Generalizable Reinforcement Learning-Based Coarsening Model for Resource Allocation over Large and Diverse Stream Processing Graphs

Lanshun Nie, Yuqi Qiu, Fei Meng
Harbin Institute of Technology

Mo Yu
IBM Research

Jing Li
New Jersey Institute of technology

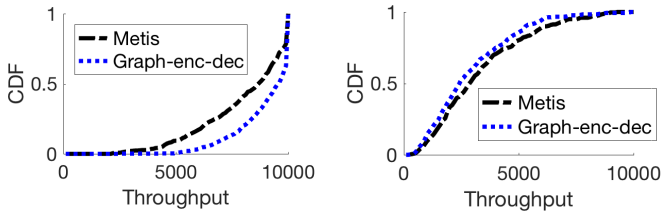
***Abstract*—Resource allocation for stream processing graphs on computing devices is critical to the performance of stream processing. Efficient allocations need to balance workload distribution and minimize communication simultaneously and globally. Since this problem is known to be NP-complete, recent machine learning solutions were proposed based on an encoder-decoder framework, which predicts the device assignment of computing nodes sequentially as an approximation. However, for large graphs, these solutions suffer from the deficiency in handling long-distance dependency and global information, resulting in suboptimal predictions. This work proposes a new paradigm to deal with this challenge, which first coarsens the graph and conducts assignments on the smaller graph with existing graph partitioning methods. Unlike existing graph coarsening works, we leverage the theoretical insights in this resource allocation problem, formulate the coarsening of stream graphs as edge-collapsing predictions, and propose an edge-aware coarsening model. Extensive experiments on various datasets show that our framework significantly improves over existing learning-based and heuristic-based baselines with up to 56% relative improvement on large graphs.**

***Index Terms*—resource allocation, reinforcement learning, stream processing, graph neural network**

I. Introduction

Stream processing, which enables the analysis of online arriving data, has been widely used in many industrial domains, such as transportation, telecommunication, and data analytics [1]–[4]. The incoming data flow of stream processing consists of structured data items called data tuples. The computation and communications in stream processing can be represented as a Directed Acyclic Graph (DAG), where the nodes represent the operators performing computation on the incoming data tuples and the directed edges represent the transmission of data tuples between operators. Online stream processing receives a high rate of incoming data flow and needs to return the computed results in real time. Thus, its main performance objective is to achieve high processing *throughput* on the available computing devices via good resource allocation for the operators.

Recently, machine learning solutions for resource allocation have been developed [5]–[13]. Among them, some specifically focus on stream processing [8], [9]. These studies mainly follow an encoder-decoder framework — an encoder is used to embed the computation graphs into node embedding so that the topological contexts can be represented as vectors. Based on the node embedding, the decoder assigns each node to a device sequentially. The encoders and decoders can be made graph-aware via graph networks [14], [15] to make the model generalizable to various graph structures.



(a) Small graphs (4∼26 nodes) (b) Big graphs (100∼200 nodes)

Fig. 1. ***Cumulative Distribution Function*** (***CDF***) of throughputs of 300 stream processing graphs with different topologies under the resource allocation of Metis vs. Graph-encoder-decoder. In CDF, (5000, 0.5) means that 50% of graphs have throughputs less than 5000/s. Thus, a curve with a smaller ***Area-Under-Curve*** (***AUC***), i.e., more skewed towards the right, has higher throughputs for stream graphs and shows better performance.

Although the encoder-decoder paradigm works well for a single large graph (i.e., training and testing on the same graph) [6], it faces challenges when learning a ***generalizable resource allocation strategy*** for large graphs (i.e., testing on large graphs with topologies unseen during training). Because the resource allocation problem is a joint optimization with global dependency among nodes, large graphs make the encoder-decoder models insufficient to capture such global information needed for optimal predictions. For example, a good allocation for a stream graph with lightweight transmission requires putting a balanced workload of stream processing nodes on each device. This essentially requires the node embedding to be aware of the whole graph, which is usually beyond the receptive fields of existing graph encoders. Figure 1 verifies this issue with a pilot study of our implementation of a graph encoder-decoder [9]. Compared with a non-learned graph partitioning algorithm Metis [16], the graph-encoder-decoder model changes from outperforming to underperforming when the sizes of graphs increase from ∼20 to ∼150.

In this work, we propose a novel ***coarsening-partitioning***

This research was supported, in part, by the National Science Foundation (USA) CNS–1948457, National Key Research and Development Project 2022YFB3305500, and National Natural Science Foundation of China U20A6003. Lanshun Nie, Yuqi Qiu, and Fei Meng are co-first authors. Jing Li is the corresponding author.

paradigm for learning a generalizable resource allocation strategy over large stream graphs. The key idea is to *coarsen* a large graph to a smaller one that is easier to handle for a partitioning model. The coarsening model is trained with reinforcement learning to maximize the throughputs of resource allocations over the smaller coarsened graphs. In other words, the model learns how to generate a coarsened graph that is representative of the original graph in terms of resource allocation results.

Second, to make the coarsening process suitable for stream graphs, we formulate it as an ***edge-collapsing prediction*** — the coarsening model learns whether to merge the nodes connected by an edge. This formulation leverages the theoretical insights in stream processing that collapsing some edges can better reduce the communication between merged nodes while maintaining balanced computation loads of merged nodes. In contrast, existing graph coarsening formulations, developed in machine learning fields, cluster nodes into pre-defined groups. Their formulation is less effective in the context of stream processing, as the groups do not preserve specific meaning.

Third, we propose a new ***graph coarsening model*** with edge-aware graph-encoding to make the edge-collapsing predictions. Our model encodes the graph using node and edge features in an efficient and global manner to best capture their impacts on throughput. To ensure that nodes are merged largely because they can be allocated to the same device to avoid heavy transmission in between without overloading the device, we use the node embedding and edge features to build our edge representation and perform the edge-collapsing predictions. The coarsening model is updated via policy gradients to maximize the throughputs after partitioning.

Additionally, we propose a curriculum learning [17] framework to handle the training difficulty over large graphs on many devices. These graphs have large search spaces and are thus unlikely to have optimal allocation results sampled during training when the whole framework is trained from scratch. Our curriculum solution addresses this severe challenge in two folds. First, we train the whole framework starting from small graphs on a few devices and gradually moving to larger ones on more devices. Intuitively, when trained on the small graphs, the model can quickly converge to model the relationship between the input graphs and the required allocations with desired throughputs. Then, continued training only needs to help the model to fit the distribution of large graphs. Second, if training using smaller graphs is not possible, we add the results obtained by a heuristic algorithm (e.g., Metis) to the beam of samples for training. This guided supervision signal helps the model to more quickly get over the cold start stage, where none of the sampled allocation results have good throughputs.

Finally, we conduct extensive experiments with various data sets and compare our framework with state-of-the-art learning- and heuristic-based methods. Results show that our framework significantly outperforms all baselines in all settings with up to 46%, 56%, and 40% improvements for graphs with 100$\sim$200, 400$\sim$500, and 1,000$\sim$2,000 nodes, respectively. Our framework also leads to a huge improvement in a setting with a realistic problem where the computing resources exceed the requirement of the graphs — it learns to use a reasonable subset of devices. Our coarsening model also demonstrates great transferability and adaptability when deployed to graphs vastly different from the training set. This indicates the potential of our approach to fit the changing real-world deployment requirements and computation environments.

