DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks

Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, Sasikanth Avancha
[vasimuddin.md,sanchit.misra,guixiang.ma,Ramanarayan.Mohanty,evangelos.georganas, alexander.heinecke,dhiraj.d.kalamkar,nesreen.k.ahmed,sasikanth.avancha]@intel.com
Intel Corporation

ABSTRACT

Full-batch training on Graph Neural Networks (GNN) to learn the structure of large graphs is a critical problem that needs to scale to hundreds of compute nodes to be feasible. It is challenging due to large memory capacity and bandwidth requirements on a single compute node and high communication volumes across multiple nodes. In this paper, we present DistGNN that optimizes the well-known Deep Graph Library (DGL) for full-batch training on CPU clusters via an efficient shared memory implementation, communication reduction using a minimum vertex-cut graph partitioning algorithm and communication avoidance using a family of delayed-update algorithms. Our results on four common GNN benchmark datasets: Reddit, OGB-Products, OGB-Papers and Proteins, show up to 3.7× speed-up using a single CPU socket and up to 97× speed-up using 128 CPU sockets, respectively, over baseline DGL implementations running on a single CPU socket.

KEYWORDS

Graph Neural Networks, Graph Partition, Distributed Algorithm, Deep Learning, Deep Graph Library

ACM Reference Format:
Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, Sasikanth Avancha. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3458817.3480856

1 INTRODUCTION

Graphs are ubiquitous across multiple domains: social networks, power grids, biological interactomes, molecules etc. In the fast-emerging domain of geometric deep learning [5], a specific field called Graph Neural Networks (GNN) has recently shown impressive results across a spectrum of graph and network representation learning problems [12, 24, 36, 38]. [34] shows that GNNs are a very powerful mechanism to learn the structure of non-Euclidean data such as graphs. GNNs combine low-dimensional embeddings associated with each vertex in a graph with local neighborhood connectivity for downstream machine learning analysis, e.g., vertex property prediction, link property prediction and graph property prediction.

Scaling GNN training for large graphs consisting of hundreds of millions of vertices and edges is a huge challenge, due to high memory capacity and bandwidth requirements as well as high communication volume to ensure convergence. For large graphs, memory capacity requirements during training restrict the size of the problem that can be solved on a single socket. Mini-batch training works around these restrictions via neighborhood sampling [6, 8, 13, 36], to create a mini-batch of graph samples, which reduces the working set size. Another technique to mitigate memory capacity problems is to use the aggregate memory capacity of a distributed system [18, 22, 29]. It has been shown that, in some cases, neighborhood sampling achieves lower accuracy compared to full-batch training [18]. In this work, we focus on full-batch training and discuss distributed memory solutions that scale across mutliple Intel® Xeon® CPU sockets; we plan to address scaling mini-batch training in future work.

Challenges of Full-batch Training: GNN training poses the following specific challenges compared to traditional DL workload training: (a) Communication volume increases because vertex feature vectors and parameter gradients must be communicated, increasing pressure on the communications network, which could become a bottleneck; (b) lower flops density and sequential nature of the training operations make it challenging to overlap computation with communication, exposing communication time; (c) due to high byte-to-flop ratio, the aggregation operation tends to be memory and communication bandwidth bound. On a single CPU socket, optimal cache and memory bandwidth utilization are important factors in achieving good performance, while in a distributed setting, graph partitioning [19, 21, 32] plays a crucial role in managing the communication bottleneck among compute sockets.

Current graph processing frameworks [11, 20, 23, 33] are capable of processing large graphs, but they do not support DL primitives. Consequently, the research community has developed libraries such as DGL [31] and PyTorch Geometric (PyG) [10] with capability of message passing in the graphs. These libraries employ current DL-based frameworks such as TensorFlow [1], PyTorch [27], and MXNet [7], coupling DL primitive operations with message passing. However, a huge challenge is that key compute primitives within these libraries are typically inefficient for single-socket shared-memory and multi-socket distributed-memory CPU settings. We chose DGL due to its rich functionality and flexibility. However,

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3480856

DGL's shared memory performance is poor on CPUs and it does not support full batch training on distributed systems. In this work, we enhance DGL by developing highly optimized single-socket and distributed solutions.

Present Work. In this work, we discuss DistGNN, which consists of a set of single-socket optimizations and a family of scalable, distributed solutions for large-scale GNN training on clusters of Intel Xeon CPUs. Our key contributions are as follows:

- A highly architecture-optimized implementation of multiple variants of the aggregation primitive implemented as a customized SpMM (Sparse Matrix-Matrix Multiplication) operation for single socket CPUs. This is achieved by efficient cache blocking, dynamic thread scheduling and use of LIBXSMM [14] for loop reordering, vectorization and JITing to reduce instruction count. On a single CPU socket (Intel Xeon 8280 with 28 cores), compared to the DGL baseline, we achieve 3.66× speedup for the GraphSAGE full-batch training on the Reddit dataset [13] (this is 3.1× faster than what is reported in [3]) and 1.95× speedup on OGB-Products dataset [15].
- Novel application of vertex-cut based graph partitioning to large-graphs to achieve an optimal solution to reduce communication across CPU sockets during full-batch GNN training.
- Novel application of delayed update algorithm to feature aggregation to achieve optimal communication avoidance during full-batch GNN training. To reduce the impact of communication, we overlap it with computation by spreading it across epochs; overlap occurs at the expense of freshness – aggregation uses stale, partially-aggregated remote vertex features. Our optimizations for single socket make it even more challenging and critical to overlap communication with computation.
- First-ever demonstration of **full-batch** GNN training on CPU-based distributed memory systems. We showcase the performance of our distributed memory solution on the OGB-papers dataset [15], which contains 111 million vertices and 1.6 billion edges. Our solution scales GraphSAGE full-batch training to 128 Intel Xeon CPU sockets, achieving 97× speedup compared to the DGL (un-optimized) baseline and 83× compared to our optimized implementation, respectively, running on a single CPU socket.

The paper is organized as follows. Section 2 describes the core operation of GNN, the aggregation function, and DGL library. Section 3 positions the related literature. In Section 4, we discuss our shared memory solution. We analyze compute characteristics of GNN operations and detail various architecture-aware optimizations. Section 5 describes graph partitioning and a set of distributed algorithms. Section 6 demonstrates the performance of our solutions. Section 7 summarizes our work and discusses future work.

2 BACKGROUND

GNNs learn graph structure through low-dimension embeddings of vertices or edges. They compute these embeddings using an *aggregation function*, which recursively gathers multi-hop neighborhood features to encode vertex or edge features. The aggregation function also learns the shared weights by training a shallow neural network on the gathered features. Depending on the application, neighborhood aggregation precedes or succeeds the neural network layers. The aggregation function operates via *message-passing* between vertices or/and edges using an *Aggregation Primitive* (AP), which constitute a core part of the aggregation function.

2.1 Aggregation Primitive

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be an input graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$, and let $f_\mathcal{V}$ and $f_\mathcal{E}$ be the vertex and edge features, respectively. The sizes of $f_\mathcal{V}$ and $f_\mathcal{E}$ are $|\mathcal{V}|\times d$ and $|\mathcal{E}|\times d$, respectively, where d is the feature vector size. In effect, the AP is a tuple $(f_\mathcal{V}, f_\mathcal{E}, \otimes, \oplus, f_O)$, where $\otimes$ and $\oplus$ are element-wise operators on $f_\mathcal{V}$ or $f_\mathcal{E}$ (or a combination thereof) to produce output features f_O.

In AP, operator $\otimes$ can be an element-wise binary or unary operator. In binary form, it operates on a pair of inputs; valid pairs are $(f_\mathcal{V}, f_\mathcal{V})$ and $(f_\mathcal{V}, f_\mathcal{E})$, in an appropriate order. Operator $\oplus$ acts as element-wise *reducer* that reduces the result of binary operation on to the final output. Mathematically, AP can be expressed as,

$$\mathsf{AP}(x, y, \otimes, \oplus, z) : \oplus(\otimes(x, y), z), \quad \forall x, y, z \in f_\mathcal{V}, f_\mathcal{E} \tag{1}$$

If one of the inputs is NULL, then operator $\otimes$ takes the unary form; it copies the input features and reduces them to the final output. Mathematically, assuming $y = \phi$, (where ϕ stands for NULL) AP using the unary operator is,

$$\mathsf{AP}(x, \phi, \otimes, \oplus, z) : \oplus(copy(x), z), \quad \forall x, z \in f_\mathcal{V}, f_\mathcal{E} \tag{2}$$

Given a graph, Equations 1 and 2 can be formulated as SpMM: $f_O = A \times f_X$, where A is the graph adjacency matrix and f_X is the dense feature matrix. DGL featgraph [17] models the AP primitive as SpMM and provides a single template API for AP computations. In section 4, we describe variants of SpMM for different forms of AP and rigorous architecture-aware optimizations to accelerate it.

