GwAC: GNNs with Asynchronous Communication

Lukas Faber
ETH Zurich, Switzerland
lfaber@ethz.ch

Roger Wattenhofer
ETH Zurich, Switzerland
wattenhofer@ethz.ch

Abstract

This paper studies the relation between Graph Neural Networks and Distributed Computing Models to propose a new framework for Learning in Graphs. Current Graph Neural Networks (GNNs) are closely related to the synchronous model from distributed computing. Nodes operate in rounds and, at the same time, receive aggregated neighborhood information. Our new framework, on the other hand, proposes GNNs with Asynchronous Communication: Every message is received individually and at potentially different times. We prove this framework must be at least as expressive as the existing synchronous framwork. We further analyze GwAC theoretically and practically with regard to several GNN problems: Expressiveness beyond 1-Weisfeiler Lehman (1WL), Underreaching, and Oversmoothing. GwAC shows promising improvements for all problems. We finish with a practical study on how to implement GwAC GNNs efficiently.

1 Introduction

Graph Neural Networks (GNNs) have become the model of choice for applying neural networks to graphs in many domains [8, 21, 23, 26, 28, 53, 60]. Almost all current GNNs follow the message passing framework [6, 21]. In this framework, nodes communicate akin to the synchronous distributed computing model. In synchronous GNN communication, (i) nodes do not receive individual messages of neighbors but an aggregation of all neighborhood messages; (ii) every node acts at the same time and, generally, for the same number of times (although in some GNNs, nodes can elect to drop out from further updates).

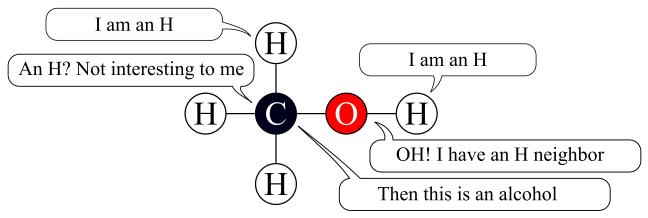


Figure 1: Detection of an alcohol (a C atom with an OH group) with GwAC. The C atom initially messages every neighbor. H neighbors reply, but the C atom discards the replies. The O atom reacts and searches for an H in its neighborhood. If O receives a reply from an H, the molecule is an alcohol.

We argue that synchronous communication is not always ideal. Aggregation (i) may bury a single important message between many irrelevant messages. Such a message becomes the proverbial needle-in-a-haystack for the receiving node to find. Forcing nodes to act the same number of times (ii) can become an issue when the learning problem requires communication over long distances. Consider a line graph $v_1, v_2, \ldots v_n$ where we want to send a message from v_1 to v_n. Every node needs participate for $n-1$ synchronous rounds while it only needs to forward a message once.

Distributed computing knows an antipodal paradigm to the synchronous round-based model: asynchronous communication. In this paradigm, every node reacts to individual messages; at different times; and a different number of times than other nodes. In this paper, we introduce GwAC as a framework for GNNs following such an asynchronous approach. Figure 1 illustrates how such an interaction can play out. The GwAC framework does not aggregate messages, removing potential haystacks (i). Furthermore, nodes can act at different times and a different number of times (ii). We will theoretically and practically show benefits with regards to Expressiveness [19, 61], Underreaching [5], and Oversmoothing [31, 32, 41]. We summarize our contributions as follows:

- We introduce GwAC, a novel framework for GNNs. Nodes follow the asynchronous communication model: They react to individual messages, act at different times, and act a different number of times. We prove that GwAC must be at least as powerful as the synchronous model. We further analyze the complexity to be comparable.
- We theoretically analyze GwAC with regard to expressiveness in the Weisfeiler-Lehman (WL) framework. We prove that GwAC can separate graphs beyond $1-$WL. We experimentally validate these findings on expressiveness benchmarks where even a simple GwAC mode achieves quasi-perfect results. Furthermore, the same model performs competitively against state-of-the-art GNNs on real-world datasets.
- We theoretically compare the efficiency of GwAC and synchronous GNNs for propagating information across many hops in a graph. We show that GwAC has an asymptotic advantage and is independent of distance. We experimentally validate these findings on a reduced shortest path problem. We analyze the results with regard to Underreaching and Oversmoothing and confirm GwAC's rubustness.
- We finish with a theoretical and practicual study on GwAC's efficiency. The complexity is comparable to powerful GNNs. We show a multithreading approach to practically compensate the inability to benefit from GPU acceleration.

2 Related Work

This paper takes inspiration from communication paradigms in distributed computing and their relationship to GNNs. Distributed knows both the synchronous and asynchronous paradigm [43, 57]. In the synchronous approach, all nodes operate in rounds. In every round, every node sends a message to every neighbor. Sato et al.[46] and Loukas et al. [34] show that message passing GNNs [6, 21] follow this framework. Following the initial work of Scarselli et al. [48], different implementations for the individual steps in the message passing framework have been proposed over the years, e.g., [9, 23, 28, 40, 53, 61, 62]. However, past works identified shared problems across the synchronous GNNs:

1-WL Limit. Xu et al. [61] and Morris et al. [39] show that GNNs are limited in their expressiveness by the 1-Weisfeiler-Lehman test (1-WL), a heuristic algorithm to evaluate graph isomorphism [50]. However, there exist simple structures that the 1-WL test cannot distinguish that we want to detect with GNNs [19]. Therefore, several augmentations to GNNs exist that include additional features, such as ports, IDs, overlapping subgraphs, or angles between edges for chemistry datasets [20, 34, 46, 47, 58]. Other methods run multiple rounds over slight perturbations of the same graph [7, 42, 55], or use higher-order information [13, 36, 39]. In the asynchronous GwAC framework, nodes do not act simultaneously. We show this allows to separate graphs beyond $1WL$. Out of scope of this paper, is the study of emerging expressiveness frameworks other than WL, such as Biconnectivity [65], VC [38], or induced subgraph WL [56].

Oversmoothing. A limitation that quickly emerged with GNNs is that of shallow architectures [31, 32]. Each layer averages and smooths the neighborhood information and the node's features. This effect leads to features converging after some layers [41], known as the Oversmoothing problem. Several works address the Oversmoothing problem, for example, by sampling nodes and edges to use in message passing [18, 24, 44], leveraging skip connections [12, 62], or additional regularization terms [11, 66, 67]. We argue the message aggregation aggravates this problem. Aggregation further hides the important information in a haystack of irrelevant messages. In GwAC, listens to individual messages should be more robust against Oversmoothing.

Underreaching. Using normal GNN layers, a GNN with k layers only learns about nodes at most k hops away. A node fails predictions if it needs information that is $k + 1$ hops away. This problem is called Underreaching [5]. There exist countermeasures, for example, having a global exchange of features [21, 59] or spreading information using diffusion processes [29, 48]. We believe the an aggravating issue for Oversmoothing is that every node always acts, even when there is no new information. In GwAC, because of asynchrony, some nodes can be involved in the communication much more often than others; this helps GwAC to gather information from further away, which is a countermeasure against Underreaching.

Oversquashing. In many graphs, the size of k-hop neighborhoods grows substantially with k. Larger neighborhoods require squashing more and more information into a node embedding of static size. Eventually, this leads to the congestion problem (too much information having to pass through a bottleneck) that is well known in distributed computing (e.g. [45]) and goes by the name of Oversquashing for GNNs [2, 52]. One approach to solve Oversquashing is introducing additional edges that function as shortcuts to non-direct neighbors [10]. Dropping-based methods [18, 24, 44] can also reduce Oversquashing by reducing the size of the neighborhoods. If the bottleneck lies in a node embedding, GwAC will struggle similarly. If the bottleneck lies in passing information, GwAC will clearly help against Oversquashing since we pass one message at a time instead of all at once.

There are some works that do not fully adopt the synchronous communication model. Amizadeh et al. [3] propose an architecture for DAGs that sequentially executes the DAG structure. Schaefer et al. [49] employs some asynchrony to decide which nodes in the graph require processing. Martinkus et al. [37] let a few agents walk on the graph and only update nodes where agents are. However, node still update synchronously. We believe that only full asynchrony with individual processing of messages can yield all the advantages, as also observed by Dudzik et al. [15]. As anectodical evidence, similar observations hold for inference in probabilistic graphical models: Elidan et al. [16] observed that inference in such models may not converge if nodes update synchronously. Asynchronous updates improved stability, which also further authors found [1, 30].

