Utrecht University

The Think Like A Vertex approach in a parallel graph neural network

Master's Thesis

Author:	Katharina Klein
Student Number:	6977146
Field of Study:	Mathematical Sciences
Supervisor:	Prof. dr. R.H. Bisseling
Second Reader:	Dr. Sjoerd Dirksen

Abstract

Neural networks have been on the rise in the past years and are being applied in many different areas. Since in many applications the underlying data naturally has a graph-like structure, a type of network called graph neural network (GNN) has been proposed in 2009. This network type is able to capture dependencies within the graph by updating the state of each vertex based on the states of its neighbors, a process which has more recently been introduced as message passing. As graphs in practical applications are becoming considerably large, training and applying a GNN is computationally expensive and time-consuming and parallelization is therefore expected to be worthwhile. In the context of implementation, the Think Like A Vertex (TLAV) framework is a reasonable approach as it fits nicely with the BSP model for parallel programming. First introduced in 2010, the TLAV approach focuses on alternating local computations in a vertex and exchanging information with neighboring vertices. In this project, we combine TLAV with the BSP model in order to design a parallel algorithm implementing the message passing process of a GNN. We test this parallelization with an update function based on the Gated Recurrent Unit (GRU). Our experiments indicate that a reasonable speedup is obtained with respect to a sequential implementation, showing that the running time can be significantly reduced when applied to graphs of different sizes and with different properties.

Contents

Notation

Notation	Meaning
$G = (V, E)$	Graph on vertex set V and edge set E
$n = \|V\|$, $m = \|E\|$	Number of vertices and edges, respectively
$N(v)$	Set of neighbors of a vertex v
$E(v)$	Set of edges to which a vertex v is attached
$\mathbf{x}_v$, $\mathbf{x}_{(v,u)}$	Feature vector of a vertex v or edge (v,u)
$\mathbf{h}_v$, $\mathbf{h}_v^t$, $\mathbf{h}_{(v,u)}$, $\mathbf{h}_{(v,u)}^t$	Hidden state (at time t) of a vertex v or edge (v,u)
M_t, U_t, R	Message, update and readout functions in the diffusion process
t	Current time step in the network
T	Final time step, i.e. number of iterations in the diffusion process
w, $\mathbf{w}$, $\mathbf{W}$	Learned network parameter, parameter vector or parameter matrix
k	Current iteration in the gradient descent algorithm
d	Dimension of the vertex states
$\tilde{d}$	Dimension of the output layer
r, g, l	BSP parameters
p	Number of processors used in the parallel algorithm
s, $P(s)$	Processor identifiers
$\phi : V \to \{0, \ldots, p-1\}$	Mapping of vertices to processors; $\phi(v)$ is the owner of vertex v
$\psi : V \to \mathscr{P}(\{0, \ldots, p-1\})$	Mapping of vertices to processors; $\psi(v)$ is the set of processors requesting v

1 Introduction

Artificial neural networks are a popular area of interest both in theoretical research and in industry. Since first being introduced, different neural network models have been developed and successfully found their way into in a variety of domains. Nowadays, they are being applied, for instance, for classification purposes, in text and speech recognition, or on different prediction tasks. Many commonly used network models require that the data to be processed is represented in a simple form such as a sequence of values or a feature vector. For applications involving complex data, this means that a pre-processing of the relevant input data is necessary, with the result that structural dependencies are typically not (fully) accounted for. Graph neural networks (GNNs) have been developed in order to capture such dependencies [1].

Often described as a generalization of convolutional neural networks, GNNs are particularly suited for problems involving data which can be represented as a graph. Currently, they are being applied in a range of areas, from the analysis of social networks, where relationships and different kinds of interactions can be modeled as edges between vertices, to recommender systems for products or online content [2], and the detection of fake news through spreading patterns [3]. Further examples of applications are traffic prediction [4] and problems in chemistry and physics, such as drug development [5] or the analysis of particle systems [6]. A frequent issue with GNNs, as with most neural network types, is that they oftentimes require extensive training data and are therefore very expensive to train. This problem is aggravated by the fact that many graphs based on real-world networks are becoming increasingly large and memory issues may arise. Such challenges motivate parallelization of GNN models.

Parallel computing, in turn, is an extensive area of research which can nowadays be encountered virtually everywhere. Whether one strives for increased performance or the processing of larger amounts of data: even with optimized algorithm design, traditional sequential computing quickly reaches its limits and parallelism comes into play. The central goal of parallel computing, then, is to improve performance through increased efficiency.

In this thesis, we develop a parallelization approach that will allow for implementing a parallel GNN. The algorithm is based on a message passing model resembling a diffusion process within a graph and is combined with a Think Like A Vertex (TLAV) approach of adopting a local view in the program design. It can be modified to fit the needs of specific applications and combined with different training methods for the purpose of developing a functional neural network. The rest of this work is structured as follows. In Section 2, relevant aspects of the three different concepts we combine in this project are introduced. Section 3 outlines the sequential and parallel algorithms as well as some further implementation details. We provide experimental results for our algorithm in Section 4 and finish with concluding remarks and directions for further research in Section 5.

2 Preliminaries

2.1 Graph Neural Networks

The first proposed neural network model processing graph-structured data, introduced by Scarselli et al. in 2009 [7], takes as input a graph $G = (V, E)$, where V is the set of vertices of the graph and E is the set of edges connecting vertices, and both vertices and edges can be associated with additional features. In their paper, the authors emphasize that a variety of graph types and in particular both directed and undirected graphs can be processed using their model. For the purpose of this project, however, we will mostly focus on the latter.

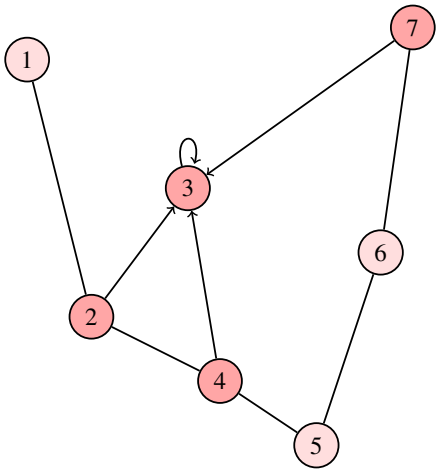


Figure 2.1: An undirected input graph with seven vertices. Directed edges and intensified color indicate information flow in the state update of vertex 3.

The model works on G by assuming that each vertex $v \in V$ has a fixed state $\mathbf{h}_v \in \mathbb{R}^d$ which depends on the features $\mathbf{x}_v$ of the vertex itself and the features $\mathbf{x}_{(v,u)}$ of the edges $(v,u) \in E(v)$ adjacent to v, as well as the states $\mathbf{h}_u$ and features $\mathbf{x}_u$ of neighbors $u \in N(v)$ of v. This interpretation of the information flow is illustrated in Figure 2.1 for one vertex of an undirected graph. Concretely, $\mathbf{h}_v$ is computed via a transition function

$$\mathbf{h}_v = f_{\mathbf{w}}(\mathbf{x}_v, \mathbf{h}_v, \mathbf{x}_{E(v)}, \mathbf{h}_{N(v)}, \mathbf{x}_{N(v)})$$

parameterized by $\mathbf{w}$. Here, $\mathbf{x}_{E(v)}, \mathbf{h}_{N(v)}, \mathbf{x}_{N(v)}$ concisely represent the features and states of neighboring edges and vertices.

The crucial assumption of the model is that the transition function is a contraction mapping and converges to its unique fixed point, so that the state $\mathbf{h}_v$ can for every vertex v be computed iteratively as

$$\mathbf{h}_v^{t+1} = f_{\mathbf{w}}(\mathbf{x}_v, \mathbf{h}_v^t, \mathbf{x}_{E(v)}, \mathbf{h}_{N(v)}^t, \mathbf{x}_{N(v)})$$

until convergence is attained. Consequently, the model returns an output $\mathbf{o}_v = g_{\mathbf{w}}(\mathbf{h}_v, \mathbf{x}_v)$ for every vertex $v \in V$. For graph-focused tasks, an additional node can be added to the graph for providing the graph output.

2.1.1 Neural Message Passing

Though the GNN model proposed by Scarselli et al. [7] is suitable for different types of graphs and allows for both linear and nonlinear transition and output functions, it is restrictive in that it relies on

contraction mappings and thus the choices for the diffusion process are limited. Furthermore, it differs from other types of neural networks as it is dynamic, in the sense that the number of time steps in the diffusion is not fixed. Different variants of GNNs that do not have these limitations have been developed in the past years. In 2017, Gilmer et al. [8] presented a model based on neural message passing which generalizes many of these variants. The key features of this Message Passing Neural Network (MPNN) model are a message passing phase and a readout phase in the diffusion process.

The message passing phase is executed for a fixed number T of time steps; in particular, T is independent of when (or if at all) the vertex states converge to a fixed point. At each time step, the network performs an initial aggregation $\mathbf{m}_v^{t+1}$ of *messages* from the neighborhood of every node $v \in V$. These messages are computed based on the current state $\mathbf{h}_v^t$ of the node itself and the current states $\mathbf{h}_u^t$ of its neighbors $u \in N(v)$ as well as on the features $\mathbf{x}_{(v,u)}$ of the corresponding edges. The vertex state is then updated as

$$\mathbf{m}_v^{t+1} = \sum_{u \in N(v)} M_t(\mathbf{h}_v^t, \mathbf{h}_u^t, \mathbf{x}_{(v,u)})$$
$$\mathbf{h}_v^{t+1} = U_t(\mathbf{h}_v^t, \mathbf{m}_v^{t+1})$$

Here, M_t and U_t are message and update functions, respectively, which depend on parameters $\mathbf{w}$ that are to be learned through training.

The readout phase produces the output of the network. For graph-focused applications, this would be a feature vector

$$\mathbf{o} = R(\{\mathbf{h}_v^T : v \in V\})$$

with R being some readout function, possibly parameterized. The function R is preferably invariant to the order of the vertices, such that the network is invariant to graph isomorphism. The model can also be adapted to produce an output $\mathbf{o}_v = R(\mathbf{h}_v^T)$ for each vertex individually, as is required in vertex-focused applications.

Different instances of the MPNN, in particular several GNN models suggested in the literature, can be obtained from the general framework by choosing different message, update and readout functions M_t, U_t and R.

