DISTRIBUTED HYPERGRAPH NEURAL NETWORKS

Stefan Kramer, Simon Wachter, Yannick Wattenberg

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

We implement a distributed-memory full-batch version of "Hypergraph Neural Networks" by Feng et al [1]. This makes it possible to train on larger graph datasets without using mini-batch techniques, which can introduce additional biases and often cost some precision [2, 3]. We show that our implementation achieves the same accuracy as the original HGNN model. Further, we compare runtimes to the highly optimized and non-distributed model implemented using the torch framework.

1. INTRODUCTION

Graph neural networks (GNN) have grown immensely popular for many different use cases such as modeling of chemical or biological processes, most often in the form of protein folding, social networks analysis, or image analysis. However, the structure of GNNs induces a major bias, namely that interactions between vertices are limited to be pairwise. This is due to the limitation that in graphs only two vertices can be part of a discrete edge. This can be a limiting factor in many fields where this assumption does not hold in the underlying setting. One such example is social network analysis, where most interactions are with a group of people and not just between two people. Another one is image or point-cloud analysis where many pixels might interact with each other. To solve this problem Feng et al. [1] introduced hypergraph neural networks (HGNNs). In a hypergraph, an edge can be connected to many different nodes, which means more complex structures, beyond pairwise interactions, can be modeled. Another limiting factor however is not addressed. By switching to a hypergraph we also increase the dataset size as each edge can incorporate multiple vertices. This in many cases means that the model does not fit in the memory of commonly available systems, examples are the large datasets of OGB shown in table 1.

In our work, we address this issue by implementing a distributed full-batch version of the HGNN model presented by Feng et al. [1].

The author thanks Jelena Kovacevic. This paper is a modified version of the template she used in her class.

Dataset	Nodes	Edges	Size
MAG240M	244,160,499	1,728,364,232	167GB
WikiKG90Mv2	91,230,610	601,062,811	89GB
PCQM4Mv2	52,970,652	54,546,813	≈8GB

Table 1. Examples of large graph datasets by OGB, Stanford [4].

2. BACKGROUND

Graph Neural Networks. GNNs are commonly built from a propagation module, a pooling module and a sampling module. The propagation module accumulates features from the neighborhoods of the vertices. Pooling modules are used to increase the reception field of the neural network. Sampling modules are used to deal with the exponential increase in the size of the neighborhood of nodes through layers. However, as we do not use mini-batch training, we do not need sampling. The two major types of propagation modules divide into spectral and spatial approaches.

Spectral approaches use the spectral representation of a graph. The graph is transformed using the Fourier transform, after which a convolution is applied and the resulting signal is transformed back into the graph domain. Henaff et. al. [5] introduced a parametrization using smooth coefficients to localize the filters spatially. Defferrard et al. [6] use Chebyshev polynomials to approximate the filter based on the work of Hammond et al. [7]. Further, Kipf and Welling [8] simplify the Chebyshev polynomials to the first order to prevent overfitting. Which then yields the original formulation:

$$\mathcal{F}^{-1}(g \star \mathcal{F}(x)) \approx w \left(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}\right) x \qquad (1)$$

A is the adjacency matrix of the graph. And can be further simplified to

$$X^l = \overline{D}^{-\frac{1}{2}} \overline{AD}^{-\frac{1}{2}} XW \qquad (2)$$

Where $\mathcal{F}$ is the Fourier transform, $\star$ is the convolution operator, g, w are the filter, and learned parameters respectively with $\overline{A} = A + I_N$. $\overline{D}_{ii}$ is the column sum of the i-th row

of A and has a regularizing effect. X is the input and W are the learned parameters. The activation, function which is applied elementwise, is omitted in this equation.

