Convolutional graph neural network training scalability for molecular docking

Kevin Crampon, Alexis Giorkallos, Stephanie Baud, Luiz Angelo Steffenel

Convolutional graph neural network training scalability for molecular docking

Kevin Crampon*[†][‡]   Alexis Giorkallos*   Stephanie Baud[‡]   Luiz Angelo Steffenel[†]

Center for Excellence in Advanced Computing
Atos SE
Echirolles, France

[‡] *CNRS, MEDyC*
Université de Reims Champagne-Ardenne
Reims, France

[†] *LICIIS*
Université de Reims Champagne-Ardenne
Reims, France

Abstract—Deep learning use is growing in many numerical simulation fields, and drug discovery does not escape this trend. Indeed, before proceeding with *in vitro* and then *in vivo* experiments, drug discovery now relies on *in silico* techniques such as molecular docking to narrow the number of experiments and identify the best candidates. This method explores the receptor surface and the ligand's conformational space, providing numerous ligand-receptor poses. All these poses are then scored and ranked by a scoring function allowing to predict the best poses among all, then compare different ligands regarding a given receptor or different targets regarding a given ligand. Since the 2010s, numerous deep learning methods have been used to tackle this problem. Nowadays, there are two significant trends in deep learning for molecular docking: (i) the augmentation of available structural data and (ii) the use of a new kind of neural network: the graph convolutional neural networks (GCNs). In this paper, we propose the study of training scalability of a GCN—a molecular complex scoring function—on an increasing number of GPUs and with a variety of batch sizes. After a hyperparameter analysis, we achieve an 80% reduction in the training time, but this improvement sometimes involves a performance metrics degradation that the final users must ponder.

Index Terms—Scalability, Graph Convolutional Neural Network, GPUs, PyTorch

I. Introduction

Discovering new treatments requires substantial financial means and equipment, so to reduce them, numerous numerical methods using sophisticated models and optimization sets were proposed. One of them is the molecular docking simulation, whose common use case is drug discovery. It involves looking for active principles (hereafter called ligands) that interact with a macromolecule of interest (the receptor), such as a protein. Finding interacting molecules may have several objectives including, for example, discovering a ligand inhibiting the receptor effect on human health. A prerequisite for that interaction is performing docking between the ligand and the receptor, hence molecular docking is performed on the 3D structure of both molecules, to predict if the ligand will bind to the receptor. In the remainder of this paper, we present methods developed for proteins as receptors.

Molecular docking simulations are less expensive than *in vitro* and *in vivo* experiments and are essentially based on a receptor's surface screening. In comparison to classical docking methods, deep learning in this space provides more accurate guidance in the search for the zone of binding [2].

For some years, a new kind of neural network has grown in popularity: the graph convolutional network (GCN) [11], which relies on graph input data. A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a set of nodes $\mathcal{V}$ connected by edges $\mathcal{E}$. Each node and edge can be augmented by adding features to respectively $\vec{v_i}$ and $\overrightarrow{e_{i \to j}}$.

GCNs are trendy because they use data in graph format. This format is the one that best represents a molecular structure. In particular, if we compare it to 1D representation (SMILES or sequence) or to discretization on a 3D grid where the information on the links between the atoms is lost.

Graph convolution may be either spectral [12] or spatial. Spectral convolutions use the eigendecomposition of the graph's Laplacian matrix, as shown in Eq 3.

$$L = D - A \qquad (1)$$

$$L = V \Lambda V^T \qquad (2)$$

$$H^{(l+1)} = V(V^T H^{(l)} \odot V^T W^{(l)}) \qquad (3)$$

Where $.^T$ means transposition, A is the adjacency matrix, D is the degree matrix, L is the Laplacian matrix (Eq. 1) whose eigendecomposition in Equation 2 provides V the eigenvectors matrix, and Λ the eigenvalues diagonal matrix. Finally, the convolution is processed with the node features matrix H at layer l and the associated weights matrix W as inputs to produce the output $H^{(l+1)}$, $\odot$ is the element-wise multiplication operator.

Nowadays, the most prominent convolution type is spatial convolution [4], it can be seen as a generalization of convolution over a regular cartesian grid, as in spatial convolution, each node aggregates information (balanced by learnable weights) from its direct neighbors. If the graph's edges have features, they also influence the information, as presented in Equation 4 and Figure 1.

$$h_i^{(l+1)} = U_l(h_i^{(l)}, \sum_{j \in \mathcal{N}(i)} M_l(h_i^{(l)}, h_j^{(l)}, e_{j \to i})) \qquad (4)$$
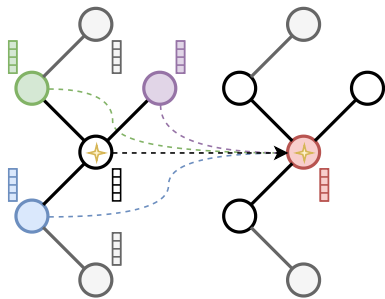
Fig. 1. In spatial convolution, the current node here identified by a star aggregates information coming from its direct neighbors here represented in green, purple, and blue. The current node feature vector is then updated as shown in the graph on the right.
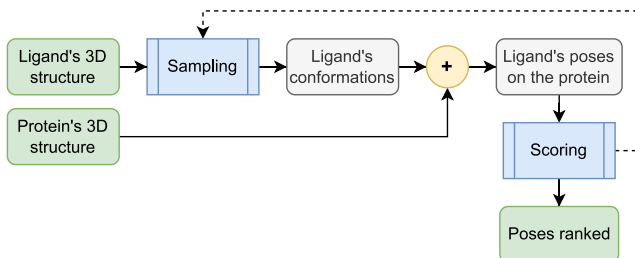

Fig. 2. A simplified scheme of the semi-rigid molecular docking process. First, the ligand conformational space is sampled, allowing the exploration of conformation and position, all conformations are then associated with the protein to generate docked complexes. Finally, a scoring function assesses complexes to rank them. Some sampling methods use the scoring output to guide them and thus restrict conformational space or protein surface area.

