Communication-Efficient Graph Neural Networks with Probabilistic Neighborhood Expansion Analysis and Caching

Tim Kaler [*, 1, 2] Alexandros-Stavros Iliopoulos [*, 1, 2] Philip Murzynowski [*, 1, 2] Tao B. Schardl [1, 2]
Charles E. Leiserson [1, 2] Jie Chen [2, 3]

Abstract

Training and inference with graph neural networks (GNNs) on massive graphs has been actively studied since the inception of GNNs, owing to the widespread use and success of GNNs in applications such as recommendation systems and financial forensics. This paper is concerned with minibatch training and inference with GNNs that employ node-wise sampling in distributed settings, where the necessary partitioning of vertex features across distributed storage causes feature communication to become a major bottleneck that hampers scalability. To significantly reduce the communication volume without compromising prediction accuracy, we propose a policy for caching data associated with frequently accessed vertices in remote partitions. The proposed policy is based on an analysis of vertex-wise inclusion probabilities (VIP) during multi-hop neighborhood sampling, which may expand the neighborhood far beyond the partition boundaries of the graph. VIP analysis not only enables the elimination of the communication bottleneck, but it also offers a means to organize in-memory data by prioritizing GPU storage for the most frequently accessed vertex features. We present SALIENT++, which extends the prior state-of-the-art SALIENT system to work with partitioned feature data and leverages the VIP-driven caching policy. SALIENT++ retains the local training efficiency and scalability of SALIENT by using a deep pipeline and drastically reducing communication volume while consuming only a fraction of the storage required by SALIENT. We provide experimental results with the Open Graph Benchmark data sets and demonstrate that training a 3-layer GraphSAGE model with SALIENT++ on 8 single-GPU machines is 7.1$\times$ faster than with SALIENT on 1 single-GPU machine, and 12.7$\times$ faster than with DistDGL on 8 single-GPU machines.

1 Introduction

Graph neural networks (GNNs) are an important class of machine learning models that incorporate relational inductive bias for effective representation learning on graph structured data (Li et al., 2016; Kipf & Welling, 2017; Hamilton et al., 2017; Veličković et al., 2018; Xu et al., 2019). These neural networks have been successfully applied in a number of use cases, including product recommendation, traffic forecasting, and financial forensics (Ying et al., 2018; Li et al., 2018; Weber et al., 2019). In many applications, data are continuously collected and the resulting graph grows rapidly, calling for efficient GNN training and inference systems that can scale with the explosive increase of graph data.

This work considers minibatch training and inference with neighborhood sampling in the distributed setting, where vertex data (features) are partitioned across machines. As opposed to full-batch optimization, minibatch optimization is typical for training neural networks, but it poses a unique challenge for GNNs because of the exponential increase of neighborhood size across network layers (Chen et al., 2018). Neighborhood sampling is a popular and effective approach to mitigating this issue (Hamilton et al., 2017; Chen et al., 2018; Ying et al., 2018; Zou et al., 2019; Zeng et al., 2020; Ramezani et al., 2020; Dong et al., 2021). In distributed GNN training, each machine computes the training loss for a minibatch of vertices that are local to the machine's partition. At every step of the training optimization, a set of partition-wise minibatches forms a "distributed minibatch" which is used to update the GNN model parameters. Distributed GNN inference is organized similarly.

Even with neighborhood sampling, minibatch neighborhood expansion creates a communication bottleneck as each machine needs to access remote data for sampled vertices. The communication pattern is stochastic because of the random nature of minibatch and neighborhood sampling. Communication time often dominates the time for training computations. An example of this bottleneck effect is shown in Table 1, which we will walk through shortly.

[*]Equal contribution [1]MIT CSAIL [2]MIT-IBM Watson AI Lab [3]IBM Research. Correspondence to: Jie Chen <chenjie@us.ibm.com>.

Proceedings of the 6th MLSys Conference, Miami Beach, FL, USA, 2023. Copyright 2023 by the author(s).

To overcome the communication bottleneck, we propose an analysis of neighborhood access patterns via vertex inclusion probabilities, as well as a caching policy based on the analysis. We refer to this analysis as "VIP analysis." In contrast to commonly used heuristics for estimating vertex access probabilities — e.g., based on vertex degree and expanded boundary frontiers (Lin et al., 2020), random walks (Dong et al., 2021; Min et al., 2021), or simulated GNN computations (Yang et al., 2022) — VIP analysis estimates access probabilities for all graph vertices based on an analytical model of the actual stochastic neighborhood expansion process. We derive the specific VIP model for the important class of node-wise sampling schemes (Hamilton et al., 2017; Ying et al., 2018; Veličković et al., 2018; Xu et al., 2019; Liu et al., 2020) and find that the resulting caching policy drastically reduces the communication volume in distributed GNNs that employ node-wise sampling with only a modest memory overhead. Moreover, VIP analysis offers a means to organize in-memory data by prioritizing GPU storage for the most frequently accessed vertex features, reducing host-to-device data transfers.

Our system, SALIENT++, is developed over SALIENT (Kaler et al., 2022), an efficient GNN system that achieves state-of-the-art performance through performance-engineered neighborhood sampling, shared-memory parallel batch preparation, and data-transfer pipelining. SALIENT achieves per-epoch time that is nearly equal to the GPU time for model computation alone (effectively hiding the cost of minibatch construction, data transfer, and gradient communication), and it scales well with additional machines. But SALIENT has the drawback of replicating the entire data set on each machine, imposing a hard limit on the data set size. Extending SALIENT to distribute vertex feature data requires addressing the feature communication bottleneck.

SALIENT++ uses the VIP-analysis-driven caching policy and a deep pipeline to achieve scalability and efficiency. It nearly matches the performance of SALIENT with only a fraction of SALIENT's memory requirements. To highlight the efficacy of SALIENT++, Table 1 lists the resulting performance of progressive modifications on top of SALIENT. This example uses the ogbn-papers100M data set (Hu et al., 2020) and partitions the graph using METIS (Karypis & Kumar, 1997) with an edge-cut minimization objective and balancing constraints for the number of training, validation, and overall vertices, as well as the total number of edges, in each partition. Accompanying Table 1, Figure 1 illustrates the key differences between SALIENT and the successive optimizations in SALIENT++ using simplified computation profiles and storage requirement depictions.

We first explore the distributed performance of SALIENT with a disjoint partitioning of feature data across machines. We modify SALIENT to only replicate the graph structure and communicate vertex features on demand and in bulk for each minibatch and its sampled neighborhood. Although distributing the vertex features reduces SALIENT's memory requirement substantially, it also leads to an immediate performance degradation. Specifically, we observe slowdown of roughly 2–3$\times$ on two or four machines and 3.5$\times$ on eight machines relative to SALIENT with full-replication. This slowdown presents itself in the form of high latencies between GNN model computations (row 2 in Figure 1). Overlapping feature communication with other GNN operations (sampling, local feature slicing, host-to-device transfers, training computations, and model updates) through pipelining offers some improvement, but the system remains bottlenecked by communication (row 3 in Figure 1).

Table 1. Per-epoch runtime of a progressively more sophisticated distributed GNN training system on the undirected ogbn-papers100M data set, using a 3-layer GraphSAGE architecture with sampling fanouts (15,10,5) and a hidden layer dimension of 256. For the system with remote feature caching, the size of the cache relative to the size of local-vertex features for each machine was 8% (2 machines), 16% (4 machines), or 32% (8 machines).

	No. of machines			
System	1	2	4	8
SALIENT (Full replication)	20.7s	10.76s	6.02s	3.08s
+ *Partitioned features*	—	33.04s	15.98s	10.85s
+ *Pipeline communication*	—	16.12s	8.73s	5.43s
+ *Feature caching*	—	10.51s	5.45s	2.91s

If we allow a small memory overhead to cache frequently accessed remote features locally, however, the combined effects of reduced communication volume and pipelining boost performance substantially. The sheer reduction of communication volume due to caching makes it possible for pipelining to (nearly) fully hide communication (4th row in Figure 1). On K machines, SALIENT++ reduces the total feature memory footprint of SALIENT by almost a factor of K while matching the performance achieved by SALIENT's full-replication strategy.