2.1.2 Relation with Classical Neural Networks

In essence, a GNN consists of a diffusion process yielding the graph or node outputs. This process resembles the propagation of information through a "classical" neural network with T layers, as Figure 2.2 shows. The crucial difference with other types of neural networks is that the connections between network nodes are part of the input to the GNN. Whereas in a simple neural network (fully-connected or not), the network architecture defines which connections exist and every instance of input data is processed with the same set of connections, the layers of an unrolled GNN will differ based on the edges in the input graph. This is illustrated in Figure 2.3.

As in other neural network settings, training a GNN aims at finding parameters $\mathbf{w}$ such that the correct output is obtained for every graph or vertex (depending on whether the task at hand is graph-focused or vertex-focused). To this end, the learning algorithm takes a training set $S = \{(G_i, v_{i,j}, \mathbf{y}_{i,j}) : i \in \{1, \ldots, m\}\}$. Here, $G_i = \{V_i, E_i\}$ denotes an input graph, $v_{i,j} \in V_i$ one of its vertices, and $\mathbf{y}_{i,j} \in \mathbb{R}^{\tilde{d}}$ is a target output for vertex $v_{i,j}$. In vertex-focused applications, multiple vertices $v_{i,j}$ of the same graph G_i and their respective target outputs may be used in the training of the network. This is the case, for example, for large-scale applications such as those involving extensive social networks. In graph-focused applications (for instance, predicting chemical properties of molecules), on the other hand, the training

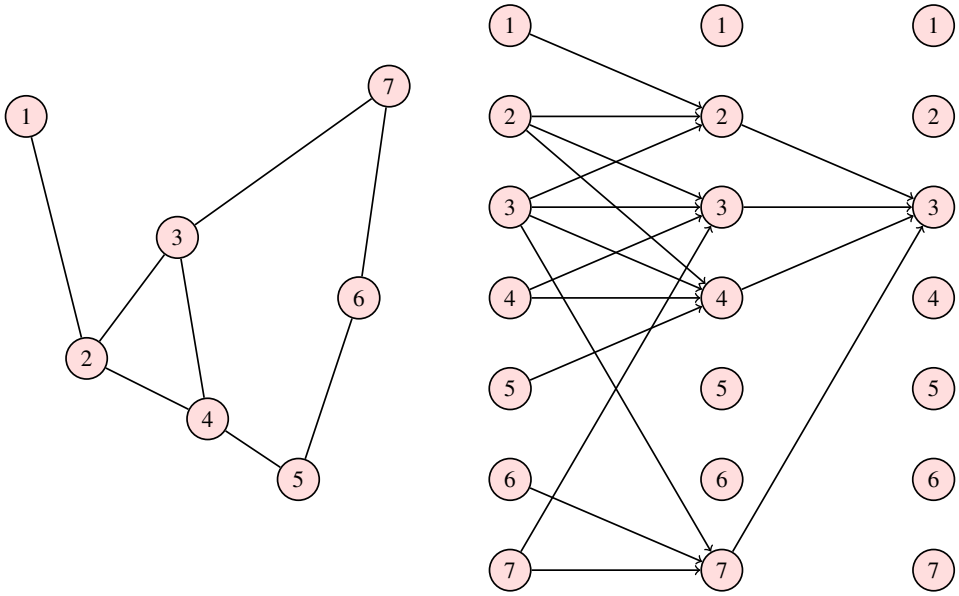


Figure 2.2: An undirected input graph with seven vertices (left) and the corresponding network unrolled (right). For simplicity, only such information flow is shown which is relevant in the calculation of the state of vertex 3.

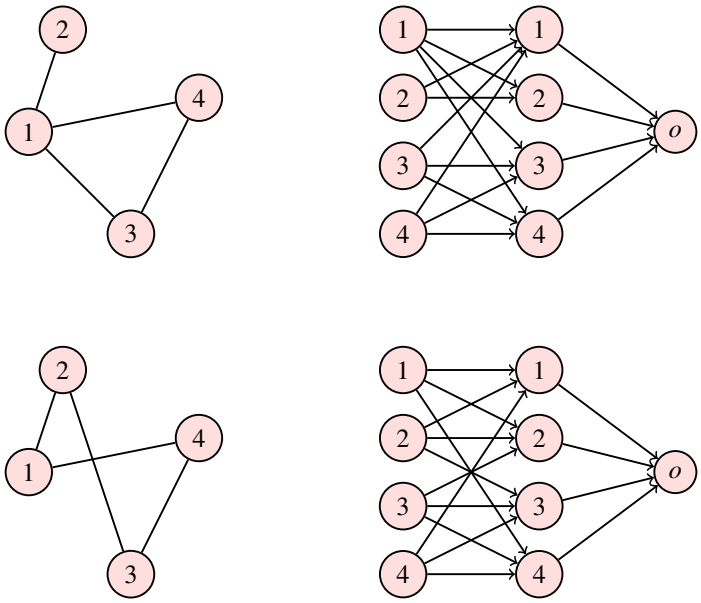


Figure 2.3: Two input graphs with four vertices each (left) and their respective diffusion processes (right). Here, the GNN model assumes $T = 1$ and produces a graph output.

set typically contains distinct graphs G_i and the corresponding target vertex $v_{i,j} = v_i$ corresponds to an output vertex pooling all other vertex outputs into a graph output.

Oftentimes, GNNs are trained in a mini-batch fashion in order to increase efficiency by performing the message passing phase on a number of subgraphs of the input graph at a time, instead of on the entire graph at once. To this end, a set of target vertices is sampled for each mini-batch, to which the network and subsequent backpropagation of the error are then applied. Given that the number of vertices

involved in this computation may increase exponentially with the number of network layers (time steps in the message passing phase), it is possible to furthermore sample subsets of the neighborhoods to use in the update at each time step [9]. With this approach, one can view the training set to contain individual graphs consisting of the target vertices together with their respective neighborhoods. The training itself can be done via common gradient-descent approaches such as Stochastic Gradient Descent (SGD).

2.2 Parallel Computing

2.2.1 Bulk Synchronous Parallel (BSP)

Parallel architectures that enable separate processes to run simultaneously are an important aspect in the advances that have been made with regard to speed and functionality. In addition to being equipped with the hardware supporting such parallelism, a user needs to decide on software and applications to be able to make use of parallel architectures, as well as an approach to designing parallel programs. For the purpose of this thesis, algorithms are based on the Bulk Synchronous Parallel (BSP) model. This model was proposed by Leslie Valiant in 1990 as a unifying model for parallel computing [10]. Its aim is to serve as a standard enabling hardware and software designers to each develop their products while maintaining portability.

In the (basic) BSP model, a computer consists of p processors (or components), each of which can perform operations on data from its local memory storage. A communication environment ensures that processors can communicate with each other to exchange data. This is visualized in Figure 2.4. Additionally, a synchronization mechanism allows for synchronization of the processors at regular intervals of l time steps. A BSP computer is characterized by the parameters p and l along with two parameters g and r. The former of these is a measure of the communication throughput of the network and the latter denotes the computing rate (in floating-point operations per second) of a single processor.

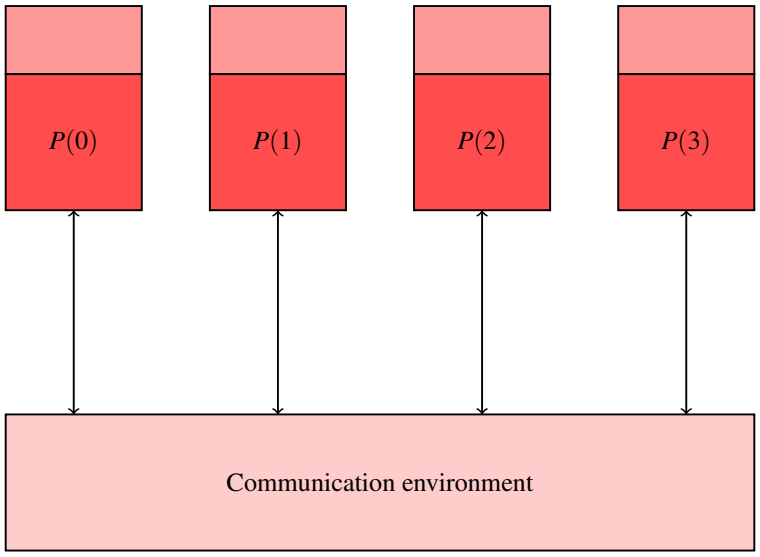


Figure 2.4: A BSP Computer consisting of four processors. Each box labeled $P(0), \ldots, P(3)$ represents one processor with its local memory storage.

An algorithm that is based on the BSP model is structured in *supersteps* separated by synchronizations. In a given superstep, processors can individually perform computations and send and receive data. Supersteps only involving basic operations are referred to as computation supersteps, whereas supersteps

exclusively meant for transferring data are called communication supersteps. Supersteps can also be mixed if they contain both computation and communication. In this case, communication is delayed until all computation has been finished. Figure 2.5 shows an example of the structure of a BSP algorithm.

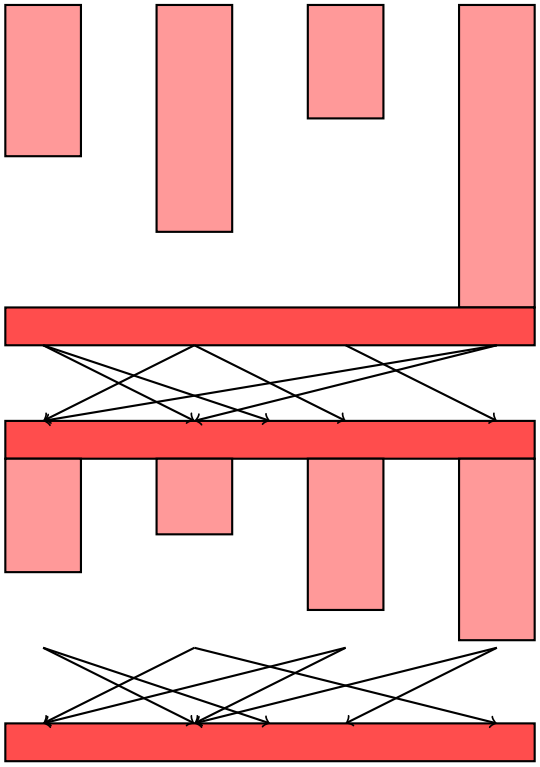


Figure 2.5: An example of a BSP algorithm consisting of three supersteps: one computation superstep followed by one communication superstep and one mixed superstep.

