transfer Learning

Transfer learning [10] aims to support tasks on a target domain of interest when there are sufficient training data from a related but different source domain, but only a limited amount of training data in the target domain.

More precisely, a domain $\mathcal{D} = \{\mathcal{X}, P(\boldsymbol{x})\}$ consists of an m-dimensional feature space $\mathcal{X}$ and a marginal probability distribution $P(\boldsymbol{x})$, where $\boldsymbol{x} = \{x_1, ..., x_m\} \in \mathcal{X}$ [10]. A task $\mathcal{T} = \{\mathcal{Y}, f(\cdot)\}$ consists of a label space $\mathcal{Y}$ and a predictive function $f(\cdot)$ [10]. $f(\cdot)$ is not observed but can be learned from feature vector and label pairs $\{\boldsymbol{x}_i, y_i\}$, where $\boldsymbol{x}_i \in \mathcal{X}$, $y_i \in \mathcal{Y}$, and i is the index of the training samples. Then, $f(\cdot)$ is used to predict the label $f(\boldsymbol{x})$ of a new sample $\boldsymbol{x}$. From a probabilistic viewpoint, $f(\cdot)$ can be written as the conditional probability distribution $P(y|\boldsymbol{x})$. As illustrated in Fig.3(a), transfer learning aims to learn a predictive function for the target domain $\mathcal{D}_T$ by exploiting the knowledge from a related source domain $\mathcal{D}_S$.

In this paper, we find that for the base modeling of DIGP source domains, deep learning models perform better than traditional machine learning models on average (Section 5). Hence, previous work on transfer learning for traditional machine learning models (e.g., random forest [23]) is not applicable. We focus on deep transfer learning methods, where the base model is a deep neural network.

An example deep transfer learning case is as follows. In an image classification task, each image is embedded into a pixel vector $\boldsymbol{x}$. $\mathcal{X}$ is the feature space of all image pixel vectors. $\mathcal{Y}$ is the set of all class labels.

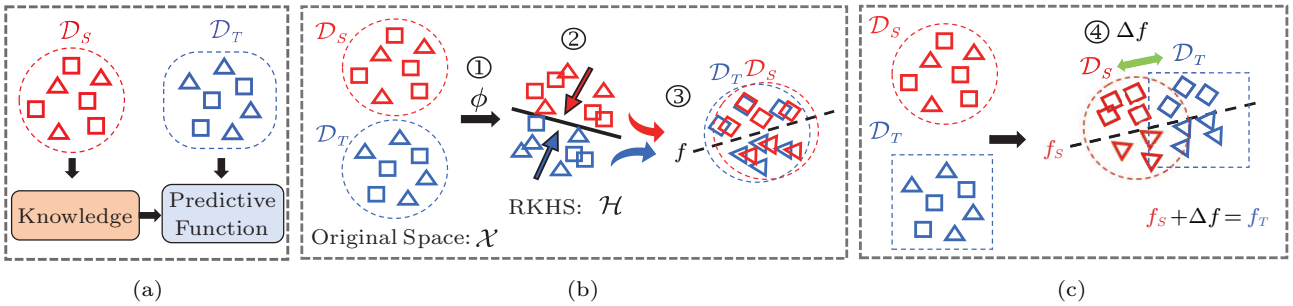

Fig.3. Deep transfer learning. (a) Concept. (b) Existing domain invariance methods. (c) Our proposal for DIGP. Existing domain invariance methods ① map source and target samples into a common subspace, ② minimize domain distance, and ③ compute a single shared predictive function. Our proposed method ④ captures domain discrepancy.

Suppose there are a large number of labeled images in a source repository (e.g., ImageNet). The images in a target repository have different sizes and exposure from the source images. The target images are unlabeled or only a small number of target images have labels. The goal is to classify images in the target repository. Here, $\mathcal{X}_S = \mathcal{X}_T$, $\mathcal{Y}_S = \mathcal{Y}_T$, but $P(\boldsymbol{x}_S) \neq P(\boldsymbol{x}_T)$.

Fig.3(b) illustrates the idea of most existing transfer learning methods. We call these methods domain invariance methods because they mainly exploit the commonality between the source and the target domain. Feature vectors of image samples in $\mathcal{X}_S$ and $\mathcal{X}_T$ are mapped to a common space $\mathcal{H}$, e.g., RKHS (Reproducing Kernel Hilbert Space), by $\phi : \mathcal{X} \to \mathcal{H}$. Then, the methods minimize certain distance metrics between the source and target domains, e.g., MMD [24], by making the marginal distributions of the two domains as similar as possible. The usual solution is to convert the problem of finding the non-linear transformation ϕ explicitly as a kernel learning problem. Finally, the methods assume that the source and the target domain share the same classifier. They cope with the limited target training samples by exploiting both the source samples and the target samples to learn a single shared classifier.

In domain invariance methods, the simplest deep transfer learning method, fine-tuning, copies lower layers of a pre-trained network and adapts them to new tasks. DDC [12] is a deep network that applies a single linear kernel to one layer to minimize MMD. DAN [13] minimizes MMD with multiple kernels (RBF kernel) applied to multiple layers. Unlike these methods that match the marginal distributions across domains, JAN [14] aligns joint distributions of multiple domain-specific layers across domains based on a Joint Maximum Mean Discrepancy (JMMD) criterion. CORAL [25] aligns the second-order statistics of the source and target distributions. Dcoral [15] minimizes the difference between source and target correlations with the CORAL loss. Several methods [26] adopt the adversarial domain classifier to learn the common components.

RTN [27] combines MMD with considerations of domain discrepancy. The images in the source domain are labeled. The images in the target domain are all unlabeled. The source and target domains have the same image classes (e.g., computer monitors, chairs, and other objects in office settings). RTN aims to classify the unlabled target images using knowledge from the labeled source images. As shown in Fig.4, RTN first employs the domain invariance transfer learning method. It feeds both the source and the target samples into a network to minimize MMD between the two domains. For the target domain, a loss function is applied to minimize the entropy so that for every target image, one class has a much higher probability than the other classes. In contrast, for the source domain, RTN adds two fully-connected (fc) layers to reflect the discrepancy between the two domains.

DIGP is significantly different from applications in previous transfer learning work. In DIGP, domain dis-

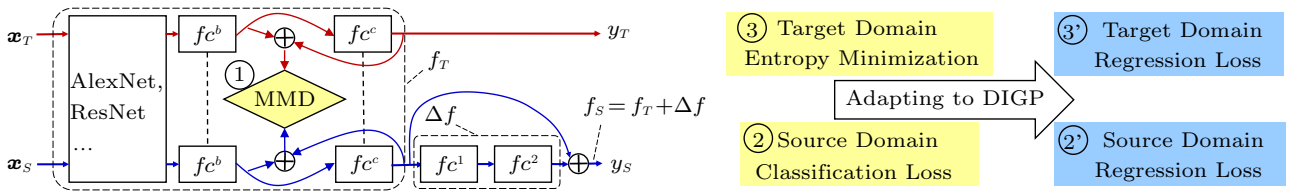

Fig.4. Adapting RTN to DIGP performance prediction.

crepancy plays a significant role, which makes domain invariance methods less effective. Moreover, the setting of a limited number of labeled target samples is different from that of RTN. As a result, previous methods applied to DIGP lead to a poor accuracy (cf. Section 5). We need a new method for DIGP modeling.