II. Related Work

Resource Allocation of Stream Processing Graphs. Stream processing systems have been extensively studied [18]–[24], and many algorithms were proposed for the resource allocation problem. For example, Metis [16] is a heuristic-based algorithm with graph coarsening and partitioning steps, which requires users to adjust the coarsen scales and resources. Amini et al. [25] applied a two-tiered approach that uses a Linear Quadratic Controller for the flow control and requires additional assumptions of the system. Fu et al. [26] instead designed a congestion-aware scheduler using queue-theoretic analysis and a fixed-size worker pool. Additional considerations, such as data ingestion [27], dispersed computing networks [28], energy saving [29], fault tolerance [30], and meeting latency targets [31], [32], have also been studied.

Learning for Resource Allocation of Computation Graphs. [5]–[8] are the first works that apply learning models to resource allocation of computation graphs. They used *sequence-to-sequence models* to predict the device placement for operations in a *single* TensorFlow graph. Using sequence-to-sequence models limited their applications to unobserved graphs, since the model can only rely on topological orders to capture the graph structure. Therefore, for each graph they consider, such as Inception-V3 for image classification, LSTM for language modeling, and word count application for stream processing, a graph-specific model is separately trained and tested on the same individual graph.

To overcome the above generalization challenge, follow-up works [9], [10] apply graph neural networks on top of reinforcement learning [33], so that the same model can handle different graph topologies. They adopt a graph-to-sequence architecture. An LSTM-based encoder is still kept, but predicting the resource allocation is based on the node embedding from a graph neural network.

For larger unseen graphs, both the graph encoder and LSTM decoder of these works become less efficient, despite their successes in training and testing on one very large graph with the same topology. Our graph coarsening model, which is orthogonal to the development of graph-to-sequence models, helps address this issue. Graph coarsening has been studied in data mining applications [34]–[37]. Our coarsening model generalizes the network design in [37], but we are fundamentally unique in the specific formulation that handles the edge-collapsing prediction task for coarsening stream graphs.

III. Our Proposed Framework

We first give a formal definition of the resource allocation problem in stream processing, then show why it challenges existing partitioning models for large graphs, and propose an

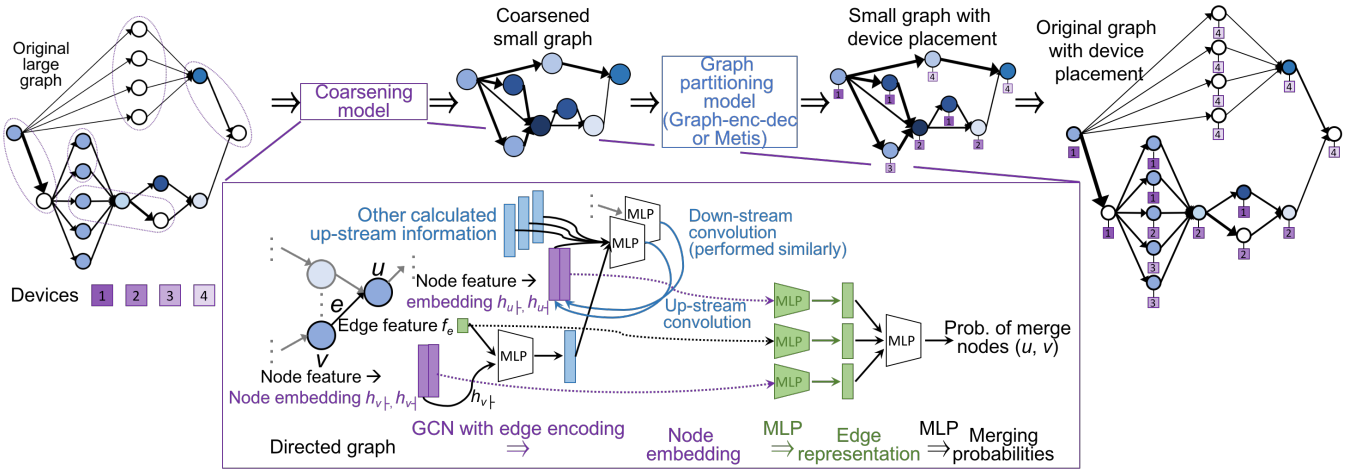


Fig. 2. Overview of our coarsening-partitioning framework for device allocation and the proposed coarsening model architecture. Our model is composed of graph encoding to learn the structured information of stream graphs for predicting whether to collapse an edge.

alternative solution to this problem by formulating the problem as graph coarsening via edge-collapsing prediction.

Problem Definition. Our data consists of different stream processing graphs $\{G_x\}$, where each graph $G_x = (V, E)$ as illustrated on the left of Figure 2. Each node $v \in V$ represents an operator, characterized by its ***CPU utilization*** (number of instructions required per second) and ***payload*** (total size of tuples produced by the operator). Each directed edge $e = (v_{in}, v_{out}) \in E$ represents the connection between operators v_{in} and v_{out}, via which v_{in} transmits its output tuples to v_{out} as input. Each edge is characterized by its communication cost, payload.

Given a graph G_x and a set of devices (e.g., CPUs) D, a resource allocation model $\mathcal{M}$ predicts a ***device placement graph*** G_y where an operator $v \in G_x$ is assigned to a device $d_v \in D$. The right of Figure 2 illustrates the target graph G_y where each node in G_x is appended with a new ***device id node*** (the square nodes in the figure) depicting its allocation.

Motivation of a New Paradigm. Existing studies [9], [10] usually perform *direct partitioning* — their models encode the original graph, on top of which the placements are directly predicted on each node. Direct partitioning is not efficient for large graphs. Essentially, resource allocation for graphs is a combinatorial search problem, with the prediction of each node having dependencies on others. The dependencies may have long distances across the whole graph. Intuitively, one important factor for a good allocation is that the workload on each device is balanced. This global dependency requires the hidden states of the encoding results to have the reception fields over the whole graph.

While most existing works try to overcome this challenge with more powerful encoders with useful inductive biases, in this paper, we propose an alternative paradigm — our key idea is to coarsen the large graphs to small enough for an existing partitioning method to handle them well. Moreover, the coarsening step is achieved with a model that predicts what edges to collapse and is optimized with reinforcement learning to maximize the performance of placement over coarsened graphs. The edge-collapsing prediction follows the theoretical intuitions that communication costs of some edges are more likely to be the bottleneck for maximizing the throughputs of stream graphs. With reasonable node embeddings, such information can be captured into edge representations without global topological structures and thus easier to learn.

Our Coarsening-Partitioning Framework. We propose a novel framework shown in Figure 2. Formally, a large input graph G_x is first fed into a ***coarsening model*** (detailed in Section IV) to predict the edge $e_{u,v}$ (connecting nodes u and v) to collapse and obtain a smaller coarsened graph S_x. As a result, this corresponds to a function $\mathcal{F}$ mapping each $v \in G_x$ to a node $v' \in S_x$. Because $|S_x| < |G_x|$, a node $v' \in S_x$ corresponds to a subset of nodes from G_x. Second, the coarsened graph S_x is fed into a ***partitioning model*** $\mathcal{M}$, e.g., existing device placement methods, to predict the coarse-grained device placement from S_x to S_y, where each $v' \in S_x$ is assigned to a device $d_{v'} \in D$. The placement S_y is mapped back (i.e., nodes $\{v|\mathcal{F}(v) = v'\}$ assigned to device $d_{v'}$) to the original graph to get the final allocation.

Training. In the resource allocation task, supervised learning of models is in general not possible, because it is difficult, if not infeasible, to get the ground truth allocation G_y for an input G_x. However, the relative quality of an allocation G_y can be accessed by calculating its throughput. Therefore, we optimize our model via reinforcement learning, with the throughput of the predicted allocation graph as the reward r.