Where $h_i^{(l)}$ is the hidden features vector of node i at layer l, $h_i^{(0)} = \vec{v_i}$, U_l and M_l are functions with learnable parameters and $\mathcal{N}(i)$ is the neighborhood of node i.

The remainder of this article focuses on spatial convolutions only, we also provide our code and results on GitHub to allow reproducibility github.com/KevinCrp/GCN_Docking_Scalability.

II. BIOLOGICAL PROBLEMATIC

The molecular docking is a 2-steps process: a sampling phase and then a scoring phase, as presented in Figure 2. Considering both molecules, the ligand and the receptor, as flexible implies a huge number of degrees of freedom, and hence a substantial computing time. To overcome that, numerous docking processes make the simplifying hypothesis that the protein is rigid on the first step, and make a flexibility assumption on the second step, to generate several protein conformations and use molecular docking for the same ligand with those conformations, but this is outside the scope of our study.

Thus, once the protein structure is fixed, the first step is to explore the ligand's conformation space. This space contains all conformations generated by the ligand's internal bond rotations. This step allows further exploration of the protein's surface by translating and rotating the ligand in space.
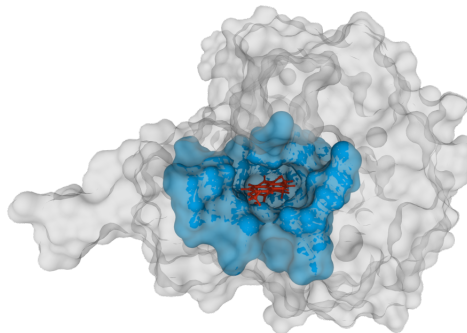

Fig. 3. Trypanosoma cruzi dihydroorotate dehydrogenase (PDB: 3W3O) with the ROE ligand in red and the associated binding site in blue.

All ligand conformations are then associated with the protein's structure to create a set of docked poses, all of which are scored with the scoring function which aims to faithfully reflect the experimental affinity.

One common way to reduce the ligand's conformational space and the explored protein's surface is to use the scoring output to guide the sampling method in an iterative process.

Numerous classical AI-free scoring functions were proposed and can be classified into three categories: a) the physics-based methods use a weighted sum of energy terms, b) the empirical methods use a weighted sum of simpler terms such as physico-chemical descriptors, and c) the knowledge-based methods use known complexes to determine the current complex affinity. Finally, a consensus method allows to combining of the scores from several of these approaches.

Since 2010, numerous machine learning or deep learning methods were introduced to tackle this problem. Some reduce the protein's explored surface by predicting the binding site, but the majority of methods just rely on improved scoring functions, as presented in our previous paper [2]. Hereafter, the presented methods belong to this category.

III. DATA

We structure our study around the PDBBind database [7]. It contains 19,443 ligand-protein complexes and their associated experimental binding affinity, and for each complex, some files are provided including a MOL2 file containing the ligand structure and a PDB file for the protein. A second PDB file defining the binding site (all protein's residues closer than 10Å of a ligand atom, as presented in Figure 3) is also provided, which we use to reduce the graph sizes and thus the computing time. Both file types provide the molecule's atoms list with their respective position and type.

We represent each complex by a graph whose nodes are the complex's atoms and edges connect each pair of nodes closer than 4.0Å. We add a set of features on each node, presented in Table I. We also add edge attributes on the molecular graph by adding a vector of ones on each edge. The aim was to increase the data size without impacting the score prediction.

Feature	Size	Description
Atom type	8	One-hot encoded (B, C, N, O, F, P, S, Others)
Hybridization	1	0: Other, 1: sp, 2: sp2, 3: sp3, 4: sq. planar, 5: trig. bipy, 6: octahedral
Heavy Degree	1	0, 1, 2, 3, 4, 5, 6+
Hetero Degree	1	0, 1, 2, 3, 4, 5, 6+
Partial Charge	1	Float
Is Hydrophobic	1	Boolean
Is in a Aromatic cycle	1	Boolean
Is HBond Acceptor	1	Boolean
Is HBond Donor	1	Boolean
Is in a Ring	1	Boolean
Belong to Protein or Ligand	2	One-hot encoded (Ligand or Protein)

Fig. 5. Distribution of the number of graphs as a function of the number of edges characterizing the training set.