3 GwAC: GNNs with Asynchronous Communication

Let us first recap the general framework for GNNs that follow the synchronous communication model, i.e., the message passing framework outlined by Battaglia et al. [6] and Gilmer et al. [21]. Algorithm 1 briefly outlines how this framework computes node embeddings h_v for every node v in a graph G. The initial embeddings h_v^0 can be set to the features of the node in the graph. We can, for example, use the final embeddings for node classification or pool the node embeddings for graph classification.

Algorithm 1: GNNs in the synchronous model.

```
1 repeat $L$ times                                   # $L$ is the number of GNN layers
2     foreach Node $v$ in parallel do
3         $g_v^{i+1} = \texttt{GATE}(h_v^i, \text{Agg}_{w \in NB(v)}\ \texttt{MESSAGE}(h_w^i))$
4         $z_v^{i+1} = \texttt{UPDATE}(h_v^i, \text{Agg}_{w \in NB(v)}\ \texttt{MESSAGE}(h_w^i))$
5         $h_v^{i+1} = g_v^{i+1} \cdot z_v^{i+1} + (1 - g_v^{i+1}) \cdot h_v^i$
```

Algorithm 2: GNNs in the asynchronous model.

```
1 unreceivedMessages = [] ;                              # Tuples (receiver, message)
2 initializeList() ;                              # e.g., a message to single or all nodes
3 repeat $M$ times                                # $M$ is the number of processed messages
4     v, message = unreceivedMessages[0]
5     $g_v^{i+1} = \texttt{GATE}(h_v^i, \text{message})$
6     $z_v^{i+1} = \texttt{UPDATE}(h_v^i, \text{message})$
7     $h_v^{i+1} = g_v^{i+1} \cdot z_v^{i+1} + (1 - g_v^{i+1}) \cdot h_v^i$ newMessage = $\texttt{MESSAGE}(h_v^{i+1}$, message)
8     foreach $w$ in NB(v) do
9         unreceivedMessages.add(w, newMessage)
```

Synchronous GNNs are parametrized by three functions: `message`: is a learnable function to apply to the node embedding before sending. `UPDATE` learns how to combine the neighborhood messages to update the node embedding. `GATE`: This function is optional and not used in every architecture: Nodes can decide that certain information is irrelevant to them, or they can elect to stop participating altogether [14, 33, 51]. We want to highlight two characteristics: (i) every node acts in parallel (line 2), and (ii) nodes only receive an aggregation version of neighbor messages (lines 3+4). These aggregations are usually permutation-invariant, for example, summation or element-wise maximum.

In contrast, look Algorithm 2 shows the GwAC framework for learning embeddings h_v for the same graph G. There are key differences to the synchronous model: The iteration is over individual messages—not rounds—. Processing stops after M messages have been processed (line 3). We process each message with the similar parametrizable functions `MESSAGE`, `UPDATE`, `GATE`. In the asynchronous case, `GATE` could also elect to discard a message and jump to the next iteration. In this model, we also need a temporal order between messages, i.e., which messages we might before before others. Algorithm 2 assumes the simplest model and handles messages chronologically. We will show in Appendix B that a more complex model using a time-ordered heap and non-uniform delays has many theoretical benefits. However, such models do not work as well practically.

3.1 Asynchronous versus Synchronous

Our main result in this section is that asynchronous models in GwAC must be at least as expressive as synchronous message-passing GNNs. We prove this through the notion of *Simulation*. Intuitively, an asynchronous GNN $\mathcal{A}$ that simulates a synchronous one $\mathcal{S}$ can compute every embedding that $\mathcal{S}$ computes. Therefore, we can always replace $\mathcal{S}$ with $\mathcal{A}$ to obtain any embedding. Formally:

Definition 1 (Simulation) *Let $\mathcal{S}$ be a synchronous message passing GNN with l layers and $\mathcal{A}$ an asynchronous GNN in GwAC. We say that $\mathcal{A}$ simulates $\mathcal{S}$ if for all input graphs G and all input features $\boldsymbol{X}$, $\mathcal{A}$ computes node embeddings h_v^l for each layer and node v in G that $\mathcal{S}$ computes.*

We will now demonstrate how to construct $\mathcal{A}$ for a given GNN $\mathcal{S}$. For simplicity, we will construct $\mathcal{A}$ and proof simulation $\mathcal{S} = \text{GIN}$ [61] and extend to other architectures afterward. For a graph G with node features $\boldsymbol{X}$, GIN computes:

$$
\begin{aligned}
h_v^0 &= \boldsymbol{X}_v \\
h_v^{l+1} &= \texttt{UPDATE}_{\mathcal{S}}(h_v^l + \sum_{w \in Nb(v)} \texttt{MESSAGE}_{\mathcal{S}}(h_w^l))
\end{aligned}
$$

We create $\mathcal{A}$ to follow the ideas from distributed computing. Awerbuch [4] presented the α synchronizer, which allows asynchronous models to simulate synchronous ones. The key notion is that of *safe* nodes: A node is safe for one round after it receives a message from every neighbor. A node transfers to the next round if it and all its neighbors are safe. We will use a similar notion and show that we can realize it with little overhead in GNNs:

If the GIN we want to simulate has L layers and an embedding size of d, we create $\mathcal{A}$ to have embeddings of size $2d + 4$ (a linear overhead). The embeddings can are tuples (s, a, w, u, i, l) with the following meaning: **s** the actual current embedding; **a** an accumulator for neighborhood messages; **w** the number of missing pulses; **u** the number of unsafe nodes; **i** if the node never acted so far; and **l** the number of layers left to simulate. Node embeddings are initialized to $(\boldsymbol{X}_v, 0, D-1, D, 1, L)$ where D is the node's degree.[1]

Messages are tuples $(m, \texttt{ack}, \texttt{pulse}, \texttt{origin}, \texttt{noop})$ where the last four entries are bits one-hot encoding the message type: `origin` used for the first message to start the execution; `pulse` messages are only sent by safe nodes to transmit the next round's information. The remaining messages are necessary to synchornize the nodes. `ack` acknowledge the receipt of neighborhood messages; and `noop` Messages containing no information, `GATE` will ignore those. Table 1 shows the `UPDATE` and `MESSAGE` function for $\mathcal{A}$. We can interpret the table as a series of cascading `if`-statements where we apply the first matching condition. The first part m of the message is always computed as $\texttt{MESSAGE}_{\mathcal{S}}$(s). The table only shows which message type we set. The appendix contains an example execution in Appendix A.

[1]We take the node degrees to be available. This does not leak information: Having degrees simplifies the proof but does not leak information to $\mathcal{A}_{\mathcal{G}}$. Nodes in $\mathcal{A}$ can easily find their degree by flooding the network once and storing the number of replies in a state entry that is never modified again.

Table 1: `UPDATE` and message type to send

Condition	s'	a'	w'	u'	i'	l'	message type
`noop`=1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
l=0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
`origin`=1	s	a	w	D	0	l	`pulse`
i=1	s	a+m	w-1	u	0	l	`pulse`
`pulse`=1 $\wedge$ w=0	UPDATE([s,(a+m)])	0	D-1	u-1	0	l-1	`ack`
`pulse`=1	s	a+m	w-1	u	0	l	`noop`
`ack`=1 $\wedge$ u=0	s	a	w	D	0	l	`pulse`
`ack`=1	s	a	w	u-1	0	l	`noop`

Lemma 1 *When node v emits its i-th `pulse` message, it must have already received $(i-1) \cdot D + 1$ pulse or origin messages.*

Corollary 1 *Nodes require D `pulse` messages between their own (Lemma 1). Thus, nodes receive one `pulse` from every neighbor.*

We provide a proof by induction in Appendix C. The base case is true after initialization, and we show that a node needs to receive D additional `pulse` messages to emit another pulse. If each node requires D pulses, no node can message their neighbor twice.

Lemma 2 *When node v emits its i-th `pulse` message, it's state s equals the synchronous GNN representation h_v^{L-i-1}.*

Appendix C shows an inductive proof. The base state h_l^0 is true by initialization. We show that after receiving a pulse by every neighbor, GwAC correctly computes the next layer's state.