2.2.2 Parallel Computing on Graphs

Since the number of computations performed in the message passing process of a GNN depends both on the size of the graph (more concretely, the number of vertices and edges) and on the number T of time steps, training a GNN on and applying it to huge graphs with many iterations can quickly become very expensive. Moreover, as the size of the graph increases, memory issues arise. It may therefore be necessary to process the input graph(s) in a distributed manner, that is, by letting multiple processors simultaneously process individual parts of the input data.

Two central considerations in the design of parallel algorithms are the distribution of the input data across the processors and the question of how the workload can be divided. For the purpose of graph processing as in the case of a GNN with input graph $G = (V,E)$, a natural choice is to assign to each of the p processors a subset of the vertex set V. Together with mini-batching and a number of optimizations, this approach to parallelism has been shown to enable the training of GNNs on graphs with millions of vertices and three billion edges [11].

In order to distribute an input graph over p processors, a p-way vertex partitioning $V = \bigcup_{s=0}^{p-1} V_s$ is used such that the V_s are nonempty and pairwise disjoint. Figure 2.6 shows an example of a graph partitioned over three processors. For a given choice of $V_0, \ldots, V_{p-1}$, a mapping $\phi : V \to \{0, \ldots, p-1\}$ defined by $\phi(v) = s$ for $v \in V_s$ indicates the processor that owns a given vertex $v \in V$. We furthermore denote by

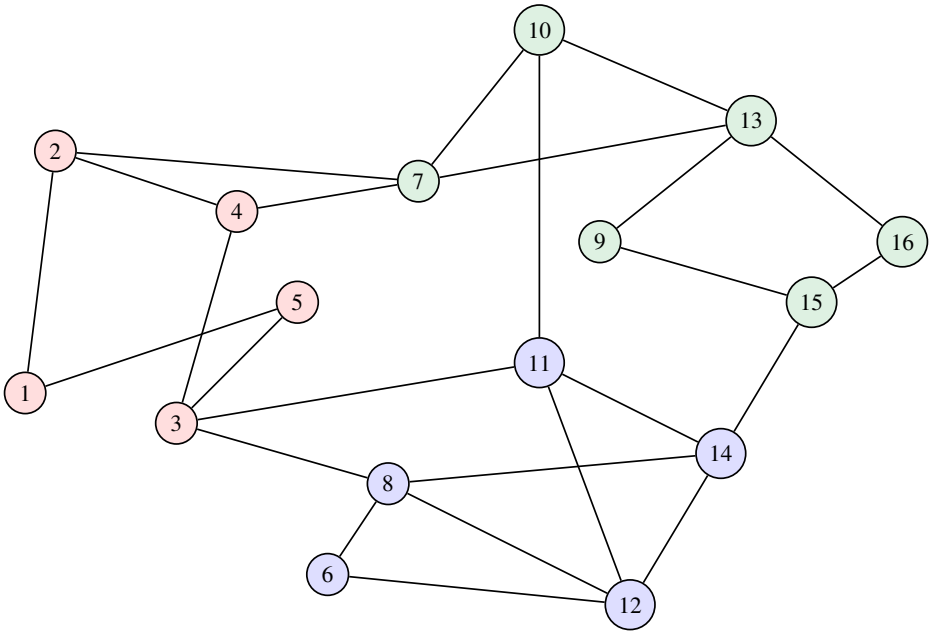


Figure 2.6: A possible partitioning of the vertices of a graph. Each color represents one of three processors over which the graph is distributed.

$E_s = \{(v,u) \in E : v,u \in V_s\}$ the set of *internal edges* of processor $P(s)$. The set $H_s = \{u \in V \setminus V_s : \exists\,(v,u) \in E$ s.t. $v \in V_s\}$ is the set of *halo vertices* of processor $P(s)$. *Halo edges* are edges $(v,u) \in E$ such that u and v are not owned by the same processor. That is, such that $\phi(u) \neq \phi(v)$.

As a result of distributing the input graph in this fashion, only limited information is available at each processor's local memory and the current states of neighboring vertices may need to be retrieved from a different processor first in order to perform the state update for a given vertex. With a poor choice of partitioning, the communication can quickly outweigh the speedup achieved from parallelizing the computation. An effective partitioning should therefore balance the workload while minimizing the between-processor communication, but its computation should not dominate the graph processing.

2.3 Think Like A Vertex

Think Like A Vertex (TLAV) refers to an approach to program design that can be encountered in different graph processing frameworks. The motivation is that many sequential algorithms developed for solving graph-related tasks adopt a global perspective and process the input graph at graph level. This usually requires the entire graph and corresponding information to be available to the algorithm at all times and hinders processing large amounts of data on distributed (parallel) system architectures. Algorithms based on a TLAV approach, on the other hand, are inherently local and therefore well-suited for execution on parallel architectures. Key feature of TLAV frameworks is the (repeated) execution of a vertex program over each vertex in the graph individually, exclusively using information that is locally available. Concretely, the vertex program receives as input only data related to the vertex itself as well as its neighbors and neighboring edges.

The first published TLAV framework was the Pregel system introduced in 2010 [12]. It is based on the BSP model and alternates between local computations and communication with neighboring vertices, in a sequence of supersteps separated by synchronizations. Information is exchanged purely via message

passing and the messages are delivered in between supersteps. The process terminates when no vertices are active (that is, performing computations or exchanging messages) anymore. Pregel has been shown to perform well in different applications of graph processing and displays good scalability. With some minor adjustments to the framework, the open-source system Apache Giraph, which is based on Pregel, can process graphs with up to one trillion edges [13].

Various other frameworks can be regarded as vertex-centric. Generally, TLAV frameworks can be characterized through their design choices with respect to four major aspects, as described in [14]. The TLAV approach can furthermore be easily combined with both the BSP model and the GNN model, as there is already some overlap between BSP and TLAV frameworks (for instance, the Pregel framework fits both approaches) and the message and update functions in the diffusion process of a GNN are inherently vertex-centric. In order to incorporate the TLAV approach, our BSP algorithm implementing the message passing process of a parallel GNN lets each processor loop over its own vertices and execute for each vertex a function which performs the message passing and state update. Synchronizations between consecutive time steps ensure that the correct information is available before the next update.

3 Implementation

With the necessary preliminaries established, we can develop a sequential and a parallel algorithm implementing a GNN. In the following, we will define the message and update functions to be used in our network, describe the resulting algorithm, and discuss some further implementation details. We first focus on the message passing phase of the network, as this is primarily relevant in the forward pass of information during neural network training as well as when applying a trained GNN to a graph. In Section 3.5, we will discuss how the algorithm can be extended so as to also perform the training of the network.

3.1 Network Model

In the literature, a relatively simple convolutional method (the Graph Convolutional Network, or GCN in short, e.g. [15]) similar to the approach used in regular convolutional neural networks appears to be a popular baseline GNN model. However, this approach is not entirely suitable for our purposes. The GCN message passing phase usually runs for only a small number of time steps, e.g. $T = 2$. As a result, only a very small neighborhood of each node is used in the computation of the node output, and long-range dependencies within the graph are not accounted for. For large-scale graphs as we are aiming at, one would therefore want to choose a larger T in the message passing phase. However, this can lead to *oversmoothing* because the state of the node itself loses importance after repeated updates.

Message and update functions based on a Gated Recurrent Unit (GRU) or Long Short-Term Memory (LSTM) approach may help to mitigate oversmoothing, similar to how they improve performance of regular recurrent neural networks with many layers, by adding *gates* that control how much influence the neighboring states have. Our algorithm is based on a GRU-like state update which has been shown to achieve good results on tasks that require, for example, a sequence of outputs [16].

The message and update functions corresponding to the forward pass utilizing a GRU update are the following. We initialize $\mathbf{h}_v^0 = \mathbf{x}_v$ for every $v \in V$, that is, the initial state of each vertex is set equal to its feature vector. We furthermore define the message function for each neighboring vertex $u \in N(v)$ as

$$M_t(\mathbf{h}_v^t, \mathbf{h}_u^t, \mathbf{x}_{(v,u)}) = \mathbf{B}\mathbf{h}_u^t$$

and the update of the state of vertex v is performed as

$$\begin{aligned}
\mathbf{a}_v^{t+1} &= \mathbf{m}_v^{t+1} + \mathbf{b} \\
\mathbf{z}_v^{t+1} &= \sigma(\mathbf{W}^z \mathbf{a}_v^{t+1} + \mathbf{U}^z \mathbf{h}_v^t) \\
\mathbf{r}_v^{t+1} &= \sigma(\mathbf{W}^r \mathbf{a}_v^{t+1} + \mathbf{U}^r \mathbf{h}_v^t) \\
\tilde{\mathbf{h}}_v^{t+1} &= \tanh(\mathbf{W}\mathbf{a}_v^{t+1} + \mathbf{U}(\mathbf{r}_v^{t+1} \odot \mathbf{h}_v^t)) \\
\mathbf{h}_v^{t+1} &= (1 - \mathbf{z}_v^{t+1}) \odot \mathbf{h}_v^t + \mathbf{z}_v^{t+1} \odot \tilde{\mathbf{h}}_v^{t+1}
\end{aligned}$$

where $\odot$ denotes element-wise multiplication and both the logistic function σ and the hyperbolic tangent tanh are applied element-wise. The vector $\tilde{\mathbf{h}}_v^{t+1}$ can be viewed as a candidate state computed based on the current vertex state $\mathbf{h}_v^t$ and the biased neighbor states $\mathbf{a}_v^{t+1}$. The vectors $\mathbf{z}_v^{t+1}$ and $\mathbf{r}_v^{t+1}$ are equivalents of the update and reset gates in a classical GRU and determine how much influence the candidate state and the current state each have on the updated state.

One major simplification that we have made for the purpose of this project, compared to the Gated Graph Neural Network as introduced in [16], is that we assume the graph G to be *undirected* and that edges are

all of the same type. If the context requires, the model can be adapted to accommodate for graphs with directed edges as well, by applying different parameter matrices $\mathbf{B}_{in}$ and $\mathbf{B}_{out}$ to incoming and outgoing edges, respectively. Similarly, distinct classes of edges can be incorporated by assigning to each edge (v,u) an edge label $\mathbf{x}_{(v,u)}$ based on its edge type and letting the parameter matrix $\mathbf{B}_{\mathbf{x}_{(v,u)}}$ depend on this label.