Spatial approaches define the convolution directly in the graph domain instead of using an additional conversion step. Spatial convolution is similar to convolution on an image, in that information is propagated in the spatial neighborhood of vertices i.e. along edges. A. Micheli [9] proposed NN4G which can be reformulated to resemble the GCN formulation in (2). Duvenaud et al. [10] first used different weight matrices for vertices of different degrees. Atwood and Towsley [11] leveraged the transition matrix to define the vertex neighborhoods and model the convolution as a diffusion process, where each node propagates its information with a certain probability.

Hypergraph. The hypergraph is a generalization of a graph, where each edge can be adjacent to multiple vertices. A hypergraph G is defined by the tuple $(\mathcal{V}, \mathcal{E}, \mathcal{W})$, where $\mathcal{V}$ are the vertices of the graph, $\mathcal{E}$ defines the hyperedges and $\mathcal{W}$ defines the edge weights. We further define the degree of a vertex as the number of edges to which it is incident and the degree of an edge as the number of vertices that it contains.

Hypergraph Neural Networks. HGNNs extend the previous GNN framework to hypergraphs. In the paper by Feng et al. [1], which is the basis for our implementation, the authors formulate a spectral approach for their graph convolution. This approach is similar to the GCN definition, however, based on the adjacency matrix of the hypergraph:

$$g \star x \approx \sum_{i=0}^{K} \theta_i T_i(\bar{\Delta}) x \tag{3}$$

Here, $\bar{\Delta}$ is the scaled hypergraph Laplacian and $T_i(X)$ the i-th order Chebyshev polynomial scaled with X. They follow the work of Kipf and Welling 2017 [8] on GCNs to limit the approximation to first-order Chebyshev polynomials. This gives the final convolution equation:

$$g \star x \approx \theta D_v^{-\frac{1}{2}} H \mathcal{W} D_e^{-\frac{1}{2}} H^T D_v^{-\frac{1}{2}} x \tag{4}$$

Here D_v, D_e denote diagonal matrices with the vertex and edge degrees respectively. Further, H denotes the incidence matrix, and θ are the learnable parameters of the convolution.

3. MODEL

Feng et al. [1] introduce their model equation as follows

$$\underbrace{D_v^{-\frac{1}{2}} H}_{3.} \overbrace{\mathcal{W} D_e^{-\frac{1}{2}} H^T D_v^{-\frac{1}{2}}}^{2.} \underbrace{X\Theta}_{1.} \tag{5}$$

1. Node feature transformation
2. Edge feature gathering
3. Node feature aggregation

Here X are the graph features or the input and Θ are learnable parameters. The node feature transformation is equivalent to a simple linear layer which is applied to all nodes in the same manner. In the edge feature gathering, all node features that are incident to an edge are summed up and rescaled. Lastly, in the node feature aggregation, all nodes gather their 'new' features from all incident edges. An illustration of this process is displayed in fig. 1.

The authors initialize $\mathcal{W}$ as the identity matrix so that all edges have the same weight. This also means that all parameters except Θ and X are constant. Hence we can further simplify the model eq. (5). We define:

$$L\mathcal{W}RX\Theta \tag{6}$$

$$\text{where } L := D_v^{-\frac{1}{2}} H \tag{7}$$

$$\text{and } R := D_e^{-\frac{1}{2}} H^T D_v^{-\frac{1}{2}} \tag{8}$$

D_v, D_e are diagonal matrices, and H is sparse, hence the defined matrices L and R are sparse matrices. A single convolutional layer of the model can be calculated as:

$$X^{(l+1)} = \sigma(D_v^{-\frac{1}{2}} H W D_e^1 H^T D_v^{-\frac{1}{2}} X^{(l)} \Theta^{(l)}) \tag{9}$$

$$= \sigma(L\mathcal{W}RX^{(l)}\Theta^{(l)}) \tag{10}$$

Here σ refers to a non-linear activation function. Diverging from the original paper we modify the model, instead of statically weighing all edges with the same identity matrix, we propose to jointly learn the edge weights with the filter parameters Θ. Wang et al. [12] also proposed to learn edge weights in a tensorized formulation of HGNNs.

