IMPROVING THE PERFORMANCE OF 1D VERTEX PARALLEL GNN TRAINING ON DISTRIBUTED MEMORY SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Kutay Tasci
July 2024

Improving the Performance of 1D Vertex Parallel GNN Training on Distributed Memory Systems
By Kutay Tasci
July 2024

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Cevdet Aykanat(Advisor)

Hamdi Dibeklioğlu

Özcan Öztürk

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

IMPROVING THE PERFORMANCE OF 1D VERTEX PARALLEL GNN TRAINING ON DISTRIBUTED MEMORY SYSTEMS

Kutay Tasci
M.S. in Computer Engineering
Advisor: Cevdet Aykanat
July 2024

Graph Neural Networks (GNNs) are pivotal for analyzing data within graph-structured domains such as social media, biological networks, and recommendation systems. Despite their advantages, scaling GNN training to large datasets in distributed settings poses significant challenges due to the complex task of managing computation and communication costs. The objective of this work is to scale 1D vertex-parallel GNN training on distributed memory systems via (i) two-constraint partitioning formulation for better computational load balancing and (ii) overlapping communication with computation for reducing communication overhead. In the proposed two-constraint formulation, one constraint encodes the computational load balance during forward propagation, whereas the second constraint encodes the computational load balance during backward propagation. We propose three communication and computation overlapping methods that perform overlapping at three different levels. These methods were tested against traditional approaches using benchmark datasets, demonstrating improved training efficiency without altering the model structure. The outcomes indicate that multi-constraint graph partitioning and the integration of communication and computation overlapping schemes can significantly mitigate the challenges of distributed GNN training. The research concludes with recommendations for future work, including adapting these techniques to dynamic and more complex GNN architectures, promising further improvements in the efficiency and applicability of GNNs in real-world scenarios.

Keywords: Graph Neural Networks, Parallel and distributed memory systems, Graph Partitioning, Load balancing, Overlapping communication with computation.

ÖZET

DAĞITIK BELLEK SİSTEMLERİNDE 1D DÜĞÜM PARALEL GNN EĞİTİMİNİN PERFORMANSININ İYİLEŞTİRİLMESİ

Kutay Tasci
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Cevdet Aykanat
Temmuz 2024

Çizge Sinir Ağları (GNNs) sosyal medya, biyolojik ağlar ve öneri sistemleri gibi graf yapılı etki alanları bünyesinde bulunan verileri analiz etmek için büyük önem taşımaktadır. Avantajlarına karşın GNN eğitimini dağıtık ortamlarda büyük veri gruplarına ölçeklendirmek, karmaşık bir görev olan hesaplama ve iletişim maliyet yönetimi nedeniyle önemli zorluklara yol açmaktadır. Bu çalışmanın amacı, 1 boyutlu düğüm-paralel GNN eğitimini dağıtık hafıza sistemlerinde (i) daha iyi hesaplama yükü dengesi için iki-kısıtlı bölümleme formülasyonu kullanarak ve (ii) iletişim yükünü azaltmak için iletişimi hesaplamayla örtüştürerek ölçeklendirmektir. Önerilen iki-kısıtlı formülasyonda ilk kısıtlama hesaplama yükü dengesini ileri yayılım esnasındaki hesaplama yükü dengesini kodlarken ikinci kısıtlama geri yayılım esnasındaki hesaplama yükü dengesini kodlamaktadır. Örtüştürme işlemini üç farklı düzeyde gerçekleştiren üç iletişim ve hesaplama örtüştürme metodu önermekteyiz. Bu metotlar, benchmark veri setleri kullanılarak geleneksel yaklaşımlara karşı test edilmiş, motel yapısı değiştirilmeden eğitim verimliliğinde iyileşme sağlanmıştır. Sonuç, çok kısıtlı çizge bölümlemenin ve iletişim ve hesaplama örtüştürme şemalarının dağıtık GNN eğitiminin zorluklarını gözle görülür miktarda azalttığını ortaya koymaktadır. Çalışma, bu teknikleri dinamik ve daha karmaşık GNN mimarilerine uygulamak da dahil olmak üzere gelecekte yapılabilecek çalışmalara yönelik öneriler sunarak ve gerçek hayat senaryolarında GNN verimliliğinde ve uygulanabilirliğinde yapılabilecek iyileştirmelere dikkat çekerek noktalanmaktadır.

Anahtar sözcükler: Çizge sinir ağları, Koşut ve dağıtık bellekli sistemler, Çizge bölümleme, Yük dengeleme, İletişim işlem örtüştürme.

Acknowledgement

I would like to thank my advisor, Prof. Dr. Cevdet Aykanat, for his guidance and support during my graduate studies. Working under his mentorship has greatly expanded my knowledge and understanding of computer science. His help and advice have been very important in completing this thesis. I am very grateful for all he has done for me.

I also want to express my gratitude to my parents for their unwavering support and encouragement throughout my studies and my brother for answering the world's most nonsense questions any time during the day. Their belief in me and constant support have been a source of strength and motivation, helping me to overcome challenges and achieve my goals. I am deeply thankful for their love and understanding.

I want to extend my gratitude to my closest friends at Bilkent University: Arçin, Gün, Cansın, and Buğra. Their friendship and support have made my time here truly memorable. A special thanks goes to Filiz for her encouragement and guidance in my academic studies, which have been invaluable throughout my journey. Lastly, thanks to Sena for evaluating my writing with her extraordinary language skills.

Computing resources used in this work were provided by the National Center for High-Performance Computing of Turkey (UHeM) under grant number 4016742023.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In recent years, Graph Neural Networks (GNNs) have transformed deep learning by providing valuable insights from graph-structured data found in areas such as social networks, biological networks, and recommendation systems [1]. Unlike traditional deep learning methods that work well with structured Euclidean data, such as images, GNNs are designed to handle unstructured non-Euclidean data. This type of data requires complex, data-driven computations that lack clear spatial or temporal patterns [2]. Consequently, training and using GNNs in parallel and distributed computing environments is particularly challenging [3]. With the rapid growth of data volume and complexity, the need for efficient and scalable methods to process and analyze graph-structured information is increasing.

Performing deep learning tasks on graphs is particularly challenging due to the inter-sample dependency between data samples. Unlike other data types where samples are independent, each sample in a graph relies on information from its neighbors. This interconnection means that the learning process must account for these dependencies, complicating parallel and distributed computations. Full-graph training is often preferred for Graph Neural Networks (GNNs) because it ensures that all inter-sample dependencies are considered. However, this method demands significant computational resources and memory, as the entire graph

must be loaded simultaneously. Parallel processing is crucial here, as it distributes the workload across multiple processors to handle these high demands. This approach is essential in distributed memory systems, where each processor has its local memory. By efficiently dividing tasks and managing communication between processors, parallel processing on distributed memory systems can significantly enhance performance and scalability, making it possible to tackle large-scale problems in various domains such as scientific computing and big data analytics.

The core operation in GNN training revolves around the Sparse Matrix-Matrix Multiplication (SpMM) kernel, a fundamental computational kernel widely used in various iterative solvers [4]. One common approach to parallelizing sparse matrix operations on distributed memory systems is to represent the sparse matrix as a (hyper)graph and partition it, thereby distributing the matrix and associated computation among multiple processors. While there is extensive literature on this topic, GNNs present unique computation and workflow challenges requiring specialized strategies. Effective parallelization in GNNs hinges on two main principles: balancing computations and minimizing communication bottlenecks. Achieving these goals involves carefully partitioning the graph to ensure an even distribution of computational load while reducing the amount of data that processors need to exchange. This thesis adheres strictly to these principles, employing advanced partitioning techniques and leveraging the latest parallel processing methodologies to optimize GNN training. By doing so, we aim to enhance the efficiency and scalability of GNN computations, addressing the distinctive demands of graph-based deep learning.

Existing research on parallel processing of the GNNs shows the importance of graph partitioning and efficient parallelization [3, 5]. This thesis emphasizes improving training and inference performance by focusing on these tasks. The proposed two-constraint partitioning scheme provides an effective model for the GNN computations to achieve better load balance. This improvement in load balancing results in enhanced scalability and runtime performance in parallelization. Additionally, to further improve the parallelization of the SpMM kernel, communication and computation overlapping algorithms are used [6]. This work

presents three schemes for implementing communication and computation overlapping in GNNs, enabling efficient training and inference. When used together, the proposed methods provide efficiency and improved scalability in parallel GNN training within distributed memory environments.

The organization of the thesis is as follows: Chapter 2 provides background information about GNNs and graph partitioning, outlining the core concepts and methodologies that support our research. A literature survey is included in Chapter 3, where existing works on parallel GNNs training methods, graph partitioning, and communication with computation overlapping algorithms are discussed. Chapter 4 describes developing and implementing our innovative two-constraint partitioning method and the overlapping communication and computation techniques. Chapter 5 details the experimental setup, including the datasets used, the experimental design, and the metrics for evaluating performance. Chapter 6 presents the results of our experiments, providing a comprehensive analysis of the effectiveness of our proposed methods compared to traditional approaches. Finally, Chapter 7 concludes the thesis, summarizing key findings and discussing potential avenues for future research that could build on the groundwork laid by this study.

Chapter 2

Background and Preliminaries

2.1 Graph Neural Networks

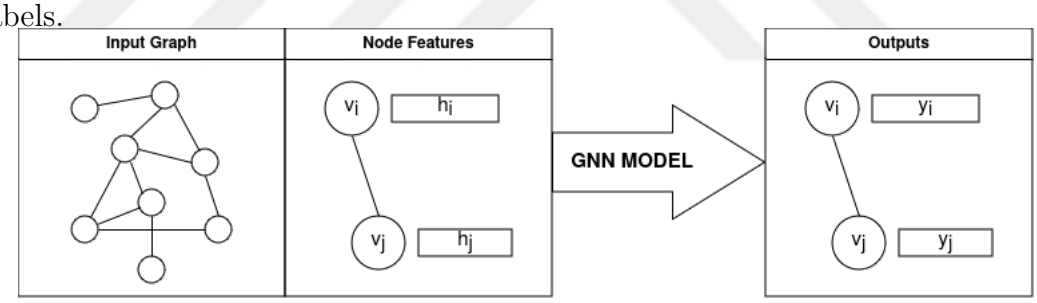
Graph neural networks (GNNs) are increasingly recognized as powerful tools in the deep learning landscape, adept at handling various classification and regression tasks on graph-structured data. These tasks encompass a wide range of applications, such as node classification, where individual nodes are categorized into classes; link prediction, which involves predicting the likelihood of a connection between nodes; and graph classification, where entire graphs are classified based on their structure and node attributes [7, 8]. The versatility of GNNs allows them to effectively model network data from diverse domains. Examples include social networks, where they can identify influential users or predict network growth patterns; citation networks, used to recommend articles or predict citation counts; biological networks, such as protein-protein interactions, where they help in understanding cellular processes; and communication networks like 5G, where they optimize routing and improve service quality [3].