The readout function in this model was chosen to be equal to $R(\mathbf{h}_v^T) = \frac{1}{d}\sum_{i=1}^{d}(\mathbf{h}_v^T)_i$ for simplicity, but can be adjusted based on the application.

With this choice of functions in the propagation phase, we arrive at a sequential algorithm which is summarized in Algorithm 1. Here, $\mathrm{GRU}(\mathbf{h}_v^t, \mathbf{m}_v^{t+1})$ denotes the update as defined above. The parameters that would need to be trained for this particular model choice are the bias vector $\mathbf{b} \in \mathbb{R}^d$ and the weight matrices $\mathbf{B}, \mathbf{W}^z, \mathbf{W}^r, \mathbf{W}, \mathbf{U}^z, \mathbf{U}^r, \mathbf{U} \in \mathbb{R}^{d\times d}$.

Algorithm 1 Sequential Message Passing

Input: Graph $G = (V,E)$, vertex features $\mathbf{x}_v$ for $v \in V$
Output: Vertex outputs $\mathbf{o}_v$ for $v \in V$

```
for v ∈ V do
    h_v^0 = x_v
for t = 0 to T − 1 do
    for v ∈ V do
        m_v^{t+1} = Σ_{u∈N(v)} B h_u^t
        h_v^{t+1} = GRU(h_v^t, m_v^{t+1})
for v ∈ V do
    o_v = (1/d) Σ_{i=1}^{d} (h_v^T)_i
```

3.2 Data Distribution

Though our aim is to apply the GNN to a single large-scale graph, network training may involve performing the propagation process for a number of smaller subgraphs G_i individually, whether these are individual disconnected graphs or neighborhoods of sampled vertices. Nevertheless, for simplicity, we will assume that the GNN works on (a subset of the vertices of) only a single input graph, by setting $G = (V,E)$ with $V = \bigcup_i V_i$ and $E = \bigcup_i E_i$. We will parallelize the algorithm by distributing the vertices of G over the available processors $\{0,\ldots,p-1\}$ as discussed in Section 2.2.2. Each processor $P(s)$ has a copy of the set of parameters used in the message and update functions and subsequently executes the vertex function only for its own vertices $v \in V_s$.

We need to account for the fact that not all neighbors of each vertex reside on the current processor. As can be seen in Figure 3.1, information which is communicated along halo edges needs to be exchanged between different processors. In our algorithm, this will be achieved by adding a communication superstep prior to every update step.

Notice, however, that there may be multiple halo edges connecting a processor to one of another processor's vertices. For instance, the red processor in Figure 3.1 has connections to vertex 7 on the green processor through both 2 and 4. Similarly, the blue processor needs the state of vertex 3 on the red processor for the updates of both 8 and 11. If the respective vertex states were sent along the explicit halo edges, i.e. if each processor sent the state of its vertices to each neighboring non-local vertex individually, then this would result in the states of 7 and 3 being communicated multiple times to the same processor. We will avoid such redundant communication by replicating halo vertices on the processors.

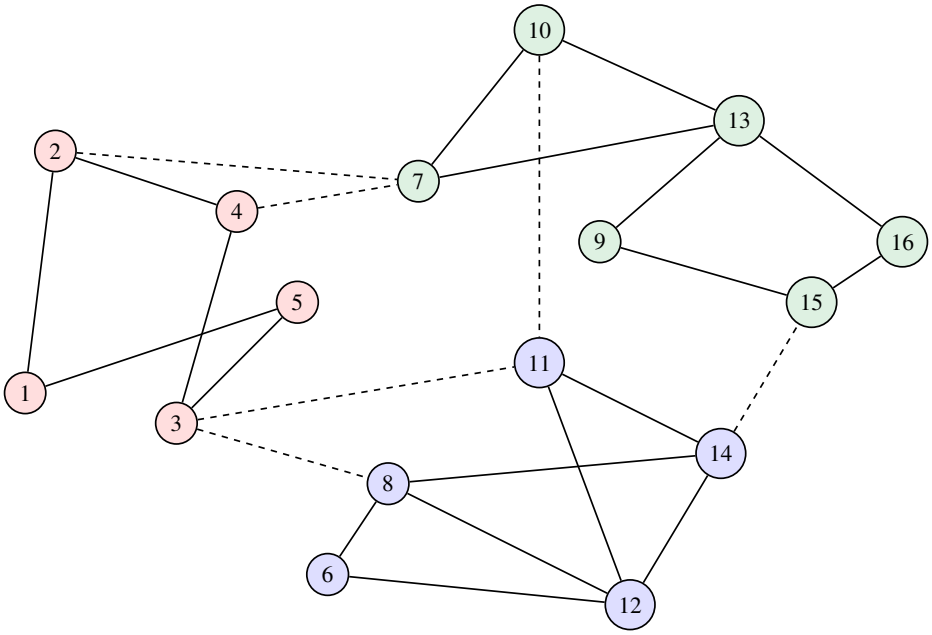


Figure 3.1: Halo edges in a graph distributed over three processors. Solid lines represent internal edges, dashed lines indicate halo edges that invoke communication.

That is, the subgraph G_s which a processor $P(s)$ has access to consists of the sets of local vertices V_s (according to the chosen partitioning of the graph) and halo vertices H_s, as well as the edge set induced by $V_s \cup H_s$. This approach results in a local view of the graph as visualized in Figure 3.2. Here, we have $V_s = \{6, 8, 11, 12, 14\}$ and $H_s = \{3, 10, 15\}$ for the blue processor $P(s)$.

It remains to answer the question of how to distribute the vertices of the graph in such a way that communication between processors is minimized while balancing the amount of computation each processor has to perform. In order to reach the latter goal, we would need to ensure that the cardinalities of the sets V_s are similar across all processors, such that each processor executes the vertex function for a roughly equal number of vertices. Furthermore, the combined degrees of all internal vertices V_s should not vary much so as to balance the computation of the messages $\mathbf{m}_v^{t+1}$.

A simple block distribution may achieve an almost equal division of the workload in the update step as it assigns a close to equal number of vertices to each processor. The block distribution is defined as $\phi(v_j) = P(j \operatorname{div} b)$, where $v_j \in V$ has index $j \in \{0, \ldots, n-1\}$ and $b = \lceil \frac{n}{p} \rceil$ is the block size, calculated such that the cardinalities of the V_s differ by at most $p-1$. Although distributing the vertices over the processors in this fashion divides the workload in the vertex update evenly, it will likely lead to a lot of between-processor communication caused by halo edges. Likewise, the number of internal edges per processor could differ by a lot.

Since the propagation of information through the GNN has similarities to matrix-vector multiplication, with the vector representing vertex states and the matrix being the adjacency matrix corresponding to the graph, we can instead apply a matrix partitioning algorithm for deriving a suitable vertex partitioning. In this project, we will use the Mondriaan partitioner [17] for sparse matrices. Mondriaan is based on hypergraph partitioning and aims at distributing the vertices of a hypergraph (in which vertices are connected by hyperedges, i.e. subsets of the vertex set) over a number of processors in such a way that the number of hyperedges spanning multiple processors is minimized while adhering to some given balance constraint. Since it is possible to bound the communication volume of a matrix-vector multiplication from above by the edge cut in a corresponding hypergraph [18], using a distribution of the vertices

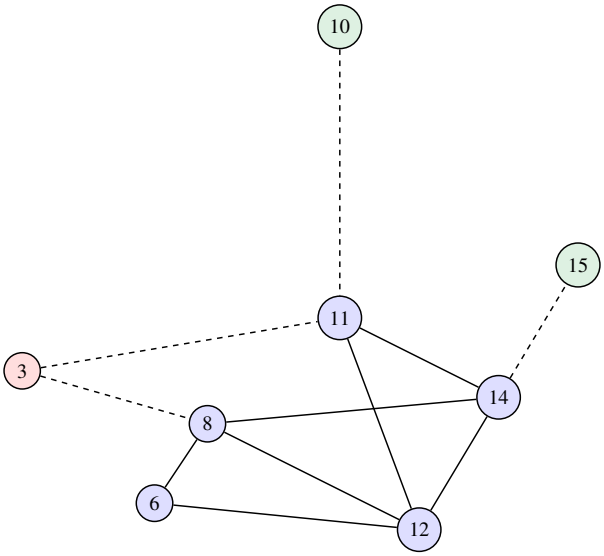


Figure 3.2: The locally available subgraph for the blue processor in Figure 3.1. Vertices colored blue are local vertices $v \in V_s$, vertices colored red or green are replicated halo vertices $v \in H_s$. Solid lines indicate internal edges, whereas dashed lines represent halo edges.

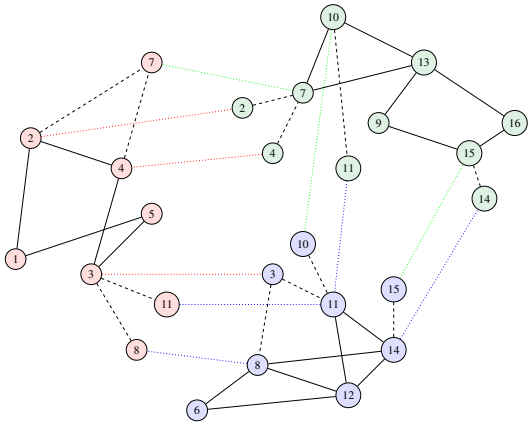
according to a hypergraph partitioning should enable us to reduce communication to a near-minimum and at the same time balance the computational workload.

3.3 Parallel Algorithm

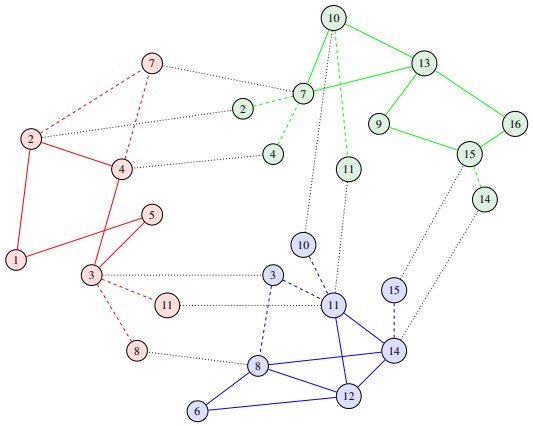
As previously noted, the message passing process in the GNN model is essentially vertex-centric, and an algorithm based on the TLAV approach would loop over each vertex in the graph in order to perform the state update. We can therefore use Algorithm 1 as our baseline algorithm. With the distribution of the vertices as discussed, we design a parallel algorithm which alternates between communication and computation supersteps.