2.2 GNN Libraries

DGL and PyG are the two prominent emerging libraries for performing GNN operations. Due to its rich set of features, in this work, we use DGL to implement our single-socket and distributed algorithms.

DGL provides graph abstractions with rich set of functionality for manipulation and utilization of graph objects. It defines general computations on graphs as the message passing paradigm. For computations on vertices, the message-passing functionality is formulated as SpMM. For computations on edges, the message-passing functionality is formulated as sampled dense-dense matrix multiplication (SDDMM). It provides in-built support for various binary and reduction operators as well as support for user-defined functions. The distillation of the core GNN operations as a few matrix multiplication operations enables parallelization and various other optimizations, such as vectorization. Additionally, DGL relies on popular DL frameworks for its neural network operations. It provides

the flexibility of selecting from popular backend DL frameworks such as TensorFlow, PyTorch, and MXNet.

3 RELATED WORK

Prior art in this field encompasses both full-batch and mini-batch training approaches. Due to small model sizes relative to other DL workloads, distributed GNNs employ data parallelism, by partitioning the input graph across CPU sockets or GPU cards.

A few approaches have been proposed for GPU-based distributed memory systems, which we describe briefly below. NeuGraph [22] describes a single node multi-GPU parallel system for training large-scale GNN. It introduces a new programming model for GNN, leveraging a variant of vertex-centric parallel graph abstraction GAS [11]. NeuGraph partitions the input graph using min-cut Metis partitioning. Roc [18], a distributed multi-GPU GNN training system, applies an online regression model to get the graph partitions. Coupled with sophisticated memory management between the host and the GPU and a fast graph propagation optimized GPU tool called Lux, Roc showcases scalable performance for large graphs on distributed GPU system. Leveraging their scalable solution, Roc trains more complex GNN model architecture to achieve better model accuracy. Roc demonstrates the efficacy of their solution on small benchmark datasets of Reddit and Amazon. CAGNET [29] implemented a suite of parallel algorithms: 1D, 1.5D, 2D, and 3D algorithms, for GNN training using complete neighborhood aggregation. Inspired by SUMMA algorithm [30] for matrix multiplication, they apply different kinds of matrix blocking strategies for work division among the compute nodes. Their solution uses optimized cuSPARSE for the SpMM computations; however, it suffers from poor scaling due to communication bottlenecks.

For CPU-based distributed systems, all the approaches proposed so far have been for mini-batch training as described below. AliGraph [35] presents a comprehensive framework building various GNN applications. It supports distributed storage, sampling, and aggregator operators, along with a suite of graph partitioning techniques including vertex- and edge-cut based ones. AliGraph implements and showcases the efficacy of the interesting concept of caching the neighbors of important vertices to reduce the communication load. However, AliGraph does not report the scaling of their solution. Dist-DGL [37], a distributed GNN processing layer added to the DGL framework. It holds the vertex features in a distributed data server which can be queried for data access. Due to the large execution time of their inefficient graph sampling operation, it manages to overlap communication with the sampling time. Dist-DGL demonstrates linear scaling on the largest benchmark dataset, the OGBN-Papers.

Various graph processing frameworks for graph analytics and machine learning have been proposed in the literature [11, 20, 23, 33]. The most notable property of these frameworks is the Gather-Apply-Scatter (GAS) model. The aggregate step in GNN is a simple message passing step between the neighbors with synchronization at the end of the step. These frameworks using synchronous GAS can naturally implement the aggregate step of GNN; however, they lack the support for implementing various graph neural network operations as well as the graph attention models.

4 DISTGNN: SHARED-MEMORY ALGORITHM

AP accounts for a majority of the run-time in GNN applications. In this section, we describe techniques we have created to accelerate their implementations within DGL for a shared memory system.

4.1 Baseline Implementation of Aggregation Primitive in DGL

Alg. 1 provides a pseudo code of the AP in DGL. It uses a customized SpMM-like formulation. Given an input graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, let A be its adjacency matrix in CSR format. $A[v]$ gives the list of neighbors of v from which there is an edge incident on v. For each edge $e_{uv} \in \mathcal{E}$, we call u the source vertex and v the destination vertex. DGL pulls messages from vertex u and edge e_{uv} and reduces them into vertex v. More formally, for each edge e_{uv} incident from vertex u to vertex v, DGL computes the binary operator $\otimes$ element-wise between corresponding vertex feature vector of u ($f_{\mathcal{V}}[u]$) and edge feature vector of e_{uv} ($f_{\mathcal{E}}[e_{uv}]$). Subsequently, it reduces the result with the vertex features of v in the output feature matrix ($f_O[v]$) by computing the reduction operator $\oplus$ element-wise and stores it into $f_O[v]$. $\otimes$ can also be a unary operator. In that case, $\otimes$ is computed on either $f_{\mathcal{V}}[u]$ or $f_{\mathcal{E}}[e_{uv}]$ and reduced with $f_O[v]$. Table 1 details various binary/unary operators ($\otimes$) and reduction operators ($\oplus$) used in DGL.

The computation is parallelized by distributing destination vertices across OpenMP threads. This way, there are no collisions because only one thread *owns* the feature vector $f_O[v]$ at each destination vertex (v) and reduces the *pulled* source feature vectors into $f_O[v]$.

Algorithm 1 Aggregation Primitive in DGL

Require: Matrix A of size $|\mathcal{V}| \times |\mathcal{V}|$ in CSR format
Require: Input feature matrix $f_{\mathcal{V}}$ of size $|\mathcal{V}| \times d$ (Vertex feature set)
Require: Input feature matrix $f_{\mathcal{E}}$ of size $|\mathcal{E}| \times d$ (Edge feature set)
Require: Output feature matrix f_O of size $|\mathcal{V}| \times d$ (Initialized to zeros)
Require: Unary/Binary operator: $\otimes$, Reduction operator: $\oplus$
1: **for** $v \in \mathcal{V}$ in parallel **do**
2: **for** u in $A[v]$ **do**
3: $f_O[v] \leftarrow f_O[v] \oplus (f_{\mathcal{V}}[u] \otimes f_{\mathcal{E}}[e_{uv}])$
4: **end for**
5: **end for**

Table 1: Binary/Unary and Reduction Operators in DGL

Unary/Binary Operator ($\otimes$)	operands	Reduction Operator ($\oplus$)	output (z)
add/sub/mul/div	x,y	sum/max/min	$z \oplus (x \otimes y)$
copylhs	x, ϕ	sum/max/min	$z \oplus (\text{copy}(x))$
copyrhs	ϕ, y	sum/max/min	$z \oplus (\text{copy}(y))$

4.2 Efficient Aggregation Primitive

In case of large graphs, the feature matrices – $f_{\mathcal{V}}$, $f_{\mathcal{E}}$ and f_O – don't fit in cache. Moreover, real world graphs are typically highly sparse. To aggregate the feature vector of a vertex, the feature vectors of all its neighbors and the corresponding edges must be accessed. Feature vectors of edges incident on a vertex can be contiguously

stored in $f_{\mathcal{E}}$ and are only accessed once in Alg. 1. This effectively makes it a contiguous access of large enough block of memory that is used once - hence, a memory bandwidth (BW) bound streaming access. On the other hand, those of the neighbors in $f_{\mathcal{V}}$ can be non-contiguous, sparsely located (leading to random gathers) and will be used as many times as the number of their neighbors. Moreover, a feature vector, $f_{\mathcal{V}}[v]$, accessed once and brought into cache may get thrashed out before it is needed again. Hence, in many cases, $f_{\mathcal{V}}[v]$ needs to be fetched from memory despite having been accessed before. Each such access adds to memory BW requirement and also has to pay the memory latency cost for the first few cache lines before the hardware prefetcher kicks in.

Cache blocking. Therefore, we need to apply cache blocking to take advantage of the *cache reuse* present in the graph, and avoid random gathers. We can either block $f_{\mathcal{V}}$ and run through entire f_O for every block of $f_{\mathcal{V}}$ or block f_O and run through entire $f_{\mathcal{V}}$ for every block of f_O. In the latter case, we need to parallelize across source vertices (u) leading to race conditions on destination vertices (v). Therefore, we block $f_{\mathcal{V}}$. Alg. 2 illustrates how we apply cache blocking using blocks of size B. First, we create n_B blocks by creating a CSR matrix for each block for easy access of neighbors. For each block, we go through all the destination vertices (v) in parallel ensuring that all threads work on one block of B source vertices at a time. As a result, any feature vector in $f_{\mathcal{V}}$ read by some thread t could be in the L2 cache of the CPU if/when some other thread t' reads the same feature vector. Therefore, $f_{\mathcal{V}}$ is read from memory only once but we make n_B passes over f_O. Each additional pass of f_O adds to BW requirement. Hence, the block size (B) should be as large as possible while ensuring that a block of $f_{\mathcal{V}}$ can fit into cache. Sparser graphs allow us to have larger B as not all feature vectors of $f_{\mathcal{V}}$ are active in a block. However, finding the best block size is challenging since many graphs follow a power law and there could be vertices with extremely large neighborhoods resulting in more feature vectors being active in the corresponding block of $f_{\mathcal{V}}$.