Thus we define two slightly different models, the first is the model proposed by Feng et al. as $\mathcal{W}$ is constant in this model we can further simplify the model of eq. (10) as:

$$\sigma(L\mathcal{W}RX^{(l)}\Theta^{(l)}) = \sigma(PX\Theta^{(l)}) \tag{11}$$

$$\text{where } P := L\mathcal{W}R \tag{12}$$

Where P is still a sparse matrix as $\mathcal{W}$ is a diagonal matrix. P can be precomputed before any learning and does not change over the training process. This reformulation was done only to illustrate that P can be precomputed. We will not further use this formulation and will just refer to the $\mathcal{W}$ parameter as constant or learnable to differentiate between the models.

Further, we define a model with a learnable $\mathcal{W}$ parameter. For this model the equation (10) can not be further simplified.

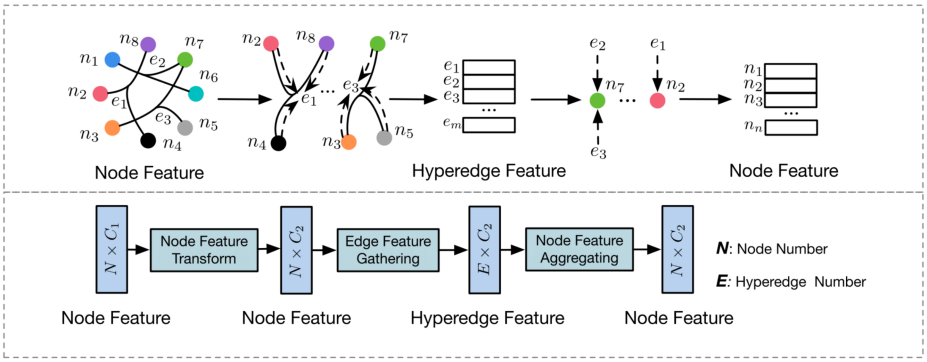


Fig. 1. The HGNN convolutional layer (Fig. 4 from Feng et al. [1])

Derivatives. Existing libraries like libtorch do not support distributed full-batch training. Hence, we have to implement the backward pass by computing the corresponding derivatives using matrix multiplication. In the following section, we will specify how we compute the derivatives of all parameters that are learnable. We use the following notation for the partial results computed in the forward pass:

$$G_1^{(l)} := LWR \tag{13}$$
$$G_2^{(l)} := X^{(l)}\Theta^{(l)} \tag{14}$$
$$G_3^{(l)} := G_1^{(l)}G_2^{(l)} \tag{15}$$
$$G_4^{(l)} := \sigma(G_3^{(l)}) \tag{16}$$
$$X^{(l+1)} = \sigma(G_4^{(l)}) \tag{17}$$
$$\hat{y} := X^{(L)} \tag{18}$$

Where L is the number of layers in the model. As we train on classification tasks, we use the cross-entropy loss $\mathcal{L}$ defined on logits:

$$\mathcal{L}(\hat{y}, y) := -\log\left(\frac{\exp(\hat{y}_y)}{\sum_{j=0}^{c-1}\exp(\hat{y}_j)}\right) \tag{19}$$
$$= -\hat{y}_y + \log\left(\sum_{j=0}^{c-1}\exp(\hat{y}_j)\right) \tag{20}$$

The derivatives that are calculated in the backward pass are then given by:

