PARALLEL GRAPH ATTENTION NETWORKS

Xinyue Chen, Boxuan Li
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{xinyuech,boxuanli}@cs.cmu.edu

1 BACKGROUND AND MOTIVATION

Graph neural networks, as a kind of modelling for structured data, have been brought to people's attention in recent years. Popular applications include node classification, and graph classification. GNNs are also integrated in other downstream tasks, such as named entity recognition, visual question answering, to help understand structured information inherent to the tasks. A few popular variants of GNNs include: graph convolution networks (GCNs (Kipf & Welling, 2016)), relational graph convolution networks (RGCNs (Schlichtkrull et al., 2017)) and graph attention networks (GATs (Veličković et al., 2017)).

The idea of GNNs is basically as follows. There is a feature tensor for each node, and each layer k of GCN has a learnable matrix parameter W_k. Each node calculates its outgoing message by applying linear transformation to its feature tensor $\{h_{k-1}\}$ with W_k and pass the message to all neighbors. Then each node aggregates the incoming messages and uses them to update its own feature tensor. This completes the message passing of a single layer. And the updated feature tensors $\{h_k\}$ are ready for use as inputs to the next layer.

In GCNs, the aggregation is instantiated as average pooling over the incoming messages. GATs further incorporate attention mechanism for aggregation. Specifically, we introduce a weight for each incoming message (from node j) specific to the receiving node i, by calculating the affinity between $W_k h_{k-1}^i$ and $W_k h_{k-1}^j$. The weights are then normalized as the coefficients for pooling.

We could also do a step further and introduce multi-head attention mechanism (Vaswani et al., 2017), where within each layer we use multiple W_k and process nodes with different W_k in parallel. Then we concatenate the resultant updated feature tensors as the canonical outputs of this layer.

We believe we could explore parallelism of multiple dimensions in the GAT model, and experiment with different parallelism toolkits and schemes to study the empirical benefits of parallelism.

2 SCOPE

In this project, we implement the forward pass of GATs using C++. We have a complete sequential version, including the support for multi-layer GATs, linear layers, and different activation functions. The OpenMP version has the same functionality as the sequential version. Our CUDA version only support single-layer GATs. In addition, we have a vertex program that supports multi-layer GATs as the sequential version.

For OpenMP and vertex program versions, we use Protein-Protein Interactions (PPI) dataset. For CUDA experiment, we randomly generate graphs of different edge densities and node scales. Also, we utilize two standard citation network benchmark datasets, Cora and Pubmed Sen et al. (2008).

3 IMPLEMENTATIONS

3.1 SEQUENTIAL VERSION

We illustrate the sequential implementation in Alg. 1. There are four major stages that are sequentially dependent. PREPAREMESSAGE calculates all the outgoing messages for each node and each attention head. HEATCALCULATION prepares the information for attention calculation. ATTNCALCULATION computes pair-wise attentions for each attention head. And finally GENERATEFEATURES concatenates the features generated from each attention head, where each head conducts for each node i attentive pooling over the incoming messages from a node's neighbors.

We use COO sparse matrix for node connectivity information. So, empirically, we only do softmax and aggregate messages over the non-zero entries in the COO sparse matrix. This significantly boosts efficiency for sparse input graphs.

To mitigate numerical unstability in the softmax operation, we extract the maximum value from the vector before applying exponential function.

Initially, we implement a more naive version, where every time we compute an attention score between two nodes, we would compute their heats respectively. In later version, we cache the heats beforehand.

Complexity analysis. Let n be the number of nodes, m be the average node degree, d_{in} be the input node feature dimension, d_{msg} be the message dimension, and k be the number of attention heads. We illustrate the time complexity in Table 3.1. When the graph is sparse, the first stage dominates the computation time.