In summary, this work makes the following contributions:

1. An analysis of vertex inclusion probabilities (VIP) during GNN neighborhood expansion with node-wise sampling. We show how VIP analysis can be used to minimize the expected communication among machines as well as the host-to-device data transfers within each machine.
2. The design and implementation of SALIENT++, a system for distributed GNN training and inference that is both efficient and scalable. The system is based on two key components: a maximum-likelihood static caching policy for remote and local features based on VIP analysis; and a deep pipeline that overlaps feature communication with other GNN operations, ensuring high utilization

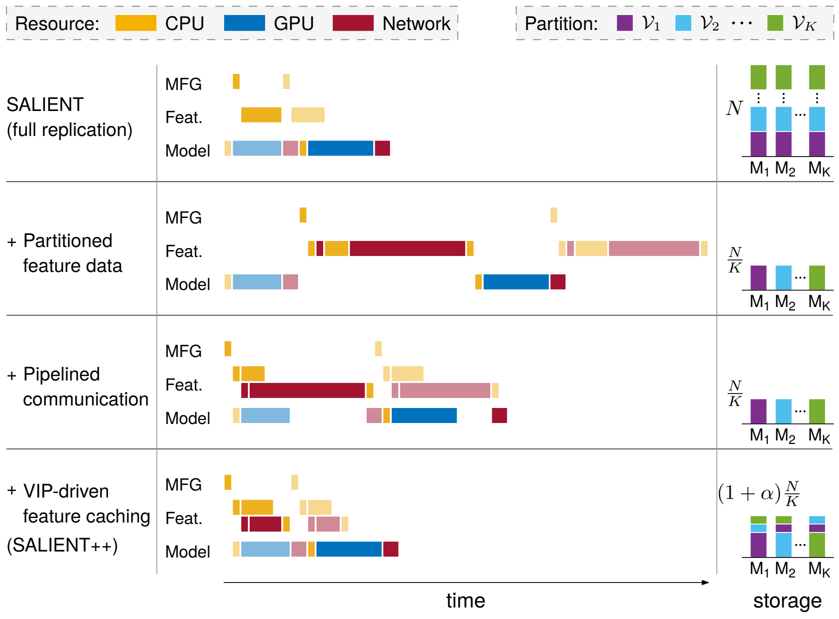

Figure 1. Illustration of distributed GNN computation profiles and feature data storage for the precursor system SALIENT and successive optimizations leading to our system SALIENT++. **Left:** Utilization of system resources (CPU, GPU, and network) in one machine. The processing of each minibatch is broken into three parts. "MFG": message-flow graph construction. "Feat.": local feature tensor slicing, feature communication and joining, and host-to-device transfers. "Model": forward and backward passes, plus communication among machines for model updates. In addition to the highlighted minibatch, the profiles also depict parts of a preceding and a succeeding minibatch in faded colors to show overlap between pipelined operations. **Right:** The color-stacked bars indicate the number of locally stored feature vectors in each of K total machines.

of available network bandwidth.

3. A systematic evaluation of the end-to-end training performance and scalability of SALIENT++ on three large benchmark graphs. On the largest of these data sets, which does not fit in-memory on one machine, SALIENT++ achieves a speedup of $1.75\times$ when scaling from 4 to 8 single-GPU machines and an additional $1.45\times$ when scaling from 8 to 16 single-GPU machines.

2 Background and Related Work

This section briefly recalls background and introduces notation for GNNs and their training/inference. It also discusses related work on systems and distributed training.

2.1 Graph neural networks

This work focuses on the class of *message passing neural networks* (MPNNs) (Gilmer et al., 2017), which encompasses many GNNs of major interest for massive graphs. Let $G = (\mathcal{V}, \mathcal{E})$ be a graph with $\mathcal{V}$ being the vertex set and $\mathcal{E}$ being the edge set. Let $X \in \mathbb{R}^{N\times D}$ be the vertex feature matrix, where N is the number of vertices and D is the feature dimension. A row of X, denoted by $x_v \in \mathbb{R}^D$, is the feature vector of a vertex v. Let $\ell = 0, \ldots, L$ denote the layer index and $\mathcal{N}_1(v)$ be the one-hop neighborhood of v. MPNNs use the following update rule to define a layer:

$$h_v^\ell = \text{UPD}^\ell\Big(h_v^{\ell-1}, \text{AGG}^\ell\big(\{h_u^{\ell-1} \mid u \in \mathcal{N}_1(v)\}\big)\Big), \qquad (1)$$

where h_v^ℓ is the layer-ℓ representation of v, AGG^ℓ is a set aggregation function, UPD^ℓ is a two-argument update function, and $h_v^0 = x_v$. One sees that the representation of a vertex v depends on the past-layer representations of v and its neighbors in $\mathcal{N}_1(v)$; and this dependence is recursive.

GNNs differ in their design of the two functions in (1). For example, in GraphSAGE (Hamilton et al., 2017), AGG^ℓ is a mean, LSTM, or pooling operator, and UPD^ℓ concatenates the two arguments and applies a linear layer. In GIN (Xu et al., 2019), AGG^ℓ is the sum of $\{h_u^{\ell-1}\}$ and UPD^ℓ is the sum of its arguments followed by an MLP. In GAT (Veličković et al., 2018), AGG^ℓ is the identity and UPD^ℓ computes h_v^ℓ as a weighted combination of $W^{\ell-1}h_u^{\ell-1}$ for all $u \in \{v\} \cup \mathcal{N}_1(v)$, where the weights are attention coefficients and $W^{\ell-1}$ is the parameter matrix of the layer.

2.2 Minibatch training

Neural networks can be trained with gradient descent (full-batch training) or stochastic gradient descent methods (minibatch training). A majority of the deep learning literature advocates minibatch training, with theoretical and empirical evidence suggesting that it converges faster, generalizes better, and is more suitable for GPU-oriented computing infrastructures (Bottou et al., 2018).

Minibatch training incurs a unique challenge for GNNs, often termed the "neighborhood explosion" problem: to compute the training loss for a vertex v, equation (1) indicates that it requests information from the neighborhood recursively, growing its size exponentially with the number of layers. For a reasonably sized minibatch, the multi-hop neighborhood quickly covers a large portion of the graph, incurring a prohibitive time cost that eclipses the saving in convergence speed. Moreover, the storage requirement easily exceeds the memory capacity of a GPU.

2.3 Neighborhood sampling

Restricting the neighborhood size by sampling is an effective method for tackling the neighborhood explosion problem. Three major types of sampling strategies are node-wise sampling (Hamilton et al., 2017; Ying et al., 2018), layer-wise sampling (Chen et al., 2018; Zou et al., 2019), and subgraph sampling (Chiang et al., 2019; Zeng et al., 2020).

Node-wise sampling approaches modify the neighborhood for each vertex v separately by taking a random subset containing at most f neighbors (called the *fanout*). Layer-wise sampling approaches collect the neighbors of all vertices in a minibatch and sample the neighborhood from their union. Such sampling proceeds recursively layer by layer and can use a nontrivial sampling distribution to control pre-activation variance while preserving unbiasedness. Nonlinear activation functions destroy unbiasedness anyway, but training convergence results can still be established based on asymptotic consistency (Chen & Luss, 2018). Subgraph sampling approaches sample a connected subgraph and compute the minibatch loss restricted to this subgraph. This work focuses on node-wise sampling for its widespread use.

2.4 Minibatch inference

Similar to training, inference can be performed in either full batches or minibatches. Whereas the choice between full batch or minibatch training centers on convergence and generalization, the choice made for inference depends on the computational pattern and implementation effort. Full-batch inference avoids neighborhood sampling but requires a separate implementation of the forward pass for different GNN architectures. On the other hand, minibatch inference can reuse the forward function for training, improving productivity for application developers and architecture designers, but the stochastic nature of neighborhood sampling may return different predictions and does not guarantee the same accuracy as the case of not performing sampling. Nevertheless, the authors of SALIENT demonstrate strong empirical results that show that prediction accuracy is not compromised with a reasonable choice of the fanouts (Kaler et al., 2022). This work follows the practice of minibatch inference.

2.5 Related work

Due to the computational pattern of neighborhood aggregation, GNN training systems on massive graphs differ substantially from those for usual neural networks in design and implementation. Many GNN systems are developed based on full-batch training for simplicity, which avoids the complication of neighborhood sampling. Examples include NeuGraph (Ma et al., 2019), Roc (Jia et al., 2020), DeepGalois (Hoang et al., 2021), FlexGraph (Wang et al., 2021a), Seastar (Wu et al., 2021), GNNAdvisor (Wang et al., 2021b), DistGNN (Md et al., 2021), and BNS-GCN (Wan et al., 2022). Some of these systems are built on common deep learning frameworks, such as TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2019), while others are built on self-developed programming models.

Examples of systems that perform minibatch training include DistDGL (Zheng et al., 2020), Zero-Copy (Min et al., 2021), GNS (Dong et al., 2021), and P^3 (Gandhi & Iyer, 2021). However, these publications report results on either multiple machines with only CPUs or a single machine with one or multiple GPUs. A newer version of DistDGL (DistDGLv2 (Zheng et al., 2021)), SALIENT (Kaler et al., 2022), and our system SALIENT++ demonstrate results in the distributed multi-GPU setting.

SALIENT++ employs edge-cut graph partitioning to distribute data across machines. An alternative approach, adopted by DistGNN (Md et al., 2021), is vertex-cut partitioning, which ensures each edge is local to some machine. A drawback of this approach is that a vertex may be assigned to multiple machines, leading to memory overhead due to feature replication that is reported to be as high as $5\times$. SALIENT++, on the other hand, achieves high distributed performance with less than 50% memory overhead due to caching in our experiments.

SALIENT++ employs a caching policy to reduce communication and data transfer. We introduce a policy based on an analysis of vertex access probabilities during neighborhood sampling. Although caching has been explored by several other systems in this context, previous caching policies are heuristic, using proxy measures such as vertex degrees (Lin et al., 2020), random walks (Dong et al., 2021; Min et al., 2021), or simulated access frequencies (Yang et al., 2022). A complementary approach to reducing communication time is taken by DGCL (Cai et al., 2021), which uses a communication-planning algorithm to optimize the communication pattern for a specific network topology.

Marius and MariusGNN (Mohoney et al., 2021; Waleffe et al., 2022) are out-of-core training systems that work on a single machine by exploiting external memory. They operate in a different setting from ours, but our vertex inclusion probability analysis may be used in complement to optimize on-disk data representation and disk I/O.