Algorithm 2 Application of Blocking on Aggregation Primitive

Require: Matrix A of size $|\mathcal{V}| \times |\mathcal{V}|$ in CSR format
Require: Input feature matrix $f_{\mathcal{V}}$ of size $|\mathcal{V}| \times d$ (Vertex feature set)
Require: Input feature matrix $f_{\mathcal{E}}$ of size $|\mathcal{E}| \times d$ (Edge feature set)
Require: Output feature matrix f_O of size $|\mathcal{V}| \times d$ (Initialized to zeros)
Require: Unary/Binary operator: $\otimes$, Reduction operator: $\oplus$
Require: Block size, B
1: $n_B \leftarrow \left\lceil \frac{|V|}{B} \right\rceil$ {Number of blocks}
2: $\{A_0, A_1, \ldots, A_{n_B-1}\} \leftarrow$ Create CSR matrices for all blocks
3: **for** $i \in 0, \ldots, n_B - 1$ **do**
4: **for** $v \in \mathcal{V}$ in parallel **do**
5: **for** u in $A_i[v]$ **do**
6: $f_O[v] \leftarrow f_O[v] \oplus (f_{\mathcal{V}}[u] \otimes f_{\mathcal{E}}[e_{uv}])$
7: **end for**
8: **end for**
9: **end for**

Multithreading. Many graphs following the power law could result in a significant difference in number of neighbors across vertices. Therefore, we use dynamic thread scheduling with OpenMP, allocating a chunk of contiguous destination vertices at a time to ensure that a thread writes to contiguous feature vectors of f_O.

Loop reordering and Vectorization with LIBXSMM. Applying SIMD to the innermost loop of Alg. 2 (lines #5-7) for the large variety of binary/unary and reduction operators in DGL using manually written intrinsics could be a time consuming task. Instead, we use the LIBXSMM library [14] that provides highly architecture optimized primitives for many matrix operations including our use-cases. LIBXSMM reorders the loop (Alg. 3) to ensure each $f_O[v]$ is written to only once per block. It generates optimal assembly code with SIMD intrinsics where applicable using JITing thus providing more instruction reduction than manually written intrinsics based code.

Algorithm 3 Reordering of the loop at lines #5-7 in Alg. 2

Require: SIMD Width, W
1: **for** $j \in 0, \ldots, d-1$, step W **do**
2: $t \leftarrow f_O[v][j : j + W - 1]$
3: **for** u in $A_i[v]$ **do**
4: $t \leftarrow t \oplus (f_{\mathcal{V}}[u][j : j + W - 1] \otimes f_{\mathcal{E}}[e_{uv}][j : j + W - 1])$
5: **end for**
6: $f_O[v][j : j + W - 1] \leftarrow t$
7: **end for**

5 DISTGNN: DISTRIBUTED-MEMORY ALGORITHM

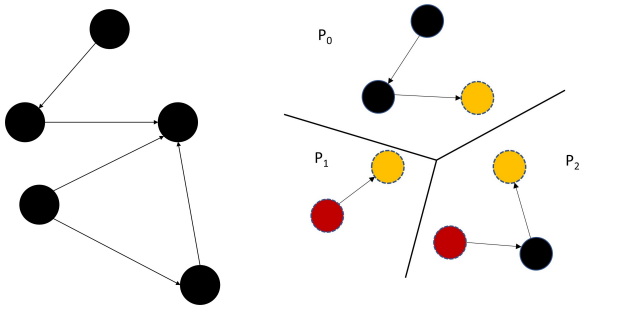
In this section, we describe our distributed algorithms using GraphSAGE GNN model. As seen in the previous sections, feature aggregation operation is the dominant runtime component. GNN models consist of a relatively small enough number of parameters, which can be processed on a single socket. However, with the increase in graph size, the aggregation operation is limited by the available memory. Hence, our distributed parallel solutions use data-parallelism. The model, being smaller in size, is replicated on the sockets and the input graph is partitioned.

It is desirable that partitions communicate less frequently during aggregation. Ideally, partitions would be fully self-contained and require no communication with each other; however, this is not practical and will result in lower training accuracy. In practice, we partition the graph to minimize communication.

5.1 Graph Partitioning

Real-world graphs follow power-law degree distribution. [2] shows that vertex-cut produces minimal cuts for power-law graphs. Distributed graph processing frameworks have shown the efficacy of vertex-cut based partitioning [11]. In this work, we use vertex-cut based graph partitions. These partitioning techniques distribute the edges among the partitions. Thus, each edge is present in only one partition, while a vertex can reside in multiple ones. Each vertex that splits due to the vertex-cut carries with it a partial neighborhood of the original vertex from the input graph. Thus, any update to such a vertex must be communicated to its clones in other partitions. The number of such clones (for each original vertex that is split) is called *replication factor*. The goal of the partitioning is

two-fold: (a) to reduce communication across partitions and (b) generate balanced partitions. The first goal can be achieved by maintaining a lower replication factor. Towards the second goal, we use uniform edge distribution as the load balancing metric. In this work, we use a state-of-the-art tool called *Libra* [32] to partition our un-weighted input graph. Libra works on a simple principle for graph partitioning. It partitions the edges by assigning them to the least-loaded relevant (based on edge vertices) partition. Libra uses a greedy heuristics-based approach to partition the edges such that replication factor is minimized. Also, Libra aims to balance partitions by randomly distributing edges across partitions. For GNN benchmark datasets, we observe that Libra produces balanced partitions in terms of edges.



(a) Original graph **(b) Partitioned graph into three partitions**

Figure 1: Example original and vertex-cut partitioned graph

5.2 Partition Setup

A partition contains two types of vertices: split-vertices and non-split vertices. Split-vertices have their own copy of the feature vector. Thus, each vertex in a partition is associated with a feature vector and takes part in local aggregation. All split-vertices communicate to receive feature vectors from their neighborhood and perform aggregation. Vertices in each partition have global and local IDs. Global IDs are the vertex IDs from the original input graph. We assign consecutive local IDs to vertices starting from partition 0 to partition $n-1$. A global data structure `vertex_map` stores the local ID range of each partition. A vertex's local ID, along with the `vertex_map`, pinpoints its partition and location within the partition. For each split-vertex, we use an array to store the local IDs of all its clones.

5.3 Delayed Remote Partial Aggregates

Once the input graph is partitioned, we assign each partition to a socket for training the model. We implement the following three distributed algorithms with varying communication intensity during the aggregation operation.

(1) `0c`: During training, at each layer, each partition first performs local aggregation. After local aggregation, each split-vertex has aggregated from its local partial neighborhood. `0c` completely avoids communication by ignoring remote partial neighborhoods of split-vertices. Due to no communication and its related pre- and post-processing computations, `0c` is the fastest of all the three proposed algorithms and provides a performance roofline for scaling GraphSAGE on the given dataset across multiple CPU sockets. It is also the most optimistic with respect to accuracy. We evaluate its accuracy in section 6.

(2) `cd-0`: During model training, at each layer, each partition perfoms local aggregation. All split-vertices communicate partial aggregates to their remote clones. Consequently, each vertex in each partition receives its complete neighborhood. `cd-0` waits for communication to be completed before moving to the next step. Since each vertex receives its complete neighborhood, it is expected to produce the same accuracy as the single socket algorithm. `cd-0` provides a lower-bound performance for scaling GraphSAGE on the given dataset across multiple CPU sockets.

(3) `cd-r`: Even after partitioning, the communication cost could be overwhelming (demonstrated in section 6). To further reduce communication volume, we apply a communication avoidance mechanism. In this algorithm, we overlap remote aggregate communication with local computation. The dependence between consecutive steps in an epoch leaves no scope for intra-epoch overlap. Consequently, we apply inter-epoch compute-communication overlap. `cd-r` delays communicating partial aggregates among split-vertices; each split-vertex starts communication in epoch i and asynchronously receives and processes aggregates in epoch $(i + r)$. We believe that delaying feature-vector aggregation during GNN training will lead to better scalability without significant loss of accuracy. This idea of delaying the aggregate updates is inspired by Hogwild! [25] which employs this scheme to delay weight updates. Communication can be further reduced by involving only a subset of split-vertices (through binning) in each epoch. We assess the impact on accuracy in section 6 and demonstrate that accuracy is within 1% of the state-of-the-art baseline for each dataset.