Algorithm 1 Sequential Forward Pass

procedure PREPAREMESSAGE
  for each head k **do**
    for each node i **do**
      msg$[k][i] \leftarrow W^k h_i$
    end for
  end for
  return msg
end procedure
procedure HEATCALCULATION
  for each head k **do**
    for each node i **do**
      self_heat$[k][i] \leftarrow a_{self}^T \cdot$msg$[k][i]$
      other_heat$[k][j] \leftarrow a_{other}^T \cdot$ msg$[k][i]$
    end for
  end for
  return self_heat, other_heat
end procedure
procedure ATTNCALCULATION
  for each head k **do**
    for each node i **do**
      attn$[k][i] \leftarrow$ Softmax(LeakyReLU(self_heat$[k][i]$+ other_heat$[k][j]$)) $\triangleright$ $j \in N_i$
    end for
  end for
  return attn
end procedure
procedure GENERATEFEATURES
  for each head k **do**
    for each node i **do**
      $h_i^k \leftarrow \sum_{j \in N_i}$ attn$[k][i][j] \cdot$ msg$[k][j]$ $\triangleright$ multiplication between a scalar and a vector
    end for
  end for
  for each node i **do**
    $h_i \leftarrow$ CONCAT $[\{h_i^k\}]$
  end for
  return $\{h_i\}$
end procedure

Table 1: Time complexity in sequential implementation.

PREPAREMESSAGE	HEATCALCULATION	ATTNCALCULATION	GENERATEFEATURES
$\mathcal{O}(knd_{in}d_{msg})$	$\mathcal{O}(knd_{msg})$	$\mathcal{O}(knm)$	$\mathcal{O}(knmd_{msg})$

3.2 CUDA VERSION

For the CUDA version, we parallelize the four stages of our sequential algorithm.

PREPAREMESSAGE. This stage is implemented as block-wise matrix multiplication between the input features and the weight matrix, $F_{n\times d_{in}} \cdot W_{d_{in}\times(k\cdot d_{msg})}$. The block width is set to 32. Within each block, the threads load a submatrix of the two matrices into the shared memory to utilize cache locality.

HEATCALCULATION. We assign the heat calculation of node-head pair to each thread and parallelize over the nodes and attention heads. We allocate shared memory for a_{self} and a_{other} of the attention of interest to utilize cache locality. The block size is set as 2×32.

ATTNCALCULATION. Again we assign the attention calculation of node-head pair to each thread and parallelize over the nodes and attention heads. The block size is set as 32×32.

GENERATEFEATURES. We parallelize each entry of the output feature matrix $F_{n\times(k\cdot d_{msg})}$. The block size is set as 32×32.

3.3 OPENMP VERSION

OpenMP implementation can be found in the same folder as C++ sequential version. We use a compiler flag to control whether the C++ source files will be compiled to a sequential version or an OpenMP version.

3.3.1 HEAD-WISE PARALLELISM

A typical attention module usually repeats its computation multiple times in parallel, thus is also known as "multi-head attention". Computations for different heads follow the same computation paths and are not interdependent, thus could easily be parallelized. In head-wise parallelism strategy, we use OpenMP to parallelize different heads. Note that the number of heads is usually a small number (in our model, it's a number ranging from 4 to 6), which makes this strategy only suitable when number of threads is not large.

The first naive approach we tried was to simply add "#pragma omp parallel for" before the beginning of every outmost for-loop. This leads to 2x speedup (on a 8-core machine) which is far from ideal. Clearly, this naive approach does not consider data dependency. As a consequence, it even leads to a

wrong result. The sequential version achieves 0.965841 micro F1 score on the PPI (Protein-Protein Interactions) dataset, which is exactly the same figure achieved by the oracle PyTorch implementation. The naive OpenMP approach, however, leads to only 0.27 micro F1 score, indicating this approach leads to completely wrong results. After removing inappropriate parallelism and adding critical sections, we get back the correct results with only 30% speedup on a 8-core machine.