3 VERTEX INCLUSION PROBABILITIES FOR GNNS WITH NODE-WISE SAMPLING

This section describes a principled approach to estimating and reducing data access costs in distributed GNNs with node-wise sampling. Node-wise sampling was notably introduced in GraphSAGE (Hamilton et al., 2017) and is used with a variety of GNN architectures (Ying et al., 2018; Veličković et al., 2018; Xu et al., 2019; Liu et al., 2020). We first introduce ***vertex-inclusion probability*** (***VIP***) analysis to calculate the probability that any vertex will be present in the sampled L-hop expanded neighborhood of a minibatch in some partition. VIP analysis takes into account the distinctive structure of neighbor accesses in GNNs with minibatches and node-wise sampling. We then demonstrate that VIP analysis enables a simple but highly effective strategy for reducing data movement during GNN computations.

The VIP model for node-wise sampling derived in this section does not apply to other sampling schemes, such as layer-wise or subgraph sampling. It may be possible to derive VIP models for these other sampling schemes in a similar way, but this is outside our scope.

3.1 Analysis of vertex inclusion probabilities in L-hop neighborhoods with node-wise sampling

We analyze the following random process, which models neighborhood expansion in GNN architectures with node-wise sampling. (i) Start from a set of vertices rather than a single vertex. (ii) For each vertex in the starting set, sample a set of direct neighbors without replacement, thereby expanding the walk frontier. (iii) Using the union of sampled neighborhoods as the new starting set, repeat step (ii). The process terminates after L repetitions (i.e., hops). We assume that sampling is independent across hops and across vertices in the current-hop set. That is, although each vertex samples among its direct neighbors without replacement, different vertices may sample the same direct neighbor.

We seek to estimate the vertex inclusion probabilities, or VIP values, in expanded neighborhoods that are obtained per the above random process. The VIP values form a vector of probabilities $p(u)$ over all graph vertices. We can express $p(u)$ via hop-wise VIP vectors $p^{[h]}(u)$ that contain the probabilities that any vertex u is sampled exactly h hops away from the starting set. (The hop-h neighborhood contains the input vertices for the ℓ-th GNN layer, $\ell = L - h$.) A vertex u is *not* sampled at hop h only if, for each of its neighbors $v \in \mathcal{N}_1(u)$, either v was not sampled at hop $h-1$ or v was sampled but did not pick u among its neighbors. The following Proposition shows how to propagate the initial-set probabilities through the graph to compute the VIP values. We give a proof in Appendix C.

Proposition 1 (Vertex inclusion probabilities in minibatch neighborhood expansion with node-wise sampling)**.** *The probability that vertex $u \in \mathcal{V}$ appears in the node-wise sampled L-hop expanded neighborhood of any minibatch is*

$$p(u) = 1 - \prod_{h=1}^{L} \left(1 - p^{[h]}(u)\right), \tag{2}$$

$$p^{[h]}(u) = 1 - \prod_{v \in \mathcal{N}_1(u)} \left(1 - t_h(u,v)\, p^{[h-1]}(v)\right), \tag{3}$$

where $p^{[0]}(v)$ is the probability that v is in the minibatch and $t_h(u,v)$ is the transition probability that v samples u as a neighbor at hop $h = 1, \ldots, L$.

The VIP model of Proposition 1 applies to any initial sampling and hop-wise transition probability function for node-wise sampling. For example, if minibatches and vertex-wise neighbors are sampled uniformly at random without replacement as in GraphSAGE, then we have: $p^{[0]}(u) = B/|\mathcal{T}|$ if $u \in \mathcal{T}$ and 0 otherwise, where B is the minibatch size and $\mathcal{T}$ is the set of training vertices; and $t_h(u,v) = \min\{1, f_h/d(v)\}$ if $u \in \mathcal{N}_1(v)$ and 0 otherwise, where $d(v)$ is the (outgoing) degree of v. Non-uniform neighbor sampling models are accommodated via the corresponding transition probability matrix or matrices.

The neighborhood expansion random process parametrizes a continuum between a random walk and full neighborhood expansion. If the initial set size and layer-wise fanouts are equal to 1, it becomes a random walk. If the fanouts are greater than the max in-degree of graph vertices, it becomes a full neighborhood expansion. The VIP model for the above two special cases is linear, while the generalized model in Proposition 1 is nonlinear; nonetheless, these models all have the same computational complexity: $O(L(M+N))$.

3.2 Vertex feature caching

We now show how to embody VIP analysis into a caching policy for minimizing communication among distributed machines and host-to-device data transfers in each machine.

Communication reduction We consider the optimal static caching policy, in the maximum-likelihood sense, for reducing inter-partition communication volume in distributed GNNs. The policy is straightforward: each machine extends its local vertex feature storage with copies of the remote features that are most likely to be accessed by the machine, thereby minimizing the total expected communication volume. For some given initial partitioning — and making no assumptions about the order in which minibatch and neighboring vertices are sampled — this policy is directly related to the VIP analysis. Specifically, we calculate partition-wise VIP values $p_k(u)$ from the corresponding initial probabilities $p_k^{[0]}(u)$. The cache contents for the k-th partition are then determined by ranking the remote vertices in order of decreasing $p_k(u)$.

We measure the size of a remote-feature cache by its corresponding ***replication factor*** α. The replication factor is defined such that the number of cached feature vectors stored in each machine is $\alpha N/K$, where K is the number of partitions or machines. That is, the fraction of cached to local vertices for each partition is α, and each feature vector is stored in $(1+\alpha)$ machines on average. We may say that a

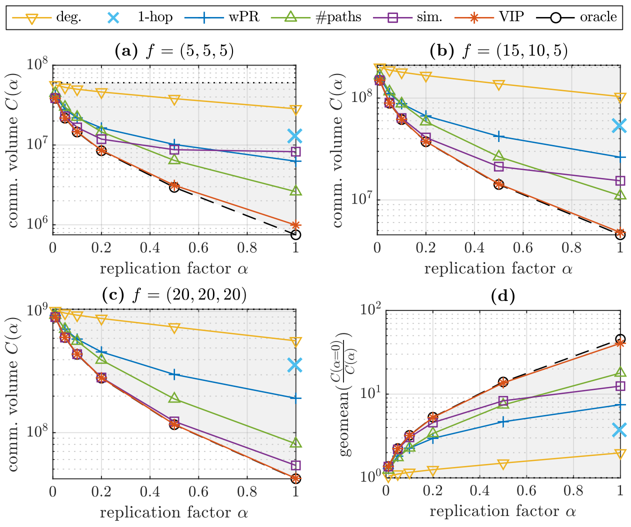

Figure 2. Comparison of caching policies with respect to remote feature communication volume. GNN: 3-layer GraphSAGE, varying fanouts f, minibatch size 1024. Data set: ogbn-papers100M (undirected). Partition: 8-way METIS with edge-cut objective plus edge and training/validation/test vertex balancing. **(a)**–**(c)** Average per-epoch communication in number of vertices over 100 epochs (log-scale). Lower is better. The feasible region between the communication upper bound (no caching) and lower bound (oracle caching) is shaded gray. **(d)** Geometric-mean improvement across fanouts, relative to no caching (log-scale). Higher is better.

partition or cache replicates a remote vertex to mean that it copies the corresponding vertex feature data.

Figure 2 compares a number of static feature-caching policies and provides evidence that the analytical VIP model of Proposition 1 effectively yields the optimal policy. The specific policies being compared are as follows: "deg." ranks vertices by degree, after filtering out remote vertices that are not reachable from a partition's training set (Lin et al., 2020); "1-hop" replicates the entire 1-hop halo of each partition; "wPR" ranks vertices by their score after 5 iterations of a weighted reverse-PageRank model (Min et al., 2021) with damping factor 0.85; "#paths" ranks vertices by the number of paths with length $\leq L$ that reach them from any local training vertex; "sim." ranks vertices by empirical VIP estimates, measured by counting vertex-wise frequencies of access over 2 simulated training epochs (Yang et al., 2022); "VIP" ranks vertices by VIP values as per Proposition 1; and "oracle" retroactively ranks vertices by their actual access frequencies after training, providing a lower bound on communication volume.[1] We make the following observations.

- *Optimality.* The analytical VIP policy yields near-optimal communication volume (within 5% of "oracle") across fanouts and replication factors. The bigger gap ($\sim$30%) for $f = (5, 5, 5)$ and $\alpha = 1.0$ is due to variance in the accesses of lower-ranked vertices; the gap narrows with more samples (e.g., bigger fanouts or more epochs).
- *Communication reduction efficacy.* VIP-driven caching is highly effective for reducing communication volume. Compared to no caching, the VIP policy achieves a geometric-mean reduction of $2.2\times$–$5.3\times$ with small replication factors $\alpha \in [0.05, 0.20]$ and more than $10\times$ with replication factors over 0.50. Compared to the other caching policies, it achieves consistently lower communication volume and its relative improvement increases with the replication factor.
- *Local information is not sufficient.* The high-degree and 1-hop halo policies, which do not take neighborhood expansion into account, offer only marginal improvements over no caching. Caching multi-hop neighbors will reduce communication further (Lin et al., 2020) but it will also blow up the replication factor.
- *Impact of tailoring the model to the expansion process.* The VIP-driven policy is up to $4\times$ and $2\times$ more effective than the "wPR" and "#paths" policies, respectively. Both of these policies take graph-structural expansion into account, but "wPR" is agnostic to the GNN fanout and number of layers, while "#paths" does not directly account for sampling.
- *Benefits and limitations of empirical estimation.* The empirical and analytical VIP estimates ("sim." and "VIP" policy, respectively) yield comparable results for low replication factors and high fanouts, but the empirical estimates become less effective as the replication factor increases or the fanouts decrease. For replication factors 0.50 and 1.0, the relative aggregate improvement of the analytical over the empirical policy is $1.6\times$ and $3.2\times$, respectively. The empirical approach has the benefit that it can be used for any sampling scheme, but it also requires increasingly many samples to accurately estimate VIP values for infrequently accessed vertices.