Table 2.1: Symbols and notations used for formulizing GNNs.

Symbol	Description
G	Graph
V	Set of vertices
E	Set of edges
A	Adjacency matrix representation of the graph G
H^{ℓ}	Vertex embeddings in layer ℓ / H^0 denotes input vertex features
G^{ℓ}	Output error of layer ℓ
$N(v_i)$	The neighborhood of a vertex v_i
ψ	Edge normalization function
$\oplus$	Permutation-invariant aggregation function
ϕ^{ℓ}	Linear projection function of layer ℓ
W^{ℓ}	Weights associated with ϕ^{ℓ}
$\mathcal{L}$	Number of layers

Node classification serves as a fundamental example for understanding and illustrating the workflow of a GNN model. Table 2.1 provides the symbols and notations used to formally define GNN computations. An example GNN model designed to perform node classification operates as follows: An input graph $G = (V, E)$ is provided, where V represents vertices and E represents edges. Input features representing the information associated with vertices and/or edges are also provided. In this example, each vertex $v_i \in V$ is associated with a feature vector h_i. This data is then processed by a GNN model comprising one or more GNN layers to produce an output feature vector y_i for each vertex $v_i \in V$. Node classification is achieved by assigning a label to each vertex v_i based on the output vector y_i. Figure 2.1 provides a high-level summary of the node classification task.

Figure 2.1: This diagram shows a Graph Neural Network (GNN) model workflow. It includes the input part with graph and node features and the output with node labels.



Graph Convolutional Networks (GCNs), proposed by Kipf and Welling [9], represent a straightforward GNN model designed primarily for node classification. The model comprises two GCN layers, separated by a ReLU activation layer. Without loss of generality, the GCN model exemplifies two fundamental operations performed in GNN models: aggregation and update operations. To articulate these two fundamental steps, Besta and Hoefler [3] proposed two different formulations: local formulation and global formulation. The local formulation offers a graph-theoretic framework, while the global formulation translates these operations into matrix multiplication operations. In practice, there are no significant differences between the two formulations.

2.1.1 Local Formulation

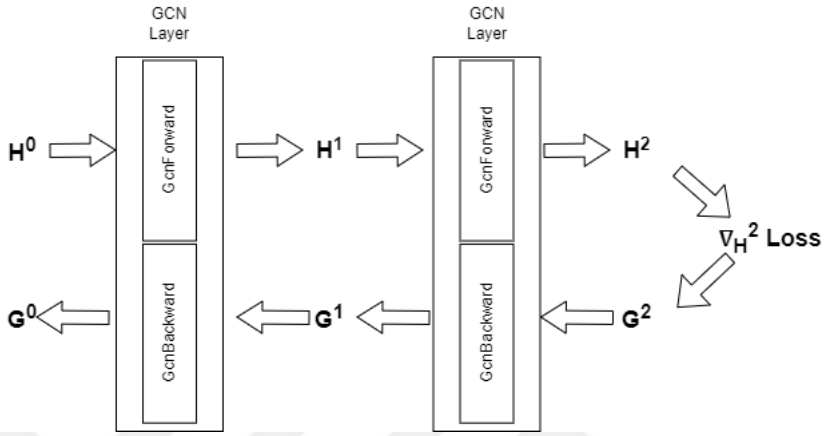
In the local formulation, a graph G is defined as a pair consisting of a set of vertices V and a set of edges E. Two vertices v_i and v_j are considered to be neighbors if there exists an edge $e_{ij} = (v_i, v_j) \in E$, showing a direct link. The neighborhood $N(v_i)$ of a vertex v_i is the set of all vertices v_j that are connected to v_i by an edge i.e., $N(v_i) = \{v_j : (v_i, v_j) \in E\}$. Each vertex v_i is associated with a feature vector h_i^0 of size f, representing its attributes. Edges e_{ij} may also be associated with a function ψ that provides additional information on the relationship between vertices.

In GNNs, each layer ℓ includes a permutation-invariant aggregation function $\oplus$ such as sum, mean, or max and a learnable linear projection function ϕ. The aggregation function $\oplus$ combines features from a vertex's neighborhood and itself to capture its local structure, while the projection function ϕ, using a weight matrix W^ℓ of size $f \times k$, updates these features into a new space. This allows the network to learn complex patterns as it deepens.

Figure 2.2 provides a visualization of the computations of a GNN model with two layers for training. Visualization clearly shows the execution and dependencies in the computations; a training iteration of a GNN model consists of one forward and one backward pass, whereas an inference iteration only performs one forward iteration. In the forward pass, the input matrix H^0 is transformed by the first GCN layer, and the output of this operation is used as the input of the second layer. At the end of the forward iteration, an output error G^2 is calculated for a given loss function. This output error is used in the backward pass as the primary input and propagated through layers in the reverse order.

The forward propagation process in GNNs, described in Algorithm 1, involves aggregating features of a vertex's neighbors and itself to effectively capture the local graph structure. This process may include transforming features through ψ to adjust the influence of neighboring vertices. After aggregation, the features are projected into a new space using W^ℓ, which directly represents a fully connected layer in traditional neural networks and enhances the network's ability to learn

Figure 2.2: The visualization of the forward and backward propagation steps of a two-layer GNN model.



from the graph structure. Since vertex features are updated, this second step can be called the update step. Through this process, a GNN layer generates a new vertex feature vector h_i^ℓ of size $1 \times k$ that encapsulates information about a vertex and its neighbors.

The back-propagation process employs an aggregation and update approach similar to forward propagation. The goal is to minimize classification error by updating the projection function ϕ and the weight matrix W^ℓ. An output error vector g_i^ℓ of size k represents the projection error for vertex v_i, and the initial output error $g_i^\mathcal{L}$ is calculated using a loss function and ground truth. The back-propagation process is described in Algorithm 2. First, a similar aggregation step is performed, differing by reversing edge directions in the graph and utilizing a function $N'(v_i)$ for reversed neighbor sets. Following that, an input error $g_i^{\ell-1}$ for the prior layer is computed using the function ϕ'. Finally, the gradient matrix Y^ℓ of size $f \times k$ is computed by calculating the outer product of the column vector $(h_i^{\ell-1})^T$ with the row vector x. This is done to adjust the weight matrix W^ℓ, optimizing the projection function to minimize classification errors.

Algorithm 1 Forward-propagation of a GCN layer for the local formulation.

Require: Aggregator Function $\oplus$, ψ
  procedure GcnForward(V, $H^{\ell-1}$, ϕ^{ℓ})
    for each $v_i \in V$ **do**
      $x \leftarrow h_i^{\ell-1}$
      for each $v_j \in N(v_i)$) **do**
        $x \leftarrow \oplus(x, \psi_{i,j}(h_j^{\ell-1}))$
      end for
      $h_i^{\ell} \leftarrow \phi^{\ell}(x)$ ▷ Corresponds to a fully-connected deep learning layer.
    end for
    return H^{ℓ}
  end procedure

Algorithm 2 Backward-propagation of a GCN layer for the local formulation.

Require: Aggregator Function $\oplus$, $H^{\ell-1}$ from GcnForward, ψ
  procedure GcnBackward(V, G^{ℓ}, ϕ')
    for each $v_i \in V$ **do**
      $x \leftarrow g_i^{\ell}$
      for each $v_j \in N'(v_i)$) **do**
        $x \leftarrow \oplus(x, \psi_{i,j}(g_j^{\ell}))$
      end for
      $g_i^{\ell-1} \leftarrow \phi'(x)$
      $Y^{\ell} \leftarrow Y^{\ell} + (h_i^{\ell-1})^T x$
    end for
    $W^{\ell} \leftarrow W^{\ell} - Y^{\ell}$ ▷ Updating the weights of linear projection function ϕ
    return $G^{\ell-1}$
  end procedure

2.1.2 Global Formulation

In the global formulation, a graph $G = (V, E)$ is considered as its $n \times n$ adjacency matrix, $A = (a_{ij})$ where rows and columns of A correspond to vertices in V and each edge $(v_i, v_j) \in E$ incurs a non-zero a_{ij} at row i and column j of A. Typically, self-loops are added to A to preserve the vertices' state, and then A's rows and columns of A are normalized [9] for additional spectral information. Note that due to the sparse nature of the graphs, A is represented as a sparse matrix in CSR format. Vertex features of the graph are given as an $n \times f$ matrix H^0 where f represents the size of the feature vectors of vertices, and each row h_i corresponds

to the features of vertex v_i. For performing linear transformations, each GNN layer ℓ is associated with a learnable weight matrix W^ℓ of size $f \times k$, again directly representing a fully connected layer.

The matrix-theoretic formulation aims to represent the GNN computation as a sequence of algebraic operations. As seen in Algorithm 3, computation is reduced to two fundamental operations. The aggregate step in the global formulation is reduced to a sparse-dense matrix multiplication operation $A^T H^{\ell-1}$, where the aggregator operation performs sum over the feature vectors and produces an intermediate X matrix of size $n \times f$. Note that the SpMM kernel needs to be modified for aggregators other than sum. Meanwhile, the normalization of A mirrors the function ψ, which scales the feature vectors in aggregation. After aggregation, we feed the resulting matrix to a fully connected layer represented as XW^ℓ that produces the output matrix H^ℓ of size $n \times k$.

In the global formulation for backward propagation, detailed in Algorithm 4 is simplified to matrix multiplication operations. The aggregation step is executed by multiplying the AG^ℓ matrix, with edge direction reversals implemented via transpose operations. To compute the input error and pass this error back to the previous GNN layer $(\ell - 1)$, the operation $X(W^\ell)^T$ is used. Weight updates are conducted in two phases: initially, gradients are calculated using $(H^{\ell-1})^T X$, followed by the weight adjustment $W^\ell \leftarrow W^\ell - Y^\ell$, streamlining the back-propagation process of GNNs in matrix-theoretic formulation.

Algorithm 3 Forward-propagation for the global formulation.

procedure GcnForward(A^T, $H^{\ell-1}$, W^ℓ)
  $X \leftarrow A^T H^{\ell-1}$
  $H^\ell \leftarrow XW^\ell$
  return H^ℓ
end procedure

Algorithm 4 Backward-propagation for the global formulation.