Parameter	Value
Atomic GAT Layers	[19, 32, 32, 32, 32, 32]
Molecular timestep	4
Learning rate	1e-3
Weight decay	1e-4
Loss function	MSE

Implementation of our network is done with PyTorch (v1.12.0) and PyTorch-Geometric [3] (v2.2.0) using the Pyglib backend. PyTorch-Geometric is a framework that allows us to deal with graph data structure and propose a comprehensive set of GCN layers. In addition, PyTorch-Lightning (v1.7.7) allows us to simplify PyTorch code by reducing the boilerplate and providing ease of parallelization on a multi-GPUs machine such as the DGX. PyTorch-Lightning allows us to easily scale the training by only setting the available number of GPUs, it should always be considered a good way to train a PyTorch model on a High Performance Computing (HPC) environment.
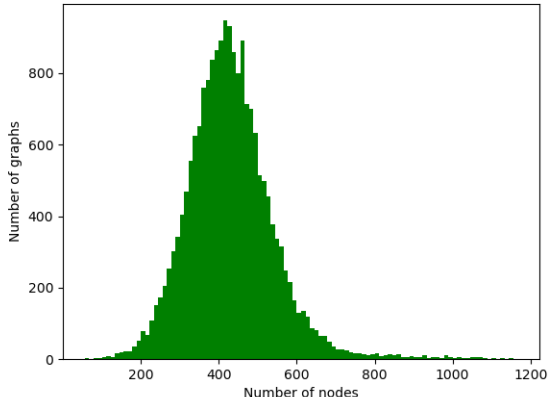

Fig. 4. Distribution of the number of graphs as a function of the number of nodes characterizing the training set.

Moreover, we do that in order to compare models using edge attributes and those not using them.

We train our model on the whole PDBBind database (19,443 protein-ligand complexes) resulting in as many graphs. The 19,443 graphs correspond to 8,375,349 nodes each having 19 physicochemical features and connected by 88,734,678 edges (*cf* Figures 4 and 5).

IV. EXPERIMENTS

A. Network's architecture

We have used the AttentiveFP [10] network for our experiments. That method uses a graph attention network (GAT) [9] to produce a molecular fingerprint. Initially used to embed a molecular graph into a fingerprint, we use it to produce a protein-ligand fingerprint. AttentiveFP may be presented in two parts, the first working on the atomic scale, which is composed of 6 layers, and the second part allows the creation of the fingerprint. That second part is composed of 4 layers, and the produced fingerprint is only a scalar, which is used as an affinity score. That model is not the best, but its performances are correct and allow us to analyze the evolution of time and metrics through the prism of scalability.
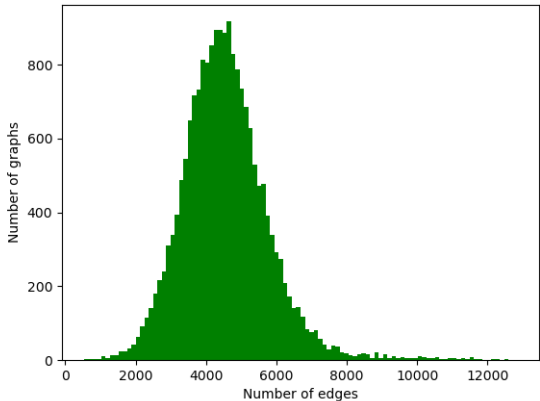
B. Metrics

The study's objective is to measure the GCN training scalability on several GPUs. With PyTorch-Lightning, we can easily distribute training over several GPUs and aggregate results with a few changes to the code. We have trained our network on 1, 2, 4, and 8 GPUs, measured the computing time and compared the impact of the parallelization on a set of use case adapted metrics: the Pearson correlation coefficient (R_p, Eq 5), and the coefficient of determination (R^2, Eq 6). The former was indicated by Su *et al.* [8] and allows us to check if the predicted affinity is linearly correlated with the experimentally determined affinity. The latter measures the prediction quality. Both have an optimal value of 1.0. Because of sensitivity issues, we run each case 10 times and use the average of the metrics to do our comparisons.

$$R_p = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 \sum_{i=1}^{n}(y_i - \overline{y})^2}} \quad (5)$$
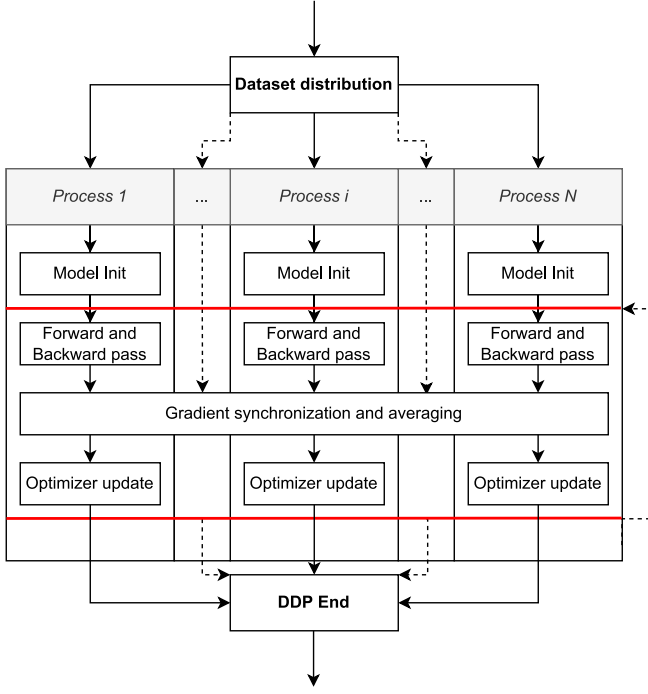
Fig. 6. The *Distributed Data Parallel* process distributes the dataset across all GPUs which all work on an independent model, the common step of gradient synchronization and averaging allows sharing of the learning. A detailed description is provided in paragraph IV-C

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - x_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2} \qquad (6)$$

Where x_i is the predicted score for complex i, $i \in [\![1;n]\!]$, y_i the experimental affinity, $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\overline{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$.