Host-to-device data transfer reduction VIP analysis can also be used to reduce the volume of host-to-device data transfers for local features on each machine. For example, assume that each GPU can retain a fraction of the local feature data in memory throughout the GNN computations (i.e., across all minibatches). We may rank the local vertices in the k-th partition by $p_k(u)$ and keep the highest-ranking-vertex features on the GPU. As with remote-vertex caching, this policy minimizes the expected data transfer volume due to local vertices in each partition.

[1]With all caching policies, we calculate vertex rankings with respect to each partition. This is different from the original setting in which some of these policies were introduced, where vertices are cached based on a single, global ranking score.

4 SALIENT++: FAST AND SCALABLE DISTRIBUTED GNN TRAINING VIA CACHING AND PIPELINING

This section describes the design of SALIENT++, a fast and scalable system for performing distributed minibatch training of GNNs on large partitioned datasets. The design of SALIENT++ is comprised of three majors components:

Vertex reordering A vertex ordering based on the graph partitions and VIP values that facilitates efficient sub-partitioning of local feature data between CPU and GPU to reduce host-to-device data transfers.

Data replication An economical data replication strategy that uses VIP analysis to reduce communication volume for fetching remote features while using only a small amount of additional memory.

Pipelined communication A pipelined distributed mini-batch preparation system that hides latencies due to sampling, data transfers, and inter-machine communication of remote feature data.

These three techniques together enable SALIENT++ to perform GNN training on partitioned data-graphs while matching the efficiency of highly optimized GNN training systems that perform distributed computations with full replication of vertex feature data across all machines. This enables SALIENT++ to perform GNN training efficiently on large datasets where full replication is impractical.

As we shall discuss in our empirical evaluation in Section 5, SALIENT++ achieves high performance for GNN training on large datasets even when operating in clusters with only modest hardware configurations.

4.1 Partitioning and reordering of vertex features

Let us describe how SALIENT++ partitions the vertex features of a graph to reduce inter-machine and host-to-device communication. SALIENT++ employs a two-level strategy for distributing the vertex features across machines and devices, as illustrated in Figure 3. The top level involves distributing vertex features based on a vertex partitioning of the graph, and the bottom level involves dividing each partition's local vertex features between GPU and CPU memory.

SALIENT++ operates on graph data sets that are partitioned using a graph partitioning algorithm such as METIS (Karypis & Kumar, 1997). The graph partitions are generated such that the training, validation, and test vertices are balanced. As a heuristic to balance work-per-partition, an additional criterion is used to balance the amount of edges per partition. Each machine is assigned a subset of training/validation/test vertices and stores vertex feature data locally according to the machine's partition.

The graph is reordered such that: (a) indices for vertices in

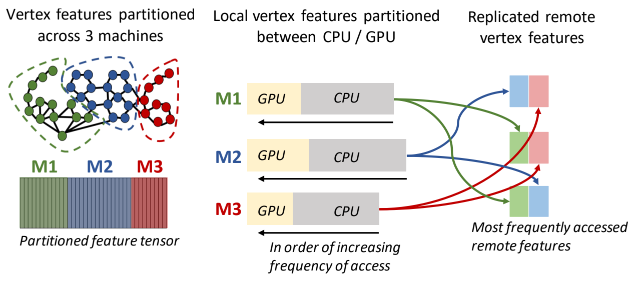
Vertex features partitioned across 3 machines
Local vertex features partitioned between CPU / GPU
Replicated remote vertex features
M1
M2
M3
Partitioned feature tensor
GPU
CPU
In order of increasing frequency of access
Most frequently accessed remote features

Figure 3. Illustration of the partitioned data set with VIP-driven local ordering and caching of remote features across machines.

the same partition are contiguous; and (b) vertices within a partition are ordered based on how beneficial it is for them to be stored on the GPU. Each machine stores a prefix of its ordered list of vertex features on the GPU, which when using VIP ordering corresponds to those vertices that are accessed most frequently by the local machine. This partitioning structure facilitates efficient determination of whether a vertex is remote or local, where a local vertex is stored on the machine, and index calculations using only a constant amount of additional memory. Figure 3 provides a graphical illustration of the VIP-reordered partitioned dataset.

4.2 Replication of remote vertex features

Each machine maintains a replica or static cache of features for vertices in remote partitions, where the size of the cache is determined by a given replication factor α. The vertices whose features are cached in each machine are determined by their VIP rank to reduce network communication, as described in Section 3.2.

Once the neighborhood sampling code prepares a batch, SALIENT++ partitions the vertices in the expanded neighborhood into local and remote vertices. Among the subset of remote vertices, an efficient lookup is performed using a hash table to determine whether the features for a remote vertex can be found in the local machine's cache. Only remote vertices that are not resident in the local cache proceed to be requested from remote machines.

4.3 Minibatch preparation pipeline

SALIENT++ implements a deep multi-stage pipeline which enables overlap between neighborhood sampling, host-to-device data transfers, network communication between machines, and training computation. SALIENT++'s pipeline is more complex than the one used in SALIENT (Kaler et al., 2022), but it maintains the latter's usability and efficiency. SALIENT++'s pipeline can be used in place of standard data loaders for GNNs by changing just a few lines of code.

Sampling and slicing SALIENT++ uses a modified version of SALIENT's optimized neighborhood sampling and slicing code to prepare minibatches using shared-memory parallelism. The original SALIENT batch-preparation code fused the neighborhood sampling and feature tensor slicing operations. In SALIENT++, these tensor-slicing operations are only performed on the subset of the vertices in the sampled neighborhood that are stored locally in the CPU memory of each machine. For the remaining vertices in the sampled neighborhood, SALIENT++ distinguishes between vertices in the following categories: vertices belonging to the local partition whose features are stored on GPU; remote vertices that are replicated on the local machine; and remote vertices that belong to other partitions.

Feature collection pipeline We briefly summarize the pipeline operations. The expanded neighborhood of each minibatch may include remote vertices, remote vertices whose features were cached on the local machine following VIP analysis, and local vertices which are split across CPU and GPU storage. Immediately following sampling, vertices are categorized by their storage location so that vertex features are properly retrieved. Communication rounds interspersed with host-to-device and device-to-host transfers are enqueued with remote machines to coordinate retrieval of remote features. Replicated and local features partitioned across CPU and GPU storage are sliced locally. Finally, all features are concatenated into a single tensor and reordered for compatability with the message-flow graph generated during neighborhood sampling.

A total of 10 mini-batches can be "in-flight" in the SALIENT++ pipeline at any time, each mini-batch being processed by a separate stage of the pipeline. Pipelining enables overlap between host-to-device data transfers and network communication, and it also allows for hiding latencies related to CPU-GPU data transfers and inter-machine communication. A more detailed stage-by-stage description of SALIENT++'s pipeline is provided in Appendix D.

5 Evaluation

In this section, we evaluate the performance of SALIENT++ and investigate the impact of its optimizations relating to pipelining and use of VIP analysis (as developed in Section 3). Additionally, we report end-to-end performance results, model accuracies, and contextualize our performance in relation to existing distributed GNN training systems.

5.1 Experimental setup

Computing environment Our experiments were run on a cluster consisting of 16 machines, each equipped with a 16-core (2-way hyperthreaded) AMD CPU with 128GB DRAM and 1 NVIDIA A10G GPU with 24GB memory. The cluster was configured using the Amazon Web Services (AWS) ParallelCluster software and employed fleets of AWS *g5.8xlarge* instances. The network SLA specified for this instance type is 25Gbps. Distributed communications were implemented via PyTorch's DistributedDataParallel module with the NCCL backend. Since our cluster comprises machines with 1 GPU each, experiments with K GPUs involve K separate machines communicating over the network.

Table 2. Summary of data sets. Edge counts reflect the number of edges in the graph after standard preprocessing (e.g., making the graph undirected). The *mag240c* data set is the papers-to-papers citation subgraph of the *lsc-mag240* heterogeneous graph data set.

Data Set	*#Vertices*	*#Edges*	*#Feat.*	*Train. / Val. / Test*
products	2.4M	123M	100	197K / 39K / 2.2M
papers	111M	3.2B	128	1.2M / 125K / 214K
mag240c	121M	2.6B	768	1.1M / 134K / 88K

Table 3. GNN architecture hyperparameters for all experiments in Section 5. Fanout is for training. Batch size is per GPU.

Data Set	*GNN*	*#Layers*	*Hidden Dim.*	*Fanout*	*Batch*
products	SAGE	3	256	(15, 10, 5)	1024
papers	SAGE	3	256	(15, 10, 5)	1024
mag240c	SAGE	2	1024	(25, 15)	1024

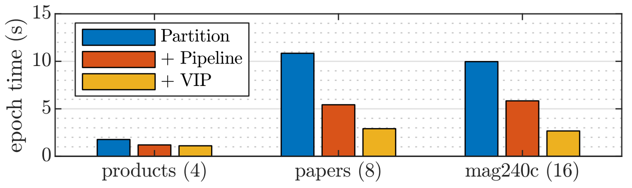

Figure 4. Impact of pipelining and VIP optimizations in SALIENT++. Illustrates the performance improvements obtained by successively more optimized versions of SALIENT++ for *products* (4 partitions), *papers* (8 partitions), and *mag240c* (16 partitions). The GNN architectures for each benchmark are listed in Table 3. The replication factors for VIP-based caching were 0.16, 0.32, and 0.32 for *products*, *papers*, and *mag240c*, respectively.