Require: H^{l-1} from GcnForward
  procedure GcnBackward(A, G^ℓ, W^ℓ)
    $X \leftarrow AG^\ell$
    $G^{\ell-1} \leftarrow X(W^\ell)^T$
    $Y^\ell \leftarrow (H^{\ell-1})^T X$
    $W^\ell \leftarrow W^\ell - Y^\ell$
    return $G^{\ell-1}$
  end procedure

2.2 Graph Partitioning

A graph $G = (V, E)$ consists of a set V of vertices and a set E of edges. Each edge $(v_i, v_j) \in E$ connects two vertices. Each vertex v_i can be associated with a weight $\omega(v_i)$ and each edge e_{ij} can be associated with cost $\omega(e_{ij})$. $\Pi(G) = \{V_1, V_2, ..., V_K\}$ is said to be a K-way partition of G if parts are mutually disjoint ($V_x \cap V_y = \emptyset$ for each pair of parts V_x and V_y) and mutually exhaustive ($V_x \cup V_y \ldots \cup V_k = V$).

In a partition Π, an edge is said to be cut if v_i and v_j are in different parts. The cut size of a partition Π is defined as the sum of the costs of the cut edges, i.e., $cutsize(\Pi) = \sum_{e_{ij} \in E_{cut}} \omega(e_{ij})$. In a partition Π, the weight of a part W_k is defined as the sum of the weights of its vertices, i.e., ($W_k = \sum_{v_i \in V_k} \omega(v_i)$). Partition Π is said to be balanced if each part V_k satisfies $\omega(V_k) \leq (1 + \epsilon)W_{avg}$, where ϵ is the maximum allowable imbalance ratio and W_{avg} is the average part weight ($W_{avg} = (\sum_k W_k)/K$). In the graph partitioning problem, the partitioning constraint is to satisfy the balance constraints among the part weights, whereas the partitioning objective is to minimize the cut size.

In multi-constraint graph partitioning, each vertex is associated with multiple weights $\omega^c(v_i)$ inducing multiple weights for each part $W_k^c = \sum_i \omega^c(v_i)$, for $c = 1, \ldots, C$. In this version, a partition Π is said to be balanced if each part V_k satisfies all of C constraints, $\omega^c(V_k) \leq (1 + \epsilon)W_{avg}^c$, where W_{avg}^c is the average part weight over the c^{th} vertex weights ($W_{avg}^c = (\sum_k W_k^c)/K$).

The graph partitioning problem is known to be NP-Hard [10]. On the other

hand, there exists successful graph partitioning tools utilizing the multi-level paradigm, such as METIS [11]

Chapter 3

Literature Survey and Discussions

This section will cover several key topics related to this thesis and how they are used and improved. Firstly, we'll discuss how GNNs are trained in parallel and distributed computing environments to enhance the efficiency of the process. After that, we'll discuss graph partitioning, a technique that helps optimize GNN performance and efficiency. Finally, we'll explore how to improve efficiency by overlapping communication with computation when training GNNs. Each part of this section aims to help us understand how GNNs work and how we can improve them.

3.1 Parallel and Distributed GNN Training

Parallel and distributed training methods have become essential for scaling Graph Neural Networks (GNNs) to accommodate large datasets typical in many real-world applications. Unlike their embarrassingly parallel counterparts, GNNs require more sophisticated methods and implementations in parallelization. These methods enable GNNs to handle large graphs by efficiently distributing the data

and computational workload across multiple processors or machines. The main advantage of parallel and distributed training is the ability to train complex GNN models on large-scale graphs within a reasonable time frame. By speeding up GNN training, we can train additional candidate models, enabling detailed tuning of architectural settings and model hyperparameters for improved reliability and efficiency. Additionally, this approach enhances the minimization of carbon footprint due to energy consumption.

A common taxonomy for parallel processing in neural networks distinguishes between two main strategies: data parallelism and model parallelism. These approaches are fundamental for efficiently training large-scale models, especially as datasets and network architectures grow increasingly complex. Data parallelism in GNNs typically involves distributing data across different processors to perform parallel computations, a method that can significantly speed up the training process of GNNs [3]. Model parallelism involves distributing different parts of a neural network model across multiple computational units, allowing for the simultaneous processing of complex and large models that exceed the memory capacity of a single device. However, data and model parallelism often intertwine due to inter-sample dependencies between data points in GNNs, where the input graph is a part of the model structure and a deciding factor in data distribution. This special type of parallelism falls into the category of graph partition parallelism.

Most graph partitioning parallelism schemes predominantly rely on assigning vertices to workers, known as "1D partitioning" or "Vertex-parallel." This approach is straightforward to implement but often faces challenges in achieving efficient load balancing. A more complex "2D partitioning" or "Edge-parallel" strategy is sometimes employed to enhance load balancing. Although this variant offers improved load distribution, it is less commonly supported and represents an area open for further development [3]. There exist works to increase the performance of graph partitioning parallelisms, such as communication-avoiding algorithms and combinations with other types of parallelism. For example, CAGNET [12] combines partitioning with replication to accelerate GNN training through communication avoidance, and models like ZIPPER [13] integrate graph partition parallelism with pipelining techniques to achieve further speedups and reduce

storage resource usage, showcasing the potential for hybrid approaches in optimizing GNN training infrastructure.

Recent studies address the load-balancing challenges in graph partition parallelism for GNNs. The work on GNNIE [14], a GNN Inference Engine, implements a novel approach to manage the variability in workload caused by the sparsity of data and power-law distribution of node degrees typical of graph data. Their approach includes a combination of load (re)distribution and a flexible architecture designed to balance the load during the weighting phase of GNN operations, thus improving overall efficiency and speed. Similarly, another work [15] highlights the inadequacies of traditional graph partitioning methods when applied to parallel GNN training due to the dynamic and irregular nature of GNN operations, particularly the SpMM operation, which is crucial for GNN efficiency. They propose an adaptive load-balancing strategy that not only initially distributes workloads through graph partitioning but also dynamically adjusts them during training to manage the computational loads more effectively. These contributions underline a shift towards more dynamic and adaptable solutions in graph partition parallelism. However, these methods propose solving the load-balancing problem during the runtime, and none of them proposes a solution to the problem in the partitioning step prior to execution.

3.2 Graph Partitioning

Graph partitioning is a crucial technique in optimizing graph neural networks (GNNs), especially for improving the efficiency of the Sparse Matrix-Matrix Multiplication (SpMM) operation, which is the fundamental operation in GNN processing. Traditional graph partitioning approaches focus on dividing a graph into subgraphs or parts such that the number of edges between these parts is minimized while maintaining a balance in the size of these parts. This approach minimizes communication overhead and balances computational load among processors. This balance is crucial to ensure an even distribution of computational load across different processors in a distributed environment.

Graph and hypergraph partitioning methods are extensively used to efficiently parallelize the SpMM kernel [4]. Although there are different partitioning methods, multi-level (hyper)graph partitioning methods have proved their superior efficiency and performance [10, 16]. Of these two methods, hypergraph partitioning provides exact modeling of the communication volume in the SpMM operation, thus making it a potentially more effective method for GNNs. However, parallel algorithms for graph partitioning such as ParMetis [17] have gained attention because of their effectiveness in handling large-scale graphs by utilizing multiple processors effectively and reducing partitioning time overhead during GNN training and inference [18].

The effectiveness of different partitioning methods in GNNs is examined under three constraints: communication time, computation time, and amortization time. The partitioning algorithm handles communication and computation time, thus aiming to decrease the overall time spent on GNN training and inference. Another factor is the amortization time for a smart partitioning scheme to be effective; the total time spent in the training should be decreased when time spent on partitioning is included. This means time spent in partitioning should be amortized during the GNN training as fast as possible before the model converges. Existing methods mostly provide feasible solutions under these constraints [5], but parallel (hyper)graph partitioners take the lead when amortization time is also considered [18]. Although there exist parallel hypergraph partitioning algorithms such as ParHIP [19], existing frameworks such as DistDGL provide support for ParMetis [20].

3.3 Overlapping Communication with Computation

In the field of high-performance computing, overlapping communication and computation has long been a crucial strategy for optimizing the efficiency of parallel systems. This approach seeks to minimize idle time and maximize resource

utilization by allowing computational tasks to proceed concurrently with data transmission processes. One of the core techniques in this strategy involves non-blocking collective operations, which enable systems to initiate a communication process and continue with computation without waiting for the communication to complete.

The concept, thoroughly explored in existing research [21, 22, 23], involves sophisticated management of communication tasks that are integrated into the computational processes using well-designed algorithms and high-level communication routines. This method significantly enhances the overall application performance by utilizing the idle time in communication. These techniques are particularly effective in environments where communication might otherwise become a bottleneck, offering an elegant solution to improve throughput and reduce runtime in complex parallel applications.

The integration of non-blocking operations into standard communication libraries like MPI (Message Passing Interface) showcases the practical application of this strategy, enabling a wide range of applications to benefit from enhanced parallelism and efficiency. This approach is not just limited to theoretical exploration but has been applied in various real-world applications, demonstrating its effectiveness in reducing latency and bandwidth costs by utilizing idle time and increasing the efficiency of data-intensive tasks. However, these non-blocking operations suffer from additional computational overheads in their implementation and integration with various hardware infrastructures [24]. These overheads are often amortized by hiding them in the communication time, but the extensive use requires optimization of these operations to become crucial to performance.

Building on existing methods for overlapping communication and computation, recent works have also focused on the SpMM kernel, the core operation of GNNs [6], particularly focusing on sparse tall-and-skinny matrix multiplication. These matrices often appear in high-dimensional data applications such as GNN training, and managing them efficiently is crucial. Efforts have focused on refining algorithms for Sparse Matrix-Matrix Multiplication (SpMM). These improvements include better ways to access memory and streamline workflows to

increase data flow and reduce delays caused by data retrieval—both critical for maintaining high performance during calculations.

Chapter 4

Efficient Vertex-Parallel GNN Training

This section presents the methodologies and techniques developed to enhance the performance and efficiency of parallel Graph Neural Networks (GNNs) on distributed-memory parallel systems through advanced partitioning and overlapping communication and computation strategies. This chapter will detail the implementation and optimization of these methods, discussing their impact on improving scalability and reducing computational overhead in distributed computing environments. The focus will be on how these innovative approaches help manage and accelerate the training of GNNs, ensuring they are fit for large-scale applications.

4.1 Vertex-Parallel GNN Training

A definition of an atomic task is needed to develop a parallel algorithm. An atomic task refers to an indivisible computation that can only be processed by a single processor. The defined atomic tasks will be processed in parallel by partitioning the related computations among the processes. The partition

$\Pi(V) = \{V_1, V_2, ..., V_k\}$ is depicted as assigning the atomic tasks V_k to process P_k without loss of generality. Hence, in the rest of the thesis, part and processor will be used interchangeably. In this case, we will define the atomic tasks as the computations associated with each GNN vertex, resulting in a coarse-grain data parallelism.