Lemma 3 (Synchronous Simulation) *$\mathcal{A}$ simulates GIN, where one round of `pulse` messages maps to one synchronous layer.*

This proof follows from Lemma 2. With every `pulse` message, nodes in $\mathcal{A}$ proceed to the next embedding h_v^l of GIN.

3.2 Other Simulation Scenarios

Disconnected Graphs The proof can be extended to disconnected graphs, for example, by sending one `origin` message per connected component.

Variable layer sizes Let us assume the GNN does have different embeddings sizes $d_1, d_2, \ldots d_L$ per layer. In such a case $\mathcal{A}$ would have a state space consisting of $s_0, a_0, s_1, a_1, \ldots s_L, a_L, w, u, i, l$. Updates to the state would not overwrite but write to the next state. This approach also allows us to model different message functions per layer or skip connections, for example.

Max Aggregation Changing to another associative aggregation function such as `max` is straightforward but changing the accumulator updates to that aggregation.

Mean Aggregation We can also easily simulate mean aggregation: When computing the update after all pulses are received we divide by the node's degree.

GCN Aggregation GCN multiplies messages by neighboring nodes' degrees. We can make sender degrees part of the message and use the `GATE` function to replicate this scaling.

GAT Aggregation Unfortunately this architecture cannot be simulated. GAT scales every incoming message with an attention factor that is dependant on all messages.

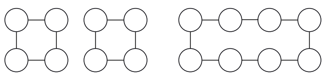
4 Expressiveness

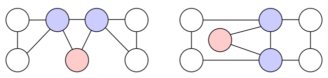
4.1 Theoretical Analysis

We investigate the expressiveness of GwAC in the Weisfeiler-Lehman (WL) framework. Our first result follows directly from Lemma 3.

Corollary 2 (1WL) *Models following the GwAC framework generally are at least as powerful as the 1 Weisfeiler-Lehman test.*

We now show that asynchronous GwAC models are even more expressive than 1-WL. GwAC models can count cycle lenghts and separate graphs as in Figure 2a or 2b, and can separate cliques of different sizes 2c.


(a) Taken from Garg et al [19]


(b) From Papp et al. [42], added red node.

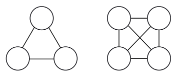

(c) $3-$ versus $4-$ clique

Figure 2: a) GwAC can separate the two graphs by identifying the $8-$ cycle and b) by identifiying a $3-$ cycle with one blue node (Lemma 4. GwAC must not start from the red node to be able to separate the graphs. c) GwAC can separate the graphs (Lemma 5). This Lemma also allows to separate Rook-Shrikande graphs which are beyond $3-$WL.

Lemma 4 *In GwAC, nodes $v_1, v_2, \dots v_k$ can determine if they are a k cycle.*

We discuss the proof in Appendix D. The idea is that a starting node can start a search along both branches of the cycle and infer the cycle length when they meet.

Lemma 5 *In GwAC, nodes $v_1, v_2, \dots v_k$ can determine if they are a k clique.*

We proof the Lemma in Appendix D. The idea is that the k nodes iteratively find out they are all in a $2, 3, 4, \dots, k$ cliques.

4.2 Expressiveness Experiments

We experimentally validate our theoretical findings on existing GNN expressiveness benchmarks[2]. We try node classification (Limits1 [19], Limit2 [19], Triangles [47], LCC [47]) and graph classification (4-cycles [34], Skip-Cycles [13], Rook-Shrikande [13]). Furthermore, we test models if they can separate certain graphs from Xu et al. [61] if we limit aggregation to `max` or `mean`. We mant to measure the effectiveness of the asynchronous communication. To this end, we only use simple linear projections for the`MESSAGE` and `UPDATE` functions and no gating. We will refer to this model as GwAC-S. We compare GwAC-S with against beyond 1-WL GNNs from literature: PPGN [36], SMP [55],[3] DropGNN [42],[4] and ESAN [7].[5], AgentNet [37], plus a GIN [61] for control. Appendix E.1 contains details on the hyperparameters and Table 2 the results.

Already the simple GwAC-S consistently solves all datasets (close to) perfectly and on the same level as the recently proposed powerful AgentNet. The other methods struggle on at least some datasets. In particular, GwAC-S solves the Skip-Cycles dataset perfectly, which requires long-range information propagation, as well as the Rook-Shrikande graphs that need power beyond $3 - WL$. The results on MAX and MEAN also showcase another benefit of not aggregation messages, where other methods with unfitting aggregations struggle.

Table 2: GwAC-S solves all beyond $1-$WL benchmarks quasi-perfect even the challenging ones require long-range propagation (Skip-Cycles) or beyond $3-$WL reasoning (Rook-Shrikande)

Dataset	GIN	PPGN	SMP	DropGNN	ESAN	AgentNet	GwAC-S
Limits1	0.50±0.00	0.60±0.21	0.95±0.16	**1.00±0.00**	**1.00±0.00**	N/A	**1.00±0.00**
Limits2	0.50±0.00	0.85±0.24	**1.00±0.00**	**1.00±0.00**	**1.00±0.00**	N/A	**1.00±0.00**
Triangles	0.52±0.15	**1.00±0.02**	0.97±0.11	0.93±0.13	**1.00±0.01**	N/A	**1.00±0.01**
LCC	0.38±0.08	0.80±0.26	0.95±0.17	**0.99±0.02**	**0.96±0.06**	N/A	**0.96±0.03**
MAX	0.05±0.00	0.36±0.16	0.74±0.24	0.27±0.07	0.05±0.00	N/A	**1.00±0.00**
MEAN	0.28±0.31	0.39±0.21	0.91±0.14	0.58±0.34	0.18±0.08	N/A	**1.00±0.00**
4-cycles	0.50±0.00	0.80±0.25	0.60±0.17	**1.00±0.01**	0.50±0.00	**1.00±0.00**	**1.00±0.00**
Skip-Cycles	0.10±0.00	0.04±0.07	0.27±0.05	0.82±0.28	0.40±0.16	**1.00±0.00**	**1.00±0.00**
Rook-Shrikande	0.50±0.00	0.50±0.00	0.50±0.00	0.50±0.00	**1.00±0.00**	**1.00±0.00**	**1.00±0.00**

[2]Code for all experiments available at `https://github.com/lukasjf/gwac/`

[3]Code for SMP and PPGN from `https://github.com/cvignac/SMP`

[4]Code for GIN and DropGNN from `https://github.com/KarolisMart/DropGNN`

[5]Code for ESAN from `https://github.com/beabevi/ESAN`

4.3 Graph Classification Experiments

We also run GwAC-S on several graph classification benchmarks: MUTAG, PTC, PROTEINS, IMDB-B, IMDB-M [64]. Table 3 shows the classification accuracy for GwAC-S and other GNN methods for these datasets. We obtain results using the protocol from Xu et al. [61]: We report the test accuracy over the best epoch of a $10-$ fold split. Biological datasets use 16 or 32 hidden units; social datasets use 64. For GwAC-S, we very the number of messages per starting node (15 or 25) and test a variant of skip connections that allows nodes to consider more than only their last state. GwAC-S achieves results that comparable to other powerful GNN models.