3 TransGPerf Design

Each DIGP computation is described by a feature vector representing the input graph, the algorithm, the DIGP platform and hardware configurations (cf. Section 4). We pay a one-time cost for collecting sufficient n_S source samples to learn a good predictive function $f_S(x)$ for the source DIGP domain. We would like to exploit transfer learning to reduce the number of samples for DIGP target domains.

Problem 1 (*Modeling DIGP Performance with Transfer Learning*). There are a large number of labeled data in a source DIGP domain $\mathcal{D}_S = \{(\boldsymbol{x}_{S_1}, y_{S_1}), ..., (\boldsymbol{x}_{S_{n_S}}, y_{S_{n_S}})\}$ but only a limited number of labeled data in a target DIGP domain $\mathcal{D}_T = \{(\boldsymbol{x}_{T_1}, y_{T_1}), ..., (\boldsymbol{x}_{T_{n_T}}, y_{T_{n_T}})\}$, where $0 < n_T \ll n_S$. $\mathcal{X}$ is the feature space of DIGP configurations. $\mathcal{Y}$ is the space of run times. $\mathcal{X}_S = \mathcal{X}_T$, $\mathcal{Y}_S = \mathcal{Y}_T$, but $P(\boldsymbol{x}_S) \neq P(\boldsymbol{x}_T)$, $P(y_S|\boldsymbol{x}_S) \neq P(y_T|\boldsymbol{x}_T)$. The goal is to learn a predictive function $f_T(\boldsymbol{x}_T)$ for the target DIGP domain.

In this section, we first analyze the limitation of previous transfer learning methods in Subsection 3.1. We then present the TransGPerf idea and learning network design in Subsection 3.2 and Subsection 3.3, respectively.

3.1 Limitation of Existing Transfer Learning Methods

Domain Invariance Methods. As discussed in Subsection 2.1, DIGP scenarios have a wide range of diversity. However, previous domain invariance methods are all based on the assumption that the source and target samples can be converted to have essentially similar distributions in a common space. Moreover, problem 1 is a regression problem, while applications in previous work are mostly classification tasks. It is relatively easier to share the same predictive function for distinct classes.

RTN. As discussed in Subsection 2.3, RTN combines a domain invariance method with a domain discrepancy design. Specifically, the common network in Fig.4 is f_T. The two extra fc layers are Δf. Then $f_S = f_T + \Delta f$. The loss function in RTN consists of three components, as marked ①–③ in Fig.4. We apply RTN to DIGP by modifiying loss component ② and ③. However, the resulting accuracy is poor (cf. Section 5). This is because that f_T (i.e., the common network) is trained using both the source and target samples, but there are much more source samples than target samples. Therefore, f_T is biased toward the source domain rather than the target DIGP domain.

3.2 Our Solution: TranGPerf

Inspired by residual learning in ResNet [16], we capture the domain discrepancy by adding a residual $\Delta f(\cdot)$ to $f_S(x)$ to model $f_T(x)$:

$$f_T(x) = \Delta f(\cdot) + f_S(x). \tag{1}$$

More precisely, ResNet [16] introduces a technique called skip connection, which directly connects the input of a neural network to the output while skipping the middle layers of the network. That is, $\mathcal{H}(x') = \mathcal{F}(x') + x'$, where x' is the input, residual $\mathcal{F}(\cdot)$ represents the middle layers, and $\mathcal{H}(\cdot)$ is the output. It is shown that the residual mapping $\mathcal{F}(x')$ can be better learned compared with directly learning $\mathcal{H}(x')$. In our case, we consider $f_S(x)$ as input x' for predicting the performance of the target DIGP. $\Delta f(\cdot)$ is residual $\mathcal{F}(\cdot)$. Output $f_T(x)$ is $\mathcal{H}(\cdot)$ in ResNet.

To reduce losing the information of target data features, we concatenate $f_S(x)$ and x as the input to $\Delta f(\cdot)$. The full model is shown below, and also illustrated in Fig.5(a):

$$f_T(x) = \Delta f(f_S(x), x) + f_S(x), \tag{2}$$

where $f_S(\cdot)$ is the source model trained in basic modeling. During the training for the target domain, the weights of $f_S(\cdot)$ can be frozen or fine-tuned. For the latter, the source data can also be processed to better exploit the source knowledge.

Compared with RTN, $f_S(\cdot)$ is the common network in our model. $\Delta f(\cdot)$ captures the domain discrepancy. Note that this part is trained using only the target samples. Therefore, unlike RTN, the resulting network can better model the target DIGP domain.

Learning Loss. The loss function consists of two components:

$$\min_{f_T} L = \min_{\Delta f} L_R + \lambda \min_{f_S} L_F.$$

Parameter λ balances the regression loss L_R for performance prediction accuracy and the feature loss L_F for learning domain-invariant feature representations.

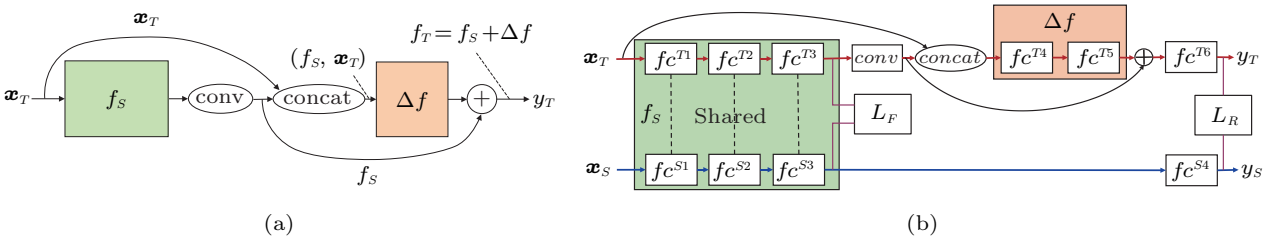

Fig.5. (a) TransGPerf design and (b) network structure.