Specifically, we aim to maximize the ***relative throughput*** $r(G_y) = \frac{T(G_y)}{I(G_x)}$ as the reward. Here, $T(G_y)$ is the throughput and $I(G_x)$ is the source tuple rate. Hence, the reward r ranges from 0 to 1 — a high reward ensures that the tuple processing rate (throughput) catches up with the source tuple rate, so there is no backpressure due to bad resource allocation.

To speed up the reward calculation, we adopt a simulator ***CEPSim*** for cloud-based complex stream processing [38] to

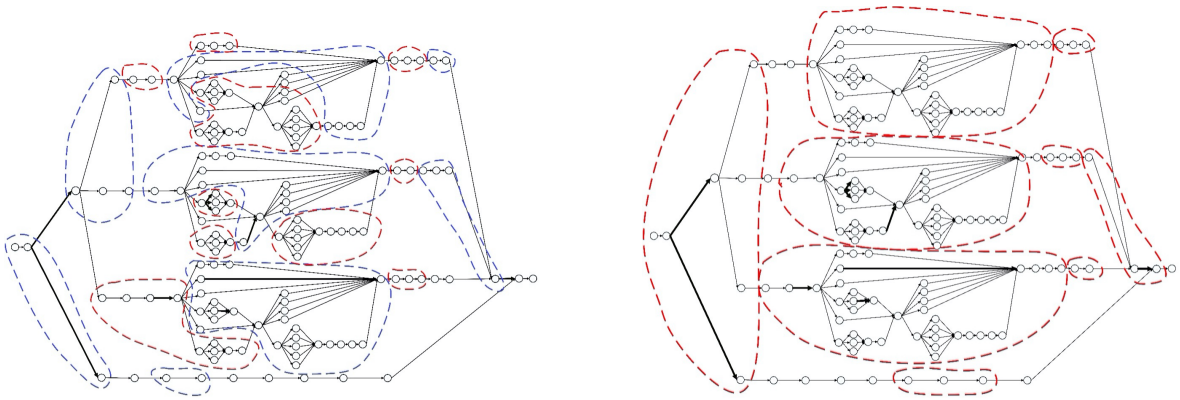


Fig. 3. An example of different graph coarsening results from the Metis coarsening algorithm (left) and our model (right). The transition load on the edge is represented by thickness. When both are partitioned by the Metis partitioning algorithm, the former gives a throughput of 3,104/s, and ours gives 9,192/s.

evaluate each allocation sample. The fidelity of CEPSim has been validated in [9], where the simulator can mimic the behavior of the real streaming systems, with consistent relative performance ranks between the throughout estimations of the simulator and a real streaming platform.

We train our framework with the REINFORCE algorithm [39] to compute policy gradients and learn network parameters with Adam optimizer [40]: $\nabla(\theta)J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\nabla\log\pi_\theta(G_y^n)\big[r(G_y^n) - b\big]$, where π_θ is the policy function outputting a distribution of possible G_y — in our case, its θ is all the learnable parameters in our framework. The baseline b is the average reward of the N on-policy and historically-found best samples, which is introduced to reduce the variance of policy gradients.

IV. Our Edge-Collapsing Coarsening Model

This section discusses how our proposed edge-collapsing formulation fits the resource allocation task for stream graphs and introduces our design of the coarsening model.

Previous works on graph coarsening [34], [41], [42] usually group nodes according to their similarity, which is essentially node-level classifications to pre-defined group IDs. However, this formulation does not fit our problem for two reasons. (1) According to insights from scheduling theory, the goal of coarsening is to collapse edges with large impacts on the throughput of a stream graph, not to find similar nodes (illustrated in Figure 3). (2) Previous coarsening models work on groups for graphs like social networks, so their node groups have specific meanings. However, there is no semantics of the groups themselves for stream graphs. Experiments (Figures 5 and 6) verify that such node-level classification formulation does not work well for our task.

Therefore, the coarsening of stream graphs for allocation should rely on edge-level decisions instead of node-level ones. The bottom of Figure 2 depicts the architecture of our proposed coarsening model. Our design aims to address two challenges: *how to make full use of the edge information for graph encoding* (Section IV-A), as our inputs are directed graphs with edge features different from existing coarsening works; and *how to use both node and edge information for the edge-collapsing prediction* (Section IV-B) to balance the impacts of computation load and communication costs.

Remark on generalizability. Our formulation also makes our coarsening model *more generalizable*, because edge-collapsing operations have similar semantics across different graphs and computing environments (e.g., different numbers of devices). Hence, our trained model can work well when directly applied to a different setting (e.g., unseen larger graphs or evaluated on a real platform instead of a simulated one) and can be further fine-tuned with a small number of iterations.

A. Edge-Aware Stream Graph Encoding

The first step of graph coarsening is to encode the graph to get its contextualized embedding. For the input directed graph G_x with edge features, the graph encoding should be edge-aware. One way to achieve this is to apply edge convolution, which requires a sparse edge set. In stream graphs, the edge sizes are usually larger than the node sizes. Thus, we take a different approach, with special treatment of edge directions and features during node convolution, and perform edge-level actions by constructing edge representations from the node embeddings and edge features.

Specifically, we build our graph encoder on top of the graph neural network (GNN) [14], [15], [43], following [9], [10]. The GNN iteratively updates a node's embedding (hidden states) with its neighbors' embeddings. We involve edge features in this process as follows. We denote the embedding of node v at the k-th iteration as $\mathbf{h}_v^k$, where $\mathbf{h}_v^0 = \mathbf{f}_v$, i.e., its node feature vector containing the CPU utilization and payload emitted from this node. This node embedding distinguishes edge directions by its two sub-vectors, $\mathbf{h}_{v_\vdash}^k$ and $\mathbf{h}_{v_\dashv}^k \in \mathbb{R}^m$, reflecting the effects of the upstream neighbors $\mathcal{N}_\vdash(v)$ and downstream neighbors $\mathcal{N}_\dashv(v)$. The edge $e_{u,v}$ has an edge feature vector $\mathbf{f}_{u,v}$, which contains the weights of transmission loads along this edge. Our GNN updates v's embedding with the following edge-aware steps, illustrated in Figure 2:

• **Information aggregation:** First, we aggregate the information from v's upstream and downstream neighbors separately. Taking the aggregation of upstream neighbors as an example,

for each $u \in \mathcal{N}_{\vdash}(v)$, we take its current embedding $\mathbf{h}_{u\vdash}^{k}$ and edge feature $\mathbf{f}_{u,v}$, feed them to a non-linear transformation

$$\mathbf{h}_{u\vdash}^{(\text{in})} = \tanh(\mathbf{W}_{\mathbf{1}}^{(\mathbf{up})}\mathbf{h}_{u\vdash}^{k} + \mathbf{W}_{\mathbf{edge}}^{(\mathbf{up})}\mathbf{f}_{u,v}),$$

where $\mathbf{W}_{\mathbf{1}}^{(\mathbf{up})} \in \mathbb{R}^{m\times 2m}$, $\mathbf{W}_{\mathbf{edge}}^{(\mathbf{up})} \in \mathbb{R}^{m\times d}$ and d is the dimension of edge features. Similarly, we aggregate the *downstream* information as:

$$\mathbf{h}_{u\dashv}^{(\text{out})} = \tanh(\mathbf{W}_{\mathbf{1}}^{(\mathbf{down})}\mathbf{h}_{u\dashv}^{k} + \mathbf{W}_{\mathbf{edge}}^{(\mathbf{down})}\mathbf{f}_{v,u})$$

• **Node update:** After getting all $\mathbf{h}_{u}^{(\text{in})}, \forall u \in \mathcal{N}_{\vdash}(v)$, we take the mean-pooling of the vectors and update the upstream-view embedding of v as ($[\cdot:\cdot]$ refers to vector concatenation):

$$\mathbf{h}_{v_{\vdash}}^{k+1} = \tanh(\mathbf{W}_{\mathbf{2}}^{(\mathbf{up})}\left[\mathbf{h}_{v_{\vdash}}^{k} : \frac{\sum_{u\in\mathcal{N}_{\vdash}(v)}\mathbf{h}_{u\vdash}^{(\text{in})}}{|\mathcal{N}_{\vdash}(v)|}\right]).$$

The downstream-view embedding is obtained similarly.