Table 3: Graph classification accuracy ($\%$). Lower GNNs can separate graphs beyond 1-WL. GwAC-S produces competitive results. *multiple versions, the score is the best-performing model

Model	MUTAG	PTC	PROTEINS	IMDB-B	IMDB-M
GraphSAGE [23]	90.4±7.8	63.7±9.7	75.6±5.5	76.0±3.3	51.9±4.9
GCN [28]	88.9±7.6	79.1±11.4	76.9±4.8	83.4±4.9	57.5±2.6
GAT [53]	85.1±9.3	64.5±7.0	75.4±3.8	74.9±3.8	52.0±3.0
GIN [61]	89.4±5.6	66.6±6.9	76.2±2.6	75.1±5.1	52.3±2.8
1-2-3 GNN [39]	86.1	60.9	75.5	74.2	49.5
DropGNN [42]	90.4±7.0	66.0±9.8	76.3±6.1	75.7±4.2	51.4±2.8
PPGN [36]*	90.6±8.7	66.2±6.5	77.2±4.7	73±5.8	50.5±3.6
ESAN [7]*	92.0±5.0	69.2±6.5	77.3±3.8	77.1±2.6	53.7±2.1
AgentNet [37]	93.6±8.6	67.4±5.9	76.7±3.2	75.2±4.6	52.2±3.8
GwAC-S	90.4±4.1	63.7±9.1	76.7±7.1	74.6±3.6	52.1±3.6

5 Underreaching and Oversmoothing.

5.1 Theoretical Analysis

Underreaching and Oversmoothing come into play when we want to propagate information through many hops in the graph. In the synchronous framework, this requires computing many synchronous rounds, for example, through many GNN layers. Let G be a directed graph with n nodes and m edges, and we want to pass information from node u to node v. Let d be the distance from u to v. A synchronous GNN model generally needs to perform at least d rounds of computation to pass the information (Shortcut edges which help tremendously but only if there is no important information along the way). In the simple case without a `GATE` function, a GNN sends dm messages. For example, gating can help nodes lying on the path from u to v that forwarded the message can decide to terminate. However, nodes that did not yet send their message need to keep a steady state. In the synchronous framework, this means sending and receiving messages. Worst case, we still need $O(dm)$ messages. Generally, d depends on n and grows with graph size.

Lemma 6 *Let G be a graph with n nodes and m edges. GwAC can send a message from any node u to any node v in $O(m)$ messages.*

Proof 1 *The following protocol sends a message from u to v. Any number of nodes $w_1, w_2, \ldots w_n$ may receive an initial message `seek`. Every node $w \neq u$ will forward a `seek` message exactly once. When u receives a `seek` message it emits a `send` message instead. Every node also forwards `send` messages exactly once. In the worst case, we have one `seek` and one `send` message traveling over every edge, which is $2m$ messages total. The actual number may be lower: If nodes receive a `send` message first, they do not need to propagate `seek` messages.*

Importantly, the number of messages is independent of d. When graphs and distances become larger, GwAC is clearly more message-efficient to propagate information. The messages that GwAC saves do not carry any information which might smooth important messages (Oversmoothing), which means it is easier to message far away nodes (Underreaching).

5.2 Experiments

Let us experimentally validate this observation by computing shortest path lengths from a node s. Computing shortest paths is a commonly-used benchmark [51, 54, 63] that needs to propagate information to the entire graph. We reduce the shortest path problem to emphasize on the propagation:

Table 4: Accuracy for predicting the parity of shortest paths to a starting node. Columns show different test graph sizes. Training size was 10 nodes. GwAC models outperform synchronous baseline, even the simpler GwAC-S without termination logic.

Model	10	25	50	100	250	500	1000
GCN	0.54±0.07	0.50±0.03	0.50±0.01	0.50±0.01	0.50±0.01	0.49±0.01	0.49±0.00
GAT	0.64±0.07	0.52±0.03	0.51±0.02	0.50±0.00	0.50±0.01	0.50±0.00	0.49±0.01
GIN	0.97±0.01	0.80±0.05	0.59±0.04	0.52±0.03	0.50±0.01	0.49±0.00	0.50±0.00
NEG	0.73±0.14	0.59±0.11	0.53±0.07	0.52±0.04	0.51±0.03	0.49±0.02	0.50±0.01
Universal	0.87±0.03	0.71±0.04	0.62±0.03	0.56±0.02	0.52±0.01	0.51±0.00	0.50±0.00
IterGNN	0.98±0.03	0.86±0.05	0.74±0.03	0.65±0.04	0.57±0.01	0.53±0.01	0.51±0.00
GwAC-S	0.99±0.00	0.91±0.05	0.82±0.07	0.76±0.10	0.64±0.10	0.58±0.13	0.53±0.10
GwAC-UT	1.00±0.00	0.99±0.00	0.98±0.02	0.97±0.03	0.96±0.05	0.95±0.05	0.94±0.06
GwAC-Iter	1.00±0.00	0.99±0.01	0.99±0.01	0.98±0.03	0.98±0.05	0.97±0.05	0.97±0.05

We make edge lengths integers, and every node predicts if its distance to s is even or odd. This binary problem removes the need for arithmetics, which can be tricky to learn properly [17, 25, 35]. We train on small graphs and increase graph size during inference. Larger graphs pose more challenging problems since the diameter grows, and information needs to travel longer distances. We provide the exact data creation and model parameters in Appendix E.2.

As baselines, we run GCN [28], GAT [54], GIN [61], NEG (Neural Execution of Graph Algorithms) [54], UT (Universal Transformers [14]), and IterGNNs [51].[6] We reuse the simple GwAC-S from the previous set of experiments, as well as conversions of UT and IterGNNs into the asynchronous framework. We call these GwAC-UT and GwAC-Iter, respectively. UT and GwACT-UT use the idea of adaptive computation time [22] for `GATE`. IterGNNs and GwAC-Iter use a similar idea, but combine gates multiplicatively instead of additively [51].Table 4 shows test set accuracies by test graph size for all architectures. All GwAC models perform consistently better than all synchronous architectures, confirming the hypothesis that asynchronous communication does indeed help for propagating information over long distances. Let us try to understand these results better in the context of Underreaching and Oversmoothing.

To investigate Underreaching, we measure accuracy split by the nodes' distances to the starting node. If we see a drop in accuracy with increasing distance, we can suspect Underreaching. At some distance, models collapse and constantly predict 0 or 1. Therefore, we bucket distances in pairs, for example, 1 and 2. Figure 3a shows the accuracy per distance. Here we see that GwAC outperforms synchronous methods, which drives the differences in Table 4. To investigate Oversmoothing, we limit the evaluation to *close* nodes. We define close nodes as nodes having a distance that also exists in the training set, which models generally solve correctly. When we increase the graph size, the model requires more computation to propagate information to nodes further away. A model suffers from Oversmoothing if information in the embeddings of close nodes is smoothed away and the embeddings stop being informative. A constant accuracy for close nodes suggests resistance to Oversmoothing. Most models show strong resistance against Oversmoothing 3b.

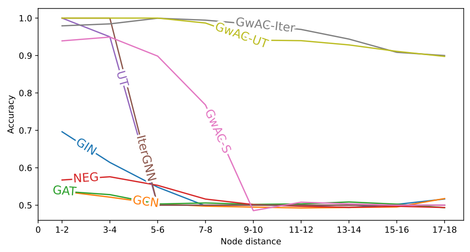
6 Runtime analysis

6.1 Complexity

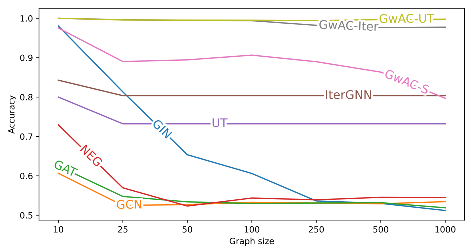
Let us compare the complexity of GNNs (Algorithm 1) and GwAC as in Algortihm 2. We measure complexity as number of messages and node updates. Computational complexity equals node update complexity (plus a constant factor) in case of uniform messages and message complexity when we learn messages based on sender plus receiver. Let G be a graph with n nodes and maximum degree D. Per layer, a synchronous GNN sends a message for every edge and updates every node. For L layers, the GNN sends $O(LnD)$ messages and does $O(Ln)$ node updates. L is usually a constant.

More expressive GNN variants, such as ESAN [7] or DropGNN [42], perform multiple computations on slightly perturbed variants on the graph. ESAN computes at least n permutations, removing one node or edge per permutation or restricting the graph to a node's ego-net. DropGNN computes r permutations, each with randomly dropped nodes. The overall complexity is $O(LnDr)$ messages and

[6]Code for IterGNN from `https://github.com/haotang1995/IterGNN`



(a)



(b)

Figure 3: (a) Accuracy per distance for the shortest path parity problem. Declining accuracy suggests Underreaching. (b) Accuracy for nodes with a training set distance for different graph sizes. Declining accuracy suggests Oversmoothing.

$O(Lnr)$ node updates. In an asynchronous GNN, we process M messages. Each processed message does one node update and sends up to D messages for totals of $O(MD)$ messages and $O(M)$ node updates. M should be in $\Omega(n)$ to obtain informative representations. For example, for Table 2, we used $M = 5n^2$ which is comparable to ESAN. For Table 3, we used linearly many messages with $M = 15n$ or $M = 25n$, which is comparable to simple GNNs.