• L_R: *Regression Loss.* Because the run time of graph computation spans a wide range from sub-seconds to several hours, absolute errors may overly emphasize large run times. Therefore, we choose Mean Absolute Percentage Error (MAPE) in the formulation. L_R includes the regression loss for both the target and the source data:

$$L_R = \frac{1}{n_T}\sum_{i=1}^{n_T}\left|\frac{f_T(\boldsymbol{x}_{T_i}) - y_{T_i}}{y_{T_i}}\right| + \frac{1}{n_S}\sum_{i=1}^{n_S}\left|\frac{f_S(\boldsymbol{x}_{S_i}) - y_{S_i}}{y_{S_i}}\right|.$$

• L_F: *Feature Loss.* Feature loss measures the distance between the distributions of the source and the target domains. We consider two types of feature loss functions: 1) no feature loss, and 2) single layer MMD with a linear kernel (following DDC [12]). The implementation can select the feature loss function that gives the best accuracy. Interestingly, we find (1) is the choice in a significant fraction of cases, which again shows the DIGP domain discrepancy. We plan to experiment with more feature loss functions, such as single layer MMD with an RBF kernel, two-layer MMD, JMMD, and CORAL, in the future work.

3.3 Network Structure

The network design is illustrated in Fig.5(b). We choose the MLP model as the base model when there are sufficient training data. There are three fully connected hidden layers in MLP, each with 128 neurons. After each fully-connected layer, we add a batch normalization and a relu activation function. The input to MLP is a 64-dimensional vector of features for DIGP.

We construct the network of the transfer learning model based on Fig.5(a). The input $\boldsymbol{x}_T$ of the network is a feature representation of the target domain sample, which is a vector of 64 dimensions. $f_S(\cdot)$ is the MLP model with the output layer removed. The output of MLP is a single predicted run time. In comparison, the intermediate input to the output layer contains 128 high-level learned features. By removing the output layer, $f_S(\cdot)$ retains such rich features.

The residual function $\Delta f(\cdot)$ is implemented as two fully-connected layers with 64 neurons, as shown in Fig.5(b). Skip connections follow the design in Fig.5(a).

In (1), "+" is performed by element-wise addition. The output of $f_S(\cdot)$ forwarded by the skip connection is added to the output of the stacked layers in $\Delta f(\cdot)$. However, the output dimensions of $f_S(\cdot)$ and $\Delta f(\cdot)$ are 128 and 64, respectively. To enable element-wise addition, we introduce a one-dimensional convolutional layer of ($num\_channels = 1$, $kernel\_size = 2$, $stride = 2$) after the output vector of $f_S(\cdot)$. This combines the 128-dimensional vector with a 64-dimensional vector. Convolutional layers are widely used in two-dimensional image processing. In this case, we apply it to the one-dimensional vector in order to re-combine and connect the discrete input features so that each neuron of the output vector contains mixed information of the original features. Consequently, the resulting vector can be added to the output of $\Delta f(\cdot)$ in the element-wise fashion.

In (2), "," is performed by concatenation. Hence, the vector $(f_S(\boldsymbol{x}_T), \boldsymbol{x}_T)$ is the concatenation of the 64-dimensional vector from the convolutional layer and the 64-dimensional input vector. Then the vector $(f_S(\boldsymbol{x}_T), \boldsymbol{x}_T)$ has 128 dimensions.

Finally, we add a fully-connected output layer for regression between the output of + and the predicted label y_T, which is different from classification softmax layer.

In addition, TransGPerf allows an optional plugin to compute domain distribution distances for supporting various feature loss functions.

4 DIGP Feature Selection

We overview feature selection for DIGP performance prediction in Subsection 4.1. Then, we describe

how to obtain important complex features in Subsection 4.2.

4.1 Features Impacting DIGP Performance

The run time of a graph computation task is composed of computation, communication, and scheduling costs. First, the computation cost consists of per-vertex computation and computation for managing messages (e.g., sorting, enqueuing, copying messages). This cost is affected by the input graph size (e.g., the number of (#) vertices and edges) and characteristics (e.g., degree distribution), graph algorithms (operations performed on a vertex), platform configurations, and hardware (CPU and memory speeds, #workers, #machines). Second, the communication cost consists of cross-machine communication and intra-process communication on a machine. It is determined by the amount of message data transferred and the network/intra-process communication bandwidth. The former is related to the graph size, the graph partitions, degree distribution, the message sizes and patterns of the graph algorithms, and the platform implementation strategies and configurations. Finally, the scheduling overhead captures the cost of waiting for global barriers and other scheduling related events.

1) *Feature Overview.* Based on our understanding of DIGP, we select 62 features in the following four categories (two influential features, i.e., #messages and graph diameter are duplicated to obtain a 64-dimensional vector for the TransGPerf network):

• input graph: e.g., #vertices, #edges, degree distribution, and diameter;

• graph algorithm: e.g., vertex/edge/message sizes, #messages, and #supersteps;

• DIGP platform: e.g., vertex/edge/message metadata sizes, and configuration parameters;

• hardware: e.g., #machines, #workers, CPU frequency, and network bandwidth.

2) *Feature Extraction.* We obtain the selected features using the following methods.

• *Simple Extraction.* A number of features are obtained directly by examining program, hardware and DIGP configurations or by invoking various tools. The SNAP package [28] computes simple graph features. R-MAT [29] parameters, which are computed by a NetMine package provided by R-MAT authors, show the most similar R-MAT graph to a given input graph. For the graph degree distribution, we count the number of vertices using buckets of exponentially growing widths.

• *Derived Formulas.* We derive formulas to compute a number of graph features. We consider the out-degree exponent as a power-law parameter [30] (see details in Appendix A.1). We use the least squares method to calculate it. We find that the SNAP package [28] reports overflow errors when solving Kronecker [31] parameters for large graphs. Therefore, we estimate the Kronecker parameters (a_k, b_k, c_k, d_k) based on R-MAT parameters (a_r, b_r, c_r, d_r). We can prove that $\frac{a_k}{a_r} = \frac{b_k}{b_r} = \frac{c_k}{c_r} = \frac{d_k}{d_r} \cong 2 \times \gamma^{\frac{1}{\log|V|}}$, where $\gamma = \frac{|E|}{|V|}$, and $|V|$ and $|E|$ are the number of vertices and edges in the graph respectively (see the proof in Appendix A.2).

• *Estimated Through Measurements and/or Models.* A couple of features cannot be directly obtained. We explain how to estimate them in Subsection 4.2.

4.2 Complex Feature Acquisition

We estimate two features, i.e., vertex computation cost and #vertex messages, through measurement and/or models. For the former, the computation on a vertex is determined by the graph analysis logic. Its cost is affected by not only the graph algorithm, but also the DIGP platform, the hardware, and the in-degree and out-degree of the vertex. For the latter, #vertex messages is strongly correlated to the graph algorithm. Different algorithms may have different message patterns.