For a vertex parallel GNN framework, computations related to each vertex v_i are defined as an atomic task t_i. As mentioned in the previous sections, a vertex v_i is associated with two fundamental operations: aggregate (SpMM) and update (GEMM). Therefore, each task t_i represents these operations performed for a vertex v_i. That is, atomic task t_i refers to (i) performing aggregation on all vertices $v_j \in N(v_i)$, (ii) feeding the resulting vector to a fully connected layer (update). In this framework, each task t_i is assigned to a specific processor p_k along with the adjacency information ($N(v_i)$) and feature vector (h_i^{l-1}) of v_i. After the computations related to the task t_i are executed, the updated feature vector h_i^l will be calculated in p_k conformally. If the feature vector of a neighboring vertex $v_j \in N(v_i)$ is kept in a different processor, communication is incurred to receive the necessary information. In matrix-theoretic formulation, this corresponds to row-wise distributing the adjacency matrix A and feature matrix H to the processors, which also corresponds to a 1D row-parallel SpMM operation.

Utilizing this task parallelism, the aggregation step can be executed in parallel across all processors. In this step, a task t_i needs all feature vectors of the vertices in $N(v_i)$; an aggregation operation can potentially incur communication between processors if two neighbors v_i and v_j are in different processors. By replicating the model weights W among all processors, an update step for each vertex can be performed locally without incurring communication since it only depends on the aggregation step's output. Among these two operations, the aggregate operation potentially incurs communication, and the update operation can be performed locally since weights W are replicated among all processors. Without a loss of generality, the update step can be performed directly on each processor without requiring a sophisticated approach.

A vertex-based parallel implementation of the aggregate operation is given in Algorithm 5. The algorithm can be divided into two steps: communication and computation. In the communication step, each process p_k performs P2P communications with the other processes to fulfill the dependencies for each task t_i. Each task t_i requires the feature vectors of every vertex in $N(V_i)$ if any vertex $v_j \in N(v_i)$ remains in a different processor, then p_k needs to receive h_j to complete the aggregation step of t_i. A vertex v_i is a boundary vertex if any $v_j \in N(v_i)$ remains in a different processor. So each processor performs an expand operation, effectively sending the feature vectors of their boundary vertices to other processors that need this information and symmetrically receiving the feature vectors required to complete its computations. At the end of the communication step, each vertex receives the information it needs to complete its aggregation operation. Most often, a synchronization step exists after the communication step to ensure that all dependencies are fulfilled. In the computation step, each vertex performs aggregation on its neighbors as expected.

Algorithm 5 1D Vertex parallel aggregate operation in graph-theoretic formulation.

procedure AGGREGATESTEP(G_{local}, H_k^{l-1}, $SendList$, $RecvList$)
  for each $p_k \in RecvList$ **do**
    $MPI\_Irecv(recvBuffer_k, p_k)$
  end for
  Fill all buffers $sendBuffer_K$
  for each $p_k \in SendList$ **do**
    $MPI\_Send(sendBuffer_k, p_k)$
  end for
  Wait on Irecv calls
  $H_k^{l-1} \leftarrow H_k^{l-1} \cup recvBuffer_K$
  $X_k \leftarrow Aggregate(G_{local}, H_k^{l-1})$
  return X_k
end procedure

4.2 Load Balancing for Vertex-Parallel GNN Training

For an efficient parallelization of a GNN model, providing a good task-to-processor assignment is crucial to minimize communication and maintain a workload balance. Previous studies highlight the importance of graph partitioning for parallel GNN processing [5]. The irregular computational dependencies of the aggregation step necessitate distributing the tasks among the processors smartly. Recall that the aggregate operation is formalized as an SpMM operation in the matrix-theoretic formulation. Graph and hypergraph partitioning models [10, 11, 25] have been used for parallel processing SpMM operation. By representing the GNN computations as a task-interaction graph, efficient task parallelization can be achieved through graph partitioning. Graph partitioning helps manage data more efficiently, significantly reducing the data exchanges between processors compared to splitting the tasks evenly. In addition to reducing the communication, the models provide a workload balance between processors since partitioning constraints include maintaining the balance of part weights.

The task-interaction graph that models the communication and computation of the GCN layer represents each task by a task vertex t_i. These vertices also represent the data associated with task t_i, which is the feature vector h_i^{l-1}. Therefore, each edge in the task-interaction graph represents the data interdependence between tasks. With proper vertex weighting in the task-interaction graph, these models can correctly encapsulate the computational load, and edges represent the interdependence. To achieve load balance in the parallelization of a GNN model, it is essential to set the vertex weight correctly in the ask-dependency graph to model the computation loads correctly.

In the aggregation step, a task t_i needs the feature vectors of their neighboring vertices in $N(v_i)$. This data dependency incurs communication between processors if v_i and v_j are assigned to different parts. For a better formulation consider a K-way partition $\Pi = \{V_1, V_2, ..., V_K\}$ an edge e_{ij} called a cut-edge if v_i and v_j are assigned to different parts. Thus, every cut edge e_{ij} resembles the communication

dependency between processors. Total communication can be represented as the cut size of Π, which is the number of the cut-edges. Minimizing the cut size also minimizes the communication between processors to complete the aggregation step, where every vertex receives the feature vectors of their neighbors. To efficiently implement parallel SpMM, consider a vertex v_i with adjacent vertex set $N(v_i)$. Let $\Lambda(v_i)$ denote the distinct set of parts with at least one vertex in $N(v_i)$. Then, the source process will expand h_i^ℓ to all processes in $\Lambda(v_i) - \Pi(v_i)$ Thus, part $\Pi(v_j)$ receives the data associated with v_i only once to prevent redundant communication. Graph partitioning fails to capture the communication model correctly, while a hyper-graph partitioning model correctly models the communication. Thus, a graph partitioning model can only provide an approximate communication model.

Understanding the computational workload of a Graph Neural Network (GNN) layer is essential for maintaining load balance in each partition. In the matrix-theoretic approach, we defined the core operations within a GNN layer, as outlined in Table 4.1, where the operational workloads are specified. The "Total work" column quantifies the cumulative work required for each operation. In contrast, the "Work-per-vertex" column details the work associated with each vertex, thus the cost of each atomic task. These operations can be separated into two computation steps, forward and backward. Since no synchronization is needed between aggregate and update steps, the sum of the workloads in each step provides the total computational work of each step.

Table 4.1: Workloads of the computations for a GNN layer in the matrix-theoretic formulation.

Equation	Total Work	Work Per Vertex	Operation	Step
$A^T H^{\ell-1}$	$nnz(A^T) \times f$	$deg(v_i) \times f$	SpMM	Forward
XW^ℓ	$n \times f \times k$	$f \times k$	GEMM	Forward
AG^ℓ	$nnz(A) \times k$	$deg(v_i) \times k$	SpMM	Backward
$X(W^\ell)^T$	$n \times k \times f$	$k \times f$	GEMM	Backward
$(H^{\ell-1})^T X$	$f \times n \times k$	$f \times k$	GEMM	Backward

$$\begin{aligned} \omega_{forward}(v_i) &= deg(v_i) \times f + f \times k \\ \omega_{forward}(v_i) &= f \times (deg(v_i) + k) \end{aligned} \tag{4.1}$$

$$\begin{aligned} \omega_{backward}(v_i) &= deg(v_i) \times k + k \times f + f \times k \\ \omega_{backward}(v_i) &= k \times (deg(v_i) + 2f) \end{aligned} \tag{4.2}$$

The objective of designing a parallel framework for Graph Neural Networks (GNNs) is to distribute computational work across processors evenly. For vertex-based partitioning, vertex workloads are defined to serve as weights in the partitioning process. These weights, derived from operations that proceed without synchronization, are aggregated into a single metric. Specifically, the computational load for a vertex v_i during forward propagation is represented as in 4.1, and for backward propagation in 4.2. These expressions can be simplified to $(deg(v_i) + k)$ for forward steps and $(deg(v_i) + 2f)$ for backward steps by removing constant factors, enabling a regularized approach to coarse-grain workload distribution in parallel GNN processing. Thus, a two-constraint partitioning is performed to maintain the load-balance of partitions in forward and backward propagation.

4.3 Overlapping Communication and Computation

The previous section provides a simple implementation of 1D vertex parallel aggregation operation. In that approach, communication and computation steps are separated by a synchronization step, which ensures that a processor receives all the information required for the computation step. During this synchronization step, a processor waits idle until it receives all the necessary messages. We propose to utilize this idle time by performing computations that are not dependent on the communication step, thus overlapping the computation with communication. In this section, the methods and algorithms that have been proposed and used to

implement efficient communication and computation overlapping algorithms for GNN computations will be discussed.

In the aggregation step, local-local computations can be separated into groups based on their dependencies on incoming messages, such as local-local and local-nonlocal computations. For a vertex v_i, an aggregation operation is performed on the feature vectors of vertices $v_j \in N(v_i)$. This operation can be performed locally without needing local-nonlocal data if these two vertices exist in the same part. These types of computations are grouped as local-local computations. If two vertices are not allocated to the same part, these computations are grouped as local-nonlocal computations dependent on the communication step. Between these two groups, local-local computations can be performed before receiving the incoming messages. The communication time can be utilized to perform local-local computations by delaying the communication synchronization.

Three different communication and computation overlapping algorithms are proposed in this work. In all algorithms, communication is performed by non-blocking P2P communication collectives, with no change in the contents of the message. Since non-blocking collectives are utilized, we can send the messages as we fill the send-buffers to take advantage of early message sending. Note that early message sending also overlaps some of the computation and communication. After initiating communication requests, synchronization is delayed until each processor performs the local-local computations. Algorithms differ in synchronization schemes and performed computations during the communication phase. Two of the algorithms overlap the local-local computations of the aggregation step only. "Partial" and "Full" algorithms separate the local-local and local-nonlocal computations. One algorithm extends this overlapping scheme with the update step for the "Partial" algorithm, which includes updating the vertex features. An extended version of the Full algorithm is also possible, but extra optimizations must be developed efficiently in the partitioning step.