At the start of the algorithm, each processor initializes the vertex states of its own vertices $v \in V_s$. Since this step can be carried out while reading the input data, we do not consider it to be part of the actual algorithm.

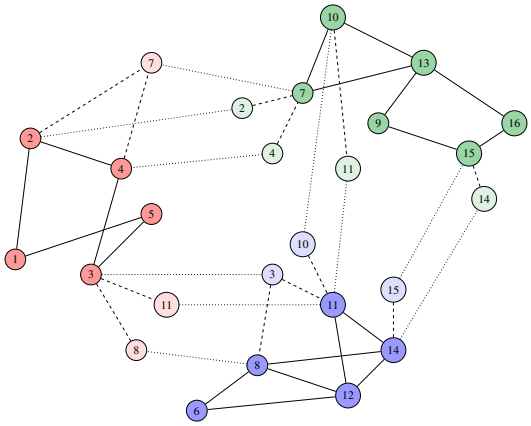
The message passing phase is the main aspect of the network. Before the update of the vertex state can be performed at a given time step t, we need to ensure that each processor has the current vertex state $\mathbf{h}_v^t$ of each of its halo vertices $v \in H_s$ at its disposal. To this end, Superstep 1 is a communication superstep that serves to make the necessary information available to the processors. In our implementation, we let each processor actively send the current states of its vertices to those processors which need them. To this end, we define $D_s \subseteq V_s$ to be the set of vertices in V_s which are replicated on at least one other processor. That is, $D_s = \{v \in V_s : \exists \tilde{s} \text{ s.t. } v \in H_{\tilde{s}}\}$. Furthermore, for every $v \in D_s$, we denote by $\psi(v) \subseteq \{0,\ldots,p-1\}$ the set of processors that have v in their set of halo vertices, i.e. $\psi(v) = \{\phi(u) : u \in N(v)\} \setminus \{s\}$. Since the distribution of the vertices does not change throughout the algorithm and every processor thus needs to obtain the vertex states of the same vertices in every update step, this approach avoids unnecessary (and redundant) communication caused by processors first requesting the information they need. In the resulting communication superstep, for each of its local vertices $v \in D_s$, processor $P(s)$ sends the vertex identifier and the current vertex state to every processor that has a copy of v. This between-processor communication is illustrated in Figure 3.3a.



(a) Communication along halo edges between processors. Colors indicate which processor initiates the communication.



(b) Communication along internal edges within each processor.



(c) Vertex updates on each processor. Only the vertex states of local vertices need to be updated.

Figure 3.3: The diffusion process in the parallel algorithm.

The information exchanged in Superstep 1 is processed at the beginning of Superstep 2. Here, processor $P(s)$ takes each ordered pair $(v, \mathbf{h}_v^t)$ it received and updates its own copy of vertex v. Next, $P(s)$ loops over its local vertex set V_s and performs the message passing (meaning that each $v \in V_s$ retrieves the current vertex states of its neighboring vertices) and the state updates. The within-processor communication and update of the local vertex states are indicated in Figures 3.3b and 3.3c, respectively.

Supersteps 1 and 2 are repeated until the final iteration T, after which the vertex output $\mathbf{o}_v$ is computed for each $v \in V_s$ in Superstep 3. The resulting parallel algorithm can be summarized as Algorithm 2.

Algorithm 2 Parallel Message Passing for processor $P(s)$

Input: Graph $G = (V, E)$, vertex features $\mathbf{x}_v$ for $v \in V_s$
Output: Vertex outputs $\mathbf{o}_v$ for $v \in V_s$

for $v \in V_s$ **do**
  $\mathbf{h}_v^0 = \mathbf{x}_v$ ▷ Initialization
for $t = 0$ to $T - 1$ **do**
  for $v \in D_s$ **do** ▷ Superstep 1
    Put $(v, \mathbf{h}_v^t)$ in $P(\psi(v))$
  for $v \in V_s$ **do** ▷ Superstep 2
    $\mathbf{m}_v^{t+1} = \sum_{u \in N(v)} \mathbf{B}\mathbf{h}_u^t$
    $\mathbf{h}_v^{t+1} = \mathrm{GRU}(\mathbf{h}_v^t, \mathbf{m}_v^{t+1})$
for $v \in V_s$ **do** ▷ Superstep 3
  $\mathbf{o}_v = \frac{1}{d} \sum_{i=1}^{d} (\mathbf{h}_v^T)_i$

3.4 Implementation Details

The GNN message passing algorithm described so far was implemented[1] in the C++ programming language using three central classes, the most important aspects of which will be summarized below.

The class `vertex` is an essential aspect of the vertex-centric approach. The class defines the update and readout functions which correspond to the vertex functions in the TLAV framework. Furthermore, each vertex stores its current as well as a temporary state. The latter is essential as it ensures that neighboring vertices can access the state computed at the previous time step: during the state update at time t, the new state $\mathbf{h}_v^{t+1}$ is stored in the variable `hidden_state_temp`, and the vertex state `hidden_state` is updated only after all vertices have completed their computations. The neighbors of a given vertex $v \in V_s$ are stored by means of a vector of pointers to the vertices adjacent to v. Their vertex state can be accessed by v by means of a public function `get_hidden_state()`. Another important attribute in the distributed setting is the vector `neighbor_processors` which represents the set $\psi(v)$ of neighboring processors owning a copy of v. Similarly, halo vertices $v \in H_s$ have an attribute `processor` storing the identifier of the processor owning v.

The class `graph` mainly serves to define the graph by creating a vector of pointers to all local vertices; in the distributed graph, the first `num_vertices` vertices in the vector point to the internal vertices $v \in V_s$, and the remaining `num_halo_vertices` vertices point to the halo vertices $v \in H_s$ replicated on processor $P(s)$. As a result, the vertices present at a processor need to be identified by means of their local indices in the vector of vertices. Figure 3.4 illustrates how the vertices are renumbered for the blue processor in Figure 3.1. In order to be able to retrieve the correct vertices during the algorithm, two

[1] The source code can be found at `https://github.com/Katharina-Klein/ParallelGNN`

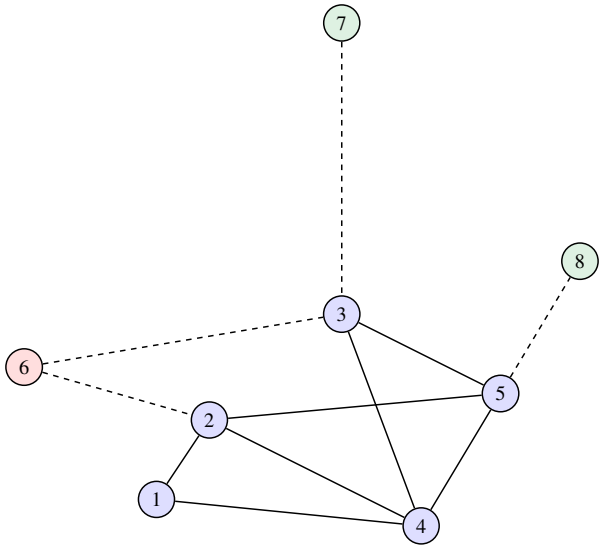


Figure 3.4: Internal vertices and halo vertices together with their local vertex indices.

vectors `local_vertex_indices` and `global_vertex_indices` are used for shifting between local and global vertex identifiers. Note furthermore that the `graph` class does not store the edges of the graph explicitly since these are not considered necessary global information in TLAV and are instead represented implicitly by means of the adjacency lists of the vertices.

Lastly, the class `gnn` implements the actual message passing process. Each processor stores the current weight parameters in the message and update functions in its own vector of weight matrices. When the function `forward_pass()` is called, the algorithm alternates between communicating the current vertex states and computing the new states as described in Algorithm 2.

The vertex states as well as the parameter matrices are implemented as column vector and matrix objects provided in the C++ library *Eigen* [19] which allows for fast matrix-vector multiplication in the update of the vertex state. In order to implement the parallel algorithm and the communication supersteps in particular, we furthermore make use of the C++ interface *Bulk*. Initially released in 2017, Bulk is an interface that enables the implementation of BSP algorithms in the C++ programming language [20]. Programs are written in a Single Program Multiple Data (SPMD) style, meaning that each processor executes the same program, but on its own data. They are suitable to be run on machines employing shared or distributed memory as well as hybrid architectures. In our algorithm, the communication is carried out by means of message queues. Since a Bulk queue does not support column vector objects of Eigen, we create a `bulk::queue<int, float[]>` to send the ordered pairs $(v, \mathbf{h}_v^t)$ to processors that need them. The `int` object represents the global index of vertex v and the `float[]` vector stores the elements of the vertex state $\mathbf{h}_v^t$.

3.5 Network Training

Until now, we have focused on the diffusion process in a GNN, that is, the propagation of information through the network. This corresponds to a full forward pass in neural network training as well as to the application of a trained GNN to an input graph. We will now outline how the parallelization of the algorithm can be extended to network training. As the GNN can be trained via backpropagation and gradient descent, we first establish the necessary computations in the backward pass of the error.