$$\frac{\partial\mathcal{L}}{\partial X_i^{(L)}} = -\delta_{iy} + \frac{\exp(\hat{y}_y)}{\sum_{j=0}^{c-1}\exp(\hat{y}_j)} \tag{21}$$
$$\frac{\partial\mathcal{L}}{\partial G_3^{(l)}} = \frac{\partial\mathcal{L}}{\partial X_i^{(l+1)}} \otimes \sigma'(G_3) \tag{22}$$
$$\frac{\partial\mathcal{L}}{\partial G_2^{(l)}} = \frac{\partial G_3^{(l)}}{\partial G_2^{(l)}}\frac{\partial\mathcal{L}}{\partial G_3^{(l)}} = \left(G_1^{(l)}\right)^T\frac{\partial\mathcal{L}}{\partial G_3^{(l)}} \tag{23}$$
$$\frac{\partial\mathcal{L}}{\partial G_1^{(l)}} = \frac{\partial\mathcal{L}}{\partial G_3^{(l)}}\frac{\partial G_3^{(l)}}{\partial G_1^{(l)}} = \frac{\partial\mathcal{L}}{\partial G_3^{(l)}}\left(G_2^{(l)}\right)^T \tag{24}$$
$$\frac{\partial\mathcal{L}}{\partial\Theta^{(l)}} = \frac{\partial\mathcal{L}}{\partial G_2^{(l)}}\frac{\partial G_2^{(l)}}{\partial\Theta^{(l)}} = X^T\frac{\partial\mathcal{L}}{\partial G_2^{(l)}} \tag{25}$$
$$\frac{\partial\mathcal{L}}{\partial X^{(l)}} = \frac{\partial\mathcal{L}}{\partial G_2^{(l)}}\frac{\partial G_2^{(l)}}{\partial X^{(l)}} = \frac{\partial\mathcal{L}}{\partial G_2^{(l)}}\Theta^T \tag{26}$$
$$\frac{\partial\mathcal{L}}{\partial w} = \frac{\partial\mathcal{L}}{\partial G_1^{(l)}}\frac{\partial G_1^{(l)}}{\partial w} = \operatorname{diag}\left(L^T\frac{\partial\mathcal{L}}{\partial G_1^{(l)}}R^T\right) \tag{27}$$

Where σ' is the derivative of the activation function that is used. For the RELU function, this means clipping all values below zero to zero. $A \otimes B$ refers to the element-wise multiplication of matrices A and B.

4. DISTRIBUTED ALGORITHMS

All matrices needed for the forward and the backward pass are distributed across the nodes. They are divided into blocks by a quadratic grid. Consequently, the training only works with a quadratic number of nodes. Every node holds one block of a matrix. Let $A \in \mathbb{R}^{n\times m}$ be some random matrix and p^2 the number of nodes. Then $A_{i,j}$ is the block of the nodes in grid-row i and grid-column j, and is of size $\frac{n}{p} \times \frac{m}{p}$. Should n or m not be divisible by p, then the nodes in the last gird-column/grid-row contain the remaining entries.
The bias is the only part that is fully available on all nodes.

As it is only a vector, its size is negligible compared to the matrices and will easily fit entirely on each node. By placing it on every node, no data has to be exchanged for the addition of the bias, which saves time. As the matrices are distributed across the nodes, their computations must be executed with a distributed matrix multiplication algorithm. Since we have to deal with dense and sparse matrices, we implemented a set of matrix multiplications to support any combination of dense and sparse matrices and exploit the characteristics of sparse matrices to reduce the amount of computation.

Fox Algorithm. The structure for all the matrix multiplication algorithms is based on the Fox algorithm by Fox et al. [13], a well-known matrix multiplication algorithm. We will quickly go over the procedure. Let again $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times r}$ be two random matrices. Let further $C = AB \in \mathbb{R}^{n \times r}$ be the product of them. All three matrices are split into blocks as explained before and distributed across p^2 nodes. The algorithm operates in p iterations. In every iteration, block matrices are broadcasted to other nodes, followed by local matrix multiplication, whose implementation depends on the type of the input matrices. In the first iteration, illustrated in fig. 2, the blocks of A in the first grid column are broadcasted along the grid row they lay in, i.e. the node that owns $A_{i,0}$ sends it to all nodes within grid row i. Similarly, the blocks of B in the first grid row are broadcasted along the grid column they lay in. The received blocks are then multiplied locally on every node. In the next iteration, the blocks of A in the second grid column and the ones of B in the second grid row are broadcasted. This process is repeated p-times. By then every block was broadcast once. The results of the local matrix multiplication summed up are equal to the blocks of C, meaning the node that is the owner of $A_{i,j}$ and $B_{i,j}$ will hold $C_{i,j}$ at the end.