C. Distribution method

We use the PyTorch-Lightning default distribution method: *Distributed Data Parallel* (DDP) [6] (Figure 6). Each GPU has an independent process and works only on a subset of the training dataset. That subset is randomly generated from the whole dataset and remains the same during the whole training process. Each process initializes a model from the same seed, making sure all weights are the same. Each process does a forward prediction pass and a backward gradient-computation pass. Finally, gradients from all processes are collected and averaged to update optimizers and models over each accelerator individually, making sure weights are synchronized and identical at each step. With that method each sub-dataset is split into several device batches, thus the main steps (between the red lines in Figure 6) are computed as many times as necessary to process the whole sub-datasets.

D. Hardware

All experiments were performed on the Nvidia DGX-1 version 5.1.0 hosted by the Regional Super Computer Center ROMEO of the University of Reims Champagne-Ardenne. That blade has 8 Nvidia Tesla V100 GPUs with each 16GB of VRAM and two sockets of 20 cores each, with 2 threads per core, and 512GB of RAM.
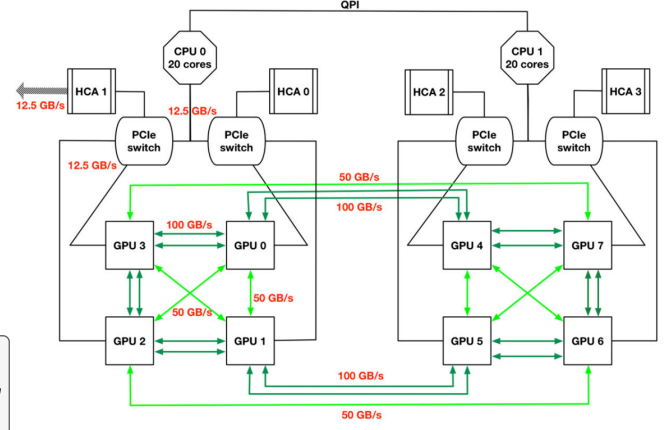

Fig. 7. The DGX-1 Tesla V100 architecture [1], composed of two NUMA (Non-Uniform Memory Access) nodes of 20 cores and 4 GPUs each. GPUs are linked together by the NVLink interconnect (green rows) which ensures an important bandwidth

TABLE III
GLOBAL AND DEVICE BATCH SIZES TESTED

| # GPUs	1	2	4	8	Figure
Constant global batch size	*Device batch size*				
64	64	32	16	8	8
128	128	64	32	16	9
256	256	128	64	32	10
Constant device batch size	*Global batch size*				
64	64	128	256	512	8
128	128	256	512	1024	9
256	256	512	1024	2048	10

Figure 7 shows that the device placement is essential. Indeed, the DDP algorithm requires sharing information during the common gradient synchronization step. Choosing close GPUs interconnect-wise is key. To summarize, we train our method on the following GPU sets: $\{0\}$, $\{0,1\}$, $\{0,1,2,3\}$, and $\{0,1,2,3,4,5,6,7\}$. In addition, we leverage the NVIDIA Collective Communication Library (NCCL) as the communication backend.