6.2 Implementation Considerations

While complexity of GwAC is comparable to synchronous GNNs we found running times to be much slower. To a large extent, we can attribute this to missing library suppport. GNNs can leverage existing libraries that lift most of the computation into libraries written in fast programming languages. On the other hand our prototypical implementation runs mostly in Python. But there is also a systemic disadvantage to GwAC. Synchronous execution in GNNs, allows to batch and parallelize computation and benefit from GPU hardware acceleration. The asynchronous communication model does not lend itself as obviously to parallelization. However, parallelization is still possible: We can execute Algorithm 2 in parallel on different graphs. We implement a proof of concept in Python and show the runtime versus available parallelism for the datasets from Section 4.3 in Figure 4. Runtime is inversely proportional to the number of available processes (within bounds because of synchronization overhead). However, the runtime is still not comparable to current GNNs and libraries.

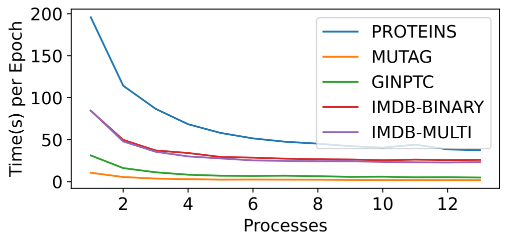


Figure 4: Runtime (y-axis) is inversely proportional to the number of processes avialable for GwAC (axis): doubling compute halves the computation time.

7 Conclusion, Limitations, and Future Work

We presented a new GNN framework: GwAC. GwAC proposes one answer how to implement asynchronously communicating GNNs. GwAC treats every message individually so that no information is lost in aggregation. Furthermore, every node can, in principle, act at different times and a different number of times. We showed theoretically and practically that GwAC's communication benefits Expressiveness, Underreaching, and Oversmoothing.

The current version of GwAC struggles with high-feature node classification tasks like Cora because it strongly overfits. In addition to improving libraries and therefore runtime, promising future work should investigate how we can *learn* GwAC models well. For example, batch normalization often helps to stabilize learning, but what would be a notion of batches in the asynchronous framework? Anoother interesting question could be how to effectively transfer residual information or how to realize dropout-based methods.

References

[1] Aksenov, V., Alistarh, D., Korhonen, J.H.: Scalable belief propagation via relaxed scheduling. Advances in Neural Information Processing Systems (2020) 3

[2] Alon, U., Yahav, E.: On the bottleneck of graph neural networks and its practical implications. In: International Conference on Learning Representations (ICLR) (2021) 3

[3] Amizadeh, S., Matusevych, S., Weimer, M.: Learning to solve circuit-sat: An unsupervised differentiable approach. In: International Conference on Learning Representations (ICLR) (2018) 3

[4] Awerbuch, B.: Complexity of network synchronization. Journal of the ACM (1985) 4

[5] Barceló, P., Kostylev, E., Monet, M., Pérez, J., Reutter, J., Silva, J.P.: The logical expressiveness of graph neural networks. In: International Conference on Learning Representations (ICLR) (2020) 2, 3

[6] Battaglia, P.W., Hamrick, J.B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al.: Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261 (2018) 1, 2, 3

[7] Bevilacqua, B., Frasca, F., Lim, D., Srinivasan, B., Cai, C., Balamurugan, G., Bronstein, M.M., Maron, H.: Equivariant subgraph aggregation networks. In: International Conference on Learning Representations (ICLR) (2022) 2, 6, 7, 8

[8] Bian, T., Xiao, X., Xu, T., Zhao, P., Huang, W., Rong, Y., Huang, J.: Rumor detection on social media with bi-directional graph convolutional networks. In: AAAI conference on artificial intelligence (AAAI) (2020) 1

[9] Brody, S., Alon, U., Yahav, E.: How attentive are graph attention networks? In: International Conference on Learning Representations (ICLR) (2022) 2

[10] Brüel-Gabrielsson, R., Yurochkin, M., Solomon, J.: Rewiring with positional encodings for graph neural networks. arXiv preprint arXiv:2201.12674 (2022) 3

[11] Chen, D., Lin, Y., Li, W., Li, P., Zhou, J., Sun, X.: Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In: AAAI Conference on Artificial Intelligence (AAAI) (2020) 2

[12] Chen, M., Wei, Z., Huang, Z., Ding, B., Li, Y.: Simple and deep graph convolutional networks. In: International Conference on Machine Learning (ICML) (2020) 2

[13] Chen, Z., Chen, L., Villar, S., Bruna, J.: On the equivalence between graph isomorphism testing and function approximation with gnns. In: Conference on Neural Information Processing Systems (NeurIPS) (2019) 2, 6

[14] Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., Kaiser, L.: Universal transformers. In: International Conference on Learning Representations (ICLR) (2018) 4, 8

[15] Dudzik, A., von Glehn, T., Pascanu, R., Veličković, P.: Asynchronous algorithmic alignment with cocycles. arXiv preprint arXiv:2306.15632 (2023) 3

[16] Elidan, G., McGraw, I., Koller, D.: Residual belief propagation: Informed scheduling for asynchronous message passing. arXiv preprint arXiv:1206.6837 (2012) 3

[17] Faber, L., Wattenhofer, R.: Neural status registers. arXiv preprint arXiv:2004.07085 (2020) 8

[18] Feng, W., Zhang, J., Dong, Y., Han, Y., Luan, H., Xu, Q., Yang, Q., Kharlamov, E., Tang, J.: Graph random neural networks for semi-supervised learning on graphs. In: Conference on Neural Information Processing Systems (NeurIPS) (2020) 2, 3

[19] Garg, V., Jegelka, S., Jaakkola, T.: Generalization and representational limits of graph neural networks. In: International Conference on Machine Learning (ICML) (2020) 2, 6

[20] Gasteiger, J., Groß, J., Günnemann, S.: Directional message passing for molecular graphs. In: International Conference on Learning Representations (ICLR) (2020) 2

[21] Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. In: International Conference on Machine Learning (ICML) (2017) 1, 2, 3

[22] Graves, A.: Adaptive computation time for recurrent neural networks. arXiv preprint arXiv:1603.08983 (2016) 8

[23] Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Conference on Neural Information Processing Systems (NeurIPS). vol. 30 (2017) 1, 2, 7

[24] Hasanzadeh, A., Hajiramezanali, E., Boluki, S., Zhou, M., Duffield, N., Narayanan, K., Qian, X.: Bayesian graph neural networks with adaptive connection sampling. In: International Conference on Machine Learning (ICML) (2020) 2, 3

[25] Heim, N., Pevny, T., Smidl, V.: Neural power units. In: Conference on Neural Information Processing Systems (NeurIPS) (2020) 8

[26] Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., et al.: Highly accurate protein structure prediction with alphafold. Nature (2021) 1

[27] Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: International Conference on Learning Representations (ICLR) (2015) 20

[28] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (ICLR) (2017) 1, 2, 7, 8

[29] Klicpera, J., Weißenberger, S., Günnemann, S.: Diffusion improves graph learning. In: Conference on Neural Information Processing Systems (NeurIPS) (2019) 3

[30] Knoll, C., Rath, M., Tschiatschek, S., Pernkopf, F.: Message scheduling methods for belief propagation. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD (2015) 3

[31] Li, G., Muller, M., Thabet, A., Ghanem, B.: Deepgcns: Can gcns go as deep as cnns? In: IEEE international conference on computer vision (ICCV). pp. 9267–9276 (2019) 2

[32] Li, Q., Han, Z., Wu, X.M.: Deeper insights into graph convolutional networks for semi-supervised learning. In: AAAI Conference on Artificial Intelligence (AAAI) (2018) 2

[33] Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. In: International Conference on Learning Representations (ICLR) (2016) 4

[34] Loukas, A.: What graph neural networks cannot learn: depth vs width. In: International Conference on Learning Representations (ICLR) (2020) 2, 6, 16

[35] Madsen, A., Johansen, A.R.: Neural arithmetic units. In: International Conference on Learning Representations (ICLR) (2020) 8

[36] Maron, H., Ben-Hamu, H., Serviansky, H., Lipman, Y.: Provably powerful graph networks. In: Conference on Neural Information Processing Systems (NeurIPS) (2019) 2, 6, 7