To exploit the parallelism better, we revised the code logic. Without surprise, we found out the original sequential implementation could be improved. In this commit, we changed the way we compute softmax for attention values. In machine learning, to find the softmax value for an array of numerical values, it is very common to subtract all of them by their max value, such that all the values are less than or equal to zero. Then, we apply exponential function to them and divide each by the sum of exponents. The subtraction step does not change the theoretical result, but is very useful and important in practice because it is better for numerical stability considering the nature of floating point representation and computation in modern computer architecture. In the oracle PyTorch implementation, the subtraction preprocessing step applies globally using a single line of code "scores_per_edge = scores_per_edge - scores_per_edge.max()". In our C++ implementation, to save memory, we computed the attention values (i.e. "scores_per_edge") on the fly. As a consequence, to compute the max, we had to calculate the attention values twice, leading to unnecessary computation cost. To eliminate redundant computations, we subtracted attentions using a neighborhood-aware approach in commit 581cce. That is, we subtracted attentions by the max attention value in their neighborhood. This led to better numerical stability, and greatly reduced computation cost. The number of cache references dropped drastically from 76440020 to 40689622 for the PPI dataset. As a consequence, the computation time for the sequential version dropped by 21.59%, but the parallell speedup remained the same. One small pitfall is that the accuracy dropped from 0.965841 to 0.924718, but we are confident that this is because the model parameters were trained in the original way of computing softmax. Due to the way the PyTorch version is implemented, it is cumbersome to align it to our C++ version and retrain the model. We leave it as an exercise if readers are interested.

We then try more parallelism strategies. In other words, we try parallelize different sections and find best regions to parallelize. In this commit, we achieve 0.35s using 8 threads on an 8-core machine, which is 4.2x faster than the sequential version tested on the same machine. We then optimize the code logic further by removing unnecessary data structures (e.g. we removed an array and used one local variable instead) and reducing computation steps (e.g. rather than calculate attention for each neighbor and then find max attention, we only calculate max heat from neighborhoods and then use max heat to calculate max attention). This leads to a slight improvement on performance, but overall speedup remains roughly the same.

3.3.2 NODE-WISE PARALLELISM

Head-wise parallelism does not scale when the number of threads exceeds number of heads. When we use a powerful machine or even a super computer where we have 64x or even 128x threads, we have to use a different parallelism strategy. In this section, we discuss the node-wise parallelism strategy we adopt.

Converting the program from head-wise parallelism to node-wise parallelism is straight-forward. In many places, we have nested for-loops in the following format:

```
for (int i = 0; i < num_heads; i++) {
    for (int j = 0; j < num_nodes; j++) {
        // do stuff
    }
}
```

To leverage node-wise parallelism, we change the loop order into the following sequence:

```
for (int j = 0; j < num_nodes; j++) {
    for (int i = 0; i < num_heads; i++) {
        // do stuff
    }
}
```

Then we add OpenMP pragma before the outer loop, and we are done.

3.4 VERTEX PROGRAM

This folder contains a Java implementation of Graph Attention Network (inference only), leveraging Apache TinkerPop Tin's vertex program framework. The trained network parameters come from a PyTorch implementation and we have verified that our vertex program achieves same accuracy compared to the inference conducted via PyTorch.

The nature of vertex program treats every vertex as an individual entity of the entire graph. The program can be executed in a distributed manner and scale well. Apache TinkerPop, as one of the most popular graph computing frameworks adopted by a variety of graph databases including Neo4j, Amazon Neptune, and JanusGraph, supports different execution backends for vertex programs.

The vertex program can be launched from a TinkerPop's gremlin console as follows:

```
graph = TinkerGraph.open()
g = graph.traversal()
```

```
g.io("/path/to/ppi-v3d0.kryo").read().iterate()
graph = g.getGraph()
result = graph.compute().program(GraphAttentionNetworkVertexProgram
.build().create()).submit().get().graph()
```

The above code snippet runs the vertex program locally with multi-threading. To run it on a Spark cluster, we just need to follow the TinkerPop official guidelines, configure the Spark environment and submit the job using "SparkGraphComputer":

```
result = graph.compute(SparkGraphComputer)
.program(GraphAttentionNetworkVertexProgram
.build().create()).submit().get().graph()
```

In both cases, the results are the same. In other words, the underlying execution engine does not impact the result but the latency.