4.3.1 Partial Communication and Computation Overlap

In this approach, local-local and local-nonlocal computations are strictly separated into two parts to be calculated separately. Algorithm 6 shows the proposed communication and computation scheme. The algorithm delays the synchronization step after the local-local aggregate computations. Thus, a processor can perform local-local operations while waiting for messages to arrive. Local-local aggregation operation for a vertex v_i is performed by aggregating feature vectors of the local neighboring vertex v_j, such that $\Pi(v_i) = \Pi(v_j)$. Then, after all messages arrive, each processor continues the aggregation with the local-nonlocal computations, where $\Pi(v_i) \neq \Pi(v_j)$. For an efficient implementation, the adjacency matrix of a given local graph is divided into two sparse matrices: A_L^k for local-local computations and A_{NL}^k for local-nonlocal computations. This distinction allows the aggregation step to be performed in two separate steps. Then, the final output can be reformed by adding local-local and local-nonlocal aggregation outputs together.

This scheme has a strict synchronization barrier between two groups of local computations. Local-local and local-nonlocal computations are executed as two distinct computations divided by this synchronization step. This provides a simple method for overlapping communication and computation. The downside of this approach is that the execution of local-nonlocal computations depends on the arrival of multiple messages from multiple parts. This means an individual process that can continue the computations; every message must wait until the last receive request arrives. This potentially leads to an inefficiency if the arrival time of one or more messages exceeds other messages. Since, as we have done in the local-local computations, all computations dependent on vertex data can be calculated when the required data is available.

Algorithm 6 Partial Communication Computation Overlap Algorithm

procedure AGGREGATESTEP(V_{local}, H_k^{l-1}, $SendList$, $RecvList$)
  for each $p_k \in RecvList$ **do**
    $MPI\_Irecv(recvBuffer_k, p_k)$
  end for
  for each $p_k \in SendList$ **do**
    Fill $sendBuffer_k$
    $MPI\_Isend(sendBuffer_k, p_k)$
  end for
  $X \leftarrow AggregateLocalLocal(V_{local}, H_k^{l-1})$
  Wait on Irecv calls
  $X \leftarrow X + AggregateLocalNonlocal(V_{local}, recvBuffer_K)$
  return X
end procedure

4.3.2 Full Communication and Computation Overlap

This approach solves the inefficiency in the previous algorithm by handling each message and related local-nonlocal computations separately. Like the local-local computations, each local-nonlocal computation can be executed without waiting for the arrival of the other messages. In the previous approach, synchronization was performed entirely on all incoming messages, ignoring the available local-nonlocal computations until all messages were received. Algorithm 7 introduces flexibility for the synchronization step to overcome this inefficiency by handling each incoming message separately. Instead of waiting for all incoming messages to proceed to local-nonlocal computations, computations related to each incoming message are executed when necessary information is acquired. To separate the aggregation operation this time, we separate a local adjacency matrix A into sub-matrices $A_1, .., A_k$ by their part. This allows the computation of the aggregation operation to be divided into sub-computations that can be performed when a communication request has been completed.

Separating the local-nonlocal computations concerning their dependent parts/processors enables the aggregation operation to be completed immediately. This flexibility allows a processor to perform computations with the currently available information rather than waiting for all incoming messages. Thus, the

strict synchronization step is loosened, and synchronization is performed separately for each incoming message instead of performing synchronization for all messages. In the worst case, runtime is bounded by the latest message arrival time and computations related to that message. This leads to minimal or no idle time in the aggregation step by overlapping the communication bound by the latest message arrival time.

Algorithm 7 Communication Computation Overlap Algorithm

procedure AGGREGATE(V_{local}, H_k^{l-1}, $SendList$, $RecvList$)
  for each $p_k \in RecvList$ **do**
    $MPI\_Irecv(recvBuffer_k, p_k)$
  end for
  for each $p_k \in SendList$ **do**
    Fill $sendBuffer_k$
    $MPI\_Isend(sendBuffer_k, p_k)$
  end for
  $X \leftarrow AggregateLocalLocal(V_{local}, H_k^{l-1})$
  for $i \leftarrow 0$ **to** $|RecvList|$ **do**
    $k \leftarrow$ Wait any of IRecv calls
    $X \leftarrow X + AggregateLocalNonlocal(V_{local}, recvBuffer_k)$
  end for
  return X
end procedure

4.3.3 Extended Partial Communication and Computation Overlap

An extended method is possible for both "Partial" and "Full" algorithms by introducing additional computation for overlapping. The extended method aims to enhance the inclusiveness of communication and computation overlap schemes from the aggregation step and include the update step. This can be achieved by performing the update operation on local-local and local-nonlocal aggregation results separately and then calculating the final results by summing up these two. When local-local and local-nonlocal computations are divided similarly to the "partial" algorithm, two update operations are performed instead of one.

Doing this doubles the update operations computational load for vertices with local-local and local-nonlocal computations.

Algorithm 8 provides an algorithm that performs aggregation and update operations using the Extended Partial Communication and Computation scheme. Unlike the previous algorithms, a complete forward operation is given for a GNN layer capable of overlapping the computation of aggregate and update steps. The communication step for the aggregation step is executed with no difference, and the synchronization step is delayed until all local-local computations are complete. In the local-local computations, aggregation is performed with the local information available, and the update step calculates the partial results with the available results. For the local-nonlocal computations, aggregation is performed with received vertex features. Note that these partial local-nonlocal aggregation results must kept separated from the local-local aggregation results, and then an update step is performed on these partial results. Finally, we can add these two products and calculate the updated vertex features.

The extended method can potentially overlap more computations compared to other methods by increasing the overall computational load. On the other hand, with the increasing size of the processor counts, it can potentially utilize the increasing communication time by doing more computations. Communication and computation overlapping algorithms are more beneficial when communication and computation times are comparable. Thus, when the communication time significantly overruns the computation time, overlapping the update step can extend the amount of GNN computations integrated into communication overlap.

Algorithm 8 Extended Partial Communication Computation Overlap Algorithm for Full GNN layer

procedure AGGREGATE(V_{local}, H_k^{l-1}, $SendList$, $RecvList$, ϕ^ℓ)
  for each $p_k \in RecvList$ **do**
    $MPI\_Irecv(recvBuffer_k, p_k)$
  end for
  for each $p_k \in SendList$ **do**
    Fill $sendBuffer_k$
    $MPI\_Isend(sendBuffer_k, p_k)$
  end for
  $X \leftarrow AggregateLocalLocal(V_{local}, H_k^{l-1})$
  $H^l \leftarrow \phi^\ell(X)$
  Wait on Irecv calls
  $X' \leftarrow AggregateLocalNonlocal(V_{local}, recvBuffer_K)$
  $H^l \leftarrow H^l + \phi^\ell(X')$
  return H^l
end procedure

Chapter 5

Experimental Setup and Datasets

In the experimental settings of this thesis, two distinct high-performance computing (HPC) cluster architectures were utilized. The first cluster features Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz with 28 cores and 128GB of RAM per node, interconnected via a 56 Gbit/s Infiniband network. The second cluster is built around 64-core 3rd-generation AMD EPYC™ CPUs, with 256 GB to 1024 GB memory, interconnected via a 200 Gbit/s Infiniband Cray Slingshot network.

We developed a 1D vertex-parallel GNN implementation using the C programming language coupled with MPI for interprocess communication. This implementation is used in our experimental investigations, offering precise control over computational behavior and optimization. Existing frameworks often have extra optimizations and bottlenecks that can affect the performance of proposed methods. Also, a more lightweight implementation offers flexibility in implementing and testing the proposed methods. For showing the correction of the framework, PyG (PyTorch Geometric) [26] is used as a baseline to compare the correction of our model. Since the methods proposed do not affect the computation related to the model, our framework correction is tested by initializing the same model with the same weights and parameters in PyG. As expected, both models converged to the same weights in full-graph training, showing the correctness of our implementation.

This custom implementation includes a flexible framework that allows for the training and inference of deep neural networks with various settings and hyperparameters. This flexibility is crucial for testing different neural network architectures. Our framework can handle multiple GCN and activation layers such as relu, tanh, and sigmoid. The choice of initial weights is tailored to each activation function, using Xavier/Glorot [27] initialization for tanh and sigmoid, and He Uniform [28] initialization for ReLU to optimize performance. Additionally, the framework supports Cross-entropy loss and the ADAM optimizer, enabling robust and effective optimization. Developing our own implementation was essential for us to precisely configure and modify various components according to our specific research needs, thus allowing us to explore in depth how different configurations impact the performance of GNNs in our experiments.

For the graph partitioning process, we primarily use METIS [11], providing robust performance and reliability. However, note that hypergraph partitioning tools like PaToH [10] and hMETIS [16] could potentially offer better solutions for this specific problem due to their advanced encoding capabilities. To further enhance the efficiency of the partitioning step, we consider parallel graph partitioners such as ParMETIS [17]. These tools are designed to significantly speed up the partitioning process, which is crucial as an effective implementation must quickly amortize the time spent on partitioning to accelerate the overall training phase. By optimizing the partitioning step, we aim to significantly improve the efficiency and speed of the training process, ensuring that the computational resources are utilized most effectively.

5.1 Datasets

For conducting experiments, we have used a selection of widely used large-scale graph datasets to perform node classification. Every dataset consists of a graph, vertex features, and vertex labels. The graph is given as a sparse adjacency matrix in CSR format. Vertex features and labels are represented as tall and skinny matrices.

Table 5.1: Properties of the GNN datasets.

Dataset	Number of		Vertex Degree			Number of	
	Vertices	Edges	Avg.	Max.	Density	Features	Classes
Yelp	716847	13954819	19.4	4886	2×10^{-5}	300	100
Reddit	232965	114848857	492.0	21658	2×10^{-3}	602	41
Amazon	1569960	264339468	168.3	75134	1×10^{-4}	200	107
Flickr	89250	989006	11.0	5426	1×10^{-4}	500	7
OGBN-Products	2449029	126167053	51.5	17482	2×10^{-5}	100	47
OGBN-Arxiv	169343	1335586	7.88	13156	4×10^{-5}	128	40

*The number of edges for all graphs is counted as if they are directed.

Graphs in these datasets possess different properties and characteristics to provide a variety in the experiments. The properties of these graphs are given in Table 5.1 for each dataset; these statistics show the general structure of the graph. These graph properties also reflect the properties of the sparse adjacency matrix used in the SpMM operation. The number of vertices $|V|$, edges $|E|$, and density and vertex degree information are provided. Density is defined as a fraction of existing edges to the number of possible edges such that $\frac{|E|}{|V|^2}$ The adjacency matrix A is a $|V| \times |V|$ matrix with $|E|$ non-zero elements for a given graph.

Information about vertex features and labels of datasets are provided in 5.1. Every vertex has an input feature vector of $1 \times f$, where f is the number of the input features. Vertex features are presented as a $|V| \times f$ dense matrix. Also, every vertex is associated with a single or multiple label(s) to be used as a ground truth during training and inference. Labels are also represented as a matrix with size $|V| \times k$, where k is the number of distinct classes presented in the node classification problem.