The operation of distributed aggregation using the Delayed Remote Partial Aggregates (DRPA) algorithm is described in Algorithm 4. A 1-level tree structure facilitates communication among the split-vertices in `cd-0` and `cd-r`. For each original vertex $i \in v_s$, we create a tree T_i in which we randomly assign one of its split-vertices as the root, while the rest of them become leaves. To synchronize aggregates across split-vertices, communication between leaves and the root occurs in two phases: (i) all leaves send their partial aggregates to the root (Line 11), (ii) the root receives and aggregates them and then communicates the final aggregates back to the leaves (Lines 13-16). Each partition performs a *pre-processing* and *post-processing* step for each partial aggregate communication. The pre-processing step involves local gather operation (Lines 10, 15); it gathers features of split-vertices. Post-processing involves a local scatter-reduce operation (Lines 14, 20). All received partial aggregates are scattered and reduced to corresponding vertices. Note that, only scatter operation is performed in the post-processing of root to leaf communication (Line 20). DRPA behaves like `cd-0` when there is no delay for the communication i.e. $r = 0$. When

$r > 0$, DRPA functions as cd-r, performing delayed partial aggregations. Ignoring all the communication and associated pre- and post-processing in DRPA produces the functionality of 0c.

Algorithm 4 Delayed Remote Partial Aggregates

Require: Graph partitions G_p along with the vertices V_p and features f_{V_p}
Require: v_s, a set of original vertices of G_p which get split
Require: Tree T_i for vertex $v_s[i], \forall i \in [0, |v_s|)$.
Require: Root features $T_i.root \in f_{V_p}$ and leaf features $T_i.leaf \in f_{V_p}$
Require: Delay parameter r

Allocate G_p per socket c
for each c in parallel **do**
  $k \leftarrow |v_s|/r$
  for i=1 *to* r **do**
    $S_i \leftarrow \{T_{i*k} \ldots T_{(i+1)*k}\}$
  end for
  for epoch e **do**
    $f_v \leftarrow \oplus(\text{copy}(f_u), f_v) \quad \forall u, v \in V_p$
    $i \leftarrow e\%r$
    $f_{v_{sl}} \leftarrow \text{gather}(S_i[j].leaf, \ \forall j)$
    async_send($f_{v_{sl}}$)
    if $e \geq r$ **then**
      $S_i[j].root \leftarrow \text{async\_recv}(f_{v_{sl}}), \forall j$
      $f_V \leftarrow \text{scatter\_reduce}(S_i[j].root), \forall j$
      $f_{v_{sr}} \leftarrow \text{gather}(S_i[j].root), \forall j$
      async_send($f_{v_{sr}}$)
    end if
    if $e \geq 2 \times r$ **then**
      $S_i[j].leaf \leftarrow \text{async\_recv}(f_{v_{sr}}), \forall j$
      $f_V \leftarrow \text{scatter}(S_i[j].leaf), \forall j$
    end if
  end for
end for

6 EXPERIMENTAL EVALUATION

6.1 Experiment Setup

We perform our single-socket experiments on Intel Xeon 8280 CPU @2.70 GHz with 28 cores (single socket), equipped with 98 GB of memory per socket; the theoretical peak bandwidth to DRAM on this machine is 128 GB/s. The machine runs CetnOS 7.6. For distributed memory runs, we use a cluster with 64 Intel Xeon 9242 CPU @2.30 GHz with 48 cores per socket in a dual-socket system. Each compute node is equipped with 384 GB memory, and the compute nodes are connected through Mellanox HDR interconnect with DragonFly topology. The machine runs CentOS 8. We use a single-socket machine with memory capacity of 1.5TB to measure the single-socket runtime for OGBN-Papers dataset.

We use GCC v7.1.0 for compiling DGL and the backend PyTorch neural network framework from their source codes. We use a recent release of DGLv0.5.3 to demonstrate the performance of our solutions and PyTorch v1.6.0 as the backend DL framework for all our experiments. We use PyTorch Autograd profiler to profile the performance of the applications.

Datasets. Table 2 shows the details of the five datasets: AM, Reddit, OGBN-Products, OGBN-Papers, and Proteins [4, 29], used in our experiments. HipMCL [4] generated Proteins graph using isolate genomes from IMG platform. It performs sequence alignment among the collected sequences to generate a graph matrix. Entries in the graph matrix are generated based on sequence similarity scores. In the absence of vertex embeddings, we randomly generate features for the Proteins dataset. The AM dataset contains information about artifacts in the Amsterdam Museum [9]. Each artefact in the dataset is linked to other artifacts and details about its production, material, and content. It also has an artifact category, which serves as a prediction target. For this dataset, in the absence of vertex features, vertex ID is assigned as the feature.

Models and Parameters. In the GraphSAGE model, we use two graph convolutional layers for the Reddit dataset with 16 hidden layer neurons. For the remaining datasets, we use three layers with the 256 hidden layer neurons. In this paper, we employed GCN aggregation operator where (i) $\oplus$ is *element-wise sum* and (ii) as a post-processing step, it adds the aggregated and original features of each vertex and normalizes that sum with respect to the in-degree of the vertex. All the reported epoch run-times are averaged over 1-10 epochs for 0c and cd-0 algorithms, while for cd-r we average run-time for epochs 10-20 due to the communication delay of 5.

Table 2: GNN benchmark datasets. Edges are directed. Each original un-directed edge of Reddit, OGBN-Products, and Proteins is converted into two directed edges.

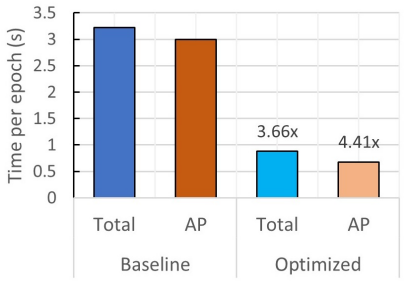
Datasets	#vertex	#edge	#feat	#class
AM	881,680	5,668,682	1	11
Reddit	232,965	114,615,892	602	41
OGBN-Products	2,449,029	123,718,280	100	47
Proteins	8,745,542	1,309,240,502	128	256
OGBN-Papers	111,059,956	1,615,685,872	128	172

Implementation in DGL. All our optimized single code is written in C++ as part of DGL's backend. Our optimizations use LIBXSMM library [14].

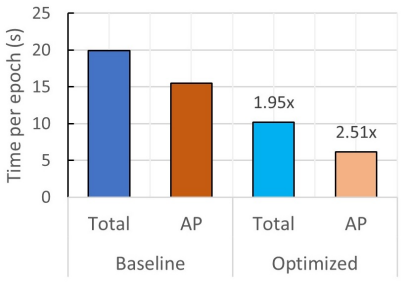
For distributed code, we use `torch.distributed` package of PyTorch coupled with Intel OneCCL [26] for efficient collective communication operations, with one MPI rank per socket; two cores on each socket are dedicated to OneCCL. We use AlltoAll collective for communicating the partial aggregates between the root and leaves in the 1-level tree. For parameter sync among the models, in each epoch, we use AllReduce collective operation. We note that this collective communication operation is synchronous *i.e.*, all compute nodes exchange weight gradients every iteration. All the distributed code is written in Python and C++ in DGL. Our code is available at https://github.com/dmlc/dgl/pull/2914 (commit: cfb73e2).

6.2 Single-Socket Performance

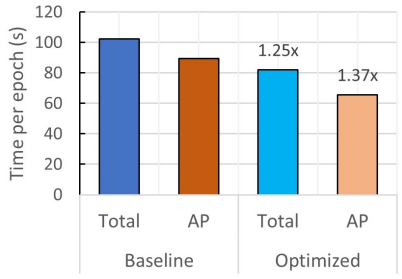
Application performance. Figure 2 compares the per epoch training time and execution time of AP for DGL 0.5.3 with our optimized implementation for four of our workloads that fit on a single socket with 98GB memory. It confirms that the runtime of many applications is dominated by AP. Our optimizations achieve up to 4.41× speedup for AP, thereby, achieving up to 3.66× speedup for end-to-end training time over DGL 0.5.3. Optimization of AP for shared memory systems was also presented in [3]. Due to lack of availability of source code, we can only compare with the results presented



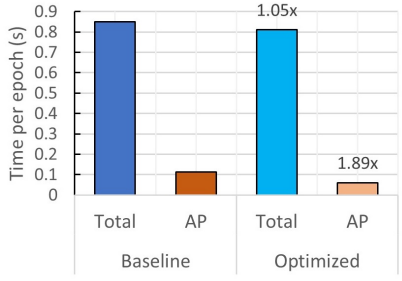
(a) GraphSAGE on Reddit



(b) GraphSAGE on OGBN-Products



(c) GraphSAGE on Proteins



(d) RGCN-hetero on AM

Figure 2: Comparison of runtime of entire training epoch (Total) and Aggregation Primitive (AP) for baseline DGL and our optimized version on the benchmark datasets. The labels on top of bars for optimized version show the speedup for Total time and AP time, respectively.