Empirically, we use shared parameters $\mathbf{W}_{\mathbf{1}}^{(*)}$ and $\mathbf{W}_{\mathbf{2}}^{(*)}$ for upstream and downstream updates. The above steps are repeated K times over all nodes in the graph. In our experiments, we find that setting K to 2 can already obtain good performance. Finally, for each v, we concatenate its upstream and downstream embeddings $\mathbf{h}_{v_{\vdash}}^{K}$ and $\mathbf{h}_{v_{\dashv}}^{K}$ as its final node representation. We denote this vector as $\mathbf{h}_{v}$ for short in the following sections.

B. Edge-Collapsing Prediction

We build our edge representation for the edge-collapsing prediction. For each edge $E_{u,v}$, its information is represented by its two nodes' representations $\mathbf{h}_u$, $\mathbf{h}_v$ and edge feature $f_{u,v}$. For a directed edge $e_{u,v}$, the influences of head node u and tail node v are different. Thus, for node embedding $\mathbf{h}$, matrix multiplication is performed with different weight matrices to obtain the representations of head and tail nodes. The edge representation is obtained via a fully connected network after concatenating the node representations with edge features:

$$\mathbf{h}^{\text{head}} = \mathbf{W}^{(\mathbf{head})}\mathbf{h}, \quad \mathbf{h}^{\text{tail}} = \mathbf{W}^{(\mathbf{tail})}\mathbf{h}$$

$$\mathbf{h}_{u,v} = \mathbf{W}_{\mathbf{1}}^{(\mathbf{merge})}\left[\mathbf{h}_{u}^{\text{head}} : \mathbf{h}_{v}^{\text{tail}} : \mathbf{W}^{(\mathbf{edge})}\mathbf{f}_{u,v}\right].$$

Next, we get the probability of merging two nodes connected by an edge via feeding the edge embedding to a nonlinear transformation:

$$P(\text{merge}(u,v)=1) = \sigma(\text{MLP}(\mathbf{W}_{\mathbf{2}}^{(\mathbf{merge})}\mathbf{h}_{u,v})),$$

where σ is the Sigmoid function. It produces a binary probability of merging nodes u and v for each edge $e_{u,v}$.

C. Curriculum Learning for Large Stream Graphs

Graph coarsening is more challenging with larger graph sizes, as larger graphs correspond to a larger search space of coarsening operations for any coarsening approach. Hence, during the cold-start stage, i.e., the beginning of the training process, it is likely that all the samples give low rewards so that none of them can provide good training signals. To address this problem, we propose two curriculum learning methods to guide our model training and help the model reach convergence in a few epochs (1∼3) with high throughputs.

Curriculum Based on the Levels of Graph Sizes. Curriculum learning [17] mimics the education processes of humans in the real world, where learning problems come in a sequence from easy to difficult. In practice, the learning model is firstly fed with easier tasks; then, starting with the local optima, the models are further trained on harder tasks to adapt to a better local optimum compared to training from scratch.

Specifically, in our work, we split tasks according to graph sizes and numbers of devices. This gives three curriculum levels, i.e., graphs with numbers of nodes between 100∼200 on 10 devices, 400∼500 on 10 devices, and 1,000∼2,000 on 20 devices, as shown in Figures 5 and 6. For curriculum learning, we start with 100∼200 on 10 devices instead of 5 devices, as this setting is not hard for our model to learn from scratch. The model is first trained on the first level directly. Because both the training and testing graphs from the previous level can be viewed as the (auxiliary) training data for the next task, we then use the model obtained for the previous level to continue training (i.e., fine-tuning) this model for the next level. On each level, the model is trained until it achieves its best performance. Empirically, compared to training the model from scratch, this approach helps the model converge much faster (1∼3 epochs) and reach much higher performance.

Metis-Guided Training Signals. Our second approach is to introduce Metis predictions to the sampled partitions in REINFORCE. Metis can provide partitions regardless of graph sizes. Though not optimal, these allocations are still more informative compared to the random allocations from the model at the start of the training. Therefore, they can provide meaningful supervision signals for the model to quickly pass the cold start stage, especially in cases where curriculum learning using smaller graphs is not possible.

We process our training graphs with Metis before the model training. These Metis partitions are added to the initial buffer of best samples and used in the same way as the regular historically-found best samples during policy gradient training. Once the model has explored enough higher throughput partitioned graphs, the Metis-guided samples will be removed from the beam and no longer affect model optimization.

To make use of Metis partitions, one obstacle to overcome is to infer which edges have been collapsed by Metis, as the algorithm does not decide for every edge whether to merge and only gives the coarsened graphs. We apply the "maximum spanning tree" algorithm to find out the collapsed-edge list. For every original subgraph with n connected operators that can map to a new operator in the Metis partition graph, we choose the top $n-1$ edges of the highest edge weight as the collapsed-edge list while guaranteeing it is a spanning tree.

V. Experimental Settings

We first introduce our evaluation benchmarks.

Data Set Generation. In addition to the data set with small stream graphs in [9], we generate 1,500 graphs with 100∼200

nodes per graph, 1,100 graphs with 400∼500 nodes per graph, and 1,500 graphs with 1,000∼2,000 nodes per graph. For each data set, we randomly select 300 graphs for testing and the remaining graphs for training.

The graphs are generated to resemble the topological structures of real-world applications in stream processing systems. Specifically, in most real-world stream applications with complicated topological structures, their processing logic can be recursively decomposed into simpler processing logic. Following research on applications of stream processing systems [19]–[24], we summarize three basic types of stream subgraphs — linear, branch, and fully connected structures, as shown in Figure 4. The more complicated processing logic, such as linear chains, loops, trees, and multi-stages, can all be represented as the combinations of the basic structures.

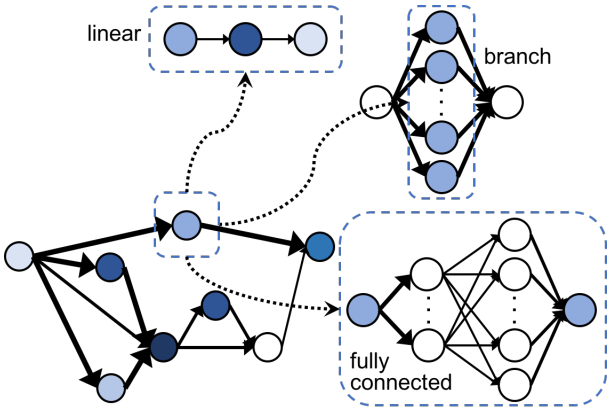


Fig. 4. Illustration of a graph generation. In each step, a newly added node in the graph will be replaced by one of the three basic subgraphs of linear topology, branch topology, or fully connected topology in the probability of 0.45, 0.45, and 0.1 in our settings. For each topology, the maximum lengths are set to 5, 1, and 3; and the maximum widths are 1, 5, and 5. This process will be recursively conducted until the node numbers meet the requirements.