Recall that given a vertex v with features $\mathbf{x}_v$, the "true" or expected output is denoted by $\mathbf{y}_v$ and the output of the network model by $\mathbf{o}_v$. We consider some general loss function $l_v = l(\mathbf{o}_v, \mathbf{y}_v)$ which quantifies the error that is produced by the network in vertex v, for instance the cross entropy loss or the square loss. The total loss l of the network may be computed as the sum or average of the l_v for all vertices v in the training set. For any parameter w in the network architecture, the parameter update in the gradient descent algorithm is carried out as

$$w^{k+1} = w^k - \eta \frac{\partial l}{\partial w^k}$$

where η is a learning rate that should ensure that the modification is neither too small nor too large. The training method therefore requires the partial derivatives of the loss with respect to the each of the parameters. That is, one needs to compute

$$\frac{\partial l}{\partial \mathbf{B}} = \begin{pmatrix} \frac{\partial l}{\partial \mathbf{B}_{1,1}} & \cdots & \frac{\partial l}{\partial \mathbf{B}_{d,1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial \mathbf{B}_{1,d}} & \cdots & \frac{\partial l}{\partial \mathbf{B}_{d,d}} \end{pmatrix}$$

$$\frac{\partial l}{\partial \mathbf{b}} = \begin{pmatrix} \frac{\partial l}{\partial \mathbf{b}_1} & \cdots & \frac{\partial l}{\partial \mathbf{b}_d} \end{pmatrix}$$

for the weight matrix $\mathbf{B}$ and the bias vector $\mathbf{b}$, and analogous derivatives for the remaining weight matrices. Note that to perform the actual weight update in the training process, the transposes of the derivative matrices (so as to simply subtract them) are needed, i.e. the gradients in denominator layout notation. For purposes of analysis, however, we stick to the numerator layout notation for now. In this notation, the derivative of a vector $\mathbf{x}$ with respect to a vector $\mathbf{y}$ refers to

$$\frac{\partial \mathbf{x}}{\partial \mathbf{y}} = \begin{pmatrix} \frac{\partial \mathbf{x}_1}{\partial \mathbf{y}_1} & \cdots & \frac{\partial \mathbf{x}_1}{\partial \mathbf{y}_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{x}_d}{\partial \mathbf{y}_1} & \cdots & \frac{\partial \mathbf{x}_d}{\partial \mathbf{y}_d} \end{pmatrix}$$

By repeatedly applying the chain rule of differentiation, we arrive (for the example parameters $\mathbf{B}$ and $\mathbf{b}$ as above) at

$$\frac{\partial l}{\partial \mathbf{B}} = \sum_v \frac{\partial l_v}{\partial \mathbf{o}_v} \frac{\partial \mathbf{o}_v}{\partial \mathbf{h}_v^T} \frac{\partial \mathbf{h}_v^T}{\partial \mathbf{B}}$$

$$\frac{\partial l}{\partial \mathbf{b}} = \sum_v \frac{\partial l_v}{\partial \mathbf{o}_v} \frac{\partial \mathbf{o}_v}{\partial \mathbf{h}_v^T} \frac{\partial \mathbf{h}_v^T}{\partial \mathbf{b}}$$

The first two factors of each term need to be determined based on the loss and readout functions, which, provided that the expected vertex output is stored at the vertex, depend only on information which is locally available and can therefore be computed by means of a vertex function. As for the last factor, we note that the update step from $\mathbf{h}_v^t$ to $\mathbf{h}_v^{t+1}$ is in essence a composition of elementary functions. The individual calculations performed in the update are visualized in Figure 3.5. Relevant intermediate steps are denoted by $\mathbf{f}_1$ through $\mathbf{f}_{15}$.

From this diagram, it can be easily deduced that as a result of the recursive update process, $\mathbf{h}_v^T$ depends on each of the parameter matrices not only directly, but also through every previous state $\mathbf{h}_v^t$ and the previous states $\mathbf{h}_u^t$ of the neighbors of v. This needs to be taken into account in the calculation of the partial derivatives. For example,

$$\frac{\partial \mathbf{h}_v^T}{\partial \mathbf{B}} = \frac{\partial \mathbf{f}_{15}}{\partial \mathbf{B}} + \frac{\partial \mathbf{h}_v^T}{\partial \mathbf{h}_v^{T-1}} \frac{\partial \mathbf{h}_v^{T-1}}{\partial \mathbf{B}} + \sum_{u \in N(v)} \frac{\partial \mathbf{h}_v^T}{\partial \mathbf{h}_u^{T-1}} \frac{\partial \mathbf{h}_u^{T-1}}{\partial \mathbf{B}}$$

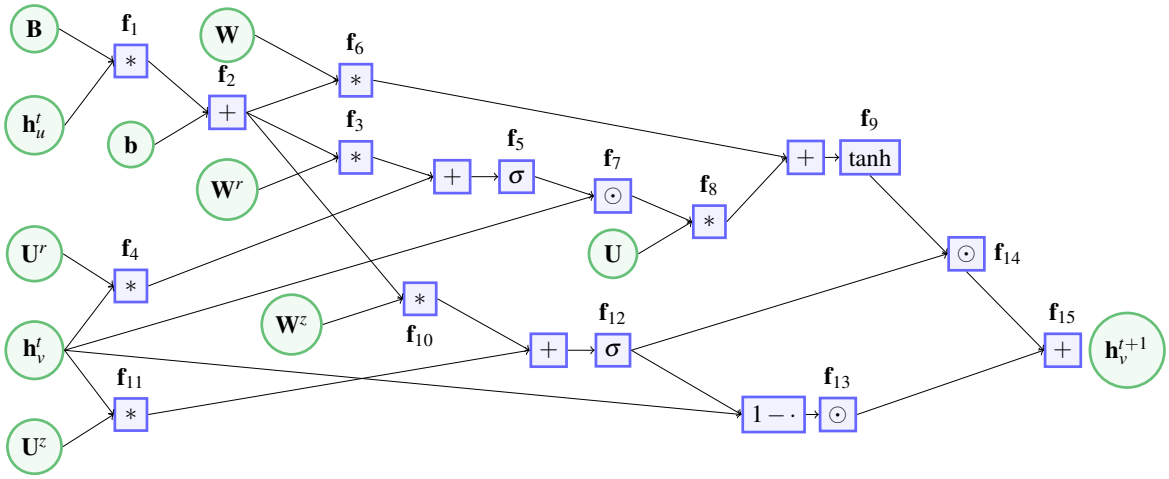


Figure 3.5: Elementary computations in the GRU update step. Notice that $\mathbf{h}_u^t$ is needed for each $u \in N(v)$ in the computation of $\mathbf{f}_1$, though we only write $\mathbf{h}_u^t$ to save space.

and this continues recursively. Thus, if $T > 1$, then the computation at a vertex v requires information from neighboring vertices $u \in N(v)$. The partial derivatives of $\mathbf{f}_{15}$ with respect to the parameters as well as $\frac{\partial \mathbf{h}_v^T}{\partial \mathbf{h}_v^{T-1}}$ and $\frac{\partial \mathbf{h}_v^T}{\partial \mathbf{h}_u^{T-1}}$, on the other hand, can be derived by means of Figure 3.5 and repeated application of the chain rule. The data needed to calculate these derivatives, in particular the intermediate vertex states $\mathbf{h}_v^t$ and $\mathbf{h}_u^t$, have already been available during the computation of the vertex output $\mathbf{o}_v$ and can be stored at the vertex v, obtained from neighboring vertices, or recomputed.

From these observations, we can conclude that the backward pass in the backpropagation in the network training can be implemented as a message passing process similar to the forward propagation through the network.

Once the partial derivatives have been computed for the vertices v in the training set, the processors need to communicate in order to obtain the total gradient and ensure that the parameters are updated in the same way on each processor. To this end, processor $P(s)$ aggregates the partial derivatives it has computed (for each of its local vertices in the training set) into a sum and broadcasts this, that is, sends it to every other processor in a communication superstep. Finally, each processor redundantly computes the total gradient and updates its parameter matrices. Communicating the local gradients and updating the parameters adds one extra communication superstep per forward-backward iteration in the training. Notice that the number of parameters is fixed and relatively small compared to the size of the graph: for our choice of model, one vector in $\mathbb{R}^d$ and seven matrices in $\mathbb{R}^{d \times d}$ need to be trained, leading to a total of $7d^2 + d$ parameters that need to be broadcast to $p-1$ processors. The resulting $(7d^2+d)(p-1)$-relation is expected to be of minor importance in the total running time of the algorithm, compared to the computational workload and communication in the forward and backward pass of a training iteration.

4 Experimental Results

The implementation of the message passing process was tested on different graphs in order to assess the performance of the parallel algorithm. The results of these experiments can be expected to give an indication of how worthwhile it is to parallelize a fully working GNN with similar message and update functions.

4.1 Setup

Tests were run on five graphs taken from the Network Data Repository [21], a collection of hundreds of networks from various domains. The graphs that were used in the experiments are listed in Table 4.1, along with their numbers of vertices and edges. In the two smaller graphs, `FBCompany` and `FBArtist`, the vertices represent Facebook pages belonging to different companies or artists, respectively, and an edge between two vertices indicates a mutual like between the two corresponding pages. `NLosm` represents the road network of the Netherlands, based on data collected by OpenStreetMap. Edges in this graph correspond to roads connecting junctions. `AmazonCo` is a graph depicting co-purchases of Amazon products, meaning that if two vertices (corresponding to different items) are connected, they are frequently purchased together. Lastly, `Amazon2008` is a graph which was obtained by use of a web crawler. It represents books sold in the Amazon store as vertices and describes similarity between them through edges.

Graph name	$\|V\|$	$\|E\|$
`FBCompany`	14 114	52 310
`FBArtist`	50 516	819 306
`NLosm`	2 216 688	2 441 238
`AmazonCo`	403 394	3 387 388
`Amazon2008`	735 323	5 158 388

Table 4.1: Properties of the graphs used in the experiments.

While the first two graphs are used primarily to investigate how the network scales as the number of time steps in the message passing process increases, the parallel algorithm is tested on the three larger graphs. If not specified otherwise, we consider the one-dimensional case, i.e. the dimensions d and $\tilde{d}$ of the vertex states and outputs, respectively, are both equal to 1.

r	g	l
0.2367 Gflop/s	11.4	326 375

Table 4.2: BSP parameters of the computer used in the experiments.

Tests are run on an AMD Ryzen 7 4700U processor (2.0 GHz, 16 GB RAM) with eight cores and no hyperthreading, using a Windows Subsystem for Linux (WSL). Due to compatibility issues with the WSL, the implementation of the parallel algorithm uses release v2.0.0 of Bulk. Table 4.2 provides the BSP parameters of the computer as measured by means of the benchmark program provided in Bulk.