Fig.6 illustrates the acquisition of the two complex features. We generate a benchmark graph set by TrillionG [32], including 500 synthetic R-MAT graphs, by varying the number of vertices from 50 to 5 000 vertices. There are 10 times as many edges as vertices. We employ five R-MAT parameter settings, i.e., $b_r = c_r = 0.25$, $d_r = s \times a_r$, $a_r + b_r + c_r + d_r = 1$, where $s = 1, \cdots, 5$. The larger the value s, the more skew the degree distribution in the generated graphs. As shown in Fig.6, we perform a DIGP run on a single machine with every benchmark graph for a given algorithm. Data collected are used to estimate the two features. A DIGP run with a benchmark graph takes 0.2 s–6.2 s. The cost of running these benchmark tests is reasonably low.

For vertex computation cost, we measure the run time of performing a given algorithm on a DIGP platform on a single machine for all the benchmark graphs. We compute the average run time as the vertex computation cost.

For #vertex messages, we compute the value for a subset of algorithms and build a black-box model to

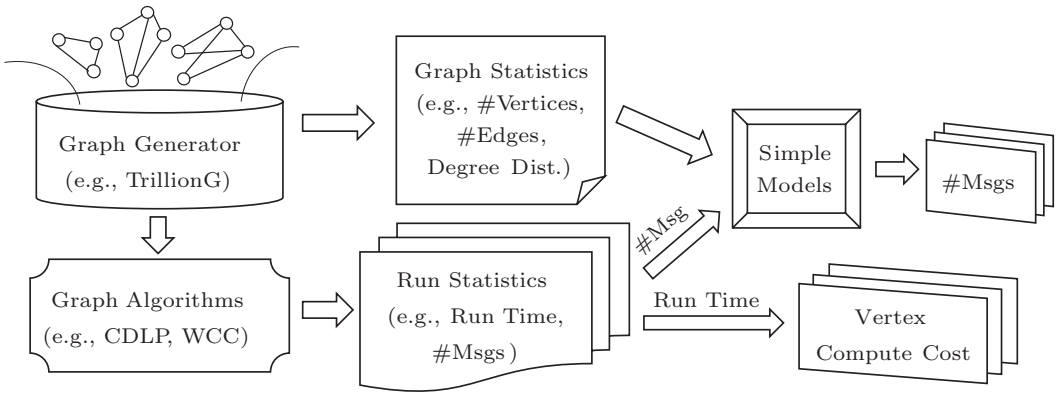

Fig.6. Complex feature acquisition.

estimate the value for the rest algorithms. In PageRank, the number of messages in each superstep is the number of graph edges. Hence, #vertex messages = #supersteps × #edges. In LCC (local clustering coefficient), the number of vertex messages is the sum of the square of each vertex's degree $\sum_v degree^2$. In SSSP and BFS, we estimate #vertex messages as #edges.

However, for algorithms, such as CDLP (community detection using label propagation) and WCC (weakly connected components), #vertex messages cannot be calculated or estimated directly. For each algorithm, we run the algorithm on the benchmark graphs and measure #vertex messages on a single machine. We use these benchmark runs as training samples and build several simple traditional machine learning and neural network models per algorithm by Weka①. We use relevant graph statistics (e.g., #vertices, #edges, degree distribution and approximate full diameter) as the model input. Among all models, SMO (sequential minimal optimization) performs the best. The MAPE errors of SMO for CDLP and WCC are 12.4% and 3.3% respectively.

5 Evaluation

We would like to answer the following questions in the evaluation.

• How does TransGPerf perform for various DIGP transfer learning tasks, including transferring to different DIGP platforms and different graph algorithms?

• How does TransGPerf compare with existing deep transfer learning methods?

• How does transfer learning compare with direct machine learning for DIGP?

5.1 Model Implementation

5.1.1 *Basic Modeling*

We implement four deep learning models using Pytorch: MLP (Multi-Layer Perceptron), CNN (Convolutional Neural Networks), RNN (Recurrent Neural Network), and LSTM (Long Short-Term Memory). We also train five traditional machine learning models using Weka: Linear Regression, K-Nearest Neighbors, Decision Tree, Bagging, and Random Forest. We train the models using the data collected on GraphLite.

Deep learning models perform better than traditional machine learning models on average. Among all deep learning models, MLP performs the best. The MAPE errors are between 7.7% and 9.0% for different graph algorithms.

In DIGP, the features do not have spatial locality, and do not represent time series. Therefore, sophisticated deep learning models, i.e., CNN, RNN, and LSTM, are not very effective. They are less accurate than the plain MLP.

5.1.2 *Transfer Modeling*

We implement eight deep transfer learning methods: 1) fine-tuning (FT), 2) DDC-l (DDC [12] with linear kernel MMD), 3) DDC-r (DDC with RBF kernel MMD), 4) DAN [13], 5) JAN [14], 6) Dcoral [15], 7) RTN [27], and 8) our solution TransGPerf (TGP). For the existing methods 1–7, we replace their pretrained models, such as AlexNet for image classification tasks, with our basic MLP model. We implement the methods as follows.

• For 1), FT inherits most parameters from the basic MLP model, i.e., the network parameters before the output layer.

①https://sourceforge.net/projects/weka/files/weka-3-8/3.8.3/, July 2021.

• For 2) and 3), DDC minimizes the domain distance measure with MMD after the third hidden layer in the basic MLP model. DDC-l is DDC using MMD with the linear kernel and DDC-r is DDC using MMD with the RBF kernel.

• For 4), DAN minimizes the domain distance measure using MMD with the RBF kernel after the second and the third hidden layers. Here, the MMD loss is computed as the sum of the domain distances after the two layers.

• For 5), JAN minimizes the same domain distance measure like DAN, but adopts the joint distributions of the two layers as the distance criterion.

• For 6), Dcoral is nearly the same as DDC except that it uses the CORAL distance as the domain distance measure.

• For 7), RTN is adapted to DIGP as described in Subsection 3.1.

• For 8), our implementation of TransGPerf follows the description in Subsection 3.3.

5.2 Experimental Setup

5.2.1 Sample Collection

We collect DIGP samples by varying the following four aspects.

• *Input Graphs.* We use 10 real-world undirected graphs from ASU①, LAW [33] and SNAP②, as listed in Table 1. The number of vertices ranges from 103 12 to 65.6 million. The number of edges ranges from 333 983 to 1.8 billion.

Table 1. Input Graphs and Time Ranges of Running DIGP Tasks on Graphs

Graph	$\|V\|$	$\|E\|$	min	max
Catalog[1]	10.3 k	334.0 k	0.1 s	3.0 min
Enron[2]	36.7 k	183.8 k	0.1 s	1.1 min
Amazon[2]	334.9 k	925.9 k	0.1 s	3.9 min
Dblp[2]	317.1 k	1.0 M	0.1 s	4.4 min
Youtube[2]	1.1 M	3.0 M	0.2 s	1.0 h
Roadnet[2]	2.0 M	2.8 M	0.6 s	21.9 min
Livejournal[2]	4.0 M	34.7 M	1.0 s	54.4 min
Orkut[2]	3.1 M	117.2 M	1.0 s	1.3 h
Hollywood[3]	2.2 M	114.5 M	0.7 s	57.8 min
Friendster[2]	65.6 M	1.8 G	13.2 s	3.1 h

Note: 1: ASU; 2: SNAP; 3: LAW.