We use the basic sub-graph structures to recursively construct larger stream graphs. In particular, we start with a simple graph and iteratively replace a node with a basic subgraph structure. Figure 4 depicts a step in the recursive graph generation process. In addition to replacing a node with a basic sub-graph, we can replicate a sub-graph for multiple times at its original place. This gives representative stream processing graphs with larger sizes and more complicated topologies.

There are two parameters of graph generation: the workload of nodes, measured by ***CPU utilization***, and payload on edges, measured by ***data saturation rate***. The CPU utilization of an operator is calculated as (IPT·R)/MIPS, where IPT is the number of instructions per tuple, R is the tuple rate of the operator, and MIPS is one million instructions per second. The data saturation rate at an edge is $(P \cdot R)$/BW, where P is the payload and BW is the link bandwidth. For operators generated by replicating a sub-graph, their properties are replicated; and the properties of other operators are randomly assigned. For task settings with different graph sizes, we set the total computing load for each graph in the data set to have the same distribution, to ensure it is within the capacity of devices.

With this construction method, our data set covers various streaming graph topologies, including a wide range of large real-world streaming processing workloads [19]–[24].

Experiments with Graphs Having Different Properties. With a large number of synthetically generated graphs having diverse topologies, we are able to compare and evaluate our framework with the following settings in the experiments.

• ***Small-Graph Setting*** from [9] where the graphs consist of 4∼26 nodes. We take the data set as a sanity check, to show that our approach also works for small graphs.

• ***Medium-Graph Setting*** – Graphs with 100∼200 nodes that we used in the motivating example in Figure 1(b).

• ***Large-Graph Setting*** – Graphs with 400∼500 nodes that are common in stream services, thus is our main setting.[1]

• ***X-Large-Graph Setting*** – Graphs with 1000∼2000 nodes.

• ***Excess-Device Setting*** – We also consider a realistic setting where the number of devices far exceeds the requirement of the stream graphs. Modern stream processing services usually consist of a large cluster of computing devices and support stream graphs with different properties. Assigning nodes to all the available devices may waste computing resources and hurt performance due to increased communication costs. Therefore, a resource allocation strategy needs the ability to find a subset of devices that optimize the throughput. We construct this setting by using the same typologies of the *Large-Graph Setting* but reducing the nodes' CPU utilization and the network bandwidth by 33% and 33%, respectively.

Experimental Environment. Due to limited computing resources, we do not have access to a distributed server cluster or a supercomputer for executing large stream processing systems. Thus, we adopt the simulator *CEPSim* [38], which has been shown to accurately reflect the behavior and relative performance of a real cloud-based stream processing system [9]. We create a cluster in *CEPSim* with homogeneous devices. The computing capacity of a device is 1.25E3 MIPS. The link bandwidth between devices is 1000 Mbps for 100∼200 and 1500 Mbps for 400∼500, 1000∼2000 nodes data sets.

VI. Experiments

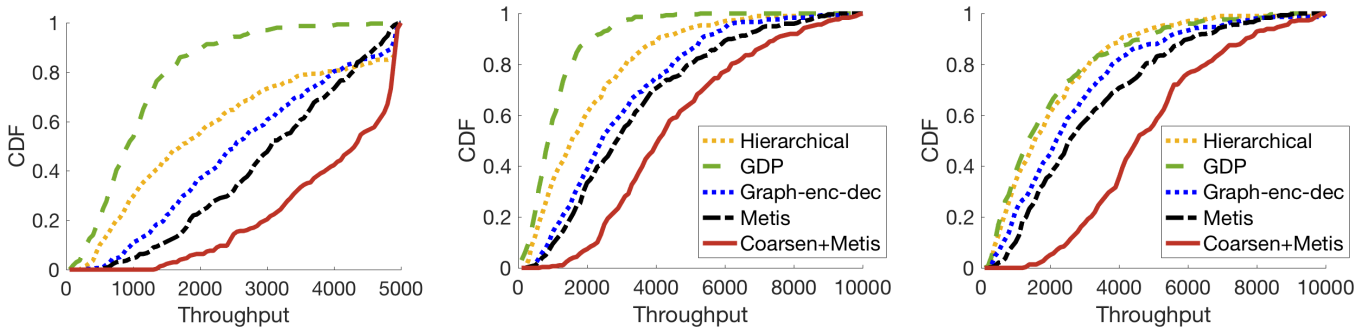
A. Baselines and Implementation Details

Baselines. We compare with the following baselines:

• ***Metis*** [16] is a graph partitioning library, which takes the input graph, the computational cost of operators, the amounts of data flowing through edges, and the number of partitions to produce. We set the number of partitions to the same as the number of available devices. Among its different heuristics, we select the one with the best performance.

• ***Graph-enc-dec*** [9] is the state-of-the-art deep learning approach for resource allocation on stream graphs. The model learns a graph encoder to capture the different graph topology

[1]Unlike a TensorFlow graph where a node is one computing operation to be assigned to one of the CPUs or GPUs, a node in a stream processing graph represents an operator consisting of many computations to be assigned to one of the distributed servers (called devices in this paper). Thus, a stream graph is more coarse-grain and has fewer nodes than a typical TensorFlow graph.



(a) Tuple rate: 5K/s, #Device: 5 (b) Tuple rate: 10K/s, #Device: 5 (c) Tuple rate: 10K/s, #Device: 10

Fig. 5. Throughput Cumulative Distribution Function (CDF) under our approach and various baseline methods in settings with 100∼200 nodes per graph, different tuple rates and available devices. The overall throughput is better if the CDF plot is more skewed towards the right (i.e., smaller Area-Under-Curve).

information and an LSTM decoder to generate the device placements for nodes sequentially.

• ***GDP*** [7] also follows the direct placement paradigm. It uses a graph-encoder followed by a sequence-to-sequence placement network based on Transformer-XL.

• ***Hierarchical*** [6] uses a hierarchical model to assign operators to groups and uses a sequence-to-sequence model for placement on the coarsened graph. We set the number of groups to 25, which has the best performance.

Hyperparameters. The number of hops K in graph embedding is 2. Empirically, we find that $K = 2$ gives the model the best performance. The lengths of node and edge embeddings are 512 and 128, respectively. We train the network for 20 epochs using Adam with a learning rate of 0.001. For the *Large Graphs* and *Extra-Large Graphs*, we perform fine-tuning for 10 and 3 epochs, respectively. At each training step, one graph is fed to the network, and 3 on-policy samples plus up to 3 samples from the memory buffer are taken.

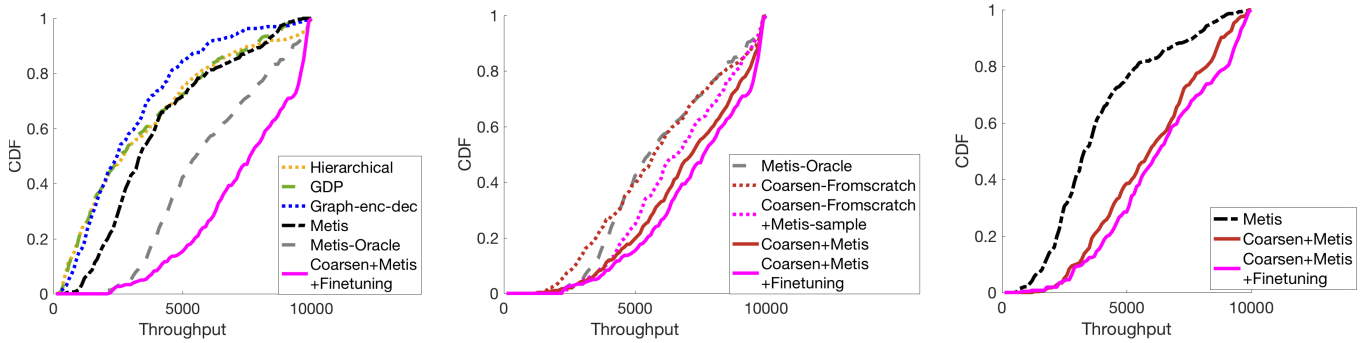
B. Evaluation Results on Various Data Sets