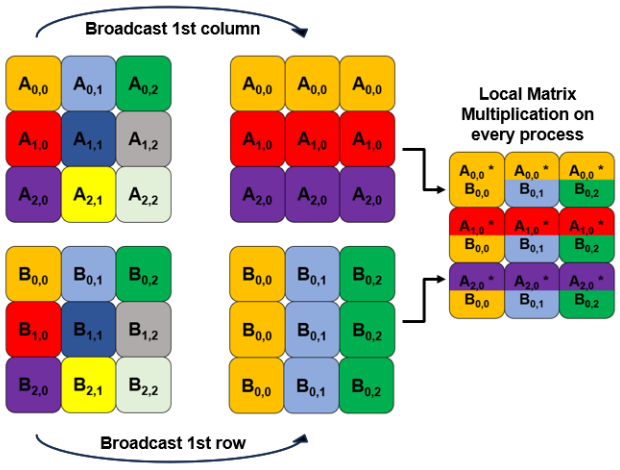


Fig. 2. Illustration of the first iteration of the Fox Algorithm on a 3x3 processor grid and 9 processes.

Table 2. Model Performance

Model	Hidden dims.	Test accuracy
`torch`	128	94.2 +/- 0.42
`torch`	256	96.0 +/- 0.26
`torch`	512	96.7 +/- 0.04
`torchW`	128	96.7 +/- 0.09
`torchW`	256	96.8 +/- 0.17
`dist`	128	95.9 +/- 0.10
`dist`	256	96.6 +/- 0.09
`dist`	512	96.5 +/- 0.06
`dist`	1024	96.6 +/- 0.14

5. EXPERIMENTAL RESULTS

Experimental setup. All our performance testing was done on the Euler cluster. We used nodes with the specifications shown in table 3. Our programs are compiled using the versions listed in table 4.

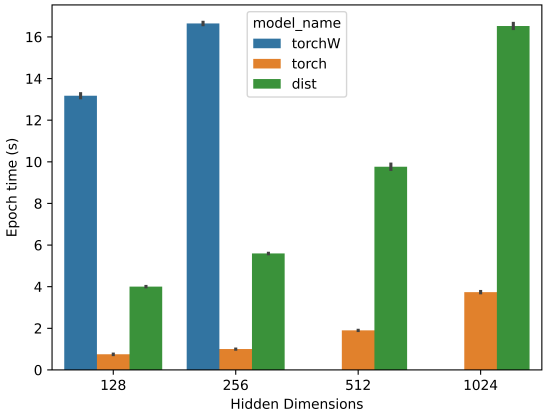
Dataset. We used the dataset that Feng et al. [1] used to keep results comparable. They used the ModelNet40[14] dataset. The node features are extracted using the MVCNN[15] and GVCNN [16] models which give a total of around 6000 features. Features are extracted using the ten nearest neighbors of each point jointly. Hyperedges then connected all of these points.

Results. Before evaluating the performance of our implementation, we want to make sure that we reach accuracy scores comparable to Feng et al. To this end, we first implemented a non-distributed memory version using libtorch. The results are shown in table 2. Our torch model "torchW 128" (with the same parameters) achieves the same accuracy as the model of Feng et al.

We can see significant differences in achieved accuracy when comparing the models that have learnable edge weights (torchW models). The "torchW 128" achieves higher accuracy compared to the "torch 256" and equal to the "torch 512" model. We also did the same testing with our distributed implementation in table 2. Here we do not implement the learnable edge weights but make up for the decrease in modeling power by increasing the hidden dimensions to at least 256. All our distributed models achieve accuracies above 96% and in about the same number of epochs as the torch models.