V. RESULTS

This study's objective is to compare two batch size strategies. The first one uses a fixed batch size on each GPU—the device batch size or DBS—regardless of the number of GPUs. It involves a growing global batch size (GBS) proportional to the number of GPUs ($GBS = \#GPU \times DBS$). The second strategy is to use a constant GBS and thus a decreasing DBS in proportion to the number of GPUs. We have tested both strategies with batch sizes of 64, 128, and 256 involving the global and device batch sizes presented in Table III. We train our model on 50 epochs to insure convergence and compare times and metrics for each batch size used with the two presented strategies: constant DBS or constant GBS. Our charts (Figures 8, 9, and 10) present the time improvement and the metric degradation, for the different strategies and batch sizes.
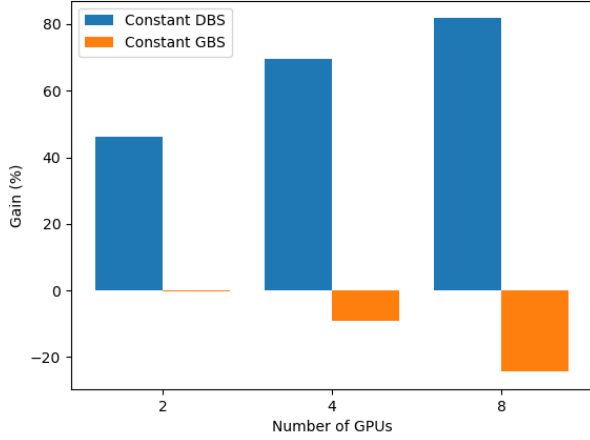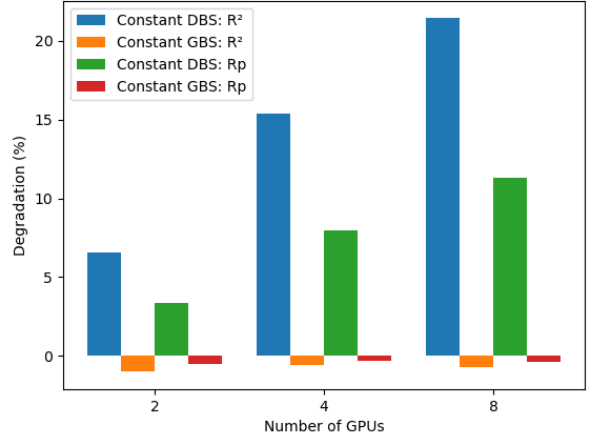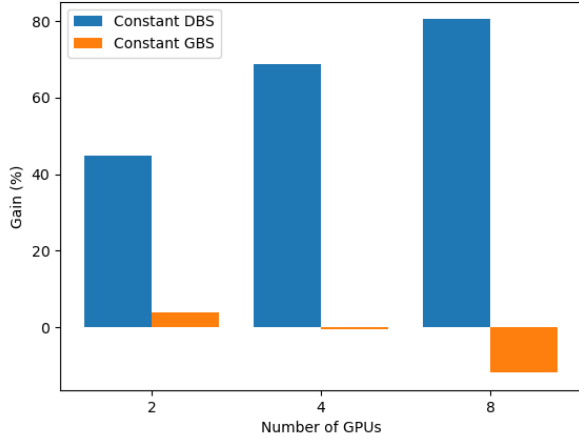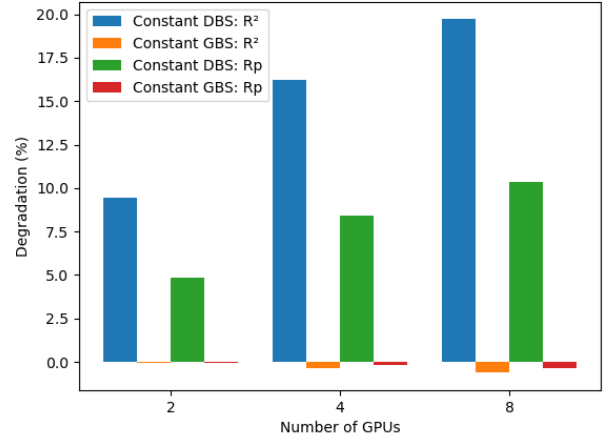
Fig. 8. Comparison of distribution method with a batch size of 64: (a) Time improvement compared to using a single GPU, (b) Degradation of the metrics compared to those obtained with 1 GPU. DBS=Device Batch Size and GBS=Global Batch Size

Fig. 9. Comparison of distribution method with a batch size of 128: (a) Time improvement compared to using a single GPU, (b) Degradation of the metrics compared to those obtained with 1 GPU. DBS=Device Batch Size and GBS=Global Batch Size

A. Strategies comparison

Whatever the batch size, the constant DBS strategy substantially reduces training time. That gain increases with the number of GPUs, reaching an 80% of time reduction for the best-case scenario. The constant GBS strategy usually increases the training time. Concerning the metrics degradation, both R_p and R^2 are degraded as the number of accelerators increases if the constant DBS strategy is used. For the constant GBS strategy, the two metrics are quite stable.

B. Batch size impact

With the constant GBS strategy, the induced DBSs are all smaller as the number of GPUs gets bigger. Thus, the use of several GPUs with a small batch brings a huge amount of communication (data copy from CPU to GPU, gradients exchange, prediction copy from GPU to CPU). All these communications need time to be treated; thus, this is the bottleneck for the training time. That is why using several GPUs is interesting only if the DBS is large enough. It is for this reason that the constant GBS strategy used with a small base batch size (64 and 128), has training times longer on several GPUs. Whereas with the constant DBS strategy time reduction is very important.