[37] Martinkus, K., Papp, P.A., Schesch, B., Wattenhofer, R.: Agent-based graph neural networks. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net (2023) 3, 6, 7

[38] Morris, C., Geerts, F., Tönshoff, J., Grohe, M.: Wl meet vc. arXiv preprint arXiv:2301.11039 (2023) 2

[39] Morris, C., Ritzert, M., Fey, M., Hamilton, W.L., Lenssen, J.E., Rattan, G., Grohe, M.: Weisfeiler and leman go neural: Higher-order graph neural networks. In: AAAI Conference on Artificial Intelligence (AAAI) (2019) 2, 7

[40] Niepert, M., Ahmed, M., Kutzkov, K.: Learning convolutional neural networks for graphs. In: International conference on machine learning (ICML) (2016) 2

[41] Oono, K., Suzuki, T.: Graph neural networks exponentially lose expressive power for node classification. In: International Conference on Learning Representations (ICLR) (2020) 2

[42] Papp, P.A., Martinkus, K., Faber, L., Wattenhofer, R.: Dropgnn: random dropouts increase the expressiveness of graph neural networks. In: Conference on Neural Information Processing Systems (NeurIPS) (2021) 2, 6, 7, 8, 20

[43] Peleg, D.: Distributed computing: a locality-sensitive approach. SIAM (2000) 2

[44] Rong, Y., Huang, W., Xu, T., Huang, J.: Dropedge: Towards deep graph convolutional networks on node classification. In: International Conference on Learning Representations (ICLR) (2020) 2, 3

[45] Sarma, A.D., Holzer, S., Kor, L., Korman, A., Nanongkai, D., Pandurangan, G., Peleg, D., Wattenhofer, R.: Distributed verification and hardness of distributed approximation. SIAM Journal on Computing (2012) 3

[46] Sato, R., Yamada, M., Kashima, H.: Approximation ratios of graph neural networks for combinatorial problems. In: Conference on Neural Information Processing Systems (NeurIPS) (2019) 2

[47] Sato, R., Yamada, M., Kashima, H.: Random features strengthen graph neural networks. In: SIAM International Conference on Data Mining (SDM) (2021) 2, 6

[48] Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE transactions on neural networks (2008) 2, 3

[49] Schaefer, S., Gehrig, D., Scaramuzza, D.: Aegnn: Asynchronous event-based graph neural networks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2022) 3

[50] Shervashidze, N., Schweitzer, P., Van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-lehman graph kernels. Journal of Machine Learning Research (2011) 2

[51] Tang, H., Huang, Z., Gu, J., Lu, B.L., Su, H.: Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. Conference on Neural Information Processing Systems (NeurIPS) (2020) 4, 7, 8

[52] Topping, J., Di Giovanni, F., Chamberlain, B.P., Dong, X., Bronstein, M.M.: Understanding over-squashing and bottlenecks on graphs via curvature. In: International Conference on Learning Representations (ICLR) (2022) 3

[53] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. In: International Conference on Learning Representations (ICLR) (2018) 1, 2, 7

[54] Velickovic, P., Ying, R., Padovano, M., Hadsell, R., Blundell, C.: Neural execution of graph algorithms. In: International Conference on Learning Representations (ICLR) (2020) 7, 8

[55] Vignac, C., Loukas, A., Frossard, P.: Building powerful and equivariant graph neural networks with structural message-passing. In: Conference on Neural Information Processing Systems (NeurIPS) (2020) 2, 6

[56] Wang, Q., Chen, D.Z., Wijesinghe, A., Li, S., Farhan, M.: N-WL: A New Hierarchy of Expressivity for Graph Neural Networks. In: The Eleventh International Conference on Learning Representations, ICLR, Kigali, Rwanda (May 2023) 2

[57] Wattenhofer, R.: Mastering Distributed Algorithms. Inverted Forest Publishing (2020) 2

[58] Wijesinghe, A., Wang, Q.: A new perspective on" how graph neural networks go beyond weisfeiler-lehman?". In: International Conference on Learning Representations (2021) 2

[59] Wu, Z., Jain, P., Wright, M., Mirhoseini, A., Gonzalez, J.E., Stoica, I.: Representing long-range context for graph neural networks with global attention. In: Conference on Neural Information Processing Systems (NeurIPS) (2021) 3

[60] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE transactions on neural networks and learning systems (2020) 1

[61] Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations(ICLR) (2019) 2, 4, 6, 7, 8

[62] Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K.i., Jegelka, S.: Representation learning on graphs with jumping knowledge networks. In: International Conference on Machine Learning (ICML) (2018) 2

[63] Xu, K., Zhang, M., Li, J., Du, S.S., Kawarabayashi, K.i., Jegelka, S.: How neural networks extrapolate: from feedforward to graph neural networks. In: International Conference on Learning Representations (ICLR) (2021) 7

[64] Yanardag, P., Vishwanathan, S.: Deep graph kernels. In: ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD) (2015) 7

[65] Zhang, B., Luo, S., Wang, L., He, D.: Rethinking the expressive power of gnns via graph biconnectivity. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023 (2023) 2

[66] Zhao, L., Akoglu, L.: Pairnorm: Tackling oversmoothing in gnns. In: International Conference on Learning Representations (ICLR) (2020) 2

[67] Zhou, K., Huang, X., Li, Y., Zha, D., Chen, R., Hu, X.: Towards deeper graph neural networks with differentiable group normalization. In: Conference on Neural Information Processing Systems (NeurIPS) (2020) 2

A An Example of Asynchronous Simulation

Let us look at an example to understand the function in Table 1 better. Let us look at the example graph
$G = (V = \{v_1, v_2, v_3, v_4\}, E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_4, v_2\}, \{v_4, v_3\}\})$. We will track the node states in tabular form.

Message	State of v_1	State of v_2	State of v_3	State of v_4
$\perp$	(s_1,0,1,2,1,L)	(s_2,0,1,2,1,L)	(s_3,0,1,2,1,L)	(s_4,0,1,2,1,L)

Now we send the first message (of type `origin`) to node v_2, which updates as follows and sends a `pulse` message in return.

$v_2 \leftarrow$ `origin`		(s_2,0,1,2,0,L)		

Let us assume that v_1 receives this message first. The state i is still true so we evaluate with the fourth equation. This leads to the following update and v_1 also sends a `pulse` message.

$v_1 \leftarrow$ `pulse`(v_2)	(s_1,s_2,0,2,0,L)			

Let us now assume that v_2 receives this message (and the message from v_2 to v_4 is still in transit). Unlike v_1, the node v_2 has already received one message before, so i is false and v_2 updates differently to the following state. Furthermore v_2 now sends a `noop` instead of a `pulse`. This message will be ignored by every node.

$v_2 \leftarrow$ `pulse`(v_1)		(s_2,s_1,0,2,0,L)		

Next, let us assume v_3 receives the `pulse` message from v_1. This is v_3's the first message and v_3 will also emit a `pulse` message in return. The message from v_2 to v_4 is still in transit.

$v_3 \leftarrow$ `pulse`(v_1)			(s_3,s_1,0,2,0,L)	

Now, v_1 may receive this `pulse` message by v_3. Since this is the last missing message from a neighbor, v_1 now computes $s'_1 = UPDATE(s_1, s_2 + s_3)$, the same update as the synchronous GIN. Further v_1 now sends an `ack` message.

$v_1 \leftarrow$ `pulse`(v_3)	(s'_1,0,1,1,0,L)			

Nodes v_2 and v_3 receive this `ack` message each and update to

$v_2 \leftarrow$ `ack`(v_1)		(s_2,s_1,0,1,0,L)		
$v_3 \leftarrow$ `ack`(v_1)			(s_3,s_1,0,1,0,L)	

Let the message from v_2 to v_4 arrive next. Then v_4 send `pulse` messages and transitions to:

$v_4 \leftarrow$ `pulse`(v_2)				(s_4,s_2,0,2,0,L)

Node v_3 receives this pulse from v_4 and transitions to the following state and sending `ack` messages. Next, v_1 receives this `ack` message. At this step, only an `ack` from v_2 is left before v_1 can proceed to the next iteration.