All the datasets are mainly used for node classification tasks and do not have extra complications in terms of computation. We perform the following tasks in the training of the following datasets: 1. categorizing types of businesses based on customer reviewers and friendship (Yelp) [29], 2. predicting communities of online posts based on user comments (Reddit) [29], 3. classifying product categories based on buyer reviewers and interactions (Amazon) [29], 4. categorizing types of images based on the descriptions and common properties of online images (Flickr) [29], 5. predicting the category of a product in a multi-class classification setup based on co-purchasing (Ogbn-Products) [30], 6. categorizing academic papers based on their citations with each other (Ogbn-Arxiv) [31].

5.2 Setup

We use a GCN model to evaluate throughput during the training process. The model consists of two GCN layers separated by a ReLU activation layer. The outputs from these layers are then transformed using a softmax activation function. For the normalization of the adjacency matrix in an undirected graph, we use the formula $D^{-\frac{1}{2}}(A+I)D^{-\frac{1}{2}}$, which exploits the graph's spectral properties. Parameters that do not influence runtime, such as the learning rate, are maintained consistent with the original GCN model specifications. It's important to note that dropout layers and L2 regularization are omitted, as our custom framework does not support these features. Only one GCN layer is measured in experiments where we test the partitioning scheme. This is because our proposed two-constraint partitioning method is developed to model the computations of a single layer.

Test, train, and validation splits exist for the given datasets. These splits are generated by using external information such as sales rank and time. For example, the `ogbn-arxiv` dataset uses papers that were published until 2017 as training, validates those published in 2018, and tests those published since 2019. Another example is `ogbn-products` datasets where sales rank is used, and the top %8 of the products are used for training, the next %2 percent is used for validation, and the rest for testing. In the experiments where model accuracy is irrelevant, graphs are used without splitting them. Full graph training is used to push the limits on algorithm performance and implementation limits.

In the experiments, three different partitions were used for each dataset using a random seed with METIS to ensure a robust comparison. Each experimental run was initiated with 10 epochs of warmup iterations, followed by 100 epochs of training. Multiple experiments were conducted on each dataset, and the minimum runtime was recorded for each. This approach was used to reduce the effect of potential network congestion caused by other programs in the runtime environment, which could result in slowdowns affecting both the proposed methods and the baseline algorithms.

Chapter 6

Results

This chapter comprehensively analyzes the experimental results derived from implementing various partitioning schemes and overlapping strategies in GNN training. This chapter systematically evaluates the effectiveness of different methods, including the proposed two-constraint partitioning method, compared to traditional single-constraint methods across various datasets and processor counts. We delve into the performance metrics, such as computational load balance, communication volume, and overall training speed, to assess the impact of these strategies on the efficiency and scalability of GNNs. Detailed discussions on how these methods perform under different conditions, mainly focusing on the SpMM and GeMM operations, will provide insights into their practical implications and potential benefits in real-world applications. Through tables and figures, this chapter illustrates the quantitative improvements these advanced techniques offer, underscoring their significance in optimizing GNN training processes.

6.1 Evaluation of Partitioning Schemes and Load Balancing

Table 6.1 presents the results of different partitioning schemes regarding computational imbalance metrics, as discussed in Section 4.2. This table displays the computational load balance during the forward and backward propagation steps for three different methods: the proposed two-constraint method and the single constraint methods (degree and unit weight). For the degree weight and unit weight methods, the balance based on their respective constraints is also provided. Each method is evaluated across different partition sizes for comprehensive analysis.

In these experiments, we compared our proposed two-constraint method against two one-constraint methods: one using vertex degree as weights and another using unit weights for the partitioning. Our method is designed to balance the computational load during both forward and backward propagation steps according to our formulation. The vertex degree method focuses on balancing the SpMM operation, while the unit weight approach is tailored to balance the GeMM operation. As the table indicates, our method successfully achieves a balanced computation across most cases with respect to our formulation. In contrast, while the other methods manage to provide a good balance according to their targeted metrics, they do not accurately represent the real computations as observed in the results.

Table 6.1: Comparison of computational load-imbalance ratios for different partitioning schemes for different partition sizes across datasets.

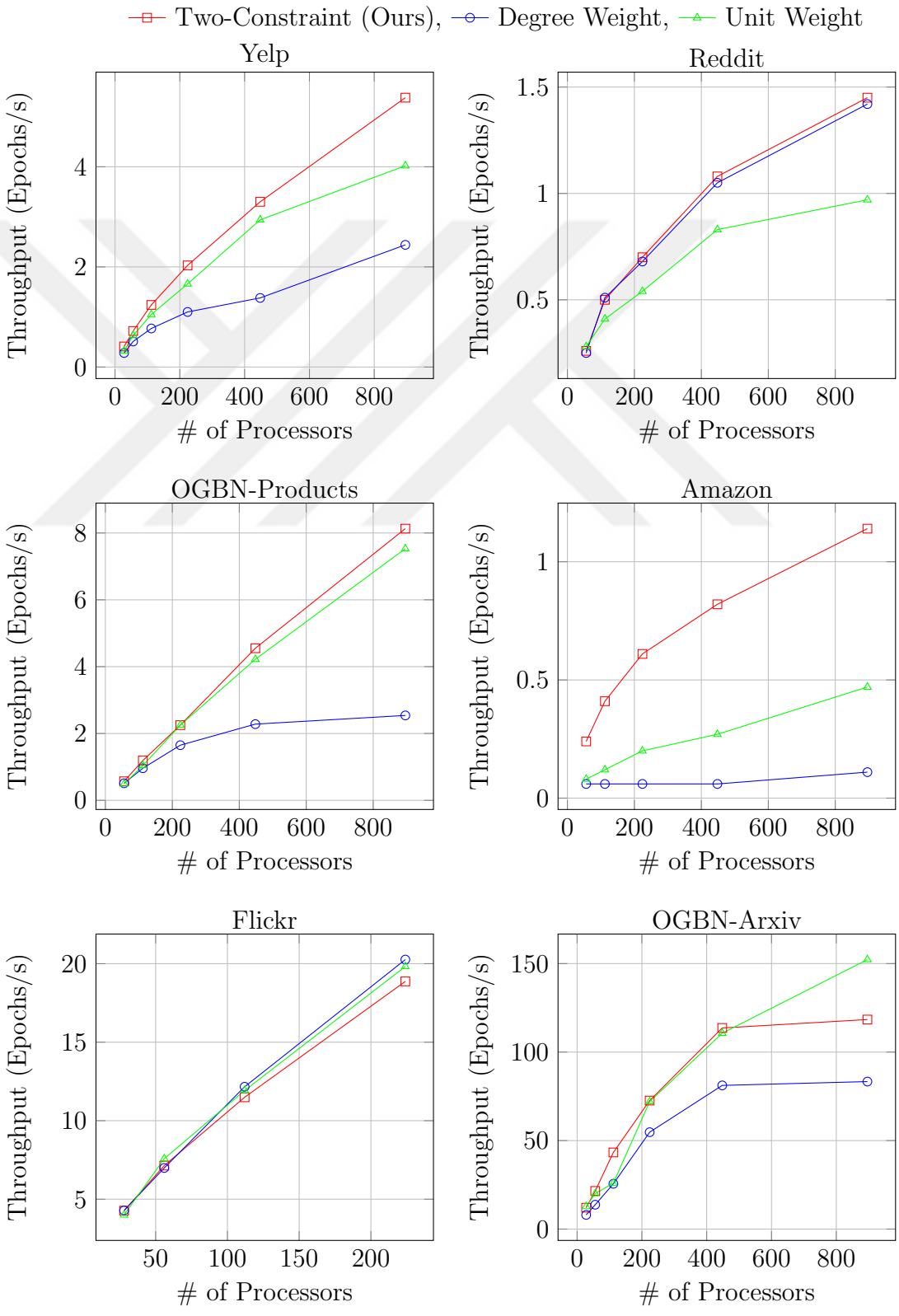
Dataset	K	Two-Constraint		One - Constraint					
		Our Method		Degree Weight			Unit Weight		
		Forward	Backward	Degree	Forward	Backward	Unit	Forward	Backward
Yelp	28	1.03	1.03	1.03	2.06	2.22	1.03	1.42	1.10
	56	1.03	1.03	1.03	1.95	2.09	1.03	1.84	1.19
	112	1.03	1.03	1.03	2.69	2.95	1.03	2.29	1.27
	224	1.03	1.03	1.03	4.17	4.67	1.03	2.88	1.39
	448	1.03	1.03	1.03	7.15	8.12	1.03	4.73	1.74
	896	1.03	1.03	1.03	7.84	8.92	1.03	7.12	2.20
Reddit	56	1.03	1.03	1.03	2.51	1.50	1.03	1.21	2.73
	112	1.03	1.03	1.03	4.21	2.03	1.03	1.22	2.78
	224	1.04	1.04	1.03	7.60	3.10	1.03	1.24	2.94
	448	1.04	1.04	1.03	6.81	2.85	1.03	1.27	3.28
	896	1.04	1.04	1.03	6.23	2.66	1.03	1.34	3.86
Ogbn-Products	56	1.03	1.03	1.03	1.58	1.95	1.03	1.45	1.18
	112	1.03	1.03	1.03	1.96	2.57	1.03	1.53	1.23
	224	1.03	1.03	1.03	2.9	4.18	1.03	1.78	1.31
	448	1.03	1.03	1.03	4.45	6.73	1.03	1.84	1.31
	896	1.03	1.03	1.03	8.52	13.51	1.03	2.29	1.52
Amazon	56	1.12	1.14	1.03	7.74	2.05	1.03	4.23	15.55
	112	1.19	1.26	1.03	13.47	2.93	1.03	7.66	30.82
	224	1.18	1.26	1.03	23.63	4.48	1.03	14.26	60.49
	448	1.24	1.28	1.03	32.46	5.83	1.03	27.66	120.41
	896	1.29	1.32	1.03	37.78	6.64	1.03	33.1	144.82
Flickr	28	1.03	1.03	1.03	1.06	1.1	1.03	1.16	1.03
	56	1.03	1.03	1.03	1.04	1.1	1.03	1.27	1.03
	112	1.03	1.03	1.03	1.07	1.17	1.03	1.52	1.04
	224	1.03	1.03	1.23	1.16	1.36	1.03	1.87	1.04
Ogbn-Arxiv	28	1.03	1.03	1.03	1.98	2.25	1.03	1.31	1.09
	56	1.03	1.03	1.03	2.07	2.38	1.03	1.45	1.12
	112	1.03	1.03	1.03	2.19	2.54	1.03	1.72	1.17
	224	1.03	1.03	1.19	2.2	2.54	1.03	1.96	1.22
	448	1.03	1.03	2.37	2.34	2.73	1.03	2.5	1.33
	896	1.55	1.04	4.75	2.43	2.86	1.03	2.87	1.4

Figure 6.1 shows the experimental results for demonstrating the performance of our two-constraint formulation. The experimental results show that our two-constraint formulation provides a better computational model for a GCN layer. In some cases where values of feature size f and hidden parameters k are effectively large or small, other one-constraint formulations can provide a compatible performance. For larger values, vertex weights converge to proportionally close values, thus reducing the effect of the vertex degree in the computations. Unlike the previous case, very small values remain unimportant when compared to the degree of a vertex. Important to note that using a two-constraint formulation significantly reduces the solution space of the partitioning algorithm, thus potentially providing a better performance in the communication step.