• *Graph Algorithms.* We consider six representative graph analysis algorithms, which are supported by the LDBC Graphalytics benchmark [6]: PageRank (PR), Single Source Shortest Paths (SSSP), Breadth First Search (BFS), Weakly Connected Components (WCC), Community Detection using Label Propagation (CDLP), and Local Clustering Coefficient (LCC). These algorithms cover a wide range of computation and communication patterns.

• *DIGP Platforms.* We consider three representative DIGP platforms: PowerGraph (PG) [2], Spark GraphX (GX) [3], and GraphLite (GL) [4]. PowerGraph is a popular DIGP platform supporting the GAS programming paradigm. GraphX is a DIGP implementation on top of Spark. GraphLite is an open source C/C++ DIGP platform. It supports Pregel-like computation natively. It implements the gather and apply optimizations to mimic the GAS behaviors. Configuration parameters can be used to control the fraction of vertices optimized with the gather and apply optimizations. We use PowerGraph v2.2 and GraphX in Graphalytics③ v1.0.0-0.2, YARN and HDFS in hadoop v3.2.0, and GraphLite in the Github directory④.

• *Hardware.* We run the experiments in a cluster of 12 machines. Each machine is equipped with two Intel® Xeon® E5-2650 v3 CPU @ 2.30 GHz (10 cores, 2 threads/core) and 128 GB DRAM. The machine runs stock Ubuntu 16.04 with Linux Kernel version 4.4.0-112-generic. The g++ version is 5.4.0 and Java version is 1.8. The machines are connected through 10 Gbps Ethernet.

We use graph computation on GraphLite as the source DIGP platform domain. We collect a large number (76 433) of samples on GraphLite. For the other DIGP platforms, we obtain fewer samples. We collect 4 163 and 1 892 samples for PowerGraph and GraphX, respectively. We get fewer samples on GraphX than on PowerGraph because compared with PowerGraph, it often takes longer time to run a GraphX task on the same graph, and GraphX runs out of memory for a large portion of computation tasks on large graphs.

5.2.2 Modeling Methodology

As described in Section 4, we extract 62 features and duplicate two influential features (i.e., #messages and the graph diameter) to compose a 64-dimensional vector as the input to the learning networks. As the fea-

①http://socialcomputing.asu.edu, July 2021.

②http://snap.stanford.edu/data, July 2021.

③https://www.graphalytics.org/, July 2021.

④https://github.com/schencoding/GraphLite, July 2021.

ture values have very different ranges, we apply feature-scaling using z-score normalization.

For a DIGP transfer learning task, we randomly choose part of the samples from the target domain as the training set, and the rest as the test set. By default, we use 80% training data and 20% test data for all the transfer learning methods. For TransGPerf, we perform an additional set of experiments with 40% training data and 60% test data to understand TransGPerf's capability in handling smaller training sets. We use TGP80/TGP to denote experiments with 80% training data, and TGP40 to denote experiments with 40% training data. We use $MAPE$ as the metric to evaluate the models.

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}|\frac{predicted_i - real_i}{real_i}|.$$

Smaller MAPE values mean lower prediction errors, and a better model accuracy.

5.3 Evaluation Results

5.3.1 *Transfer to Different DIGP Platforms*

Fig.7 compares TransGPerf with previous deep transfer learning methods for transferring to different DIGP platforms. The source DIGP platform is GraphLite. The target DIGP platforms are PowerGraph and GraphX, respectively. The sample data contains mixed computation runs for all algorithms.

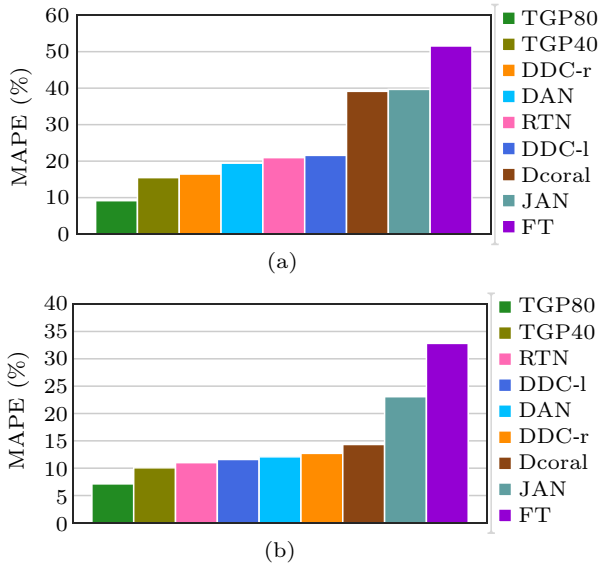

Fig.7. Transfer to different DIGP platforms. (a) PowerGraph. (b) GraphX.

In Fig.7, we see that 1) TGP80 and TGP40 perform the best among all methods. TPG80 achieves a MAPE of 9.2% and 7.1% for PowerGraph and GraphX, respectively. 2) Using 40% training data, TGP40 is only slightly worse than TGP80. 3) Compared with previous methods, TGP performs significantly better. TPG80 reduces MAPE by 7.3%–42.4% and 3.88%–25.69% for PowerGraph and GraphX, respectively.

Previous deep transfer learning methods perform differently on different computation scenarios. The domain invariance methods perform poorly because they cannot capture the domain discrepancy in DIGP. Whether inheriting parameters from the basic MLP (fine-tuning) or minimizing the domain distance (DDC, DAN, JAN, Dcoral), mainly focuses on the similarity between the source and target domains. However, it is not enough to transfer distributed graph computation scenarios. In DIGP, domain discrepancy plays a significant role, which makes domain invariance methods less effective. For RTN, f_T (i.e., the common network) is trained using both the source and the target samples. Since there are much more source samples than target samples, f_T is biased toward GraphLite rather than PowerGraph and GraphX. In comparison, our proposed TransGPerf models $f_T(x)$ by combining $f_S(x)$ and residual $\Delta f(\cdot)$. $f_S(x)$ takes advantage of prior knowledge of the source domain, while $\Delta f(\cdot)$ captures the domain discrepancy and is trained using the target data. In this way, TGP gives a better model for the target.

Interestingly, we see higher MAPE values for PowerGraph. This is because DIGP for large graphs often runs out of memory on GraphX. Therefore, the samples collected for GraphX cover smaller graph diversity than those for PowerGraph.