$v_3 \leftarrow$ `pulse`(v_4)			(s'_3,0,1,0,0,L)	
$v_1 \leftarrow$ `ack`(v_3)	(s'_1,0,1,0,0,L)			

So let us assume that v_2 receives the `pulse` message from v_4 next, and updates and sends an `ack`:

$v_2 \leftarrow$ `pulse`(v_4)		(s'_2,0,1,0,0,L)		

Node v_1 now receives this `ack` message and proceeds to simulate the next GIN layer. It decrements the layer counter and emits a `pulse` for its neighbors. Since these neighbors already received all data (but not all `ack` messages) from the previous iteration they can safely store this message in the accumulator. For example, let v_2 receive the `pulse` from v_1.

$v_1 \leftarrow$ `ack`(v_2)	(s'_1,0,1,2,0,$L-1$)			
$v_2 \leftarrow$ `pulse`(v_1)		(s'_2,s'_1,0,0,0,$L-1$)		

For the network to proceeed further, v_4 needs to receive the `pulse` from v_3, so it can complete the first layer simulation and send `ack`. This will allow v_2 and v_3 to start simulation the next layer.

B GwAC with Random Message Delays

Algorithm 3: GNNs in the asynchronous model.

```
openMessages = [] ;                          # Triples (delay, receiver, message)
initializeList() ;                           # e.g., a message to single or all nodes
repeat $M$ times                             # $M$ is the number of processed messages
    delay, v, message = openMessages[0]
    $g_v^{i+1} = \texttt{GATE}(h_v^i)$, message
    $z_v^{i+1} = \texttt{UPDATE}(h_v^i$, message)
    $h_v^{i+1} = g_v^{i+1} \cdot z_v^{i+1} + (1 - g_v^{i+1}) \cdot h_v^i$ newMessage = MESSAGE($h_v^{i+1}$, message)
    foreach $w$ in NB(v) do
        openMessage.add(delay + randomDelay(), w, newMessage)
```

We will now look at a variant of GwAC where message delays are not uniform but random. Algorithm 3 shows an adapted algorithm where we use a delay-sorted heap instead. These delays can act as symmetry breakers in symmetric graphs like star graphs. The problem we need to solve is bringing order in the equal neighbors of center node v, which we do by exploiting the random delays.

We connect the neighbors of v.[7] When v `offers` an ID, nodes without an ID will reply to try to `claim` it. Some nodes will receive this reply before they receive the message from v. These nodes will `surrender` this ID. After some attempts to offer the ID, all but one node will surrender. Then v gives the ID to the one non-surrendering node (implicitly by starting to offer the next ID). Then the protocol restarts for the next ID (minus the nodes that are `done` and have an ID). We will transform this idea into a network:

We have four states, `offering` (which is only used by v), `claiming` and `surrendering` (which are used by the outer nodes), and `done` used by all nodes that no longer need to participate. We have five message types `offer`, `confirm`, `claim`, `surrender`, `origin`, and `noop`.

Nodes have states that are tuples $(\texttt{state}, \texttt{try}, \texttt{ID}, \texttt{c}, \texttt{w}, \texttt{x})$: `state` is one of above states, `try` identifies different ID assignment attempts from each other, and `ID` is the ID of the node. The following three entries are only used by central node v: `c` a binary feature if the ID was already claimed in the current attempt, `w` contains the number of neighbors in the current attempt that did not reply yet, `x` contains the number of neighbors that do not have an ID yet. Initially, every node starts in the state (`surrendering`, -1, -1, 0, 0, 0).

Messages are tuples $(\texttt{type}, \texttt{CID}, \texttt{attempt})$: `type` is one of the above messages types, `CID` is a candidate ID that v tries to currently assign, `attempt` is the current attempt to assign an ID.

Table 5 shows the node update and message functions in GwAC for the center node, Table 6 for the outer nodes. There is actually only one function. Nodes judge by their state if they are the center node or node. For readability, we split them here. If the reaction function ρ decides to ignore a message, the tables show a row of blanks ($\perp$).

Lemma 7 *GwAC with the functions in Table 5 and 6 can create unique identifiers (IDs) for every node in a star graph.*

Proof 2 *We prove that (i) that no two nodes receive the same identifier and (ii) every node receives an identifier.*

(i) The center node takes ID 0 for itself and only proposes IDs of 1 and above to its neighbors. If a neighbor does not change its ID, it stays with -1. Thus ID 0 is unique. Suppose that two nodes w_1, w_2 received the same ID. Since IDs are increasing w_1 and w_2 must have received the ID at the same time. Since the `confirm` message is only sent when there was exactly one node left, w_1 and w_2 must have received the ID via the second to last condition in Table 6a. This is only possible if they

[7]We can do without those edges since v could propagate messages between outer nodes, but they make the idea more concise and the algorithm faster

Table 5: `UPDATE` (a) and `MESSAGE` function (b) for the central node v in GwAC to assign unique IDs in a star graph.

	(a)						**(b)**		
Condition	state	try	ID	c	w	x	type	CID	attempt
noop	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
done	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
try $\neq$ attempt	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
origin	offering	0	0	0	D-1	D-1	offer	1	0
claim $\wedge$ c = 0	offering	try	0	1	w-1	x	noop	$\perp$	$\perp$
claim $\wedge$ c $>$ 0	offering	try+1	0	0	x-1	x	offer	CID	try+1
surrender	offering	try	0	c	w-1	x	noop	$\perp$	$\perp$
w = 0 $\wedge$ x $>$ 0	offering	try+1	0	0	x-2	x-1	offer	CID+1	try+1
x = 0	done	0	0	0	0	0	confirm	$\perp$	$\perp$

Table 6: `UPDATE` (a) and `MESSAGE` function (b) for the outer nodes in GwAC to assign unique IDs in a star graph.

	(a)						**(b)**		
Condition	state	try	ID	c	w	x	type	CID	attempt
noop	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
done	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
surrendering $\wedge$ offer $\wedge$ CID=ID	surrendering	attempt	ID	$\perp$	$\perp$	$\perp$	surrender	CID	attempt
offer $\wedge$ attempt $>$ try	claiming	attempt	CID	$\perp$	$\perp$	$\perp$	claim	CID	attempt
claiming $\wedge$ attempt $>$ try	surrendering	attempt	CID	$\perp$	$\perp$	$\perp$	surrender	CID	attempt
claiming $\wedge$ offer $\wedge$ ID $\neq$ CID	done	0	ID	$\perp$	$\perp$	$\perp$	noop	$\perp$	$\perp$
claiming $\wedge$ confirm	done	0	ID	$\perp$	$\perp$	$\perp$	noop	$\perp$	$\perp$

were in state `claiming`, which means that they both send a `claim` message. However, in such cases, v would have started another try with that ID.

(ii) The center node receives ID 0. We have to show all other nodes also receive an ID. Node v counts internally how many IDs it could offer successfully and only stops when this counter reaches 0. We initialize the counter with v's degree. Therefore, v will offer sufficiently many IDs. Let us suppose that w did not receive an ID. Node w will always try to claim an ID unless it is already surrendering this ID. Therefore w surrenders for all IDs, even the last one. However, this is not possible since there is only w left, and no node could have sent a `claim` message to make w surrender.

We can generalize this proof from star graphs to arbitrary connected graphs. We start with any node that assigns IDs in its 1-hop neighborhood (a star graph). Next, the node with the next-highest ID repeats the process — skipping nodes that already have an ID. Eventually every node is adjacent to a center node and receives an ID. The algorithm stops after every node assigned IDs, and finding a successor is not longer possible.

We can further extend this algorithm to unconnected graphs by initally adding a master node that is connected to every node. We can conclude:

Lemma 8 *GwAC with random message delays can create IDs in arbitrary graphs.*

We can now define the expressiveness of GwAC with random message delays.