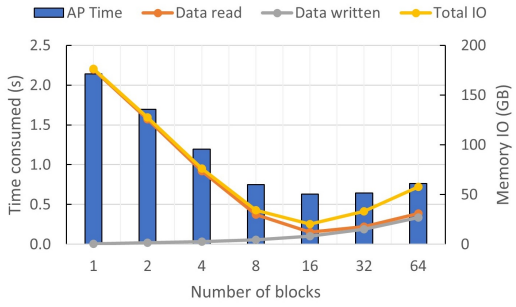
in that paper. [3] reports training time of 2.7 secs/epoch for Reddit achieving a speedup of 7.7× over DGL 0.4.3, while we consume only 0.88 secs/epoch. Compared to the DGL 0.4.3, our current performance on Reddit is nearly 24× faster.

In the following, we explain the factors that affect performance gain of AP kernel using the two benchmarks that achieve the highest speedup. Analysis of access to edge features is straightforward since they are accessed in streaming fashion. Therefore, we focus only on the vertex features by using ⊗ as copylhs. Without loss of generality, we use ⊕ as sum in our analysis.

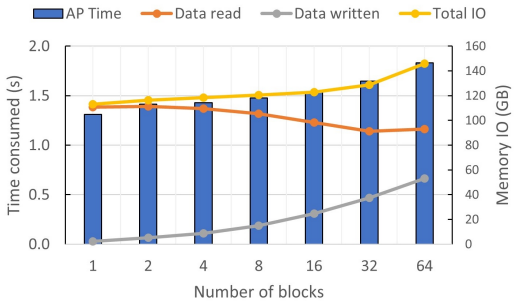
Table 3: Cache reuse achieved for the AP kernel with respect to density of graph and the number of blocks (n_B). Density is defined as the number of non zero cells divided by total cells in the adjacency matrix. Ideal cache reuse is the average vertex degree of the graph – 492 for Reddit and 50.5 for OGBN-Products.

		n_B						
Dataset	Density	1	2	4	8	16	32	64
Reddit	0.002	3.1	4.3	7.3	16.1	27.0	16.7	9.6
OGBN-Products	0.00002	2.3	2.2	2.2	2.1	2.1	2.0	1.8

Effect of block size. Table 3 and Figure 3 illustrate the effect of block size (B), and hence, the number of blocks (n_B). In the ideal scenario, every feature vector ($f_{\mathcal{V}}[u]$) would be loaded only once from memory and would be used to update feature vectors in f_O of all its neighbors. Similarly, every feature vector in f_O would also be



(a) Reddit



(b) OGBN-Products

Figure 3: Time consumed and amount of data read, written and total memory IO (data read + written) for AP with respect to the number of blocks (n_B).

written to memory only once. Therefore, maximum average reuse of the feature vectors in $f_\mathcal{V}$ and $f_\mathcal{O}$ is the average vertex degree of the graph. While ideal reuse is not possible, our blocking method tries to maximize reuse. Clearly, the sparser the graph, the less likely it is to reuse the feature vectors. When we use just one block, the blocks of $f_\mathcal{V}$ start getting thrashed out of cache after processing a few rows of the adjacency matrix, thus, preventing reuse. As the blocks get smaller, chances of blocks of $f_\mathcal{V}$ staying in the cache increase, thereby, increasing reuse in $f_\mathcal{V}$. At the same time, we need more passes of $f_\mathcal{O}$ decreasing its reuse. This gets reflected in data read from and written to memory as clearly seen in Figure 3. The best performance is at the sweet spot where the sum of data read and written is the smallest. For denser graphs (like Reddit), the sweet spot is more to the right compared to sparser graphs (like OGBN-Products).

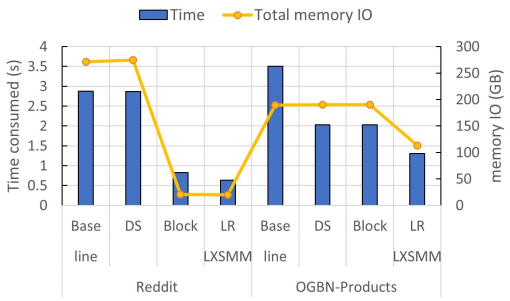


Figure 4: Effect of Dynamic Scheduling (DS), Blocking (Block) and Loop Reordering with LIBXSMM (LR LXMM) on the total memory IO and execution time of AP for Reddit and OGBN-Products.

Breakup of speedup with respect to optimizations. Figure 4 shows the breakup. There is a clear correlation between memory IO required and execution time. Dynamic scheduling has no effect for Reddit but plays a major role in improving performance for OGBN-Products. On the other hand, blocking has a massive impact for Reddit but has no effect for OGBN-Products where we end up using only one block. Loop reordering and JITing with LIBXSMM improves the performance in both cases.

6.3 Distributed Algorithm Performance

Table 4: Average replication factor due to vertex-cut based graph partitioning using Libra

Datasets/ #Partitions	2	4	8	16	32	64	128
Reddit	1.75	2.94	4.66	6.93	-	-	-
OGBN-Products	1.49	2.16	2.98	3.90	4.85	5.74	-
Proteins	1.33	1.65	1.91	2.11	2.27	2.37	-
OGBN-Papers	-	-	-	-	4.63	5.63	6.62

Graph Partition. As seen in the previous sections, data movement during aggregation is directly related to the number of edges, and hence the run-time. Thus, we use the simple criteria of equal edge allocation among the partitions as the load balancing mechanism. Libra, despite having no hard constraints on maintaining an equal distribution of edges to the partitions, produces highly balanced partitions in terms of the number of edges.

Table 4 shows the average vertex replication for a different number of partitions produced by Libra. Reddit, the densest of the benchmark datasets, results in relatively more split-vertices during partitioning than all other datasets. Proteins results in a significantly smaller replication factor. It exhibits natural clusters of protein families (sequence homology), thus leading itself to high-quality partitioning. OGBN-Products and OGBN-Papers, with the least average vertex degree, have similar replication factors. A lower replication factor is desirable. On single socket, a higher replication factor leads to a sparser partitioned graph which results in more pressure on the memory bandwidth, whereas in a distributed setting, an increase in replication factor with the number of partitions leads to more communication and hampers the scalability of the solution.

Scaling. In all our experiments, we run `cd-r` algorithm with delay of $r = 5$ epochs. Figure 5 shows the per epoch time and speed-up of our solutions with increasing socket-count. Owing to the high replication factor in Reddit partitioning, the decrease in partition size from 2 to 16 partitions is highly sub-linear. This directly leads to a sub-linear decrease in local and remote aggregation time. For 16 sockets, we observe 0.98×, 2.08×, and 2.91× speed-up using `cd-0`, `cd-5`, and `0c`, respectively compared to the optimized DGL single-socket performance. In contrast to Reddit, the Proteins dataset exhibits nearly linear decrease in partition size from 2 to 64 partitions. For 64 sockets, we observe 37.9×, 59.8×, and 75.4× speed-up using `cd-0`, `cd-5`, and `0c`, respectively compared to the optimized DGL single-socket performance. Due to the usage of memory from multiple Non-Uniform Memory Access (NUMA) domains, the single-socket run is slower, leading to a super-linear speedup for `0c`.

The quality of partitions for OGBN-Products, as reflected in its replication factor, is in-between those of Reddit and Proteins datasets. OGBN-Papers has slightly better quality partitions compared to OGBN-Products. For OGBN-Products, for 64 sockets, we observe 6.3×, 9.9×, and 16.1× speed-up using `cd-0`, `cd-5`, and `0c`, respectively compared to the optimized DGL single-socket performance.

For OGBN-Papers, for 128 sockets, we observe 27.43×, 83.16×, and 123.13× speed-up using `cd-0`, `cd-5`, and `0c`, respectively compared to the optimized DGL single-socket performance. Compared to un-optimized DGL on single socket, we observe 32×, 97×, and 143× speed-up using `cd-0`, `cd-5`, and `0c`, respectively. Here, due to very high memory requirements (1.4TB), a single-socket run uses memory from multiple NUMA domains and thus runs slower. Similarly, 32 and 64 socket runs also use memory from multiple NUMA domains and run slower. Owing to high memory demands, we could not run OGBN-Papers dataset below 32 sockets. Note that for multi-socket runs, two cores per socket are reserved for OneCCL library.