Main Results. We compare our approach with baselines in multiple settings that vary in tuple rates, number of devices, and number of operators per graph. Note that, unlike other systems where an application only runs once and the goal is to minimize its makespan (i.e., runtime), stream processing continuously processes incoming data nonstop, so achieving high throughput is its main performance objective for each stream graph. To compare the generalizability and performance of different resource allocation strategies over a large number of stream graphs, we present the Cumulative Distribution Function (CDF) curve of throughputs. The quantitative and statistical performance comparison is further made by calculating the Area-Under-Curve (AUC) of CDF curves. Here, a smaller AUC means that more stream graphs achieve higher throughput, i.e., better performance. Table I summarizes the performance of different approaches via Area-Under-Curve (AUC) scores, as well as the relative improvement of the best baseline and our framework with regard to Metis. Note that for large graphs (with >100 nodes), Metis becomes the best baseline that beats existing learning-based solutions.

Table I. Area-Under-Curve and relative improvement of our framework with respect to Metis for (Tuple rate, #Device, #node/graph).

	AUC	Imp. wrt Metis
Metis (10K/s, 5 devices, 4∼26 nodes)	1983	-
Graph-enc-dec (*best baseline*)	1021	-
Coarsen+Metis	786	60%
Metis (5K/s, 5 devices, 100∼200 nodes)	1973	-
Coarsen+Metis	1082	45%
Coarsen+Graph-enc-dec	1060	46%
Metis (10K/s, 5 devices, 100∼200 nodes)	6778	-
Coarsen+Metis	5564	17%
Coarsen+Graph-enc-dec	5790	13%
Metis (10K/s, 10 devices, 100∼200 nodes)	6949	-
Coarsen+Metis†	5200	25%
Coarsen+Graph-enc-dec	5481	21%
Metis (10K/s, 10 devices, 400∼500 nodes)	6060	-
Coarsen+Metis (direct prediction with †)	3036	50%
Coarsen+Metis (+curriculum)‡	2681	56%
Coarsen+Metis-oracle (+curriculum)	1974	67%
Metis (10K/s, 20 devices, 1K∼2K nodes)	6167	-
Coarsen+Metis (direct prediction with ‡)	4135	33%
Coarsen+Metis (+curriculum)	3682	40%

First of all, we observed that on the small-graph benchmark [9] used in previous research, i.e., with 4∼26 nodes, our coarsening-based framework already outperforms the previous best learning-based approach (Graph-enc-dec) and becomes the new state-of-the-art. Secondly, when the graphs become larger, while previous learning-based approaches lag behind Metis, our framework is the only one that manages to significantly improve over Metis. Thirdly, when the graphs become further larger (i.e., over 400 nodes), our curriculum learning strategy (see Section IV-C) gives a huge boost on top of our standard framework, leading to a 56% improvement over Metis. Finally, we find that using Metis or Graph-enc-dec for partitioning in our framework does not show much difference (see Table II for details). Intuitively, given well-coarsened small graphs, allocating nodes to devices becomes a simpler



(a) 400∼500 nodes, #Device: 10, baseline (b) 400∼500 nodes, #Device: 10, ablation (c) 1000∼2000 nodes, #Device: 20

Fig. 6. Generalizability of different methods. All learning-based models, except for Coarsen-Fromscratch and Coarsen-Fromscratch+Metis-sample, are trained by data with smaller graphs while all methods are evaluated on larger graphs. In (a) and (b), they are trained by data with 100∼200 operators per graph, 10K tuple rate, and 10 devices, while evaluated by data with 400∼500 operators per graph, 10K tuple rate, and 10 devices. In (c), they are trained by data with 400∼500 operators per graph, 10K tuple rate, and 10 devices, while evaluated by data with 1000∼2000 operators per graph, 10K tuple rate, and 20 devices.

problem where heuristic-based methods can already work well.

Comparison on Medium Graphs. Next, we take a closer look at the performance of different baselines. Figure 5 shows the results in the three settings with 100∼200 nodes from Table I. As discussed earlier, the previous model, Graph-enc-dec, fails to capture the topology information of graphs with such sizes. The Hierarchical and GDP also fail to handle the resource allocation of large graphs well. In all settings with medium graphs, the classical graph partitioning algorithm Metis outperforms all neural baselines; and our proposed coarsening+partitioning framework clearly beats all baselines.

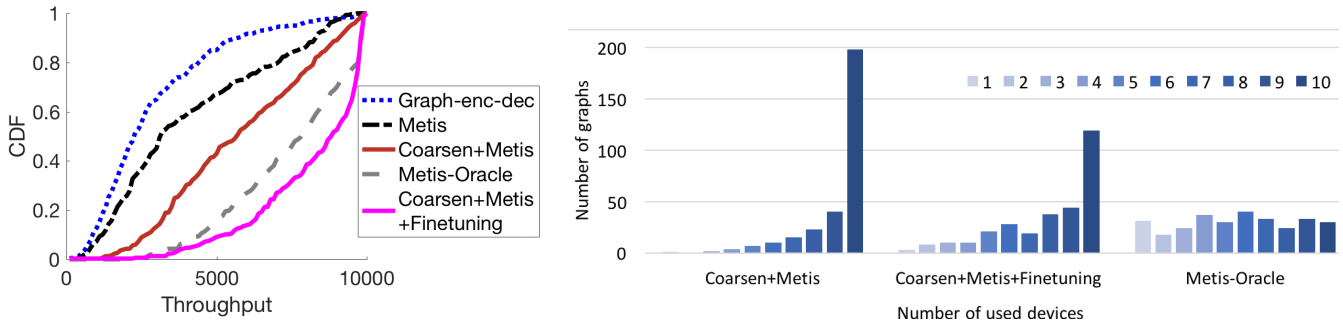
Moreover, it is worth mentioning that Hierarchical follows the same coarsening+partitioning framework but performs badly, despite its success in minimizing the makespan of one very large TensorFlow graph. In addition to the difference between makespan minimization for TensorFlow and maximizing throughput for stream processing, there is a fundamental difference between optimizing the performance of a single graph and learning a generalizable model for unobserved graphs with varying topologies.

Specifically, the general-purpose coarsening formulations, including the one used by Hierarchical, essentially predict each node's cluster label and require the clusters to have predetermined meanings across the dataset. However, the coarsened nodes do not have predetermined meanings in resource allocation. Only when the dataset consists of a single graph, may it be possible to obtain consistent meanings of the coarsened nodes. When applying Hierarchical to multiple graphs, we have tried various extensions to Hierarchical, but none of them improves its performance. For example, to better represent the coarsened nodes, we tried to add the CPU and payload information of the coarsened nodes as features. We also tried to change its loss function to the one used in our framework, but the results remained the same. Similarly, training the Grouper and Placer of Hierarchical together or alternatively does not affect its performance. This result validates that the graph coarsening task in this work is much more challenging and demands a novel formulation and model.