The running times presented in Section 4.2 are obtained by executing the respective program five times and taking the average. An evaluation of the standard deviation showed that the running times tend to be very consistent across several runs of the same program. Timings were taken before and after the message passing phase and therefore do not include initialization steps (specifically, building the graph)

or the calculation of the readout. We also provide the speedup and efficiency of the parallel program; these are measures indicating the added value of parallelizing an algorithm. Concretely, if Time_{seq} is the total running time of a (good) sequential program and Time_p is the total running time of a parallel program using p processors, then the resulting speedup can be calculated as

$$\text{Speedup}_p = \frac{\text{Time}_{seq}}{\text{Time}_p}$$

A speedup of $\text{Speedup}_p = p$ would indicate perfect parallelization. Consequently, the efficiency can be calculated as

$$\text{Efficiency}_p = \frac{\text{Speedup}_p}{p}$$

and one would aim for an efficiency close to $\text{Efficiency}_p = 1$.

4.2 Running Times

We first present some experimental results for the sequential algorithm. The sequential program was tested on `FBCompany` and `FBArtist` for different choices of the number T of time steps in the message passing process. Results for the one-dimensional case $d = 1$ are visualized in Figure 4.1 and two main conclusions can be immediately drawn from them. First, the running time appears to be increasing quickly as the size of the graph increases. In this case, `FBArtist` has roughly three times as many vertices as `FBCompany` and more than 16 times as many edges, and displays a running time that is about 6.5 times longer than that for `FBCompany`. This reinforces our motivation to parallelize the algorithm, since many graphs corresponding to real-life applications are much larger than these two. Second, the running time increases linearly as the number of time steps grows, which was to be expected given that every iteration in this process incurs the same cost. Based on this conclusion and the fact that the message passing phase dominates the full algorithm, we can from now on consider a fixed number of time steps in the message passing phase and assume that the total running time for a different T will scale linearly.

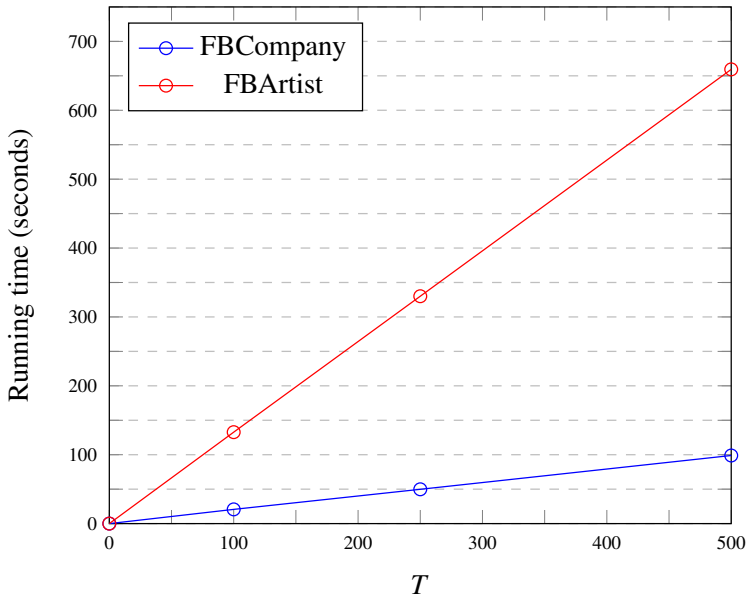


Figure 4.1: Running times of the sequential program, plotted against the number of time steps in the message passing phase.

We now turn towards the parallel algorithm, applied to the three larger graphs `NLosm`, `AmazonCo` and `Amazon2008`. For all of the following experiments, we fix $T = 25$ and consider results for both the sequential algorithm and the parallel algorithm with different numbers of processors. We consider both the block distribution and a Mondriaan distribution. The latter was determined using the Mondriaan partitioner with a maximum load imbalance of 3%. If not indicated otherwise, experiments were run using the resulting Mondriaan partitioning.

Figure 4.2 displays the running times for the sequential program and the parallel program with $p = 2, 4, 8$. As can be seen, a larger number of processors does speed up the algorithm, with the program being executed the fastest for $p = 8$. This is the case for all three of the graphs, indicating that the parallel algorithm works well for graphs of different sizes and with different properties. Moreover, we note that the number of vertices in the graph has a more pronounced influence on the running time of the (sequential) algorithm than the number of edges. Despite the fact that both `AmazonCo` and `Amazon2008` have more edges than `NLosm`, the algorithm needs significantly longer when applied to the latter graph. This is not surprising as with this particular choice of model, the computations within the vertices dominate the algorithm as long as the average degree of vertices in the graph is not too large.

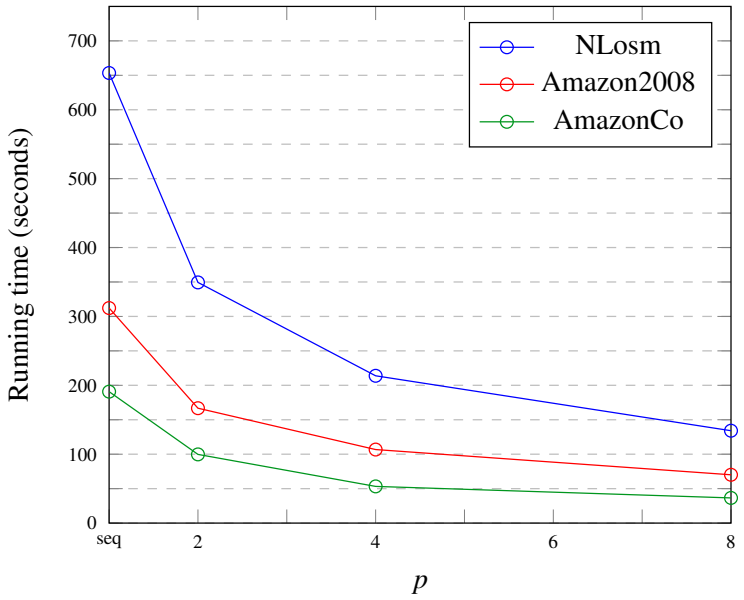


Figure 4.2: Running times of the sequential and parallel programs, plotted against the number of processors used. For the parallel algorithm, the graphs were partitioned over the processors using a Mondriaan distribution with a maximum load imbalance of 3%.

For easier interpretation, the running times along with the speedup and efficiency are furthermore provided in Table 4.3. As the results indicate, the parallelized algorithm achieves a similar speedup and efficiency for all graphs, with slight differences which can be assumed to be due to different graph structures as well as varying quality of the partitionings. As such, the best results are obtained for `AmazonCo`, with a near-optimal efficiency of almost 96% in the case of $p = 2$ and a slightly lower efficiency of around 90% in the case of $p = 4$. Generally, the performance of the parallel algorithm decreases for a larger number of processors used. While the efficiency lies above 93% for $p = 2$ on all graphs, it drops to about 73% for $p = 4$ on `Amazon2008` and ranges between only 55% and 65% for $p = 8$. This is due to the fact that while the computational workload per processor is reduced as p increases, the amount of communication does not necessarily decrease by the same factor. With a good choice of partitioning, this effect may be mitigated to some extent.

		Running time (seconds)	Speedup	Efficiency
NLosm	Sequential	653.37		
	$p = 2$	348.17	1.88	93.8%
	$p = 4$	209.88	3.11	77.8%
	$p = 8$	134.26	4.87	60.8%
AmazonCo	Sequential	190.8		
	$p = 2$	99.71	1.91	95.7%
	$p = 4$	53.25	3.58	89.6%
	$p = 8$	36.53	5.22	65.3%
Amazon2008	Sequential	312.17		
	$p = 2$	166.78	1.87	93.6%
	$p = 4$	106.74	2.92	73.1%
	$p = 8$	70.12	4.45	55.7%

Table 4.3: Running times, speedup and efficiency obtained for $p = 2, 4, 8$ processors.

In order to get an idea of whether the results obtained thus far transfer to higher-dimensional situations, we test the algorithm on NLosm and AmazonCo for three-dimensional input data, i.e. with $d = 3$. As shown in Table 4.4 for the case of four and eight processors, the parallel algorithm achieves a comparable or even larger speedup on both graphs. For NLosm and both $p = 4$ and $p = 8$, in particular, we observe speedups of almost 3.5 and 6 and an efficiency close to 87% and 75%, respectively. Only for AmazonCo and $p = 4$, the efficiency of the parallel algorithm drops when applied to the three-dimensional data.

The improvement in speedup in the other cases may be due to the computational workload increasing by a larger factor than the amount of data exchanged between processors when the dimension d increases. Namely, for each vertex whose state is required for computations at a neighboring processor, a total of $d + 1$ data words (d vector components and one vertex index) need to be sent to that processor. On the other hand, the GRU update involves several matrix-vector multiplications of $d \times d$ matrices and d-dimensional vectors, so that the total number of computations in the vertex function grows a little more than linearly. This could explain why the effect is more pronounced for the graph NLosm, since it is more sparse than AmazonCo and the computations at the vertices dominate the message passing process.

		Running time (seconds)	Speedup	Efficiency
NLosm	Sequential	491.36		
	$p = 4$	141.37	3.48	86.9%
	$p = 8$	83.56	5.88	73.5%
AmazonCo	Sequential	160.22		
	$p = 4$	51.85	3.09	77.3%
	$p = 8$	29.79	5.38	67.2%

Table 4.4: Running times, speedup and efficiency obtained for $p = 4, 8$ processors when applying the network to input data of dimension $d = 3$.

Notice, however, that both the sequential and the parallel algorithms display faster execution times in the case $d = 3$ than in the one-dimensional case $d = 1$. For example, on the road network NLosm the sequential algorithm takes about 650 seconds to perform the message passing phase with $T = 25$ and with one-dimensional vertex states. The running time then reduces to close to 500 seconds for three-dimensional vertex states, despite the fact that more computations need to be carried out. We have not been able to investigate this more closely, but we suspect that the use of the Eigen ColVector and

`Matrix` types might create some overhead that results in suboptimal performance when the dimension of the vertex states is small.

Next, we investigate the influence of the graph partitioning by comparing the Mondriaan distribution used until now with a simple block distribution. One would expect the Mondriaan distribution to perform better, given that it aims at minimizing the communication volume while retaining a balanced workload. As Table 4.5 and Figures 4.3 and 4.4 show for the one-dimensional case $d = 1$, however, this is not always the case. For the road network `NLosm`, the algorithm is faster when using a block distribution. We can observe this effect for both four and eight processors, although the difference is more pronounced in the latter case. For the graph `Amazon2008`, a similar situation occurs when using $p = 8$, with the block distribution leading to an almost 15% faster running time than the Mondriaan distribution. On the other hand, the Mondriaan distribution appears to be the better choice on `Amazon2008` with four processors, as well as on `AmazonCo` with both $p = 4$ and $p = 8$.