As described in Section 5, the aggregation step involves local and remote aggregation sub-steps in `cd-0` and `cd-r`, whereas for `0c`, it comprises of only the former sub-step. Remote aggregation

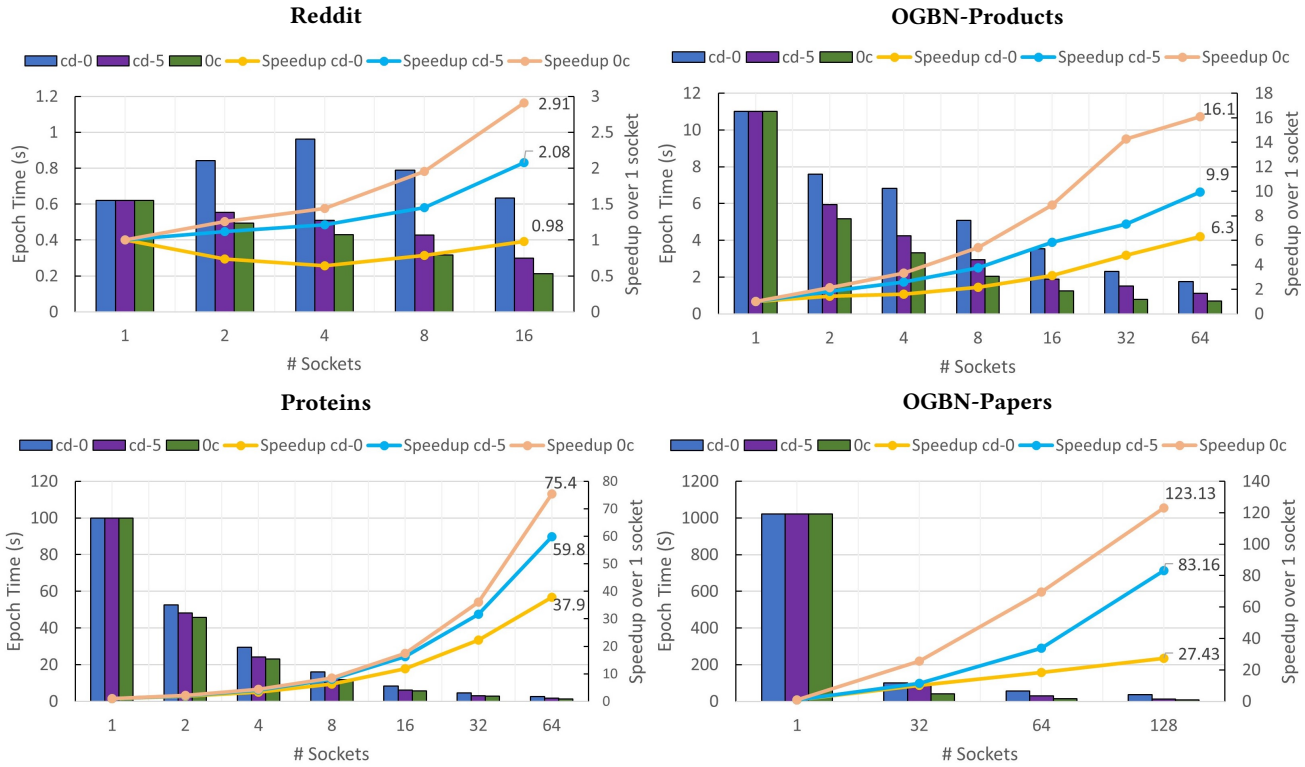


Figure 5: Runtime performance and speedup of three distributed algorithms of DistGNN on the benchmark datasets.

not only involves actual communication but also pre- and post-processing. Local aggregation does not involve communication, therefore local aggregation time (LAT) remains the same across cd-0, cd-r, and 0c. Figure 6 shows that in the forward pass, LAT scales linearly with the number of sockets, except for Reddit. It also shows that remote aggregation time (RAT) scales poorly with the increase in the number of sockets; this is solely an artifact of the replication factor and number of split-vertices per original vertex. For cd-5, we observe that a negligible amount of time is spent in waiting for asynchronous overlapped communication; thus, RAT is purely composed of pre- and post-processing times. For Reddit and Proteins, local aggregation dominates remote aggregation. For OGBN-Products, remote aggregation consumes a significant portion of the overall execution time beyond 16 sockets, reducing AP scalability. For OGBN-Papers, RAT is always higher than LAT, due to much higher pre- and post-processing cost compared to other datasets. Due to high communication volume in cd-0, RAT is higher than LAT for all datasets, except Proteins.

Accuracy. The GraphSAGE model reports single-socket test accuracy of 93.40% and 77.63% for Reddit and OGBN-Products respectively. We evaluate the test accuracy of all the three distributed algorithms on a number of partitions (Table 5). We use the delay factor $r = 5$ for our cd-r experiments. For both the datasets, all the distributed algorithms report the accuracy within 1% of the best accuracy. For 8 and 16 sockets using Reddit dataset, we observe that accuracy recovers with increased training time from 200 to 300 epochs. Algorithms cd-5 and 0c, for some cases, on both the datasets, report accuracy better than single-socket accuracy. One possible reason for this increase is clustering due to partitioning. In all the experiments with cd-r, we do not see any discernible improvements in accuracy with values of $r < 5$, while large values of r (e.g., $r = 10$) degraded the accuracy due to increasingly stale feature aggregates.

Due to the absence of the features and labeled data for the Proteins dataset, we were unable to validate training accuracy with test data. For OGBN-papers, we use GraphSAGE as the primary GNN model – same as the Dist-DGL. We show how to scale OGB-papers across 128 sockets. While the accuracy of vanilla GraphSAGE (Table 5) for this dataset is significantly lower than that reported in OGB leaderboard [16], there are techniques (such as [28]) to bridge the accuracy gap, which we plan to explore. Table 5 also reports the test accuracy attained by our distributed algorithms. Further experiments with hyperparameters tuning are required for OGBN-Papers dataset.

Memory and Communication Analysis. In this section, we chart out the memory requirements of the GraphSAGE model and report the actual memory consumption for the OGBN-Papers dataset. As discussed in Section 6.1, the GraphSAGE model contains three layers; within each layer, the neighborhood aggregation step is followed by the multi-layer perceptron (MLP) operations. Let N, f, h_1 and h_2, and l be the number of partition vertices, features, first hidden layer neurons, second hidden layer neurons, and labels,

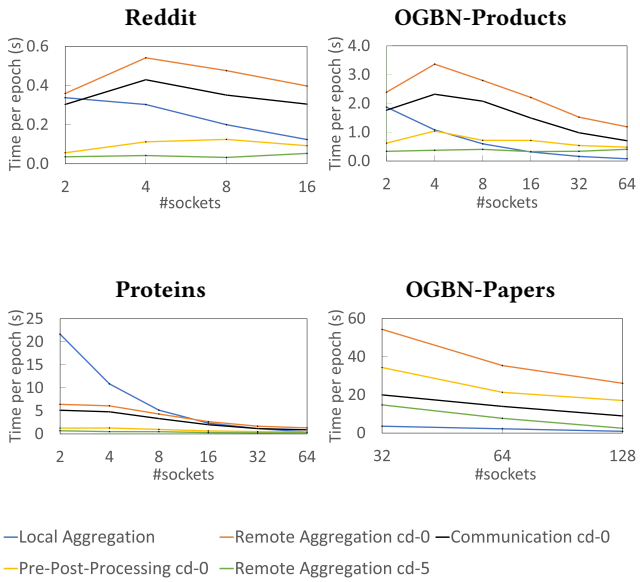


Figure 6: Forward pass scaling performance of local and remote aggregate operation of `cd-0`, `cd-5`, and `0c` on benchmark datasets. The remote aggregation involve pre- and post-processing time for communication. Due to the absence of communication in `0c`, its time is same as local aggregation time.

respectively. Also, let w_1, w_2, and w_3 be the weight matrices at each of the three layers of the `MLP`. Memory required by the GraphSAGE model is as follows. (1) The weight matrix dimensions, w_1: $f \times h_1$, w_2: $h_1 \times h_2$, and w_3: $h_2 \times l$. (2) The input feature matrix dimensions, $N \times f$. (3) The neighborhood aggregation output dimensions at each of the three layers, $N \times f$, $N \times h_1$, and $N \times h_2$. (4) Similarly, the `MLP` operation output dimensions at each of the three layers, $N \times h1$, $N \times h2$, and $N \times l$. The intermediate results at each layer need to be stored to facilitate the backpropagation of the gradients. Additionally, in a distributed setting, memory is required for buffering the data for communication. In `cd-0` and `cd-r` algorithms, the amount of communication per partition is directly proportional to the number of split-vertices in the partitions. Table 6 shows the memory consumption of the distributed algorithms and the percentage of split-vertices per partition for a different number of partitions for OGBN-Papers dataset.