Our two-constraint method performs better than other methods in four out of six datasets, especially in the larger ones. It shows particularly strong results in the Yelp and Amazon datasets, where it enhances both scalability and overall performance. For the `Reddit` and `Ogbn-products` datasets, our method proves to be more versatile compared to others. The figures indicate that the SpMM operation is more critical for the Reddit dataset, while the GeMM operation becomes key in the Ogbn-products dataset. Overall, our two-constraint methods perform the best in these conditions, showing that they are more flexible and effective across different scenarios.

For several reasons, our two-constraint method doesn't perform as well with smaller datasets. First, in smaller datasets, the time spent on communication becomes more significant than computation time. Here, the one-constraint method's wider solution space in partitioning is an advantage over the two-constraint approach. Secondly, in the case of the `Flickr` dataset, the specific values of $f = 500$ and $k = 7$ are quite extreme. These settings allow the degree weight to effectively balance the load during the forward step and the unit weight to do the same during the backward step. As a result, besides being more efficient in communication, these models also manage to maintain a decent balance in computational load.

Figure 6.1: Comparison of throughput performance across different datasets using different schemes in partitioning.



This section has provided a detailed examination of the efficacy of different partitioning strategies in balancing computational loads for Graph Neural Networks, particularly highlighting the advantages of the two-constraint method over traditional one-constraint methods in various dataset sizes and types. The evidence from the experiments suggests that the two-constraint method not only offers superior performance in larger datasets, such as those from `Yelp` and `Amazon` but also demonstrates greater versatility across diverse scenarios, such as in the `Reddit` and `Ogbn-products` datasets where specific operations like SpMM and GeMM are critical. However, the two-constraint approach falls back in smaller datasets where the cost of communication are predominant compared to computations. This highlights the importance of carefully choosing partitioning strategies that fit well with each dataset's specific features and needs. The lessons learned from these experiments emphasize how crucial it is to select the right partitioning methods to improve both the efficiency and effectiveness of the models. Doing so can make Graph Neural Networks more scalable and useful in a variety of real-world situations.

6.2 Performance Analysis of Overlapping Techniques

Table 6.2 provides the communication statistics and runtime metrics across various datasets and partition sizes. This table details the communication volume metrics, including message volume and message size for both send and receive calls, where the average values for send and receive volumes are identical due to a corresponding receive call for every send. Message counts are similarly aligned. Additionally, the table presents runtime improvements for three methods: Full-Overlap (FO), Partial-Overlap (PO), and Extended Partial-Overlap (EPO). Table 6.3 shows the average of these parallel runtime improvements. These improvements are expressed as the execution time ratio of each method relative to the no-overlap method, with smaller values indicating better performance. As seen in the tables, the overlapping methods provide faster training on average.

Table 6.2: Communication metrics and parallel runtime metrics across various datasets and partition sizes.

Dataset	K	Message Volume			Mssg. Count		Improvement		
		Send Max	Recv Max	Avg.	Max	Avg	FO	PO	EPO
Yelp	28	92270	105227	71901	27	27	1.00	1.00	1.02
	56	67401	90681	49723	55	54	0.97	0.96	1.03
	112	47627	66572	31653	111	111	0.87	0.86	0.92
	224	34849	50380	19820	222	221	0.77	0.78	0.83
	448	21924	30430	12028	446	434	0.85	0.88	0.89
	896	15251	25033	7035	868	746	0.97	1.03	1.01
Reddit	56	104459	125554	70271	55	55	1.00	1.03	0.95
	112	86794	98879	53861	111	111	0.86	0.85	0.86
	224	69323	84915	39892	223	223	0.61	0.65	0.65
	448	54251	66967	30255	447	441	0.72	0.80	0.80
	896	40092	44850	22062	895	836	0.98	1.05	1.05
Ogbn-Products	56	150541	157073	89102	55	55	0.99	1.03	0.95
	112	107540	104918	58702	111	111	0.94	0.85	0.86
	224	70038	76484	38237	222	213	0.89	0.65	0.65
	448	51851	61785	25157	442	374	0.83	0.80	0.80
	896	35317	56844	15511	827	531	0.85	1.05	1.05
Amazon	56	242353	366963	153105	55	55	0.93	0.98	1.03
	112	189107	281927	114147	111	111	0.86	0.83	0.86
	224	145104	231486	86016	223	222	0.71	0.75	0.75
	448	115010	197489	63458	444	432	0.75	0.79	0.80
	896	88833	158743	46139	875	806	0.94	0.97	0.99
Flickr	28	14742	15322	12105	27	27	1.00	0.98	0.95
	56	8610	9925	7061	55	55	1.00	0.99	1.03
	112	4897	6809	3988	111	111	0.79	0.79	0.81
	224	2774	5819	2204	223	220	0.70	0.76	0.77
Ogbn-Arxiv	28	17912	23379	11040	27	27	0.97	0.99	1.12
	56	12732	18021	7252	55	55	0.93	0.96	1.07
	112	8568	15815	4503	111	105	0.95	0.97	1.05
	224	5633	17095	2752	215	177	0.88	0.95	0.92
	448	3702	13595	1668	398	250	0.87	0.88	0.88
	896	3514	13105	891	711	234	0.53	0.55	0.54

Figure 6.2: Communication and computation times of aggregation (SpMM) step for various datasets and partition sizes.

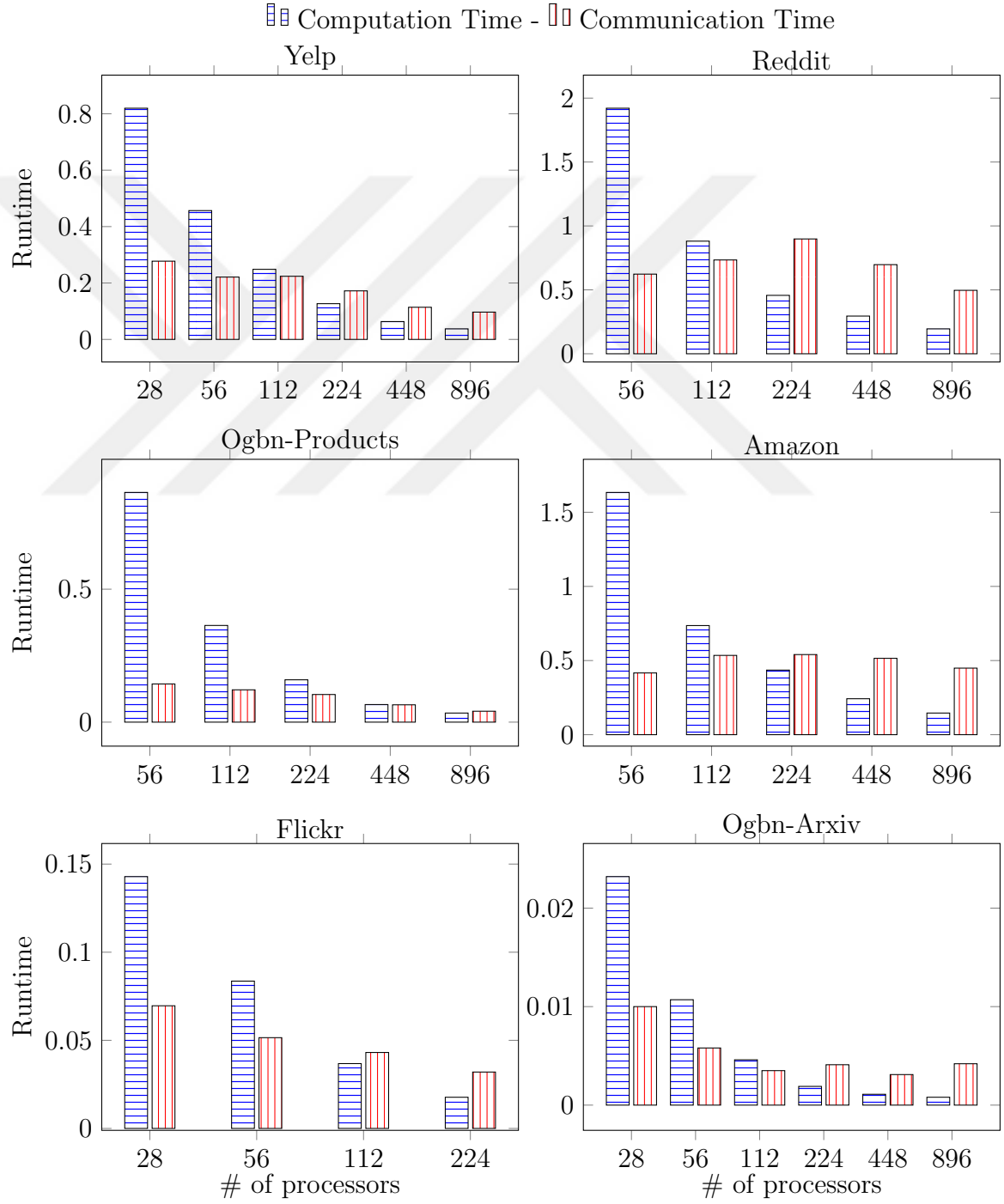


Table 6.3: Average parallel runtime improvement of all datasets over different processor counts.

K	Avg. Improvement		
	FO	PO	EPO
28	1.00	0.99	0.98
56	0.97	0.99	1.01
112	0.88	0.86	0.89
224	0.76	0.76	0.76
448	0.80	0.83	0.83
896	0.85	0.93	0.93

Communication volume decreases with the increasing partition size, as expected. Meanwhile, the message count increases with the partition size. In these terms, message volume directly affects the communication time regarding bandwidth cost, and message count affects the latency costs. Due to the high volumes of communication in GNN training, it is mostly bounded by bandwidth costs; however, additional overheads related to the message count exist in our communication and computation overlapping schemes. Non-blocking communication costs in the MPI collectives have additional overheads compared to normal counterparts [24], which is directly related with message count.