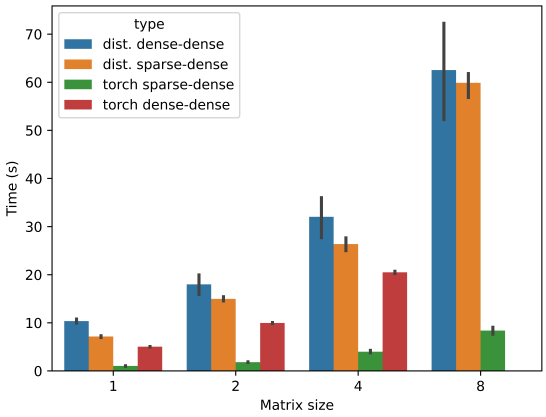
All of our models have similar test accuracy to epoch curves. For our performance testing, we are looking at the average epoch times across all epochs and multiple test runs, shown in fig. 3. It is apparent, that the introduction of the learnable W parameters has a massive impact on performance. With runtimes being over 10x slower than the model without W. The runtime for our distributed implementation is significantly slower than our torch implementation.

Fig. 3. Average time it takes to compute one epoch with 4 nodes or threads for our distributed and torch models respectively



This is due to the communication overhead incurred by the distributed nature of the matrix multiplication. We are still faster in achieving sufficient accuracy when compared to the "torchW" models. Further, our model scales roughly linearly with respect to the hidden dimensions of the model. The same holds for the torch implementations.

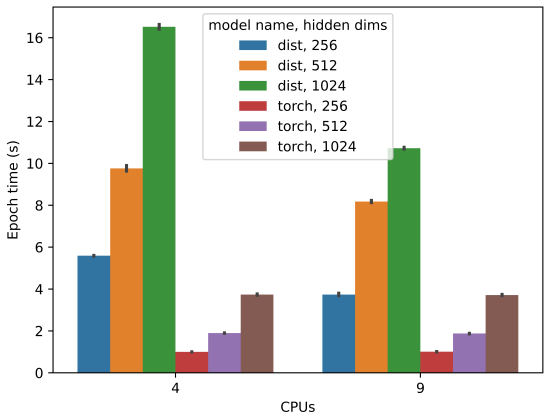
Fig. 4. Matrix multiplication benchmarks for different implementations and matrix sizes using 4 nodes or threads for our distributed and torch models respectively. The sparse matrix is the adjacency matrix of the dataset and the dense matrix is the feature matrix with 300M and 75M entries respectively. The x-axis denotes the scale of the matrices where 2 means we scale the number of samples by 2 while keeping the nnz ratio of the sparse matrix at 0.08%



All our models are based on the calculation of dense-dense and sparse-dense matrices. Thus we also evaluated the underlying computation algorithms, shown in fig. 4. Both our distributed dense-dense and sparse-dense multiplication scales linearly with increased matrix size. The two matrix multiplication algorithms used by torch also scale linearly with the matrix size.

Figure 5 further shows that the larger our model and with it the matrices in our computation get, the better our model scales when increasing the number of tasks (in a 1-to-1 ratio CPUs).

Fig. 5. Comparing average times for one epoch across different numbers of nodes or threads for our distributed and torch models respectively



6. CONCLUSIONS

We have implemented the first distributed memory framework for HGNNs. This enables full batch training on large datasets so that no sampling techniques are necessary. Yielding better accuracy than common sampling methods. Our performance being lower than torch is natural, as the dataset used by Feng et al. [1] fits in the memory of a single machine. However, our version is specifically designed to accommodate larger datasets that would not fit onto a single machine. Our implementation scales roughly linear in the number of learnable parameters. This is to be expected, as the most time-consuming operation is the communication of the matrix subblock, possible future work might focus on this issue. Further, we introduce a slightly modified version of HGNNs that jointly learns the edge weights with the convolution filter. This new model does improve test accuracy at the cost of significantly slower epoch times. We showed that using learnable edge weights is roughly equivalent to increasing the hidden dimensions by a factor of four in terms of final test accuracy.