5.3.2 *Transfer to Different Graph Algorithms*

Fig.8 compares TGP with previous deep transfer learning methods for transferring to different graph algorithms. The DIGP platform is GraphLite. For each of the six algorithms, we use the rest of the five algorithms to build the source MLP model. Then we use the source model to build a transfer learning model for the given algorithm.

In Fig.8, we see similar trends as in Fig.7. TGP80 and TPG40 perform the best among all methods. TGP80 achieves a MAPE of 6.4%–16.9% for transferring to one of the algorithms. Compared with previous methods, TGP80 reduces MAPE up to 36.4%. Finally, using smaller sample datasets, TGP40 actually achieves an accuracy similar to TGP80.

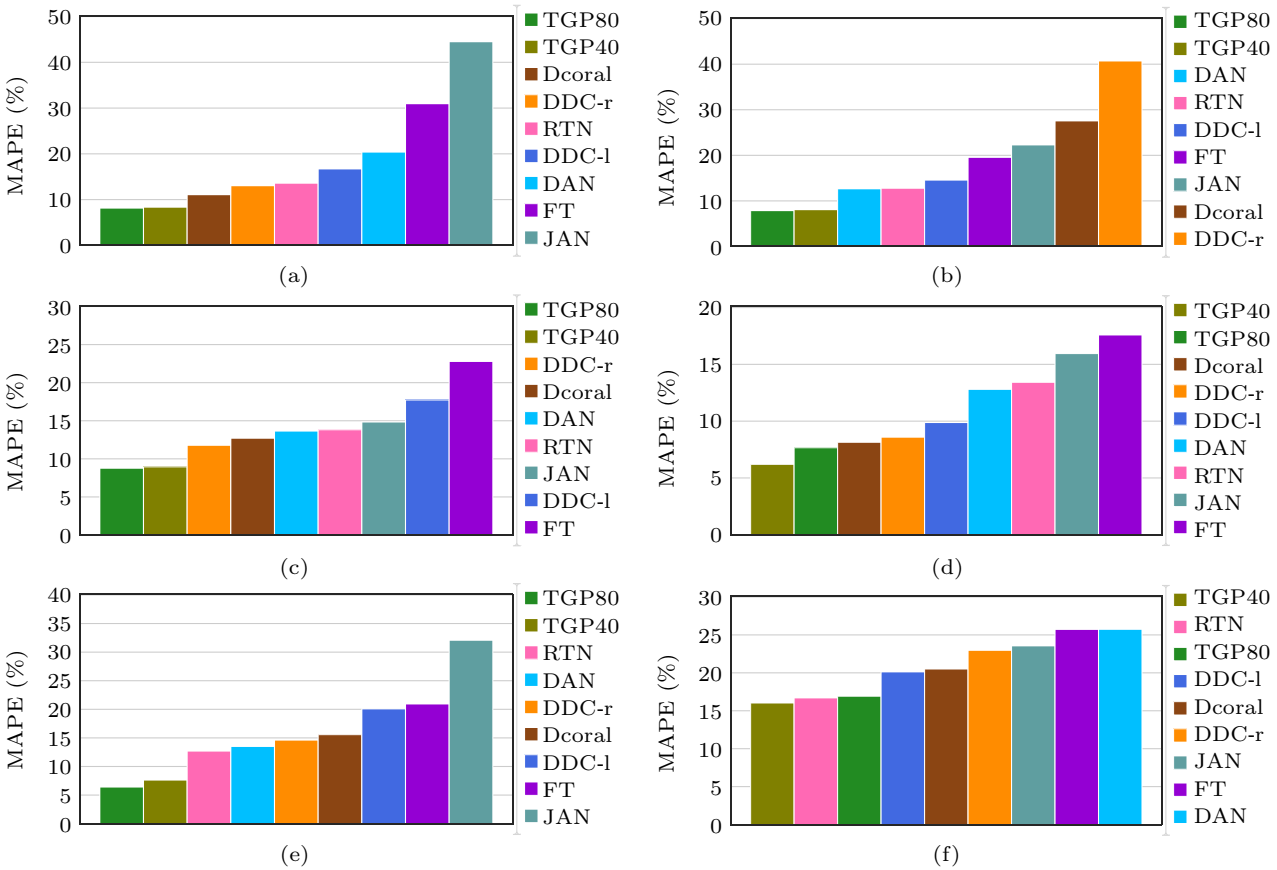

Fig.8. Transfer to different graph algorithms. (a) PR. (b) SSSP. (c) BFS. (d) WCC. (e) CDLP. (f) LCC.

5.3.3 *Predicted with Basic Models*

To illustrate the difference between the source and the target, we use the basic models trained with the source to predict the target. Table 2 and Table 3 show the MAPE of MLP trained with Source (MLP-Src) and TGP models for target DIGP platforms and graph algorithms respectively. MLP-Src is the source MLP models in experiments of transfer to different DIGP platforms and graph algorithms. The difference between MAPE of MLP-Src and TGP can reflect the transfer difficulty and the large gap from the source to the target.

Table 2. MAPE (%) of MLP Trained with Source and TGP Models for Target DIGP Platforms

Model	PG	GX
MLP-Src	1 013.4	142.4
TGP	9.2	7.1

Table 3. MAPE (%) of MLP Trained with Source and TGP Models for Target Graph Algorithms

Model	PR	SSSP	BFS	WCC	CDLP	LCC
MLP-Src	59.5	325.0	824.6	67.8	40.0	1 051.1
TGP	8.2	7.9	8.8	7.7	6.4	17.9

5.3.4 *Transfer to Different Graph Datasets*

We transfer graphs with low average degrees (avg.dgr.) to other graphs with high degrees. Firstly, we transfer graph samples on GraphLite of Amazon (avg.dgr. 2.76) to Catalog (avg.dgr. 32.39) and Friendster (avg.dgr. 27.53) in Table 1. TGP achieves the test MAPE of 10.93% and 39.67% respectively for Catalog and Friendster. Though the average degrees of graphs Catalog and Friendster are both approximate to 30, the transferring effects from the same graph Amazon are different. Then we transfer graph samples on GraphLite of Youtube (avg.dgr. 2.63) to Friendster in Table 1 and TGP achieves the test MAPE of 21.82%. The transfer accuracy of target Friendster has improved a lot from Amazon to Youtube as the source. Though the average degrees of graphs Amazon and Youtube are similar, the graph scale and time ranges of tasks on Youtube are closer to those of Friendster than to those of Amazon.

The results indicate that, besides average degree, other graph characteristics (e.g., graph scale, degree distribution, time ranges of tasks on the graph) can also have influence on the transferring effects of graph

datasets. The more similar the two domains are, the better the transferring effects will be. Previous good results of transferring graph platforms and algorithms by TGP mean that 1) the source properties are well learned, and 2) the discrepancy between domains can be well characterized by the small residual network. Therefore, the source can be well transferred to the target. However, for transferring graph datasets, our experiments show it depends on the similarity. For example, if the graph scales or time ranges of tasks on the graphs differ a lot, the transferring effects of different graphs are not so good. This is probably because the discrepancy between domains cannot be characterized by the small residual network. Hence the boundary between transfer learning and machine learning is the domain similarity. If the source and the target domains differ a lot, we had better choose traditional machine learning, or collect more similar samples with the target for the source domain.