4 EXPERIMENTS

4.1 CUDA VERSION

The experiment is conducted in a CMU GHC machine, an 8-core Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz machine.

In the section, we make use of Cora and Pubmed (statistics given in Table 4.1). The density is the ratio between the number edges in the dataset and the number of edges in a fully connected graph with the same number of nodes. We also randomly generate graphs with the number of nodes being 400, $1,200$ and $5,000$ nodes, and density between 0.01, 0.1 and 0.3, combinatorially.

Table 2: Dataset statistics of Cora and Pubmed.

Dataset	**#Nodes**	**#Edges**	**Avg. Degree**	**Density (d)**	d_{in}
Cora	2,708	13,264	4.9	0.18%	1433
Pubmed	19,717	108,365	5.5	0.03%	500

We run five trials for all experiments and give their mean and standard deviation. For all experiments we set $k = 8$ and $d_{msg} = 64$. The reported runtime includes all the kernel runtime and the memory copying from host to device or device to host.

For the first set of experiments, we compare the sequential and CUDA runtime of Cora, Pubmed, and our synthetics graphs, and compute the speedups. For Pubmed, we obey the dataset setup and set d_{in} as 500, for Cora, $d_{in} = 1433$. For the synthetic datasets, we set $d_{in} = 1433$. The results are shown in

Fig. 4.1. Overall, as the number of nodes n grows larger, we could achieve higher speedup. The graph density also affects the speedup with respect to the total number of edges, because we use sparse matrix and only compute for the non-zero entries. But the influence of the number of edges is much less than that of the number of nodes. This is because empirically, the first matrix multiplication stage dominates the computation. To verify this hypothesis, we calculate the runtime for the five stages respectively. As shown in Table 4.1, the first stage accounts for ~ 90% of the runtime.

Table 3: Runtime comparisons on all datasets. All numbers are in milliseconds (ms).

	Cora	**Pubmed**	-
Seq	9854.37 ± 3.21	25225.43 ±6.58	-
CUDA	147.38 ± 3.83	292.96 ± 4.57	-
Speedup	66.86x	98.11x	-
	n = 400, $\mathbf{d}$ = $1e^{-2}$	n = 400, $\mathbf{d}$ = $1e^{-1}$	n = 400, $\mathbf{d}$ = $3e^{-1}$
Seq	1456.28 ±1.77	1490.41 ±1.39	1565.20 ±0.30
CUDA	74.64 ±0.66	281.00 ±3.00	74.74 ±0.42
Speedup	19.51x	18.40x	20.94x
	n = 1200, $\mathbf{d}$ = $1e^{-2}$	n = 1200, $\mathbf{d}$ = $1e^{-1}$	n = 1200, $\mathbf{d}$ = $3e^{-1}$
Seq	4390.29 ±2.93	4702.63 ±0.53	5409.46 ±17.87
CUDA	110.44 ±2.79	111.38 ±4.32	113.19 ±3.24
Speedup	39.75x	42.22x	47.79x
	n = 5000, $\mathbf{d}$ = $1e^{-2}$	n = 5000, $\mathbf{d}$ = $1e^{-1}$	n = 5000, $\mathbf{d}$ = $3e^{-1}$
Seq	18734.43 ±61.71	25116.16 ±1640.34	37327.55 ±929.92
CUDA	222.57 ±2.65	233.73 ±0.31	288.29 ±5.72
Speedup	84.17x	107.46x	129.48x

Table 4: Runtime breakdown of a CUDA trial, with n = 5000 and $\mathbf{d}$ = $1e^{-1}$, measured in milliseconds (ms).