Datasets We evaluate SALIENT++ on three standard benchmark data sets: ogbn-products (*products*), ogbn-papers100M (*papers*) (Hu et al., 2020), and lsc-mag240 (*mag240c*) (Hu et al., 2021). The graph and training set in these data sets vary in size, with *papers* and *mag240c* being two of the largest open benchmarks at the time of this work. See Table 2 for detailed information. All graphs were made undirected (if originally not), as is common practice. For *mag240c*, we train on the homogeneous papers-to-papers component of the graph.

GNN architectures We evaluated SALIENT++'s performance using standard GraphSAGE (Hamilton et al., 2017) architectures with each data set's most commonly used hyperparameters. Table 3 lists the GNN architectures and hyperparameters that were used for each data set.

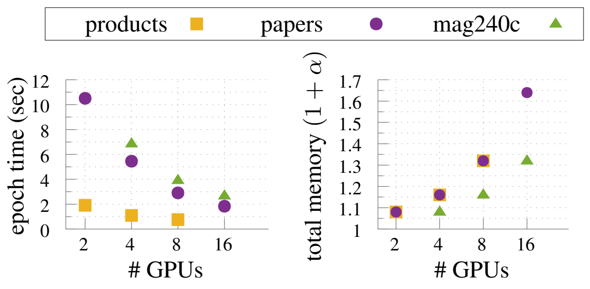

Figure 5. Scalability and total memory used by all machines for *products*, *papers*, and *mag240c*. Total memory is plotted as a multiple of the unreplicated data set size $1 + \alpha$ where α is the replication factor used for the K-GPU execution.

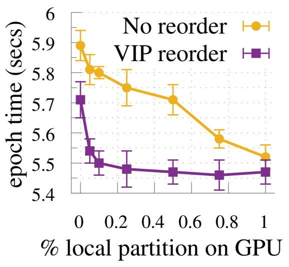

Figure 6. Impact of VIP-based local vertex ordering on per-epoch runtime. Experiment run on 4 GPUs with the *papers* dataset and replication factor $\alpha = 0.15$. Reported numbers are the mean over 10 epochs and vertical lines show the standard deviation.

5.2 Performance analysis of SALIENT++

We analyze the performance of SALIENT++ by investigating the impact of its component optimizations, and measuring end-to-end performance and scalability across three large graph data sets.

Summary of performance improvements Figure 4 summarizes how SALIENT++'s optimizations result in increasingly better end-to-end training performance on *products*, *papers*, and *mag240c* in the distributed setting. Significant improvements are observed for both the *papers* and *mag240c* benchmarks. In relative terms, the *papers* benchmark benefits equally from pipelining and VIP-based caching, while the *mag240c* benchmark benefits slightly more from caching on top of pipelining. This is mainly due to the larger feature dimensionality for *mag240c*, which causes the communication of remote feature data to be relatively more throughput-bound than for *papers*.

Scalability Figure 5 (left plot) illustrates the reduction in per-epoch runtime when using SALIENT++ to execute the *products*, *papers*, and *mag240c* benchmarks on 2–16 GPUs. For all three benchmarks, the reported runtimes include warm-up time to fill up the pipeline at the start of each training epoch, which results in diminished scalability once per-epoch runtimes become less than a second. The *products* benchmark achieves $1.7\times$ speedup when going from 2 to 4 GPUs, but does not significantly benefit from more scaling. The *papers* benchmark achieves $1.9\times$ speedup from 2 to 4 GPUs and another $1.9\times$ from 4 to 8 GPUs. The *mag240c* benchmark achieves $1.75\times$ speedup going

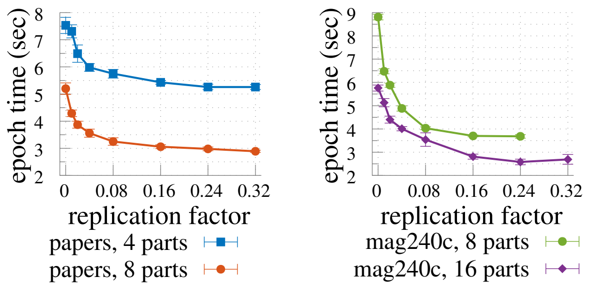

Figure 7. Replication factor impact on per-epoch runtime. Results are shown for *papers* (left) for 4 and 8 partitions, and for *mag240c* for 8 and 16 partitions for replication factors α varied between 0 and 0.32. The percentage of the local partition stored on GPU for this experiment was 90% for *papers* and 10% for *mag240c*. Reported numbers are the mean value over 10 epochs and vertical bars indicate the standard deviation across epoch runtimes.

from 4 to 8 GPUs, and $1.45\times$ speedup from 8 to 16 GPUs. SALIENT++'s performance matches that of prior fast systems which perform distributed GNN training with full replication (Kaler et al., 2022), while SALIENT++ uses substantially less memory. Figure 5 (right plot) shows the total memory for storing vertex feature data (including replicated vertices) across all machines. Note that full replication with K machines corresponds to replication factor $\alpha = K - 1$.

Local partition storage on CPU versus GPU Figure 6 illustrates the performance impact of increasing the percentage of the local partition stored on GPU while running the *papers* benchmark. Given an ordering of the vertices in the local partition and a percentage $\beta\%$ of local data to store on GPU, SALIENT++ stores the first $\beta\%$ of vertex features on the GPU. The results labeled "no reorder" show an essentially linear reduction in per-epoch runtime as a function of $\beta\%$ when ranking local vertices by their initial ordering. The results labeled "VIP reorder," on the other hand, show that when ranking local vertices by their VIP values, data transfer bottlenecks are effectively eliminated with as little as 10% of the local partition data on the GPU.

Impact of replication factor Figure 7 illustrates the impact on per-epoch runtime of increasing the number of replicated vertex features for *papers* on 4 and 8 partitions (left) and on *mag240c* for 8 and 16 partitions (right). Figure 7 shows that modest replication factors of 0.08–0.16 and 0.16–0.32 are sufficient for minimizing per-epoch runtime when using 4 and 8 partitions respectively on the *papers* dataset.

Performance breakdown for SALIENT++ Figure 8 illustrates the performance bottlenecks in SALIENT++ before and after incorporating pipelining and caching via VIP analysis.[2] A comparison of the "pipelining on" and "pipelining

[2]The performance-breakdown experiments in Figure 8 store all local node features on the GPU. Consequently, the "pipelin-

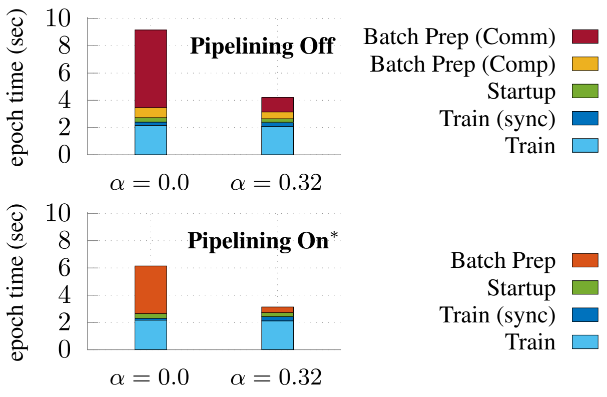

Figure 8. Performance breakdown for SALIENT++ (with added synchronization) for an 8-GPU execution of *papers* with all local node features on GPU. The "pipelining off" breakdown is best able to isolate the time spent performing each operation. The "pipelining on" breakdown adds minimal synchronization to isolate distributed batch preparation from training computation. *Batch Prep (Comm)* and *Batch Prep (Comp)* are the time for communication and computation, respectively, in distributed batch preparation. These categories are combined into *Batch Prep* for the "pipelining on" breakdown. *Startup* is the time to prepare the first batch and/or fill the pipeline. *Train (sync)* is the time for parallel workers to synchronize at the first collective operation of the training backward pass. *Train* measures GPU computation for training.

off" breakdowns for $\alpha = 0$ illustrates that network communication is the primary bottleneck for distributed GNN training, and it remains the bottleneck even when communication is pipelined to overlap with computation. When using caching with $\alpha = 0.32$, the time spent performing communication is sufficiently small so that it can be overlapped nearly perfectly with other computation in the program.