		Running time (seconds) Mondriaan distribution	Running time (seconds) block distribution
`NLosm`	$p = 4$	209.88	**206.94**
	$p = 8$	134.26	**110.9**
`AmazonCo`	$p = 4$	**53.25**	69.11
	$p = 8$	**36.53**	50.11
`Amazon2008`	$p = 4$	**106.74**	114.37
	$p = 8$	70.12	**60.46**

Table 4.5: Running times for the Mondriaan and block distributions for $p = 4, 8$ processors. Boldface indicates faster running times.

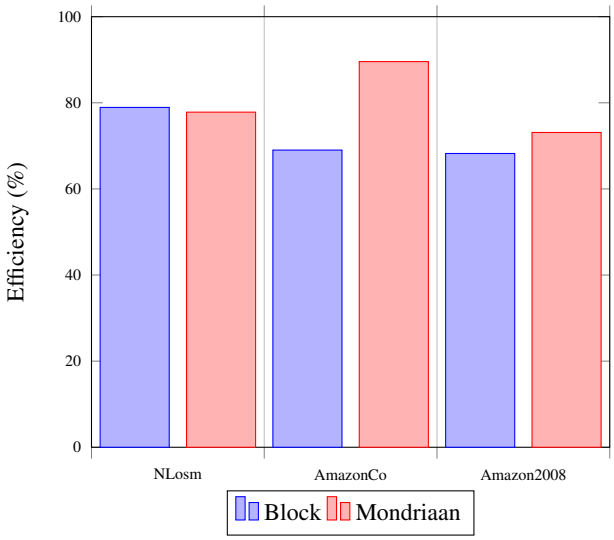


Figure 4.3: Efficiency of the parallel algorithm with $p = 4$ compared to the sequential algorithm when employing a block or Mondriaan distribution.

These results are surprising in light of the generally good performance of the Mondriaan partitioner. Indeed, Table 4.6 shows for $p = 4$ that the partitioning based on the block distribution should incur a much higher communication cost as both the maximum number of halo vertices and the maximum number of requested vertices (the larger of which determines the value of h in the h-relation) are significantly larger. Analogously, the maximum number of halo and internal edges combined is larger for the block

Figure 4.4: Efficiency of the parallel algorithm with $p = 8$ compared to the sequential algorithm when employing a block or Mondriaan distribution.

	$p = 4$	Max. # of halo vertices	Max. # of requested vertices	Max. # of halo + internal edges	Max. # of vertices
`NLosm`	Mondriaan	1 214	1 219	621 406	568 414
	Block	106 707	93 362	684 342	554 172
`AmazonCo`	Mondriaan	37 685	38 929	933 152	106 534
	Block	196 497	97 080	1 631 860	100 849
`Amazon2008`	Mondriaan	87 149	69 892	1 452 958	183 831
	Block	176 049	115 089	1 622 280	183 831

Table 4.6: Properties of the Mondriaan and block distributions for $p = 4$. Provided are the numbers of vertices, halo and requested vertices as well as halo and internal edges combined, each obtained separately as the maximum over the four processors.

distribution in all three cases. This should lead to a (slightly) higher cost incurred by the computation of the messages $\mathbf{m}_v^{t+1}$.

We perform a similar comparison for the two graphs `NLosm` and `AmazonCo` in the three-dimensional case $d = 3$ for four and eight processors. As can be seen in Table 4.7, the algorithm does show the expected behavior for the different distributions. Namely, the Mondriaan distribution leads to a faster running time for both graphs and for both $p = 4$ and $p = 8$.

Since the difference in performance between the block and Mondriaan distributions cannot be indisputably explained by aspects related to the partitioning of the graph, one possible reason for the discrepancy may be that there is an underlying factor in the implementation that creates a large overhead in the execution of the program. Given that the effect was only observed for the two larger graphs `NLosm` and `Amazon2008`, we suspect that issues related to memory access might play a role. For example, the block distribution might benefit from positive cache effects that are not present in the case of the Mondriaan distribution. As the dimension of the vertex states grows, the overhead might then be outweighed by the cost of communication, which is (as argued earlier) much lower for the Mondriaan distribution than for the block distribution.

			Mondriaan distribution	Block distribution
NLosm	$p = 4$	Running time (seconds)	**141.37**	145.78
		Efficiency	86.9%	84.3%
	$p = 8$	Running time (seconds)	**83.56**	86.47
		Efficiency	73.5%	71%
AmazonCo	$p = 4$	Running time (seconds)	**51.85**	67.73
		Efficiency	77.3%	59.1%
	$p = 8$	Running time (seconds)	**29.79**	48.05
		Efficiency	67.2%	41.7%

Table 4.7: Running times and efficiency for the Mondriaan and block distributions for $p = 4, 8$ processors and $d = 3$. Boldface indicates faster running times.

Generally, it should be noted that we expect the added value of parallelizing the message passing algorithm to be higher if the message and update functions are more complex than the GRU-like update (in the sense that they require more computations), and lower if they are less involved. This is because for a fixed graph and a given partitioning of this graph, the amount of communication is the same regardless of the particular approach to the vertex state update. Given that the calculation of the partial derivatives in the backpropagation involves a workload similar to or possibly heavier than that in the GRU update function, parallelizing the network training is expected to yield similar speedups as were observed in our experiments.

5 Conclusion

The goal of this project was to choose and implement a parallel graph neural network model while using the Think Like a Vertex approach, thereby adopting a vertex-centric view on the graph. To this end, we chose a network resembling a neural message passing process and combined this with a vertex update based on the Gated Recurrent Unit. This message passing process was parallelized and an approach to parallelizing the network training was proposed. In particular, we argued that the backward pass in the backpropagation algorithm can be implemented as a message passing process as well. This justifies concentrating on designing an efficient parallel algorithm for the message passing.

The resulting parallel algorithm was implemented in the C++ programming language and uses the C++ library Bulk [20] for writing parallel programs. It was tested on a number of different real-life graphs and with up to eight processors. The numerical experiments show that parallelization of a GNN is worthwhile as the running time quickly increases as the size of the input graph and the number T of time steps in the message passing phase increase. The parallel algorithm displays a reasonable or good performance on all graphs it was tested on, with a peak efficiency of almost 96% for two processors on the Amazon co-purchases graph. Our tests furthermore indicate that the effectiveness of the parallelization decreases for larger values of p, but in turn increases as the dimension d of the input data grows.

A comparison of different partitionings of the graphs led to surprising discrepancies in the running times of the algorithm that cannot be explained with the quality of the partitionings alone. For higher-dimensional input data, however, a communication-reducing Mondriaan distribution outperforms a simple block distribution of the vertices over the processors. This underlines the importance of choosing a suitable partitioning of the input graph which the network is applied to.

Overall, the experimental results for our parallel message passing algorithm look promising, significantly speeding up program execution compared to the sequential version of the algorithm. With some optimizations concerning implementation details and a good choice of partitioning, we expect that our algorithm can be incorporated into a GNN in order to accelerate training and application of the network to large-scale, real-life graphs.

5.1 Future Work

The current project can be taken as a starting point for extensive follow-up work in the area of parallel GNNs, as several questions remain to be answered.

First and foremost, our implementation has yet to be tested with a larger number of processors. Though the experimental results showed an improvement in running time for up to $p = 8$ processors, the efficiency of the parallel algorithm decreased with increasing p. It is therefore necessary to investigate whether this effect carries over to larger values of p, and up to which number of processors it would be worthwhile to use the additional computational resources.

Similarly, the algorithm should be applied to a broader range of graphs with different properties, in particular very large-scale graphs. This is necessary to confirm our observation that the parallelization of the message passing process performs well on large graphs of different sizes and with different structural characteristics. Especially since we are interested in applying the network to real-life graphs which have considerably more vertices and edges than those used in our experiments, it is important to investigate if the algorithm indeed shows comparable performance on such graphs.

Further research is necessary to explain the surprising results involving different vertex distributions and input data dimensions. For one-dimensional vertex states, a graph partitioning based on the block

distribution unexpectedly yielded better results than a Mondriaan distribution aimed at reducing communication. Moreover, the sequential algorithm displayed faster running times in the case $d = 3$, despite more computations being performed. It may be that there are some implementation details that create overhead and lead to suboptimal results on one-dimensional data; for example, in this case the algorithm might benefit from using regular `float` variables instead of the Eigen `ColVector` and `Matrix` types. These aspects therefore need to be investigated further.

Lastly, the parallel message passing algorithm still remains to be adapted to the backward pass in the backpropagation algorithm, and subsequently incorporated in a fully functional GNN that can be trained on a set of training data.

References

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 974–983. Association for Computing Machinery, 2018.

[3] F. Monti, F. Frasca, D. Eynard, D. Mannion, and M. M. Bronstein. Fake news detection on social media using geometric deep learning. *ArXiv*, abs/1902.06673, 2019.

[4] B. Yu, H. Yin, and Z. Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 3634–3640, 2018.

[5] J. M. Stokes, K. Yang, K. Swanson, et al. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702.e13, 2020.

[6] V. Bapst, T. Keck, A. Grabska-Barwinska, et al. Unveiling the predictive power of static structure in glassy systems. *Nature Physics*, 16:448–454, 2020.

[7] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[8] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, page 1263–1272, 2017.

[9] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, page 1025–1035. Curran Associates Inc., 2017.

[10] L. G. Valiant. A bridging model for parallel computing. *Communications of the ACM*, 33(8):103–111, 1990.

[11] 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. *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44, 2020.

[12] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, page 135–146. Association for Computing Machinery, 2010.

[13] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, 2015.

[14] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 48(2), 2015.

[15] T. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, 2017.

[16] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow. Gated graph sequence neural networks. In *Proceedings of ICLR'16*, 2016.

[17] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005.

[18] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP*. Oxford University Press, second edition, 2020.

[19] G. Guennebaud, B. Jacob, et al. Eigen 3.4.0. http://eigen.tuxfamily.org, 2021.

[20] J. Buurlage, T. Bannink, and R. H. Bisseling. Bulk: A modern C++ interface for bulk-synchronous parallel programs. In *Euro-Par 2018: Parallel Processing*, volume 11014, pages 519–532, 2018.

[21] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. `https://networkrepository.com`.