In Figures 11, 12, and 13, which present training times and metrics value after training, we can see that metrics degradation is more important with a bigger batch size for the
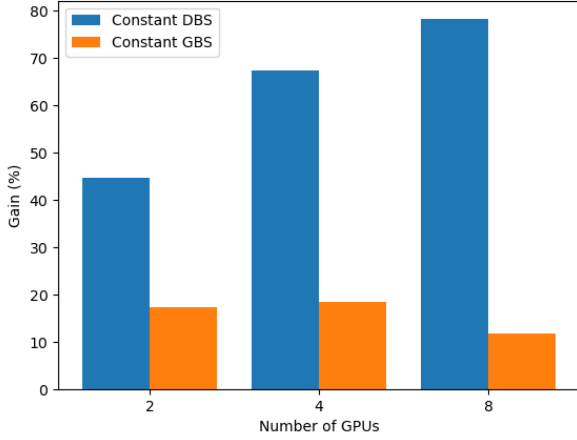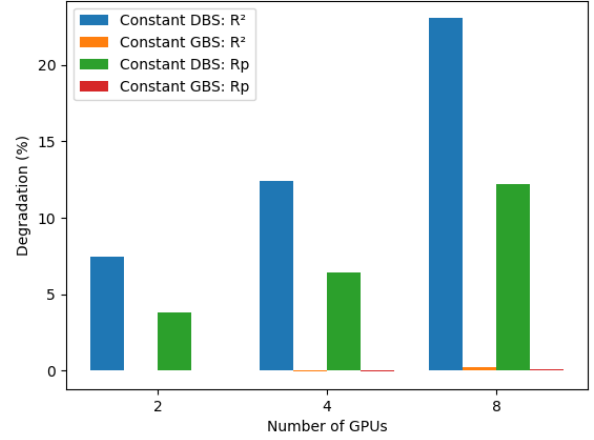
Fig. 10. Comparison of distribution method with a batch size of 256: (a) Time improvement compared to using a single GPU, (b) Degradation of the metrics compared to those obtained with 1 GPU. DBS=Device Batch Size and GBS=Global Batch Size
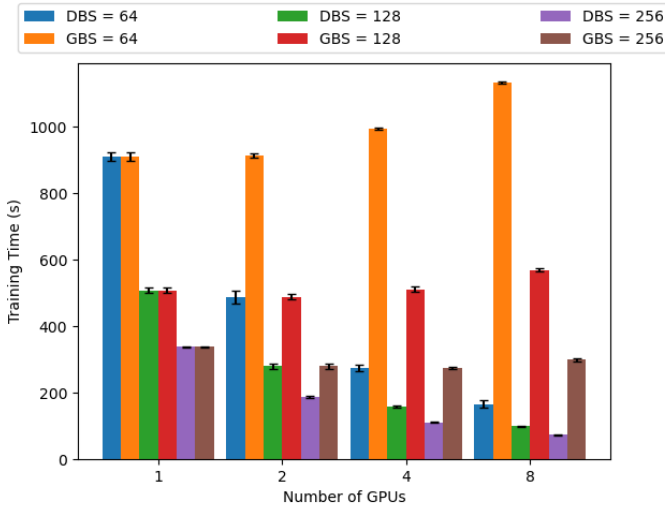

Fig. 11. Average training time for all strategies with different batch sizes. DBS=Device Batch Size and GBS=Global Batch Size
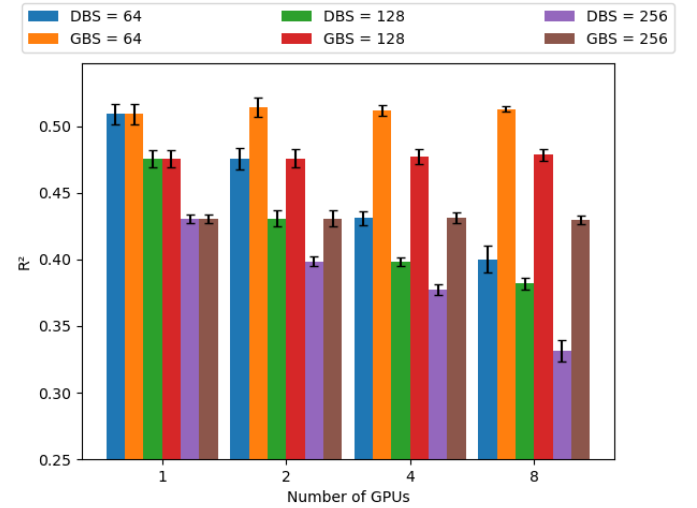

Fig. 12. Average R^2 for all strategies with different batch sizes. DBS=Device Batch Size and GBS=Global Batch Size

constant DBS strategy. Concerning the constant GBS strategy, the metrics seem to not be impacted by the batch size increase.

C. Analysis

Training time

For some experiments, the use of several GPUs may cause a loss in time performance. That gap is explained by a communication (CPU-GPU and GPU-GPU) bottleneck. The number of communications increases with the number of GPUs and with the decrease of the DBS. Thus, using 8 GPUs with a small DBS (example in Figure 8a with 8 GPUs and a constant GBS of 64 inducing a DBS of 8) does not allow compensating for communication time by the computing acceleration. All GPUs are not interconnected (Figure 7), and that bottleneck is

increased. That is why multi-GPU training must be launched on closer GPUs if possible. Another cause of time loss may be the use of unbalanced sub-datasets (some having bigger graphs) involving an unbalanced computing time between the GPUs. Thus, the other processes must wait until the end of the longer one.