Comparison on Large and Extra Large Graphs. Similarly, Figure 6(a) compares all the methods in the settings with 400∼500 nodes. All learning-based baselines suffer from further performance degradation compared to Metis, while our model with curriculum learning achieves a 56% relative improvement. Figure6(b) further provides an ablation of our curriculum learning method. Our framework trained from scratch still outperforms Metis, showing the strength of our coarsening-partitioning framework. On top of it, our curriculum approach gives further performance boost, with the best result achieved from graph size-based curriculum. Moreover, it is found that even directly applying our model trained on graphs with 100∼200 nodes gives a 50% relative improvement with regard to Metis. This demonstrates the **strong transferability** of our coarsening framework. A similar trend is observed for extra-large graphs, as show in the bottom of Table I and Figure 6(c). The results show that our model can transfer well to the unseen larger graphs, especially with few-iteration adaptation. Such ability can be beneficial when *transferring from simulated to real-platform throughputs*.

Comparison in the Setting with Excess Devices. Finally, we conduct experiments and analysis in a realistic setting where the number of available devices can be larger than the number of devices used in the optimal allocation. Figure 7(a) shows that our framework with curriculum learning far exceeds all the baselines. Moreover, it even significantly outperforms the oracle setting for Metis, where we enumerate all the numbers of used devices for Metis and use the one that achieves the highest throughput (Metis-Oracle). This confirms that our model successfully learns the best number of devices to use and the good resource allocation on these devices adaptively.

Note that when making direct inferences without fine-tuning on the target graphs (Coarsen+Metis), our model is still better than all the baselines but lags behind the Metis-Oracle. This is as expected, because the model trained on small graphs with fully utilized devices and directly transferred to large graphs with excess devices is unaware of the excess resources before fine-tuning for the target setting. Thus, it tends to use more



(a) Throughput CDFs (b) Device usage histogram

Fig. 7. Generalizability of different methods in the *Excess Setting*. All learning-based models are trained by data with 100∼200 operators per graph, 10K/s tuple rate, and 10 devices, while all methods are evaluated by data with 400∼500 operators per graph, 10K tuple rate, and 10 devices.

devices than necessary, as revealed in the histogram of used devices in Figure 7(b). We can see that for Metis, the oracle number of devices varies with graphs. Because Metis cannot produce the best allocation on a given number of devices, it tends to use fewer devices to reduce network traffics. However, this underutilizes the devices and results in lower throughput.

To validate this hypothesis, we calculate the average utilization of devices that are allocated with computation load. Our Coarsen+Metis+Finetuning has an average device utilization (and standard deviation) of 0.12 (0.16) and average bandwidth utilization (and standard deviation) of 0.11 (0.12), while Metis-Oracle has 0.18 (0.21) and 0.16 (0.15), respectively. Both the average utilization and standard deviation of Metis-Oracle are higher than those of Coarsen+Metis+Finetuning, confirming that our method achieves better load balancing.

Finally, we also run the experiment using our model with Metis-oracle, i.e., Coarsen+Metis-oracle (+curriculum) in Table I. Results show that using Metis-oracle can further improve performance compared to using Metis only. Together with Metis-oracle, our framework can produce high-quality allocation using a reasonable number of devices.

C. Performance Analysis

When Our Model Outperforms Metis. We conduct a detailed comparison between our best model, Coarsen+Metis, and the best baseline, Metis, to investigate the sources of our improvement. Figure 8 gives the relation between graph throughputs and their compressed ratios. The ranges of compressed ratios are chosen such that Metis roughly has the same number of graphs in each range. The boxplots show that our model performs better on graphs compressed 4x times or larger. Figure 9 compares the distributions of data saturation rate (i.e., the amount of data flowing through the link over link bandwidth) of coarsened graphs. We can see that more edges of graphs coarsened under our proposed model have lower data saturation rates than those under Metis. This suggests that our model can find the best edges to collapse globally to reduce the network flows and improve throughput.

Ablation of Coarsening Model. As our graph network architecture mainly differs from previous GNNs in the use of edge features, we investigate the results with these features

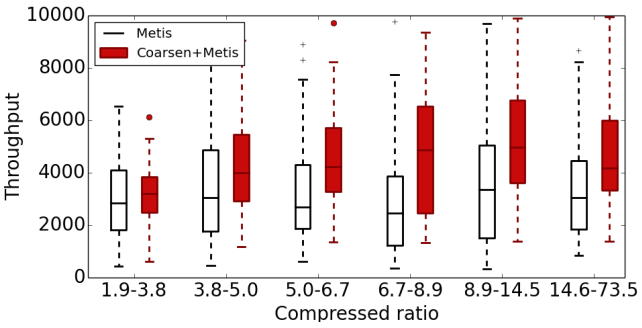


Fig. 8. Impacts of compressed ratios of coarsened graph.

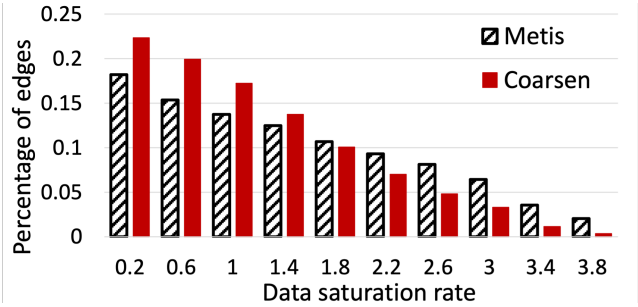


Fig. 9. Amount of data flowing between nodes of coarsened graphs under the coarsening of Metis vs. Coarsening model.

removed from our best-performed model (Coarsen+Metis) in Table II. Removing edge features from the graph-encoding or edge-collapsing modules degrades the performance. It plays a more critical role in the edge-collapsing module.

Ablation of Coarsening-Partitioning Framework. The ***Coarsen-only*** approach directly uses our coarsening model for device placement *without the partitioning model*. It merges graphs with our model until the number of remaining nodes equals the number of devices. This baseline investigates the necessity of our proposed coarsening-partitioning pipeline. As hypothesized in Section IV, our coarsening model works by grouping local sub-graphs that can run on the same device. However, since sub-graphs using the same device can be distributed in different places of the input graph, it is not

Table II. Ablation study with Tuple rate: 5K/s, #Device: 5, 100∼200 nodes. The two ablation results correspond to removing the edge features in the graph-encoding or edge-collapsing modules. Coarsen-only corresponds to removing the partitioning module in our framework.

	AUC	Improv. wrt Metis
Metis	1973	–
Our best model (Coarsen+Metis)	**1082**	**45%**
Our best model w/o edge-encoding	1295	34%
Our best model w/o edge-collapsing	1229	38%
Coarsen+Graph-enc-dec	1060	46%
Coarsen-only	1771	10%
Graph-enc-dec	2332	–

Table III. Average inference time (second) of different methods for datasets with 100∼200 and 400∼500 nodes, 10K/s Tuple rate, and 10 devices using a GeForce RTX 2060 8G GPU.

	100∼200 nodes	400∼500 nodes
Coarsen+Metis	0.286	0.315
Metis	0.001	0.003
Hierarchical	0.178	0.172
GDP	2.486	2.274
Graph-enc-dec	2.306	3.086

suitable to directly merge all these unconnected sub-graphs to a single node. Hence, a partitioning step is necessary.

The result in Table II (last row) confirms our hypothesis. Without the partitioning module, *Coarsen-only* still largely outperforms the Graph-enc-dec model, but only slightly outperforms Metis. This result suggests: (1) the advantage of our novel coarsening-partitioning framework; (2) our graph coarsening formulation is more effective compared to the previous encoder-decoder formulation. The comparison between using Metis and graph-encoder-decoder as the placement model in Table II shows that when introducing our compression model, the selection of the placement model does not result in significant differences. With high-quality coarsened graphs, the placement problem becomes relatively simple.