Performance on a slow network Figure 9 illustrates the performance of SALIENT++ on a slow network where higher replication factors are needed to alleviate communication bottlenecks. The slower network speeds were enforced using the Linux traffic control subsystem with the token-bucket filter (TBF) queuing discipline (Hubert et al., 2002). In this setting, we compare the two best caching policies from Section 3, VIP (analytic) and VIP (simulation). Recall from Section 3 that these two policies tend to perform very similarly for low replication factors, but diverge as the replication factor grows. On the *papers* dataset, the relative gap between these two policies grows up to 30% until $\alpha = 0.64$ and then starts to shrink once communication ceases to bottleneck training time. For the *mag240c* dataset, which uses node features that are $6\times$ larger than *papers*, the relative

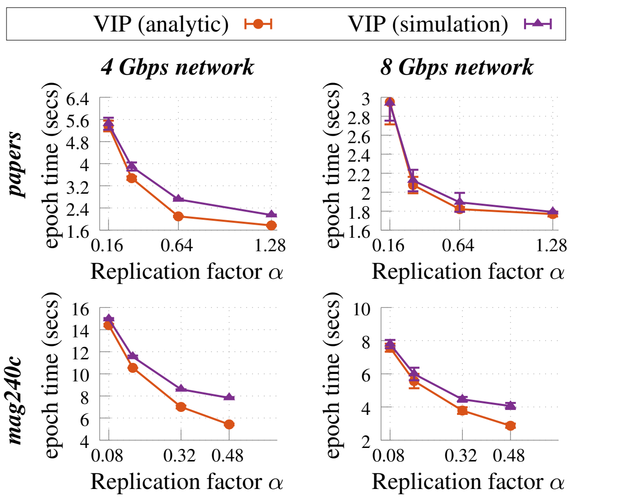

Figure 9. Comparison of VIP-analytic versus VIP-simulation on end-to-end runtime on slow networks. Illustrates the per-epoch runtime of 16-node executions on *papers* and *mag240c* when using the VIP-analytic and VIP-simulation caching policies with different replication factors.

gap between VIP (analytic) and VIP (simulation) is larger (up to 45% with $\alpha = 0.48$) and persistent. Whether or not VIP (analytic) is preferred to VIP (simulation) depends on the workload characteristics (e.g., fanout and size of node features) as well as the network bandwidth budget.

5.3 Context for SALIENT++'s performance

To conclude our empirical evaluation, we contextualize SALIENT++'s performance by reporting model accuracy results, discussing preprocessing overheads, and comparing our performance to existing systems.

Model accuracy The optimizations in SALIENT++ do not impact model accuracy. SALIENT++ is integrates with PyTorch Geometric (PyG) and can use existing GNN models implemented with PyG. Regardless, we report accuracies for *products*, *papers* and *mag240c* so that one can contextualize the performance numbers. For computing accuracies, we executed SALIENT++ on 8 machines for 30 epochs with a fixed learning rate of 0.001 and a batch size of 1024 per machine.[3] Sampling was used during inference with fanouts $(20, 20, 20)$ for *papers* and *products*, and $(25, 15)$ for *mag240c*. On *products*, we observed a test accuracy of 0.785 for the 3-layer SAGE architecture. On *papers*, we observed a test accuracy of 0.646 for the 3-layer SAGE architecture. On *mag240c*, the 2-layer GraphSAGE architecture achieves an accuracy of 0.651.[4]

ing off" system with $\alpha = 0$ is slightly faster in Figure 8 than in Table 1 where all local node features are on CPU. The "pipelining on" system uses a version of SALIENT++'s pipeline with extra synchronization, resulting in modest slowdown, to facilitate the attribution of time among multiple network operations and concurrent computations running on different CUDA streams.

[3]These hyperparameters were not tuned for accuracy, but are known to be reasonable values.

[4]Validation accuracy is reported instead of "test-dev" or "test-challenge" accuracy, which we did not measure at the time of

Table 4. Comparison of SALIENT++, DistDGL (public code), and DistDGLv2 (Zheng et al., 2022). All numbers are reported for a 3-layer GraphSAGE architecture with fanouts 15,10,5 and 256 hidden nodes run on the *papers* dataset.

System	*Time (s)*	*Notes*
SALIENT++	2.9	8 NVIDIA A10G GPUs, 25Gbps network throughput, $19.5 / hr.
DistDGL	37.0	Same hardware as is used by SALIENT++. Example distributed code from Github[5]
DistDGLv2	≈ 5	64 NVIDIA T4 GPUs, 100Gbps network throughput, $62.6 / hr.[6]

Preprocessing overheads SALIENT++'s preprocessing overheads fall into two categories: (a) runtime overheads incurred before each execution on a dataset; and, (b) dataset preparation overheads that are incurred only once and can be amortized over multiple experiments. The runtime overheads for an 8-node execution of SALIENT++ on the *papers* dataset with replication factor $\alpha = 0.32$ are as follows. Loading the dataset from disk takes approximately 10 seconds. Computing the VIP weights for fanouts $(15, 10, 5)$ takes 11.8 seconds when implemented in PyTorch with sparse transition weight matrices and dense hop-wise VIP-vectors. For large graphs, the VIP computation code streams batches of matrix rows to the GPU, overlapping communication and data transfer. The communication of remote feature vectors and the related tensor slicing operations take about 22 seconds. The dataset preprocessing overheads are highly dependent on the workflow used for partitioning graphs. Our (unoptimized) workflow for graph partitioning uses serial METIS to partition graphs and, additionally, uses machines with limited memory necessitating the use of swap files. In this setting, partitioning the *papers* dataset takes ~ 2 hours and creating a reordered dataset from that partition takes 30 minutes. Optimized workflows for partitioning can generate partitions substantially faster. For example, DistDGLv2 (Zheng et al., 2022) reports a 12 minute runtime for partitioning when using ParMETIS (Karypis et al., 1998) and 4 large multicores. SALIENT++ is agnostic to the source of the graph partitioning, and can be used in conjunction with more scalable graph partitioning codes.

Comparison to DistDGL We compared the performance of SALIENT++ to DistDGL on *papers* using the GraphSAGE architecture. A summary of this comparison is provided in Table 4. The DistDGL code was obtained from DGL's public GitHub repository and executed in the same computing environment used to evaluate SALIENT++. DistDGL's public code for distributed multi-GPU training was approximately $12.7\times$ slower than SALIENT++ on 8 GPUs (1 GPU per machine). Of course, DGL is under active development and its support for efficient GNN training (especially in distributed environments) is evolving rapidly. As such, we also report performance results from the recent work "DistDGLv2" (Zheng et al., 2022) that describes innovations in the DistDGL system for improving distributed GNN training performance. Although not all of these innovations are publicly available, their reported performance can be used to contextualize the performance we achieve with SALIENT++. A full comparison of the hardware used by DistDGLv2 and SALIENT++ shows that SALIENT++ consistently achieves similar (often better) performance than what was reported for DistDGLv2, while using substantially less resources. SALIENT++'s 8-GPU runtime is approximately $1.7\times$ faster than DistDGLv2's reported numbers for executing a training epoch on *papers* using 64 GPUs. This faster per-epoch time is achieved with $8\times$ fewer GPUs, $4\times$ smaller network bandwidth, and $3.2\times$ cheaper hardware.

submission. Prior results published on the OGB leaderboard show that this result is consistent for GraphSAGE on this dataset.

[6]Hardware details obtained from DistDGLv2 paper (Zheng et al., 2022), and pricing information obtained from Amazon Web Services for the *g4dn.metal* instance type reported as being used.

6 Conclusions and future work

We have presented a distributed multi-GPU system, SALIENT++, for GNN training and inference on massive graphs. This system is built on top of SALIENT, a prior state-of-the-art system that attains efficiency and scalability through fast sampling and pipelining, at the cost of full data replication on all machines. In SALIENT++, we distribute the vertex feature data and address the resulting communication bottleneck through analyzing the access pattern of the out-of-machine (i.e., remote) vertices and proposing a caching strategy that replicates a small amount of the most frequently accessed vertex features. Together with a deep pipelining of all operations from communication to computation, SALIENT++ retains the efficiency and scalability of SALIENT while consuming only a fraction of the storage required by SALIENT.

An avenue of future work is to further apply the access pattern analysis to improve the initial graph partitioning, with an aim of reducing the communication volume orthogonally to the use of caching. This would require incorporating the vertex inclusion probabilities in the graph partitioning objective, on top of edge cuts and load balancing. Additionally, a hierarchical graph partitioning may better leverage the higher intra-machine bandwidth among GPUs than inter-machine communication. Another line of future work is to explore distributing the graph structure across machines. Distributing the graph incurs nontrivial challenges in the multi-round communication of node features, but resolving this challenge will further reduce memory consumption and bears the potential of handling graphs that are orders of magnitudes larger than the current largest benchmarks.

ACKNOWLEDGEMENTS

This work was conducted on the SuperCloud computing cluster https://supercloud.mit.edu and the Satori computing cluster https://mit-satori.github.io. This research was sponsored by MIT-IBM Watson AI Lab and in part by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. https://www.tensorflow.org/.

Bottou, L., Curtis, F. E., and Nocedal, J. Optimization methods for large-scale machine learning. *SIAM Rev.*, 60 (2):223–311, 2018.

Cai, Z., Yan, X., Wu, Y., Ma, K., Cheng, J., and Yu, F. Dgcl: An efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pp. 130–144, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi: 10.1145/3447786.3456233. URL https://doi.org/10.1145/3447786.3456233.

Chen, J. and Luss, R. Stochastic gradient descent with biased but consistent gradient estimators. Preprint arXiv:1807.11880, 2018.

Chen, J., Ma, T., and Xiao, C. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *ICLR*, 2018.

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C.-J. Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks. In *KDD*, 2019.

Dong, J., Zheng, D., Yang, L. F., and Karypis, G. Global neighbor sampling for mixed CPU-GPU training on giant graphs. In *KDD*, 2021.

Gandhi, S. and Iyer, A. P. P3: Distributed deep graph learning at scale. In *OSDI*, 2021.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *ICML*, 2017.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. In *NIPS*, 2017.