Memory transfer

Training time may be roughly decomposed into computing time and data transfer time. In that part, we will analyze the impact of batch size strategy on the transfer time. That time is composed of a 3-step; memory copy of the graphs from the CPU to the GPU accelerators (H_to_D) through PCIe transfer buses. The device-to-device copies (D_to_D) cover the data
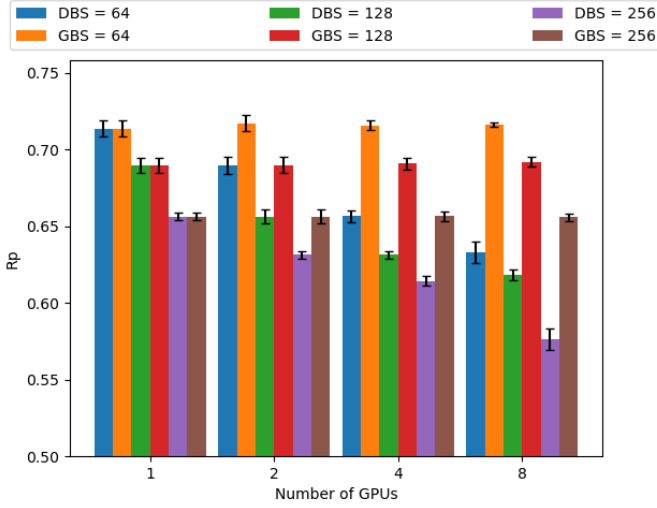
Fig. 13. Average R_p for all strategies with different batch sizes. DBS=Device Batch Size and GBS=Global Batch Size
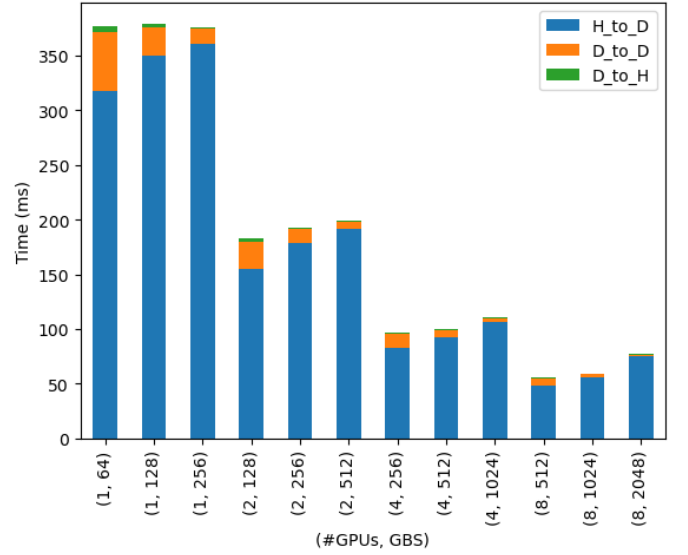

Fig. 14. Average data transfer time for 1 epoch, according to the number of GPUs and the batch size used with a constant Device Batch Size. H_to_D=Host to Device, D_to_D=Device to Device, and D_to_H=Device to Host.

Fig. 15. Average data transfer time for 1 epoch, according to the number of GPUs and the batch size used with a constant Global Batch Size. H_to_D=Host to Device, D_to_D=Device to Device, and D_to_H=Device to Host.

transfers for inter-GPUs synchronization (model gradients and weights update) which NVlink insures. And the GPU to CPU offloads (D_to_H) the results. We present the average data transfer time per epoch at a constant DBS (Figure 14) and a constant GBS (Figure 15). Data transfer metrics were obtained thanks to NVProf.

Regarding the constant DBS strategy, we can see that at iso-GPU, an increase in the DBS does not result in better overall training time. Indeed, the small reduction which may be observed on D_to_D and D_to_H parts is compensated by the augmentation of data transfer from the host to the devices. Whatever the size of the batch and for both strategies, the use of a larger number of GPUs reduces the time allocated to data transfer, both in absolute value and relative to the training time.

Figures 8a, 9a, 10a, 14, and 15 would highlight the correlation between data transfer time and overall training time. At constant DBS, the training time decreases sharply with the increase in the number of GPUs, as does the transfer time. This can be explained by the fact that the larger the DBS, the more data is loaded on the GPUs, and thus the GPUs can be better exploited, so the reduction in time occurs at both levels.

At constant GBS, even though transfer time decreases with the number of GPUs, the total training time does not follow the same trend. Because computation time is greater, the DBS decreases with the number of GPUs, which run on only a small fraction of the data, which is not ideal.

Metrics degradation

Figures 8b, 9b, and 10b show a metric degradation both for R^2 and R_p when scaling over several GPUs and using the constant DBS strategy. At constant DBS, that degradation may be caused by the induced GBS increase: at each step (*ie* a global batch processed), the network is updated from the error averaged over the whole mini-batch set. Each data point will orient learning in its direction. Because of the high degree of freedom of the system, this results in inefficient guidance (many pushes will, on average cancel out) and thus learning. For either strategy, the metrics degradation may be explained by the different means done during the training, reducing the error impact and thus slowing down the network convergence. This also explains why for a constant GBS, the metrics remain stable whatever the number of GPUs used.
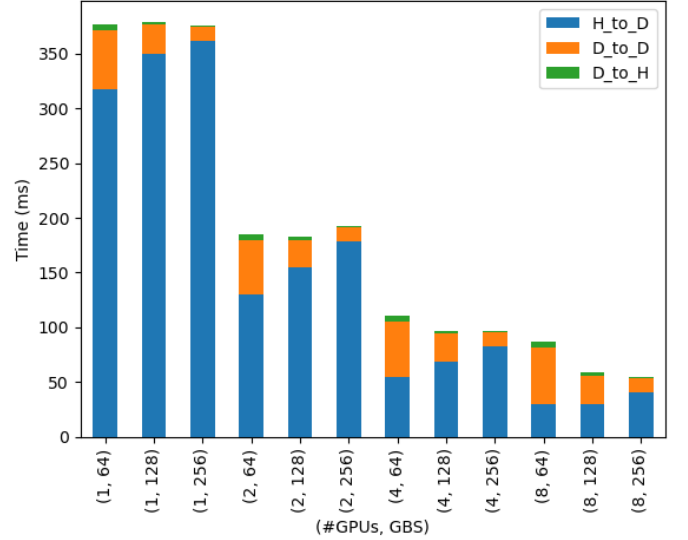
VI. Model comparison