Comparison with Current Solutions. As far we know there is no state-of-the-art for distributed full batch training on CPUs. Therefore, we compare our solution with a recent distributed solution for *mini-batch* training with neighborhood sampling on CPUs, called `Dist-DGL`. On account of a different aggregation strategy used in `Dist-DGL`, we compare our full-batch training approach with it based on total aggregation work and time per epoch. We use the same GNN model architecture as `Dist-DGL` for OGBN datasets. In `Dist-DGL`, during each mini-batch computation, the amount of aggregation work varies at each hop, depending on the number of vertices, their average degree (fan-out), and feature vector size. For full-batch training, the amount of work at each hop varies with feature vector size only.

Out of 2,449,029 vertices of OGBN-Products dataset, 196,615 are labeled training vertices. Table 7 & 8 show the number of vertices, average degree, and feature vector size per hop. In `Dist-DGL`, with equal distribution of training vertices per socket, each socket processes a roughly equal number of batches. Table 7 & 8 also highlights the total work that `Dist-DGL` and `DistGNN` do on single and 16 sockets, in terms of Billions of Ops (B Ops). The total work per hop is calculated as the product of number of vertices, feature size, and average vertex degree. Our solution performs $\approx 4\times - 13\times$ more work ($\approx 77.18e^9/19.98e^9$ ops) and ($\approx 18.8e^9/1.41e^9$ ops), respectively, per epoch than `Dist-DGL` due to complete neighborhood aggregation. However, even with this increase in amount of work, our solution reports comparable or even better epoch time on similar hardware

Table 5: Test accuracy of single socket and distributed algorithms using Reddit, OGBN-Products, OGBN-Papers dataset, with corresponding learning rate (lr) and number of epochs. We set weight decay, $wd = 5e^{-4}$ for all the experiments.

Reddit							
	`cd-0`		`cd-5`		`0c`		
#sockets	Acc.(%)	lr	Acc.(%)	lr	Acc.(%)	lr	#epochs
1	93.40	0.01	93.40	0.01	93.40	0.01	200
2	93.70	0.028	93.59	0.028	93.58	0.028	200
4	93.44	0.028	93.25	0.028	93.39	0.028	200
8	93.14	0.028	93.33	0.08	93.14	0.07	300
16	92.86	0.028	92.62	0.08	92.38	0.07	300
OGBN-Products							
	`cd-0`		`cd-5`		`0c`		
#sockets	Acc.(%)	lr	Acc.(%)	lr	Acc.(%)	lr	#epochs
1	77.63	0.01	77.63	0.01	77.63	0.01	300
2	77.12	0.05	77.65	0.05	78.42	0.08	300
4	77.35	0.05	79.14	0.07	78.91	0.08	300
8	77.49	0.08	79.18	0.07	79.10	0.08	300
16	77.47	0.08	78.00	0.08	78.95	0.08	300
32	77.45	0.07	78.00	0.08	78.37	0.08	300
64	77.25	0.07	77.64	0.08	77.76	0.08	300
OGBN-Papers							
	`cd-0`		`cd-5`		`0c`		
#sockets	Acc.(%)	lr	Acc.(%)	lr	Acc.(%)	lr	#epohs
1	41.29	0.03	41.29	0.03	41.29	0.03	200
128	37.9	0.01	37.65	0.01	36.74	0.01	200

Table 6: Per epoch peak memory requirements of the distributed algorithms and split-vertices percentage in a partition for OGBN-Papers dataset.

Partitions	32	64	128
`cd-0` Memory (GB)	199	124	78
`cd-5` Memory (GB)	311	196	120
`0c` Memory (GB)	180	112	70
Split-vertices/partition (%)	90	92	93

(Dist-DGL uses either of Intel Xeon Skylake or Cascade Lake CPU with 96 VPUs on AWS instance m5.24xlarge) (Table 9). We see a similar trend with OGBN-Papers dataset which has 1,207,179 training vertices out of 111,059,956 vertices in the graph.

Table 7: Aggregation work done (billion operations) per hop, per mini-batch, and per socket using neighborhood sampling by Dist-DGL. The mini-batch size is 2000. Dataset used: OGBN-Products.

Hops	#vertices	Avg. deg.	#feats	Total work per socket (B ops)
Hop-2	233,692	5	100	0.116
Hop-1	30,214	10	256	0.077
Hop-0	2,000	15	256	0.007
1 Mini-batch				0.202
1 Socket (99 Mini-batches per socket)				19.98
16 Sockets (7 Mini-batches per socket)				1.41

Table 8: Aggregation work done per hop and per full batch using complete neighborhood aggregation by DistGNN. Here full batch represents a partition and each socket executes one partition. Dataset used: OGBN-Products.

#sockets	Hops	#vertices/ partition	Avg. deg.	#feats	Total work per socket (B Ops)
1	Hop-2	2,449,029	51.5	100	12.61
	Hop-1	2,449,029	51.5	256	32.29
	Hop-0	2,449,029	51.5	256	32.29
	Full Batch				77.19
16	Hop-2	596,499	51.5	100	3.07
	Hop-1	596,499	51.5	256	7.86
	Hop-0	596,499	51.5	256	7.86
	Full Batch				18.80

Table 9: Training time for Dist-DGL and DistGNN on OGBN-Products.

#sockets	Dist-DGL time (s)	DistGNN (cd-5) time (s)
1	20	11
16	1.5	1.9

CAGNET, a distributed solution to GNN training, performs complete neighborhood aggregation on a GPU cluster. Due to very different cluster configuration and intra- and inter-node network topologies, we do not compare DistGNN with CAGNET.

7 CONCLUSION AND FUTURE WORK

In this paper, we present DistGNN, the first ever and a highly efficient distributed solution for full-batch GNN training on Intel Xeon CPUs. The aggregate operation is data-intensive. On a single socket Intel Xeon CPU, we accelerated it by identifying and optimizing computational bottlenecks. Our distributed solutions employ avoidance and reduction algorithms to mitigate communication bottlenecks in the aggregation operation. To reduce communication volume, we leveraged vertex-cut based graph partitioning and overlapped communication with computation across epochs as delayed partial aggregates. We demonstrated the performance of DistGNN on a set of common benchmark datasets with the largest one having hundred million vertices and over a billion edges. With our delayed partial aggregate algorithms the accuracy is sometimes better than single-socket run and remains within 1% with the increase in number of partitions. 0c algorithms showing superior accuracy on OGBN-Products datasets points to the clustering effect on the accuracy.

In future work, we expect to demonstrate highly scalable DistGNN for mini-batch training across various datasets. We will further analyze the accuracy of 0c and cd-r algorithms on various datasets and different GNN model architectures that have state-of-the-art accuracy. We also expect to extend DistGNN to different GNN models, beyond GraphSAGE. To further reduce communication volume, we will deploy low-precision data formats such FP16 and BFLOAT16, with convergence using DistGNN.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 2000. Error and attack tolerance of complex networks. *nature* 406, 6794 (2000), 378–382.

[3] Sasikanth Avancha, Vasimuddin Md, Sanchit Misra, and Ramanarayan Mohanty. 2020. Deep Graph Library Optimizations for Intel (R) x86 Architecture. *arXiv preprint arXiv:2007.06354* (2020).

[4] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic acids research* 46, 6 (2018), e33–e33.

[5] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine* 34, 4 (2017), 18–42.

[6] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[8] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 257–266.

[9] Victor de Boer, Jan Wielemaker, Judith van Gent, Michiel Hildebrand, Antoine Isaac, Jacco van Ossenbruggen, and Guus Schreiber. 2012. Supporting Linked Data Production for Cultural Heritage Institutes: The Amsterdam Museum Case Study. In *The Semantic Web: Research and Applications*, Elena Simperl, Philipp Cimiano, Axel Polleres, Oscar Corcho, and Valentina Presutti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 733–747.

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 17–30.

[12] William L Hamilton, Payal Bajaj, Marinka Zitnik, Dan Jurafsky, and Jure Leskovec. 2018. Embedding logical queries on knowledge graphs. *arXiv preprint arXiv:1806.01445* (2018).

[13] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[14] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020).

[16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020). https://ogb.stanford.edu/docs/leader_nodeprop/

[17] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. Featgraph: A flexible and efficient backend for graph neural network systems. *arXiv preprint arXiv:2008.11359* (2020).

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[19] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[20] Yucheng Low. 2013. GraphLab: A Distributed Abstraction for Large Scale Machine Learning. (2013).

[21] Guixiang Ma, Yao Xiao, Theodore L Willke, Nesreen K Ahmed, Shahin Nazarian, and Paul Bogdan. 2020. A Vertex Cut based Framework for Load Balancing and Parallelism Optimization in Multi-core Systems. *arXiv preprint arXiv:2010.04414* (2020).

[22] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 443–458.