Hoang, L., Chen, X., Lee, H., Dathathri, R., Gill, G., and Pingali, K. Efficient distribution for deep learning on large graphs,. In *GNNSys*, 2021.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. Preprint arXiv:2005.00687, 2020.

Hu, W., Fey, M., Ren, H., Nakata, M., Dong, Y., and Leskovec, J. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430*, 2021.

Hubert, B. et al. Linux advanced routing & traffic control howto. *Netherlabs BV*, 1:99–107, 2002.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with Roc. In *MLSys*, 2020.

Kaler, T., Stathas, N., Ouyang, A., Iliopoulos, A.-S., Schardl, T., Leiserson, C. E., and Chen, J. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

Karypis, G. and Kumar, V. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

Karypis, G., Schloegel, K., and Kumar, V. Parmetis—parallel graph partitioning and sparse matrix ordering library, version 2.0. *Univ. of Minnesota, Minneapolis, MN*, 10, 1998.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. In *ICLR*, 2016.

Li, Y., Yu, R., Shahabi, C., and Liu, Y. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In *ICLR*, 2018.

Lin, Z., Li, C., Miao, Y., Liu, Y., and Xu, Y. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pp. 401–415, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421281. URL https://doi.org/10.1145/3419111.3421281.

Liu, Z., Wu, Z., Zhang, Z., Zhou, J., Yang, S., Song, L., and Qi, Y. Bandit samplers for training graph neural networks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 6878–6888, 2020.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. NeuGraph: Parallel deep neural network computation on large graphs. In *USENIX ATC*, 2019.

Md, V., Misra, S., Ma, G., Mohanty, R., Georganas, E., Heinecke, A., Kalamkar, D., Ahmed, N. K., and Avancha, S. Distgnn: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

Min, S. W., Wu, K., Huang, S., Hidayetoğlu, M., Xiong, J., Ebrahimi, E., Chen, D., and mei Hwu, W. Large graph convolutional network training with GPU-oriented data communication architecture. Preprint arXiv:2103.03330, 2021.

Mohoney, J., Waleffe, R., Xu, H., Rekatsinas, T., and Venkataraman, S. Marius: Learning massive graph embeddings on a single machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 533–549. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/mohoney.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *NIPS*, 2019.

Ramezani, M., Cong, W., Mahdavi, M., Sivasubramaniam, A., and Kandemir, M. GCN meets GPU: Decoupling "when to sample" from "how to sample". In *NeurIPS*, 2020.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *ICLR*, 2018.

Waleffe, R., Mohoney, J., Rekatsinas, T., and Venkataraman, S. Marius++: Large-scale training of graph neural networks on a single machine. *arXiv preprint arXiv:2202.02365*, 2022.

Wan, C., Li, Y., Li, A., Kim, N. S., and Lin, Y. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 673–693, 2022. URL https://proceedings.mlsys.org/paper/2022/file/d1fe173d08e959397adf34b1d77e88d7-Paper.pdf.

Wang, L., Yin, Q., Tian, C., Yang, J., Chen, R., Yu, W., Yao, Z., and Zhou, J. FlexGraph: a flexible and efficient distributed framework for GNN training. In *EuroSys*, 2021a.

Wang, Y., Feng, B., Li, G., Li, S., Deng, L., Xie, Y., and Ding, Y. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *OSDI*, 2021b.

Weber, M., Domeniconi, G., Chen, J., Weidele, D. K. I., Bellei, C., Robinson, T., and Leiserson, C. E. Anti-money laundering in Bitcoin: Experimenting with graph convolutional networks for financial forensics. In *KDD Workshop on Anomaly Detection in Finance*, 2019.

Wu, Y., Ma, K., Cai, Z., Jin, T., Li, B., Zheng, C., Cheng, J., and Yu, F. Seastar: vertex-centric programming for graph neural networks. In *EuroSys*, 2021.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *ICLR*, 2019.

Yang, J., Tang, D., Song, X., Wang, L., Yin, Q., Chen, R., Yu, W., and Zhou, J. GNNLab: A factored system for sample-based GNN training over GPUs. In *EuroSys*, pp. 417–434, 2022.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, 2018.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. GraphSAINT: Graph sampling based inductive learning method. In *ICLR*, 2020.

Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. DistDGL: Distributed graph neural network training for billion-scale graphs. In *IA3*, 2020.

Zheng, D., Song, X., Yang, C., LaSalle, D., and Karypis, G. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale graphs. Preprint arXiv:2112.15345, 2021.

Zheng, D., Song, X., Yang, C., LaSalle, D., and Karypis, G. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, pp. 4582–4591, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393850. doi: 10.1145/3534678.3539177. URL https://doi.org/10.1145/3534678.3539177.

Zou, D., Hu, Z., Wang, Y., Jiang, S., Sun, Y., and Gu, Q. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *NeurIPS*, 2019.

A Artifact Appendix

A.1 Abstract

This section describes the software artifacts associated with this paper. The code is distributed via GitHub at `https://github.com/MITIBMxGraph/SALIENT_plusplus_artifact` and can be used to perform the experiments presented in the paper. We provide scripts to run: (a) the simulation experiments to compare different caching policies and generate data for Figure 2; and (b) the distributed experiments that produce data for Table 1 and Figures 4–7. Detailed instructions for running these scripts are provided in a dedicated README file for artifact evaluation located at `https://github.com/MITIBMxGraph/SALIENT_plusplus_artifact/blob/main/README.md`.

A.2 Artifact check-list (meta-information)

- **Algorithm:** SALIENT++ distributed training algorithms for GNNs and VIP analysis algorithms for caching policies for GNNs.
- **Program:** PyTorch, CUDA
- **Compilation:** `gcc/g++` version 7 or greater; `nvcc` version 11.
- **Data set:** Node classification benchmark data sets from OGB.
- **Run-time environment:** Ubuntu 20.04 (or modern Linux distribution) with NVIDIA drivers installed.
- **Hardware:** NVIDIA GPU with sufficient memory. Distributed experiments require SLURM cluster with GPU nodes. Simulation experiments require up to 200 GB of main memory.
- **Experiments:** Local simulation experiments for comparing different caching policies in terms of remote vertex communication volume, and distributed experiments for analyzing the impact on per-epoch training time of different optimizations, varying replication factor, and varying percentage of data stored in-memory on GPU.
- **How much disk space required (approximately)?:** 1.5 TB for all experiments, 100 GB for a substantial subset of experiments.
- **How much time is needed to prepare workflow (approximately)?:** 1–2 hours with prior experience and access to hardware/clusters.
- **How much time is needed to complete experiments (approximately)?:** 4–12 hours for all experiments if using pre-processed datasets.
- **Publicly available?:** Yes
- **Code licenses:** Apache License 2.0
- **Data licenses:** Amazon license and ODC-BY.
- **Archive DOI:** `https://doi.org/10.5281/zenodo.7889203`

A.3 Description

A.3.1 How delivered

The code is available on GitHub at `https://github.com/MITIBMxGraph/SALIENT_plusplus_artifact`. Within the repository, scripts for streamlining the process of exercising the artifact are provided in the `experiments` directory. Instructions for running the scripts can be found in the repo's top-level README document at `https://github.com/MITIBMxGraph/SALIENT_plusplus_artifact/blob/main/README.md`.

A.3.2 Hardware dependencies

The minimum requirements for exercising the software artifact are as follows.

The simulation experiments can be performed on the ogbn-papers100M dataset using a single machine (with or without a GPU) that has 160GB or more of main memory. On machines with lower memory capacity, it is possible to run these experiments by using swapfiles, ideally on fast SSDs. Simulation experiments on the smaller ogbn-products dataset require less than 10 GB of memory.

The distributed multi-GPU experiments require access to either a SLURM cluster with GPU nodes or a single machine with multiple GPUs. Those opting to use a single machine with multiple GPUs should pass the appropriate flags to experimental scripts to indicate they are running scripts locally. Access to a SLURM cluster with GPU nodes may be obtained through cloud services and accompanying software packages. For example, on Amazon Web Services one can use the ParallelCluster software to launch a SLURM cluster.

Depending on the used hardware and available disk space, certain experiments may not be feasible. We have made an effort to reduce the memory requirements for running performance experiments on the largest data sets, and we expect that GPUs with 16GB of memory and machines with 128GB of main memory will be sufficient for running all or almost all of the distributed experiments.

A.3.3 Software dependencies

Reasonably up-to-date NVIDIA drivers must be installed on the machine. For the distributed experiments, a SLURM cluster is required. The remaining external software dependencies can be resolved using the conda package manager.

A.3.4 Data sets

Graph data sets for node property prediction are taken from the Open Graph Benchmark (OGB) repository. To decrease the time and minimum hardware resources required for experiments, we have provided the option to download pre-processed versions of the graph data. If electing to not download the preprocessed graphs, the first execution of the code on a new graph will download it from OGB and perform pre-processing locally. Note that, for the distributed experiments, additional pre-processing is needed to generate partition labels and reorder vertices by partition; this can be achieved using provided scripts, as described in the artifact README.

A.4 Installation

We recommend referring to the installation instructions provided at https://github.com/MITIBMxGraph/SALIENT_plusplus_artifact/blob/main/INSTALL.md. We summarize the installation process here.