5.3.5 *Comparison with Ernest*

Ernest [9] builds a traditional regression model (i.e., non-negative least squares) to predict Spark performance. As GraphX tasks can be considered as Spark applications, we are interested in comparing TGP and Ernest for GraphX. For a given graph algorithm (BFS and WCC), we run the sample configurations computed by Ernest for all the 10 real-world graphs. Then, we use the Ernest package to build a performance model. The resulting MAPE for predicting DIGP performance is 654.3% for BFS and 1 120.7% for WCC. The accuracy of Ernest is much worse than that of TGP (and the other transfer learning methods) as shown in Fig.7. This is because Ernest assumes that both the job (the graph algorithm) and the input dataset (the input graph) are fixed. It cannot handle the case where multiple input graphs may be used in the computation, while our proposed TGP supports this case effectively.

6 Discussion

Computation performance has been a significant concern in distributed graph analysis. Complex graph computation tasks often take hours or even days to run on graphs with billions of vertices and edges. A good performance model can be very helpful in two aspects. First, it can help users estimate the run time of a graph computation task and understand the tradeoff between hardware cost and computation time for resource scheduling and capacity planning purposes. Second, it can help algorithm developers understand intrinsic features and underlying patterns of graph computation in order to better optimize their designs. In general, modeling performance for DIGP can help execution time prediction, resource planning, performance analysis, and computation optimization.

7 Conclusions

Performance modeling for graph computation is a challenging yet new field to explore. We proposed a novel deep transfer learning method, TransGPerf, that exploits knowledge from a source DIGP domain to build the model for a target DIGP domain with limited target samples. It reduces the cost of running a large number of target DIGP tasks. Experimental results showed that TransGPerf effectively supports a wide range of DIGP transfer learning tasks. Performance modeling in other types of distributed systems often encounters similar challenges as in DIGP. It is also time- and resource-consuming to collect a large number of training samples. We believe that our proposed method may be applicable beyond DIGP.

Acknowledgement(s) We would like to thank Deepayan Chakrabarti for the graph statistics package NetMine and Himchan Park for discussions about graph characteristics, which help a lot for our graph feature extraction.

References

[1] Malewicz G, Austern M H, Bik A J C, Dehnert J C, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. In *Proc. the 2010 ACM SIGMOD International Conference on Management of Data*, June 2010, pp.135-146. DOI: 10.1145/1807167.1807184.

[2] Gonzalez J E, Low Y, Gu H, Bickson D, Guestrin C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. the 10th USENIX Symposium on Operating Systems Design and Implementation*, October 2012, pp.17-30.

[3] Xin R S, Gonzalez J E, Franklin M J, Stoica I. GraphX: A resilient distributed graph system on Spark. In *Proc. the 1st International Workshop on Graph Data Management Experiences and Systems*, June 2013, Article No. 2. DOI: 10.1145/2484425.2484427.

[4] Niu S, Chen S. Optimizing CPU cache performance for Pregel-like graph computation. In *Proc. the 31st IEEE International Conference on Data Engineering Workshops*, April 2015, pp.149-154. DOI: 10.1109/ICDEW.2015.7129568.

[5] Han M, Daudjee K, Ammar K, Özsu M T, Wang X, Jin T. An experimental comparison of Pregel-like graph processing systems. *Proc. VLDB Endow.*, 2014, 7(12): 1047-1058. DOI: 10.14778/2732977.2732980.

[6] Iosup A, Hegeman T, Ngai W L *et al.* LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.*, 2016, 9(13): 1317-1328. DOI: 10.14778/3007263.3007270.

[7] Ngai W L, Hegeman T, Heldens S, Iosup A. Granula: Toward fine-grained performance analysis of largescale graph processing platforms. In *Proc. the 5th International Workshop on Graph Data-management Experiences & Systems*, May 2017, Article No. 8. DOI: 10.1145/3078447.3078455.

[8] Herodotou H, Lim H, Luo G, Borisov N, Dong L, Cetin F B, Babu S. Starfish: A self-tuning system for big data analytics. In *Proc. the 5th Biennial Conference on Innovative Data Systems Research*, January 2011, pp.261-272.

[9] Venkataraman S, Yang Z, Franklin M J, Recht B, Stoica I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proc. the 13th USENIX Symposium on Networked Systems Design and Implementation*, March 2016, pp.363-378.

[10] Pan S J, Yang Q. A survey on transfer learning. *IEEE Trans. Knowledge and Data Engineering*, 2010, 22(10): 1345-1359. DOI: 10.1109/TKDE.2009.191.

[11] Weiss K R, Khoshgoftaar T M, Wang D. A survey of transfer learning. *J. Big Data*, 2016, 3: Article No. 9. DOI: 10.1186/s40537-016-0043-6.

[12] Tzeng E, Hoffman J, Zhang N, Saenko K, Darrell T. Deep domain confusion: Maximizing for domain invariance. arXiv:1412.3474, 2014. https://arxiv.org/pdf/1412.3474.pdf, May 2021.

[13] Long M, Cao Y, Wang J, Jordan M I. Learning transferable features with deep adaptation networks. In *Proc. the 32nd International Conference on Machine Learning*, July 2015, pp.97-105.

[14] Long M, Zhu H, Wang J, Jordan M I. Deep transfer learning with joint adaptation networks. In *Proc. the 34th International Conference on Machine Learning*, August 2017, pp.2208-2217.

[15] Sun B, Saenko K. Deep CORAL: Correlation alignment for deep domain adaptation. In *Proc. the 2016 European Conference on Computer Vision Workshops*, October 2016, pp.443-450. DOI: 10.1007/978-3-319-49409-8_35.

[16] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In *Proc. the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, June 2016, pp.770-778. DOI: 10.1109/CVPR.2016.90.

[17] Barker K J, Pakin S, Kerbyson D J. A performance model of the Krak hydrodynamics application. In *Proc. the 2006 International Conference on Parallel Processing*, August 2006, pp.245-254. DOI: 10.1109/ICPP.2006.11.

[18] Kerbyson D J, Alme H J, Hoisie A, Petrini F, Wasserman H J, Gittings M L. Predictive performance and scalability modeling of a large-scale application. In *Proc. the 2001 ACM/IEEE Conference on Supercomputing*, November 2001, Article No. 37. DOI: 10.1145/582034.582071.

[19] Sundaram-Stukel D, Vernon M K. Predictive analysis of a wavefront application using logGP. In *Proc. the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999, pp.141-150. DOI: 10.1145/301104.301117.