We have tested another model in the same experiments to know the impact of the model architecture on our results. However, contrary to the previous model, this one does not use any edge attribute. All our experiments' results are available on our GitHub. Here, we have used a classic graph convolutional network (GCN) [5] which is composed of a series of 6 GCN layers that apply a classic graph convolution as presented in Figure 1, the first layer increases the node's feature dimension from 19 to 32, the four hidden layers realize a convolution but do not change the node's features dimension, and the output layer increases the dimension from 32 to 64. Then a multilayer perceptron (MLP) is used to reduce all the node features to only one value. Finally, a global average pooling means all the graph's node output values to a score.

With that model, the evolution of the training time is the same as for the previous model, and thus the best strategy to reduce the training is also to use the constant DBS strategy with a big batch size. Concerning the metrics degradation, the constant DBS strategy is also the strategy bringing the more important loss of performances, however, contrary to the previous model, here the constant GBS strategy brings a metrics degradation too.

Thus, that second set of experiments shows that the time reduction is not impacted by the model choice. However, this is not the case for the metrics, especially for the constant GBS strategy, which degrades them here. Nevertheless, this degradation is still much less important than with the other strategy.

VII. Conclusion

The objective of this study was to find the best strategy for training distribution. Regarding our result, the strategy choice depends on whether the priority is given to reducing training time or preserving model performance. If the main goal is to reduce the training time, thus it will be used the biggest number of GPUs, and the biggest batch size with the constant DBS strategy. That way may be used to identify an architecture and then train it with a better metrics preservation strategy. However, it is interesting to directly use a method that conserves the metrics performance (*ie* the constant GBS strategy), in that case, a large global batch size must be used to bring a time reduction with the use of several GPUs. Nevertheless, the choice of a large global batch size induces a loss regarding the metrics. That is why, either we choose a great number of GPUs with the constant DBS strategy, and thus the metrics collapse, or we choose the constant GBS strategy which allows us to lightly reduce the training time while degrading the metrics less.

However, that choice is highly conditioned by the network and its hyperparameters. Indeed, a bigger network requiring more computing time, for example, may have a more interesting trade-off with more GPUs or with bigger batch size. Another possibility is to study the use of a lower learning rate; indeed, a lower learning rate involves a more stable convergence but is slower. Moreover, the use of several GPUs involves a slower convergence, a possibility to bypass that issue is to train the model on more epochs. To conclude, we must find the best trade-off between time reduction in training and metrics degradation. Finally, the use of multi-GPU distribution training is a good way to speed up the search for better hyperparameters, thus we can find quickly the best model architecture, and then we can train our model with the best hyperparameter sets on only 1 GPU to get the best performances. To go further, we think that analyzing the impact of the GPU generation on the time and the metric degradation will be interesting. Moreover, it will be interesting to study the influence of the hardware as well as the GPU, the CPU, and the interconnect used. Although we tested both strategies on two quite different models, regarding the huge amount of available GCN models for the studied use case, it should be considered to perform this study on other models to see if the impact of the choice of strategy and batch size is the same.

References

[1] M. Bernaschi, E. Agostini, and D. Rossetti, "Benchmarking multi-gpu applications on modern multi-gpu integrated systems," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 14, p. e5470, 2021, e5470 cpe.5470. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5470

[2] K. Crampon, A. Giorkallos, M. Deldossi, S. Baud, and L. A. Steffenel, "Machine-learning methods for ligand–protein molecular docking," *Drug Discovery Today*, vol. 27, no. 1, pp. 151–164, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1359644621003974

[3] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *CoRR*, vol. abs/1903.02428, 2019. [Online]. Available: http://arxiv.org/abs/1903.02428

[4] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

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

[6] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: experiences on accelerating data parallel training," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, p. 3005–3018, Aug 2020.

[7] Z. Liu, M. Su, L. Han, J. Liu, Q. Yang, Y. Li, and R. Wang, "Forging the basis for developing protein–ligand interaction scoring functions," *Accounts of chemical research*, vol. 50, no. 2, pp. 302–309, 2017.

[8] M. Su, Q. Yang, Y. Du, G. Feng, Z. Liu, Y. Li, and R. Wang, "Comparative assessment of scoring functions: The casf-2016 update," *Journal of Chemical Information and Modeling*, vol. 59, no. 2, p. 895–913, Feb 2019.

[9] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[10] Z. Xiong, D. Wang, X. Liu, F. Zhong, X. Wan, X. Li, Z. Li, X. Luo, K. Chen, H. Jiang, and M. Zheng, "Pushing the boundaries of molecular representation for drug discovery with the graph attention mechanism," *Journal of Medicinal Chemistry*, vol. 63, no. 16, pp. 8749–8760, 2020, pMID: 31408336. [Online]. Available: https://doi.org/10.1021/acs.jmedchem.9b00959

[11] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv:1812.08434 [cs, stat]*, Jul 2019, arXiv: 1812.08434. [Online]. Available: http://arxiv.org/abs/1812.08434

[12] H. Zhu and P. Koniusz, "Simple spectral graph convolution," in *International Conference on Learning Representations*, 2020.