Lemma 9 *(Graph Isomorphism) Given two connected graphs G_1, G_2 that we connect via a node u to a graph G and diameter δ_G, GwAC with random message delays that has a width in $O(\frac{n}{2}!n^2)$ can solve graph isomorphism for any graphs G_1, G_2 simulating an appropriate GNN for δ_G many rounds.*

Proof 3 *We are reducing Lemma 9 to Corollary 3.1 from Loukas [34]. The author shows that if we have (i) unique identifiers for every node in a graph and (ii) we have a message passing GNN that has no bound in width and is at least as deep as the graph's diameter δ_G, this GNN is Turing* *universal and can compute any function on this graph (including graph isomorphism).*

Let G_1, G_2 be two connected graphs for which we want to test graph isomorphism. We create the graph G by adding a unique node u[8] that we connect to a random node from G_1 and G_2 each. Further, let $\mathcal{G}$ be a GNN of unbounded width that has at least δ_G many layers. If the GNN has access to IDs, this GNN is Turing universal and can compute any computable function. Therefore, this GNN can compute if G_1 and G_2 are isomorphic. Let us consider an GwAC model with unbounded width, which can compute graph isomorphism on G as follows: First, we employ Lemma 9 to assign every node a unique identifier. Then we leverage the synchronous Simulation Lemma 3 and simulate G with the identifiers for δ_G many rounds. Whatever representations $\mathcal{G}$ finds to be able to compute graph isomorphism between G_1 and G_2, GwAC can compute the same representations by simulation. Thus, GwAC can also compute graph isomorphism on any graph G.

Finally, let us tackle the unbounded width requirement. We are not interested in any algorithm on G, but only in determining graph isomorphism. We can estimate the required width for this problem as follows: One example algorithm to compute graph algorithms is to use the δ_G layers to build a complete representation of G in every node. If we use adjacency matrices, this requires width $O(n^2)$. Now we enumerate all possible assignments π from nodes in G_1 to G_2 — of which there are $O(\frac{n}{2}!)$ many. For each candidate, we write the permuted adjacency matrix, requiring $O(\frac{n}{2}!n^2)$ space. If any permuted adjacency matrix equals that of G_1, the graphs are isomorphic, otherwise, they are not. We can check this by subtracting the matrices and comparising against 0. This check requires constant width for each of the $O(\frac{n}{2}!n^2)$ candidates. We can conclude that the GNN can solve graph isomorphism on G with $O(\frac{n}{2}!n^2)$ width. This is the same bound for GwAC that needs only a constant factor in state and message size for simulation.

Despite the impressive theoretical results, we found that uniform delays work better in practice. Random delays cause different executions with different embeddings on repeated runs over the same graphs. This creates noisy gradients and destabilizes training. Let GwAC-SR be a variant of GwAC-S that uses random and not uniform delays. Table 7 compares the results of GwAC-SR with the results of GwAC-S on the datasets in Table 2. We can see that GwAC-S trains stably and consistently performs better.

Table 7: Comparing GwAC-S with a random delay version GwAC-SR on several expressiveness benchmarks. The random delays create too much noise. The non-random GwAC-S consistently performs better.

Dataset	GwAC-S	GwAC-R
Limits1	**1.00**±**0.00**	0.89±0.11
Limits2	**1.00**±**0.00**	**0.98**±**0.01**
Triangles	**1.00**±**0.01**	0.88±0.15
LCC	**0.96**±**0.03**	0.80±0.04
MAX	**1.00**±**0.00**	0.37±0.10
MEAN	**1.00**±**0.00**	0.71±0.11
4-cycles	**1.00**±**0.00**	**0.99**±**0.02**
Skip-Cycles	**1.00**±**0.00**	0.56±0.18
Rook-Shrikande	**1.00**±**0.00**	0.50±0.50

[8]We can also start the ID assignment in GwAC at node u, this node would always have ID 0 without loss of generality.

C Simulation Proofs

C.1 Proof for Lemma 1

We are first need to proof an additional Lemma:

Lemma 10 *Nodes receive a `pulse` message when executing the $i = 1$ condition.*

Proof 4 *Messages of type `noop` and `origin` are captured by a preceding condition. A node u could only receive an `ack` message from a neighbor v after v received a `pulse` message from every neighbor, including u. By elimination the message type must be `pulse`.*

Now we can proof Lemma 1 via induction over i.

Proof 5 *We start with $i = 1$, i.e., when nodes emit their first `pulse` message. After initialization, all nodes will send a `pulse` message after receiving one `origin` (third condition) or `pulse` message (Lemma 10).*

Suppose the lemma holds for node v that just sent its i-th pulse. Before v another pulse, it must reach the seventh condition in Table 1. For this, u must be decremented to 0, which requires reaching the fifth condition. In turn, this requires decrementing w to zero for which v must receive $D - 1$ `pulse` messages in the sixth condition and one further one in the fifth. In total, v has now received $(i - 1) \cdot D + 1 + D = i \cdot D + 1$ `pulse` messages.

C.2 Proof for Lemma 2

Proof 6 *We prove this by induction. The lemma holds for $i = 1$, when the nodes state s is $\boldsymbol{X}_v = h_v^0$ based on initialization.*

Suppose the lemma holds for node v that just sent its i-th pulse. We know from Lemma 1, that v now needs to receive D `pulse` messages (Lemma 1), one from each neighbor(Corollary 1) before it can send `pulse` $i + 1$. According to the induction hypothesis, node v receives z_w^{l-i-1} from every neighbor w in their ith `pulse` message. Upon receiving the last of these messages, v computes locally UPDATE($z_v^{l-i-1}, \sum_w z_w^{l-i-1}) = z_v^{l-i}$ following the fifth condition. This state is unchanged until v emits `pulse` $i + 1$.

D Expressiveness Proofs

D.1 Proof for Lemma 4

Proof 7 *Let $v_1, v_2, \dots v_k$ be a cycle of k nodes and let computation start from node v. Every node executes the following protocol. If a node receives a message $\texttt{COUNT}$-i and it never received a message before, it stores i and messages $\texttt{COUNT}$-$(i+1)$. If the same node then receives a $\texttt{COUNT}$-j message with $j > i$, the node ignores the message. Node v starts the computation by sending $\texttt{COUNT}$-0. The nodes on both paths store numbers as in a BFS from v. Eventually, the two BFS branches meet when one node w that first received a $\texttt{COUNT}$-i message receives a $\texttt{COUNT}$-j message with $i \geq j$. Then, w knows the circle is closed and sends a $\texttt{FOUND}$-$(j+i)$ that every node forwards once. Thus, all nodes become aware of the cycle and its length.*

D.2 Proof for Lemma 5

Proof 8 *Let $v_1, v_2, \dots v_k$ be in a k clique. The nodes iteratively find out they are in $2, 3, \dots k+1$ cliques. The starting node v will coordinate the other nodes. Initially, every node stores that they are in a 1-clique. To find a clique of size j, node v sends a $\texttt{CLIQUE}$-j message, which every neighbor of v forwards once. If neighbors of v receive $j-1$ such messages and are in a $j-1$ clique according to their state, they send a $\texttt{CLIQUE}$-j-$\texttt{ACK}$ message to all neighbors (including v) and update their state to be in a j-clique. If v receives k many $\texttt{CLIQUE}$-j-$\texttt{ACK}$ messages, it sends out a $\texttt{CLIQUE}$-$(j+1)$ message. Upon receiving k many $\texttt{CLIQUE}$-$(k+1)$-$\texttt{ACK}$ messages, v knows there exists a $(k+1)$-clique and can propagate this information to its neighbors.*

E Experiment Details

E.1 Expressiveness Benchmarks

We follow the training setup by Papp et al. [42] and use four layers of synchronous GNNs. For Skip-Cycles, we additionally try 9 layers and take the better result. For GwAC, we allow a total of $5n$ messages, with n being the size of the graph. Similar to DropGNN or SMP, we execute multiple runs for GwAC, starting with each node once. Each run computes the final embedding for the starting node. For all architectures, we use 16 hidden units for Limits1, Limits2, Triangles, LCC, and 4-cycles; and 32 units for MAX, MEAN, and Skip-Cycles. We use the Adam [27] optimizer with a learning rate of 0.01 and train for 1000 epochs.

E.2 Long-Range Communication

We create 100 graphs with $n = 10$ nodes. The base of each graph is a random spanning tree to which we add $(\frac{n}{5})$ random extra edges. We randomly pick a starting node s. We mark s for synchronous GNNs or send it the initial GwAC message. We learn over 1000 epochs with the Adam optimizer with a learning rate of 0.01 and use embeddings sizes of 30. For testing, we sample 10 graphs with $10, 25, 50, 100, 250, 500,$ and 1000 nodes, respectively.