Installation in Python environment: We provide instructions to install the artifact in a Python virtual environment. The installation can be used for both the single-machine and distributed multi-GPU experiments. The instructions for installing the artifact in a conda environment are summarized below.

```
# Download and install miniconda
wget https://repo.anaconda.com/miniconda/\
    Miniconda3-py38_4.10.3-Linux-x86_64.sh
bash Miniconda3-py38_4.10.3-Linux-x86_64.sh

# Create a conda environment for the artifact
conda create -n salientplus python=3.9.5 -y
conda activate salientplus

# Install Pytorch, PyG, OGB, prettytable
conda install -y pytorch==1.13.1 torchvision==0.14.1 \
  torchaudio==0.13.1 pytorch-cuda=11.7 -c pytorch -c nvidia
conda install -y -c conda-forge ogb
conda install -y pyg -c pyg -c conda-forge
conda install -y pytorch-sparse -c pyg
conda install -y -c conda-forge nvtx
conda install -y -c conda-forge matplotlib
conda install -y -c conda-forge prettytable

# Install fast_sampler
cd fast_sampler
python setup.py install
cd ..

# (Optional) Install METIS
# - omitted, see INSTALL.md in the artifact repository
```

A.5 Experiment workflow

We recommend referring to the artifact evaluation documentation located in the GitHub repository at https://github.com/MITIBMxGraph/SALIENT_plusplus_artifact/blob/main/README.md. We summarize the experimental workflow here. Unless otherwise noted, all file paths are relative to the experiments directory in the repository.

Initial setup for experiments

We provide two utility scripts to streamline the process of downloading the pre-processed datasets and partitionings. A configuration script (configure_for_environment.py) determines which datasets to download based on the available disk space and the maximum number of GPUs (and partitions) to use for the experiments. A separate script (download_datasets_fast.py) downloads the datasets selected during configuration. Please see the artifact repository README for instructions on using these setup scripts.

Simulation experiments

The script run_sim_experiments_paper.sh runs simulation experiments on the ogbn-papers100M dataset to reproduce the results in Figure 2. This script assumes that the 8-partition ogbn-papers100M dataset was downloaded using the setup scripts described above.

The driver script for running custom simulation experiments is run_sim_experiments.sh. The custom simulation script options are documented in the artifact README.

Distributed multi-GPU experiments

These experiments require the use of a SLURM cluster with GPU nodes or a local machine with multiple GPUs. Each of the distributed-experiment scripts provides options to customize the experiment, and will display a table of results in the terminal after the experiment has completed.

A summary of the provided scripts is as follows:

- experiment_optimization_impact.py reproduces the experiment in Table 1 and Figure 4 to show the end-to-end impact of different optimizations on per-epoch runtime.
- experiment_vary_replication_factor.py reproduces the experiments in Figures 5 and 7 to show the performance scalability with increasing number of distributed nodes and the relationship between per-epoch runtime and replication factor.
- experiment_vary_gpu_percent.py reproduces the experiment in Figure 6 to analyze the performance impact of storing features on GPU.
- experiment_accuracy.py performs end-to-end training on the datasets and reports accuracy on the validation and test sets.

If running these scripts on a SLURM cluster, it is necessary to configure the provided experimental driver script `exp_driver.py` by editing the `SLURM_CONFIG` variable at the top of the file. Additional details are provided in the artifact evaluation guide in the repository. If running these scripts locally on a single machine with multiple GPUs, one must pass the command line argument `--run_local 1` to the script.

Users who wish to run custom experiments may use the core experiment driver script `exp_driver.py` directly.

A.6 Evaluation and expected result

Upon completion of the simulation experiments, a table will be produced with the GNN training communication volume for different caching policies. This reproduces the data shown in Figure 2.

Upon completion of the distributed GPU experiments, multiple tables will be produced with the results of Table 1 and Figures 4–7. Participants might opt to execute these experiments on different datasets or for different parameters depending on their hardware and time constraints.

A.7 Experiment customization

The following experiment customizations are possible. The simulation and distributed experiment scripts provide command-line options to run on different datasets and with different parameters. The experimental driver script `exp_driver.py` may be used directly to run custom performance experiments in the multi-machine multi-GPU setting on different datasets, using different model architectures, and training parameters. The software may be used separately from the experimental driver scripts. The included pre-processing utility scripts can be used to partition graphs and reorder datasets according to a graph partitioning.

B Code Repository

In addition to the artifact repository that focuses on benchmarking and reproducibility, an implementation of SALIENT++ for general-purpose usage is available at https://github.com/MITIBMxGraph/SALIENT_plusplus.

C Proof for layer-wise probabilities in Proposition 1

We start by considering the probability that some vertex $u \in \mathcal{V}$ is sampled as a 1-hop neighbor of some minibatch $\mathcal{B}$. This is equal to the probability that, for any vertex v, all of the following are true: v is included in the minibatch, u is a neighbor of v, and u is sampled among v's direct neighbors. If we denote by $\mathcal{N}_h^s(\mathcal{B})$ the vertices in the expanded neighborhood that are sampled after exactly h expansion steps away from $\mathcal{B}$, we have:

$$
\begin{aligned}
p^{[1]}(u) &= \Pr[u \in \mathcal{N}_1^s(\mathcal{B})] \\
&= \Pr\left[\bigcup_{v \in \mathcal{N}_1^{\mathsf{T}}(u)} \Big((v \in \mathcal{B}) \cap (u \in \mathcal{N}_1^s(v))\Big)\right] \\
&= 1 - \Pr\left[\bigcap_{v \in \mathcal{N}_1^{\mathsf{T}}(u)} \Big((v \notin \mathcal{B}) \cup (u \notin \mathcal{N}_1^s(v))\Big)\right] \\
&= 1 - \prod_{v \in \mathcal{N}_1^{\mathsf{T}}(u)} \Pr[(v \notin \mathcal{B}) \cup (u \notin \mathcal{N}_1^s(v))] \\
&= 1 - \prod_{v \in \mathcal{N}_1^{\mathsf{T}}(u)} \Big(\Pr[v \notin \mathcal{B}] \\
&\qquad\qquad + \Pr[(u \notin \mathcal{N}^s(v)) \cap (v \in \mathcal{B})]\Big) \\
&= 1 - \prod_{v \in \mathcal{N}_1^{\mathsf{T}}(u)} \Big(1 - p^{[0]}(v) + \big(1 - t_1(u,v)\big)\, p^{[0]}(v)\Big) \\
&= 1 - \prod_{v \in \mathcal{N}_1^{\mathsf{T}}(u)} \Big(1 - t_1(u,v)\, p^{[0]}(v)\Big).
\end{aligned}
$$

The product in the 4th line follows from the assumption of neighborhood-sampling independence between vertices in the minibatch. Each factor therein is the probability that u is not sampled as a neighbor of v. For directed graphs, when we write $\mathcal{N}_1^{\mathsf{T}}(u)$ for the union, intersection, and product indices, we mean the outgoing direct neighbors of u. For undirected graphs, $\mathcal{N}_1^{\mathsf{T}}(u) = \mathcal{N}_1(u)$.

By a similar reasoning, the probability that a vertex u is sampled exactly ℓ hops away when expanding the neighborhood of some minibatch is

$$
\begin{aligned}
p^{[h]}(u) &= \Pr[u \in \mathcal{N}_h^s(\mathcal{B})] \\
&= \Pr\left[\bigcup_{v \in \mathcal{N}_1^{\mathsf{T}}(u)} \Big((v \in \mathcal{N}_{h-1}^s(\mathcal{B})) \cap (u \in \mathcal{N}_1^s(v))\Big)\right] \\
&= \cdots \\
&= 1 - \prod_{v \in \mathcal{N}_1^{\mathsf{T}}(u)} \Big(1 - t_h(u,v)\, p^{[h-1]}(v)\Big),
\end{aligned}
$$

where the omitted steps are similar to the ones for $p^{[0]}(u)$ above.

D Walkthrough of SALIENT++'s pipelining stages

The minibatch preparation process proceeds in the following stages. Stage 1 obtains the next processed minibatch by

SALIENT++ from the neighborhood sampler. Stage 2 performs an all-to-all communication to broadcast the number of remote vertices each machine will send/receive. Stage 3 transfers the metadata from stage 2 to the CPU so that appropriately sized tensors can be allocated. Stage 4 performs an all-to-all communication in which each machine i sends to each machine j a list of nodes whose features are local to machine j and needed by machine i. Stage 5 receives the list of nodes requested by other machines, maps their global identifiers to local identifiers, and performs a device-to-host transfer so that the list of requested nodes is accessibly from CPU memory. Stage 6 launches an asynchronous worker thread that performs a "masked selection" operation to split each list of requested node indices into two groups based on whether the referenced node is stored in the local CPU or GPU memory. The background thread launched by Stage 6 also performs tensor slicing for the requested nodes whose features are stored in CPU memory. Note that the, seemingly innocuous, operation of performing a "masked selection" operator to divide indices into a CPU group and a GPU group can induce a device synchronization if it is performed on the GPU in Stage 5, which is why we assign this task to the background CPU thread. Stage 7 starts a host-to-device data transfer to send the results of Stage 6 to the GPU. Stage 8 performs tensor slicing on GPU for requested node features stored locally on GPU, and combines the CPU/GPU results so that features requested by each remote machine are in a single tensor. Additionally, Stage 8 slices the machine's local feature cache to extract the remote features needed by the current machine [7] Stage 9 performs an all-to-all communication to communicate all requested remote features. Stage 10 combines the received remote features into a single tensor that is permuted so as to be consistent with the locally-generated message-flow graph for the batch.

[7]The placement of these operations in the pipeline is somewhat arbitrary, as these operations do not depend on non-local information.