7. FUTURE WORK

Our current model uses the Fox algorithm and derivatives for the dense-dense and sparse-dense matrix multiplications. However, there has been extensive research in the field of distributed matrix multiplication in general and dense-dense and sparse times tall and skinny dense matrix multiplications especially, there are more efficient algorithms that can be used. As the main workload comes from these multiplications it would likely benefit the implementation to use other specialized algorithms such as the ones implemented by Selvitopi et al. [17]. Another possible method to accelerate the matrix multiplication that should be explored is leveraging accelerators for the local multiplication. This however might quickly run into memory bandwidth issues.

Further, implementing the learnable W parameter would be beneficial. This is however not trivial to do efficiently, as implementing the derivatives directly would lead to the creation of dense matrices of size $|E| \times |E|$. That is quite inefficient, as can be seen in the slowdown of our torch implementation.

8. REFERENCES

[1] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao, "Hypergraph neural networks," 2019.

[2] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu, "Dgcl: An efficient communication library for distributed gnn training," in *Proceedings of the Sixteenth European Conference on Computer Systems*, New York, NY, USA, 2021, EuroSys '21, p. 130–144, Association for Computing Machinery.

[3] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha, "Distgnn: Scalable distributed training for large-scale graph neural networks," 2021.

[4] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec, "Ogb-lsc: A large-scale challenge for machine learning on graphs," *arXiv preprint arXiv:2103.09430*, 2021.

[5] Mikael Henaff, Joan Bruna, and Yann LeCun, "Deep convolutional networks on graph-structured data," 2015.

[6] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," 2017.

[7] David K Hammond, Pierre Vandergheynst, and Rémi Gribonval, "Wavelets on graphs via spectral graph theory," 2009.

[8] Thomas N. Kipf and Max Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[9] Alessio Micheli, "Neural network for graphs: A contextual constructive approach," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 498–511, 2009.

[10] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," *CoRR*, vol. abs/1509.09292, 2015.

[11] James Atwood and Don Towsley, "Diffusion-convolutional neural networks," 2016.

[12] Maolin Wang, Yaoming Zhen, Yu Pan, Zenglin Xu, Ruocheng Guo, and Xiangyu Zhao, "Tensorized hypergraph neural networks," 2023.

[13] G.C Fox, S.W Otto, and A.J.G Hey, "Matrix algorithms on a hypercube i: Matrix multiplication," *Parallel Computing*, vol. 4, no. 1, pp. 17–31, 1987.

[14] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao, "3d shapenets: A deep representation for volumetric shapes," 2015.

[15] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik G. Learned-Miller, "Multi-view convolutional neural networks for 3d shape recognition," in *Proc. ICCV*, 2015.

[16] Yifan Feng, Zizhao Zhang, Xibin Zhao, Rongrong Ji, and Yue Gao, "Gvcnn: Group-view convolutional neural networks for 3d shape recognition," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 264–272.

[17] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç, "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, New York, NY, USA, 2021, ICS '21, p. 431–442, Association for Computing Machinery.

[18] Stefan Kramer, Yannick Wattenberg, Simon Wachter, and Han Xi, "Ywattenberg/distributed-hgnns," .

A. CONFIGURATIONS

Table 3. Compute Node Specifications for Euler VIII

Component	Specifications
Processor	
Model	AMD EPYC 7742
Amount	2 per node
Cores	64 per processor
Clock Speed	2.25 GHz (nominal), 3.4 GHz (peak)
Memory	
Type	DDR4
Capacity	512 GB
Clock Speed	3200 MHz
Network	
Type	100 Gb/s Ethernet

Table 4. Compiler and library versions compilation

Compiler/Library	Version
`gcc`	11.4.0
`openmpi`	4.1.4
`openblas`	0.3.20
`cmake`	3.26.3
`libtorch`	2.1.2 (cpu)
`combblas`	2.0[1]

[1]For implementing our algorithms we had to make changes to the base library. These are included in our Repository[18] as a submodule