[23] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[24] Sameh K Mohamed, Vít Nováček, and Aayah Nounu. 2020. Discovering protein drug targets using knowledge graph embeddings. *Bioinformatics* 36, 2 (2020), 603–610.

[25] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *arXiv preprint arXiv:1106.5730* (2011).

[26] Intel OneCCL. 2020. https://github.com/intel/torch-ccl

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[28] Yunsheng Shi, Zhengjie Huang, Shikun Feng, and Yu Sun. 2020. Masked label prediction: Unified massage passing model for semi-supervised classification. *arXiv preprint arXiv:2009.03509* (2020).

[29] Alok Tripathy, Katherine Yelick, and Aydin Buluc. 2020. Reducing Communication in Graph Neural Network Training. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 987–1000.

[30] Robert A Van De Geijn and Jerrell Watts. 1997. Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

[31] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[32] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis.. In *Nips*, Vol. 27. 1673–1681.

[33] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*. 1–6.

[34] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[35] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3165–3166.

[36] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 974–983.

[37] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. *arXiv preprint arXiv:2010.05337* (2020).

[38] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. 2018. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics* 34, 13 (2018), i457–i466.

,

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Installation steps (DistGNN):

1. Copy dgl/setup_env.sh, dgl/install_extra_dep.sh, && dgl/env.sh to a desired location XYZ (After this you may discard dgl folder, as the scripts below will setup dgl separately)

2. cd XYZ

3. Set compiler, gcc 8.3.0

4. Run "XYZ/setup_env.sh"

a. It creates a XYZ/sub407 sub-folder

b. It installs all the dependencies (anaconda, OneCCL, Pytorch, and other dependencies) as well as DGL in a new conda environment called "sub407".

c. The DGL code is now present in sub407 sub-folder

d. The conda environment can be enabled as "source sub407/miniconda3/bin/activate sub407"

5. Run "source XYZ/env.sh"

a. It activates the conda environment "sub407" and sets up all the environment variables

6. Run "XYZ/install_extra_dep.sh"

7. The DGL (DistGNN) installation is ready to run the single socket as well as distributed experiments, with DGL code present in XYZ/sub407/dgl (follow "How to run" described in section D).

8. If you wish to rerun setup_env.sh then remove sub407 folder and rerun the scripts.

How to run (Instructions are also present in <path_to_dgl>/dgl/examples/pytorch/graphsage/experimental/README.md):

1. Single Socket experiments

Notes: Here, numactl is used for best performance in multi-numa domains compute node. If the memory is not sufficient to run the experiments, the run will be killed by the system. Memory requirements for these runs: Reddit (8 GB), ogb-products (29 GB), Proteins (175 GB), AM (6 GB).

cd <path_to_dgl>/dgl/examples/pytorch/graphsage

numactl -N 0 -m 0 python train_full.py –n-epochs 200 –dataset reddit

numactl -N 0 -m 0 python train_full_ogbn-products.py –n-epochs 300 –dataset ogbn-products

numactl -N 0 -m 0 python train_full_proteins.py –n-epochs 200 –dataset proteins

cd <path_to_dgl>/dgl/examples/pytorch/rgcn-hetero

numactl -m 0 -N 0 python entity_classify.py -d am –l2norm 5e-4 –n-bases 40 –testing –gpu -1 –n-epochs 200

2. Distributed-memory experiments

cd <path_to_dgl>/dgl/examples/pytorch/graphsage/experimental

Step 1: Perform input graph partitioning (2.1)

Step 2: Perform distributed-memory runs (2.2)

2.1 Graph partitioning

Notes:

a. Output partitions are created in the current directory.

b. By default it creates 2, 4, 8,16 partitions for Reddit; 2,4,8,16,32,64 partitions for OGBN-Products; 2,4,8,16,32,64 partitions for Proteins; 32,64,128 partitions for OGBN-Papers100M. The number of partitions can be changed in dgl/python/dgl/distgnn/partition/main_Libra.py:213.

c. As of now the Libra partitioning code is single-threaded python code (which also involves data format conversions), so for large datasets (Proteins, OGB), it takes time (in hrs) to produce the partitions.

python ../../../../python/dgl/distgnn/partition/main_Libra.py cora (small example as a demo) python ../../../../python/dgl/distgnn/partition/main_Libra.py reddit python ../../../../python/dgl/distgnn/partition/main_Libra.py ogbn-products python ../../../../python/dgl/distgnn/partition/main_Libra.py proteins python ../../../../python/dgl/distgnn/partition/main_Libra.py ogbn-papers100M

2.2 Distributed-memory runs

Note: By default, the partitions are read from the current directory. Either you can use the scripts below for distributed runs or you can use the command-line below.

I .Scripts:

Reddit: dist_reddit.sh – The script runs 2,4,8,16 partitions. Allocation of 16 compute nodes is required.

OGBN-Products: dist_ogbn_products.sh - The script runs 2,4,8,16,32,64 partitions. Allocation of 64 nodes is required.

Proteins: dist_proteins.sh - The script runs 2,4,8,16,32,64 partitions. Allocation of 64 nodes is required.

OGBN-Papers100M: dist_ogbn_papers.sh - The script runs 32,64,128 partitions. Allocation of 128 nodes is required.

Note: DistGNN runs are resource-intensive (compute, memory, disk, network). The disk space can reach in TBs, while we used 384 GB memory per node for our experiments. In case of limited resources, the scripts would fail. Command-line execution of each experiment is also described below.

II. Manual:

Reddit:

cd-0:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym.py –dataset reddit –n-epochs 200 –nr 1 –lr 0.03

cd-5:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym.py –dataset reddit –n-epochs 200 –nr 5 –lr 0.03

0c:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym.py –dataset reddit –n-epochs 200 –nr -1 –lr 0.03

OGBN-Products:

cd-0:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_ogbn-products.py –dataset ogbn-products –n-epochs 300 –nr 1 –lr 0.03

cd-5:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_ogbn-products.py –dataset ogbn-products –n-epochs 300 –nr 5 –lr 0.03

0c:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_ogbn-products.py –dataset ogbn-products –n-epochs 300 –nr -1 –lr 0.03

Proteins:

cd-0:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_proteins.py –dataset proteins –n-epochs 200 –nr 1 –lr 0.03

cd-5:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_proteins.py –dataset proteins –n-epochs 200 –nr 5 –lr 0.08

0c:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_proteins.py –dataset proteins –n-epochs 200 –nr -1 –lr 0.08

OGBN-Papers100M:

cd-0:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_ogbn-papers.py –dataset ogbn-papers100M –n-epochs 200 –nr 1 –lr 0.08

cd-5:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_ogbn-papers.py –dataset ogbn-papers100M –n-epochs 200 –nr 5 –lr 0.08

0c:

sh run_dist.sh -n <num_nodes> -ppn 1 python train_dist_sym_ogbn-papers.py –dataset ogbn-papers100M –n-epochs 200 –nr -1 –lr 0.08

Installation steps (baseline DGL, for establishing single-socket baseline performance):

1. Copy dgl/setup_env_baseline.sh, dgl/install_extra_dep.sh, && dgl/env_baseline.sh to a desired location XYZ (After this you may discard dgl folder, as the scripts below will setup dgl separately)

2. cd XYZ

3. Set compiler, gcc 8.3.0

4. Run “XYZ/setup_env_baseline.sh”

a. It creates a XYZ/sub407_baseline sub-folder

b. It installs all the dependencies (anaconda, Pytorch, and other dependencies) as well as DGL in a new conda environment called “sub407_baseline”.

c. The DGL code is now present in sub407_baseline sub-folder

d. The conda environment can be enabled as “source sub407_baseline/miniconda3/bin/activate sub407_baseline”

5. Run “source XYZ/env_baseline.sh”

a. It activates the conda environment “sub407_baseline” and sets up all the environment variables

6. Run “XYZ/install_extra_dep.sh”

7. The DGL (Baseline) installation is ready to run the single socket experiments, with DGL code present in XYZ/sub407_baseline/dgl (follow “How to run” described in section D.1).

8. If you wish to rerun setup_env.sh then remove sub407_baseline folder and rerun the scripts.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/dmlc/dgl/pull/2914
↪ (Commit: cfb73e2)
Artifact name: DistGNN
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Xeon 8280/9242 CPU; Memory: 384GB

Operating systems and versions: CentOS 7.6/8.0

Compilers and versions: GCC 8.3.0

Applications and versions: DGLv0.6.0; PyTorch 1.7.1

Libraries and versions: OneCCL (git commit: d386f73); Python version: 3.7.10 ; LIBXSMM (git commit:55c6a9f); cmake-3.19

Key algorithms: Optimized SPMM, Libra graph partitioning

Input datasets and versions: Reddit, OGBv1.1.1, AM, Proteins