[20] Bhattacharyya A, Hoefler T. PEMOGEN: Automatic adaptive performance modeling during program runtime. In *Proc. the 23rd International Conference on Parallel Architectures and Compilation*, August 2014, pp.393-404. DOI: 10.1145/2628071.2628100.

[21] Bhattacharyya A, Kwasniewski G, Hoefler T. Using compiler techniques to improve automatic performance modeling. In *Proc. the 2015 International Conference on Parallel Architectures and Compilation*, October 2015, pp.468-479. DOI: 10.1109/PACT.2015.39.

[22] Calotoiu A, Beckingsale D, Earl C W, Hoefler T, Karlin I, Schulz M, Wolf F. Fast multi-parameter performance modeling. In *Proc. the 2016 IEEE International Conference on Cluster Computing*, September 2016, pp.172-181. DOI: 10.1109/CLUSTER.2016.57.

[23] Sun J, Sun G, Zhan S, Zhang J, Chen Y. Automated performance modeling of HPC applications using machine learning. *IEEE Trans. Computers*, 2020, 69(5): 749-763. DOI: 10.1109/TC.2020.2964767.

[24] Pan S J, Tsang I W, Kwok J T, Yang Q. Domain adaptation via transfer component analysis. *IEEE Trans. Neural Networks*, 2011, 22(2): 199-210. DOI: 10.1109/TNN.2010.2091281.

[25] Sun B, Feng J, Saenko K. Return of frustratingly easy domain adaptation. In *Proc. the 30th AAAI Conference on Artificial Intelligence*, February 2016, pp.2058-2065.

[26] Tzeng E, Hoffman J, Darrell T, Saenko K. Simultaneous deep transfer across domains and tasks. In *Proc. the 2015 IEEE International Conference on Computer Vision*, December 2015, pp.4068-4076. DOI: 10.1109/ICCV.2015.463.

[27] Long M, Zhu H, Wang J, Jordan M I. Unsupervised domain adaptation with residual transfer networks. In *Proc. the 30th Annual Conference on Neural Information Processing Systems*, December 2016, pp.136-144.

[28] Leskovec J, Sosič R. SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.*, 2016, 8(1): Article No. 1. DOI: 10.1145/2898361.

[29] Chakrabarti D, Zhan Y, Faloutsos C. R-MAT: A recursive model for graph mining. In *Proc. the 4th SIAM International Conference on Data Mining*, April 2004, pp.442-446. DOI: 10.1137/1.9781611972740.43.

[30] Faloutsos M, Faloutsos P, Faloutsos C. On power-law relationships of the Internet topology. In *Proc. the 1999 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 30–September 3, 1999, pp.251-262. DOI: 10.1145/316188.316229.

[31] Leskovec J, Chakrabarti D, Kleinberg J M, Faloutsos C. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *Proc. the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases*, October 2005, pp.133-145. DOI: 10.1007/11564126_17.

[32] Park H, Kim M. TrillionG: A trillion-scale synthetic graph generator using a recursive vector model. In *Proc. the 2017 ACM International Conference on Management of Data*, May 2017, pp.913-928. DOI: 10.1145/3035918.3064014.

[33] Boldi P, Vigna S. The webgraph framework I: Compression techniques. In *Proc. the 13th International Conference on World Wide Web*, May 2004, pp.595-602. DOI: 10.1145/988672.988752.

Songjie Niu received her B.E. degree from Beijing Institute of Technology, Beijing, in 2014. She is a Ph.D. candidate at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests include graph computation, database systems, and big data processing. She is a student member of CCF.

Shimin Chen received his Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, in 2005. He is a full professor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include data management systems, big data processing, and computer architecture. He is a senior member of IEEE.

Appendix

A.1 PowerLaw Equivalence

Four kinds of power-laws are introduced in [30]. We focus on rank exponent and out-degree exponent. We have proved these two kinds of power-laws are equivalent to some extent. The out-degree exponent power-law is defined as: the frequency f_d of an out-degree d is proportional to the out-degree to the power of a constant $\mathcal{O}$, i.e., $f_d \propto d^{\mathcal{O}}$. The rank exponent power-law is defined as: the out-degree d_v of a node v is proportional to the rank of the node r_v to the power of a constant $\mathcal{R}$, i.e., $d_v \propto r_v^{\mathcal{R}}$, which can also be written as $d_v^{\frac{1}{\mathcal{R}}} \propto r_v$. Assuming $r_v = c_v d_v^{\frac{1}{\mathcal{R}}}$, if we derive r_v with respect to d_v, we can get $r'_v = \frac{c_v}{\mathcal{R}} d_v^{\frac{1}{\mathcal{R}}-1}$. Notice this formula has a similar format to the out-degree exponent power-law. It is not difficult to understand that f_d is the rate of change of r_v relative to d_v. Therefore we can get $\frac{1}{\mathcal{R}} - 1 \approx \mathcal{O}$.

A.2 Kronecker Calculation

We estimate the Kronecker parameters of large graphs by means of R-MAT parameters as below. For a graph, suppose the number of vertices and edges are denoted as $|V|$ and $|E|$ respectively, the four parameters of R-MAT are (a_r, b_r, c_r, d_r), and the four parameters of Kronecker are (a_k, b_k, c_k, d_k).

The probability that R-MAT produces an edge is

$$P_r(e) = |E| {a_r}^{|a_r|} {b_r}^{|b_r|} {c_r}^{|c_r|} {d_r}^{|d_r|},$$

where $|a_r| + |b_r| + |c_r| + |d_r| = \log|V|$ and $a_r + b_r + c_r + d_r = 1$.

The probability that Kronecker produces an edge is

$$P_k(e) = {a_k}^{|a_k|} {b_k}^{|b_k|} {c_k}^{|c_k|} {d_k}^{|d_k|},$$

where $|a_k| = |a_r|, |b_k| = |b_r|, |c_k| = |c_r|$, and $|d_k| = |d_r|$.

For an edge, there will be $P_r(e) = P_k(e)$, then we can get $(\frac{a_k}{a_r})^{|a_r|}(\frac{b_k}{b_r})^{|b_r|}(\frac{c_k}{c_r})^{|c_r|}(\frac{d_k}{d_r})^{|d_r|} = |E|$. Assume $\frac{a_k}{a_r} = \frac{b_k}{b_r} = \frac{c_k}{c_r} = \frac{d_k}{d_r} = \lambda$, then there will be $\lambda^{\log|V|} = |E|$. Suppose $|E| = \gamma|V|$, then we can get $\lambda = 2{\times}\gamma^{\frac{1}{\log|V|}}$, namely $\frac{a_k}{a_r} = \frac{b_k}{b_r} = \frac{c_k}{c_r} = \frac{d_k}{d_r} = 2{\times}\gamma^{\frac{1}{\log|V|}}$.