Qualitative Examples. Figure 3 gives an example where our model predicts superior coarsened graphs. Each dashed circle indicates a merged node. To arrive at the pre-defined coarsening ratio used by its coarsening algorithm, Metis greedily and local-optimally decides edge-collapsing, which leads to cutting off some heavy edges. In contrast, our model is based on global topology information and automatically decides the coarsening ratio. When predicting whether to collapse an edge, the embeddings, which capture wider contexts of nodes and edges, help the model balance the workload in different coarsened nodes and avoid cutting heavy edges.

Complexity and Efficiency of Our Model. The theoretical complexity of our model is $O(Nm^2 + Em^2)$, where N is the number of nodes, E is the number of edges, and m is the size of the hidden states. The computation can run in parallel on GPUs. We measured the average inference time of coarsen model and other baselines in different datasets using an GeForce RTX 2060 8G GPU, as shown in Table III. Note that our model is trained offline. Upon deployment in stream-processing systems, the trained model can directly perform inference (i.e., no online training). The average inference time for 300 graphs, each with 400 to 500 nodes, is 0.315s. In comparison, the heuristic-based Metis is much faster (3ms). However, most stream-processing systems, once deployed, continuously process incoming data flow from the same graph for hours. Thus, the inference overhead is negligible.

Time Cost of Training, Finetuning and CEPSim. During training and finetuning, each epoch (containing 800 different graphs) takes about 108min for graphs with 400∼500 nodes. The major cost of 98min comes from evaluating the allocation using the CEPSim simulator. When training from scratch for graphs with 100∼200 nodes, the model takes 14 epochs to converge. In contrast, when finetuning for graphs with 1k∼2k nodes using the model trained on smaller graphs, the model takes just 3 epochs to converge.

VII. Conclusion

We propose a new coarsening-partitioning paradigm to handle the challenges of resource allocation for large stream graphs. A learnable graph coarsening model is proposed to group the nodes, resulting in a smaller graph suitable for existing graph partitioning approaches. Then, the partitioning module generates the resource allocation for the coarsened graph and maps the allocation back to the original graph. Extensive experiments for various data sets show that our framework has clear advantages over all baseline approaches. In the future, we plan to extend the proposed model to heterogeneous devices and other resource allocation problems.

Acknowledgment

We thank Qiqing Luo for the works that initiate this paper.

References

[1] IBM, "IBM Streams," https://ibmstreams.github.io, 2019.

[2] A. Flink, "Apache Flink," http://flink.apache.org, 2019.

[3] L. Hoang, R. Dathathri, G. Gill, and K. Pingali, "Cusp: A customizable streaming edge partitioner for distributed graph analytics," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 439–450.

[4] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond analytics: The evolution of stream processing systems," in *ACM SIGMOD international conference on Management of data*, 2020, pp. 2651–2658.

[5] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *Proceedings of International Conference on Machine Learning (ICML)*, 2017, pp. 2430–2439.

[6] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *International Conference on Learning Representations*, 2018.

[7] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. C. Ma, Q. Xu, M. Zhong, H. Liu, A. Goldie, A. Mirhoseini *et al.*, "Gdp: Generalized device placement for dataflow graphs," *arXiv preprint:1910.01578*, 2019.

[8] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *Proceedings of the VLDB Endowment*, vol. 11, no. 6, pp. 705–718, 2018.

[9] X. Ni, J. Li, M. Yu, W. Zhou, and K.-L. Wu, "Generalizable resource allocation in stream processing via deep reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 857–864.

[10] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 270–288.

[11] P. Zhang, J. Fang, T. Tang, C. Yang, and Z. Wang, "Auto-tuning streamed applications on intel xeon phi," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 515–525.

[12] Y. Zhou, X. Li, J. Luo, M. Yuan, J. Zeng, and J. Yao, "Learning to optimize dag scheduling in heterogeneous environment," in *23rd IEEE International Conference on Mobile Data Management*, 2022, pp. 137–146.

[13] G. Zhou, W. Tian, and R. Buyya, "Deep reinforcement learning-based methods for resource scheduling in cloud computing: A review and future directions," *arXiv preprint arXiv:2105.04086*, 2021.

[14] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[15] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 1024–1034.

[16] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[17] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 41–48.

[18] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–11.

[19] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[20] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik, "Cola: Optimizing stream processing applications via graph partitioning," in *International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2009, pp. 308–327.

[21] Z. Wang and M. F. O'Boyle, "Partitioning streaming parallelism for multi-cores: a machine learning based approach," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 307–318.

[22] S. Markidis, I. B. Peng, R. Iakymchuk, E. Laure, G. Kestor, and R. Gioiosa, "A performance characterization of streaming computing on supercomputers," *Procedia Computer Science*, vol. 80, pp. 98–107, 2016.

[23] T. Buddhika and S. Pallickara, "Neptune: Real time stream processing for internet of things and sensing environments," in *IEEE international parallel and distributed processing symposium (IPDPS)*, 2016, pp. 1143–1152.

[24] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018.

[25] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," in *26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006, pp. 71–71.

[26] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "EdgeWise: A better stream processing engine for the edge," in *USENIX Annual Technical Conference (ATC)*, 2019, pp. 929–946.

[27] O.-C. Marcu, A. Costan, G. Antoniu, M. Pérez-Hernández, B. Nicolae, R. Tudoran, and S. Bortoli, "Kera: Scalable data ingestion for stream processing," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1480–1485.

[28] P. Rahimzadeh, J. Lee, Y. Im, S.-C. Mau, E. C. Lee, B. O. Smith, F. Al-Duoli, C. Joe-Wong, and S. Ha, "Sparcle: Stream processing applications over dispersed computing networks," in *IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 1067–1078.

[29] M. Danelutto, D. De Sensi, and M. Torquati, "Energy driven adaptivity in stream parallel computations," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2015, pp. 103–110.

[30] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM symposium on operating systems principles (SOSP)*, 2013, pp. 423–438.

[31] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2015, pp. 399–410.

[32] L. Xu, S. Venkataraman, I. Gupta, L. Mai, and R. Potharaju, "Move fast and meet deadlines: Fine-grained real-time stream processing with cameo," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021, pp. 389–405.

[33] S. Munikoti, D. Agarwal, L. Das, M. Halappanavar, and B. Natarajan, "Challenges and opportunities in deep reinforcement learning with graph neural networks: A comprehensive review of algorithms and applications," *arXiv preprint arXiv:2206.07922*, 2022.

[34] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," *arXiv preprint arXiv:1806.08804*, 2018.

[35] H. Gao and S. Ji, "Graph u-nets," in *international conference on machine learning*. PMLR, 2019, pp. 2083–2092.

[36] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3734–3743.

[37] T. Ma and J. Chen, "Unsupervised learning of graph hierarchical abstractions with differentiable coarsening and optimal transport," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021, pp. 8856–8864.

[38] W. A. Higashino, M. A. Capretz, and L. F. Bittencourt, "Cepsim: Modelling and simulation of complex event processing systems in cloud environments," *Future Generation Computer Systems*, vol. 65, pp. 122–139, 2016.

[39] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[40] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[41] H. Chen, B. Perozzi, Y. Hu, and S. Skiena, "Harp: Hierarchical representation learning for networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[42] A. Loukas, "Graph reduction with spectral and cut guarantees." *J. Mach. Learn. Res.*, vol. 20, no. 116, pp. 1–42, 2019.

[43] K. Xu, L. Wu, Z. Wang, M. Yu, L. Chen, and V. Sheinin, "Exploiting rich syntactic information for semantic parsing with graph-to-sequence model," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 918–924.