TOTAL	PREPAREMESSAGE	HEATCALCULATION	ATTNCALCULATION	GENERATEFEATURES
275.53	183.28	0.08	7.41	6.72

Finally, we study the impact of d_{in} upon runtime. We compare Table 4.1 and the third section of Table 4.1. Clearly smaller d_{in} results in smaller speedup.

Table 5: Runtime with d_{in} = 50 and n = 1200.

	$\mathbf{d}$ = $1e^{-2}$	$\mathbf{d}$ = $1e^{-1}$	$\mathbf{d}$ = $3e^{-1}$
Seq	190.58 ±0.69	505.57 ±0.75	1200.27 ±1.74
CUDA	61.03 ±2.16	62.93 ±3.64	64.32 ±2.85
Speedup	3.12x	8.03x	18.66x

4.2 OpenMP Version

4.2.1 head-wise Parallelism

The experiment is conducted in a CMU GHC machine, an 8-core Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz machine.

Number of Threads	Time	Speedup
1x	1.4975s	1x
4x	0.5224s	2.87x
6x	0.3532s	4.24x
8x	0.3508s	4.27x

We can see that the program scales well when the number of threads is smaller than or equal to 6. This is because the max number of attention heads is 6. Adding more threads does not help here. The fact that 8x threads perform slightly better than 4x threads is due to the activation function parallelism. In activation function, we parallelize the computations by nodes, partly because there is no presence of ”head” in activation step.

4.2.2 node-wise Parallelism

The experiment is conducted in a Pittsburgh Supercomputing Center (PSC) machine.

Number of Threads	Time	Speedup
1x	1.5825s	1x
4x	0.4303s	3.68x
16x	0.1469s	10.77x
64x	0.0814s	19.44x
128x	0.1959s	8.08x

We also record the elapsed time for the critical section in the code: 0.043 seconds. According to Amdahl's law, the theoretical maximum speedup that can be achieved using N processors is: S(N) = 1/((1-P)+(P/N)), where P is the portion of the program that can be made parallel. In our case, P = 0.9728. Therefore, S(64) = 23.58, meaning that the maximum speedup we can achieve for 64 threads is 23.58, which is close to the value we achieve in the experiment.

Interestingly, 128x is much slower than 64x threads and slightly slower than 16x threads. Due to the lack of profiling tool on PSC machine, we cannot analyze cache references and cache misses, but our hypothesis is that the increasing number of threads causes more and more cache contentions and misses.

4.3 Vertex Program

Running the vertex program is significantly slower than running the sequential version due to a couple of facts. First, running a vertex program has additional overheads including graph partitioniong, task

allocation, message passing. Second, a vertex program uses Bulk Synchronous Parallel (Valiant, 1990) model, meaning that all computations must finish before next iteration can start. Third, the dataset is too small with only 3k nodes and 100k edges.

The major benefits of vertex programs include: 1. No need to dump graph data for neural network computation. Users can run GAT inferences on any TinkerPop-enabled graph database directly, as opposed to dumping graph data to somewhere else for computation. 2. Scalability. With a huge graph, the vertex program can be run on many computation nodes.

We don't have enough time to create a gigantic graph and benchmark the GAT computation on a Spark cluster, but it is certainly possible and is left as future work.

REFERENCES

Apache tinkerpop. https://tinkerpop.apache.org/.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2016. URL https://arxiv.org/abs/1609.02907.

Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017. URL https://arxiv.org/abs/1703.06103.

Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008. URL http://www.cs.iit.edu/~ml/pdfs/sen-aimag08.pdf.

Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8): 103–111, 1990.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017. URL https://arxiv.org/abs/1710.10903.

5 CONTRIBUTION

Joint efforts: Sequential version, dataset, report

Xinyue Chen: CUDA version

Boxuan Li: OpenMP version, vertex program

Overall credit distribution is 50% - 50%.