Figure 6.2.2 illustrates the distribution of time spent on communication and computation during the aggregation (SpMM) step. As the number of processors increases, computation time decreases. However, communication time does not consistently scale with increased parallelization, likely due to the scale-free nature of the datasets. The strategy of overlapping computation with communication aims to obscure computation time within the communication time, thereby ensuring that the total execution time is limited by either the communication or computation time. It is important to note that these statistics do not account for the additional costs associated with non-blocking collectives. Due to their nature, accurately measuring these costs is challenging, as they tend to be obscured within the communication time when utilized.

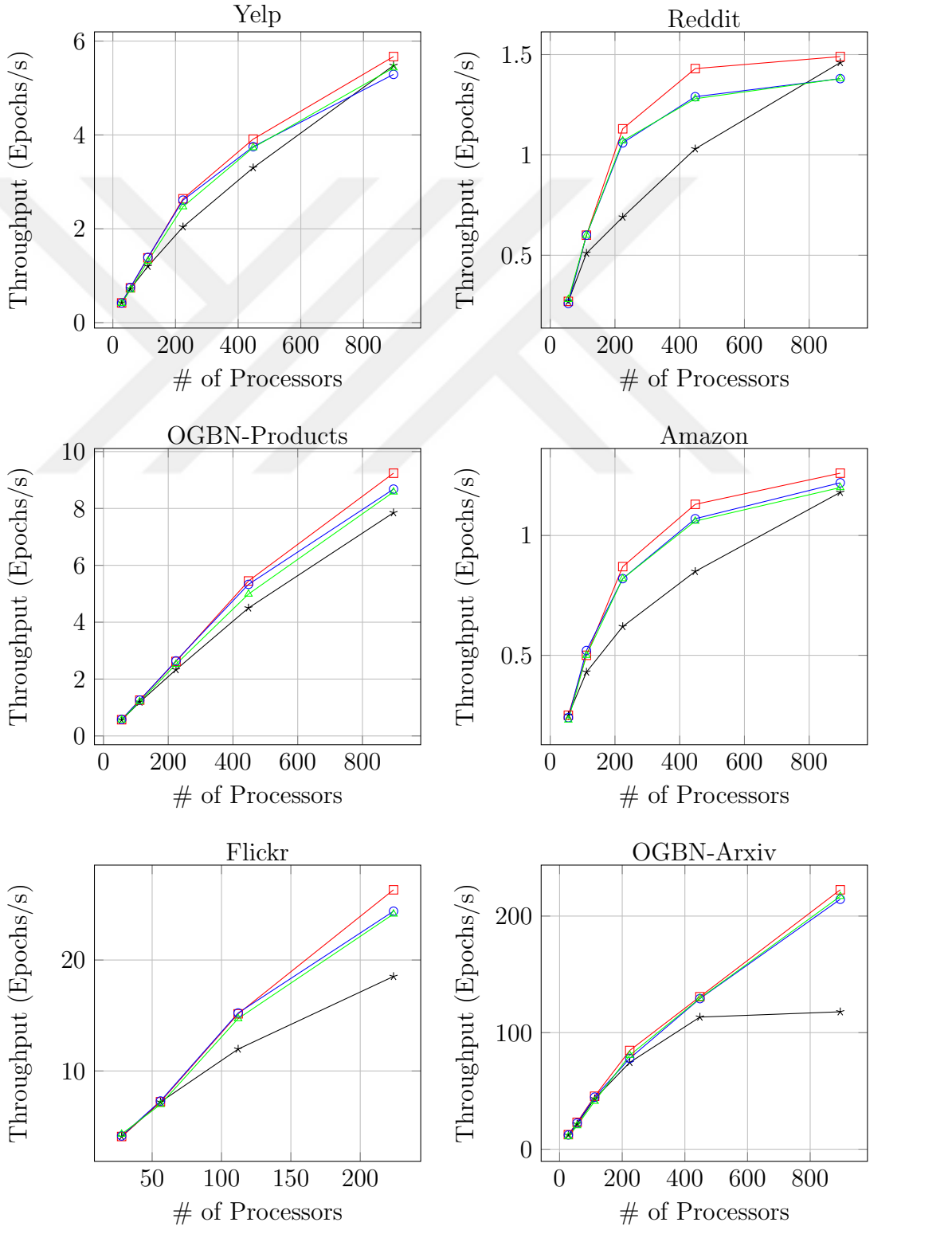
Figure 6.3 displays runtime statistics for four methods discussed in Section

4.3: No Overlap (Baseline), Full Overlap, Partial Overlap, and Extended Partial Overlap. These methods are evaluated for various partition sizes, with their throughput values illustrated in the figure. The Full Overlap method consistently delivers superior performance, particularly notable when communication and computation times are comparable. While the No Overlap, Partial Overlap, and Extended Partial Overlap methods also improve with increased parallelization, they do not match the efficiency gains observed with the Full Overlap method. Despite this, all methods demonstrate scalability, indicating that each benefit from enhanced parallel processing capabilities, the amount of overlapping, and the costs of non-blocking collectives affect their performance. This highlights the Full Overlap method's effectiveness in optimizing runtime across diverse computational settings.

Theoretically, the runtime increase stops when insufficient computation load is left to hide in the computation time. In practice, the additional overheads caused by the non-blocking collectives bring additional computation costs, which become a significant cost in computation. These overheads differ from latency and bandwidth costs that can be treated as a computation load [24]. Our overlapping methods are also capable of hiding this type of computational cost, but these additional loads can completely use the excess time in communication. Performance deterioration in the Amazon, Yelp, and Reddit datasets is caused by these overheads due to large message counts and communication volume. Table 6.2 shows that these are the top three datasets regarding maximum and average message count.

As expected, the Full Overlap method provides better performance than partial methods, meaning that receiving messages asynchronously provides more room for computation. On the other hand, the Extended Partial Overlap method performs similarly to the Partial Overlap method. This behavior shows that the partial method reaches its full potential by only using the computations in the aggregation (SpMM) step, and additional computations in the update(GeMM) operation stay outside of communication time.

Figure 6.3: Comparison of throughput performance of overlapping communication and computation across different datasets for various processor counts.



Chapter 7

Conclusion and Future Work

This thesis has delved into the challenges and solutions of optimizing Graph Neural Network (GNN) training on high-performance computing environments. Central to this exploration was the development and evaluation of an innovative two-constraint partitioning method designed to manage computational demands across distributed systems efficiently. This method significantly improved the performance and scalability of GNNs, particularly when handling large datasets where the traditional one-constraint methods were less effective.

A key aspect of our approach was the strategic overlap of communication and computation phases, which was critical in enhancing the throughput and efficiency of GNN training. By implementing overlapping methods we effectively minimized the idle times typically associated with sequential processing. These overlapping schemes allowed simultaneous data transmission and local computation, thereby reducing the overall execution time and enhancing resource utilization.

Our experiments demonstrated that intelligent partitioning combined with overlapping communication and computation could significantly mitigate the performance bottlenecks traditionally seen in parallel GNN training. The two-constraint method, in particular, proved sufficient at balancing computational

loads across nodes during both the forward and backward propagation phases, optimizing the use of network bandwidth and processing power.

Bibliography

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[2] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[3] M. Besta and T. Hoefler, "Parallel and distributed graph neural networks: An in-depth concurrency analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[4] S. Acer, O. Selvitopi, and C. Aykanat, "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems," *Parallel Computing*, vol. 59, pp. 71–96, 2016.

[5] N. Merkel, D. Stoll, R. Mayer, and H.-A. Jacobsen, "An experimental comparison of partitioning strategies for distributed graph neural network training," *arXiv preprint arXiv:2308.15602*, 2023.

[6] O. Selvitopi, B. Brock, I. Nisa, A. Tripathy, K. Yelick, and A. Buluç, "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 431–442, 2021.

[7] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning

on graphs," *Advances in neural information processing systems*, vol. 33, pp. 22118–22133, 2020.

[8] V. P. Dwivedi, C. K. Joshi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," *Journal of Machine Learning Research*, vol. 24, no. 43, pp. 1–48, 2023.

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

[10] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on parallel and distributed systems*, vol. 10, no. 7, pp. 673–693, 1999.

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

[12] A. Tripathy, K. Yelick, and A. Buluç, "Reducing communication in graph neural network training," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, IEEE, 2020.

[13] Z. Zhang, J. Leng, S. Lu, Y. Miao, Y. Diao, M. Guo, C. Li, and Y. Zhu, "Zipper: Exploiting tile-and operator-level parallelism for general and scalable graph neural network acceleration," *arXiv preprint arXiv:2107.08709*, 2021.

[14] S. Mondal, S. D. Manasi, K. Kunal, and S. S. Sapatnekar, "Gnnie: Gnn inference engine with load-balancing and graph-specific caching," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 565–570, 2022.

[15] Q. Su, M. Wang, D. Zheng, and Z. Zhang, "Adaptive load balancing for parallel gnn training," 2021.

[16] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in vlsi domain," in *Proceedings of the 34th annual Design Automation Conference*, pp. 526–529, 1997.

[17] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of parallel and distributed computing*, vol. 48, no. 1, pp. 71–95, 1998.

[18] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. Karypis, "Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4582–4591, 2022.

[19] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2625–2638, 2017.

[20] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: Distributed graph neural network training for billion-scale graphs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 36–44, IEEE, 2020.

[21] T. Hoefler and A. Lumsdaine, "Overlapping communication and computation with high level communication routines," in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 572–577, IEEE, 2008.

[22] T. Hoefler, P. Gottschling, and A. Lumsdaine, "Leveraging non-blocking collective communication in high-performance applications," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pp. 113–115, 2008.

[23] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pp. 125–es, 2006.

[24] T. Hoefler and A. Lumsdaine, "Optimizing non-blocking collective operations for infiniband," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, IEEE, 2008.

[25] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing*, vol. 110. Benjamin/Cummings Redwood City, CA, 1994.

[26] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[27] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, JMLR Workshop and Conference Proceedings, 2010.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[29] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," *arXiv preprint arXiv:1907.04931*, 2019.

[30] K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma, "The extreme classification repository: Multi-label datasets and code," 2016.

[31] K. Wang, Z. Shen, C. Huang, C.-H. Wu, Y. Dong, and A. Kanakia, "Microsoft academic graph: When experts are not enough," *Quantitative Science Studies*, vol. 1, no. 1, pp. 396–413, 2020.