15-418/618
Course Project: Parallelized Graph Convolutional Networks Proposal

Liwei Cai (Andrew ID liweicai)
Chengze Fan (Andrew ID chengzef)

April 9, 2019

Abstract

In this project, we will implement parallelized Graph Convolutional Networks running on multi-core CPU, including a multithreaded version using OpenMP and possibly a distributed version using MPI. Our implementation will include both forward computation and backward propagation for training and inference. We will then analyze the scaling behavior of our implementation and compare its performance to the reference TensorFlow implementation.

1 Background

Analyzing and understanding graph data like social networks, citation networks and protein interaction networks are of great practical interest and thus become a popular research topic. In recent years, deep learning have achieved great success in learning from data with relatively simple structures, like tabular data, images and texts. However, designing the appropriate neural network for graph-structured data have long been a challenge.

Graph Convolutional Networks (GCN), proposed by Kipf and Welling (2016) [1], is one of the first successful attempts in applying deep neural networks to arbitrary graphs.

The original paper applied GCN to the task of *semi-supervised node classification*, where a graph and features of each node are given, but only a small portion of nodes are labeled with a class, and the remaining nodes are left for classification. Other algorithms for machine learning tasks on graphs, like label propagation (Zhu and Ghahramani, 2002) and DeepWalk (Perozzi et al., 2014), have been implemented as projects of this course in previous semesters.

Following the success of GCN, many variants of it have been proposed, and similar mathematical principles have been used in designing convolutional networks for manifolds. However, since the focus of this project is on parallel computing rather than machine learning, we decided to follow its original and simplest form.

[1] Also see the author's blog post for a high-level explanation.

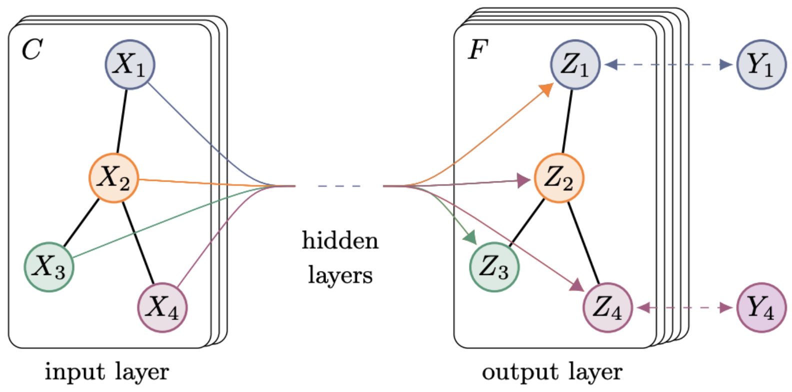

Figure 1: GCN architecture, taken from Figure 1(a) of Kipf and Welling (2016). Schematic depiction of multi-layer Graph Convolutional Network (GCN) for semi-supervised learning with C input channels and F feature maps in the output layer. The graph structure (edges shown as black lines) is shared over layers, labels are denoted by Y_i

2 Computational Model of GCN

We extract the computational model of GCN from Kipf and Welling (2016). The whole process is also shown in Figure 1.

There exists a graph that has an adjacency matrix A. We first make some transformation independent of nodes' information and get $\hat{A}$.

we further assume that Each of the nodes in a graph has a feature vector X of length C and output vector Z of length F. Y_i is the label of node i. The forward model for inference can be expressed as:

$$Z = f(X, A) = \text{softmax}(\hat{A}\ \text{ReLU}(\hat{A}XW^{(0)})W^{(1)}) \tag{1}$$

Here, $W^{(0)} \in \mathbb{R}^{C\times H}$ is an input-to-hidden matrix and $W^{(1)} \in \mathbb{R}^{H\times F}$ is an hidden-to-output matrix. These two matrices are the parameters we are supposed to train.

The loss is a cross-entropy error over all labeled examples and the function is shown as follows:

$$L = -\sum_{l\in X}\sum_{f=1}^{F} Y_{lf} \ln Z_{ln} \tag{2}$$

where X is the set of nodes that has labels.

Each training iteration is performed over the labeled nodes and tries to minimize the loss using batch gradient descent in the original paper. We will respect and mimic this method in our implementation. If we need to test on larger datasets, batch gradient descent may become problematic in terms of memory and we may use mini-batch gradient descent instead.

3 Challenges

3.1 Parallelism Pattern

Besides parallelizing common operations like matrix multiplication that exist in all neural networks, the graph structure of GCN imposes extra challenges to parallelization.

In each layer, the output of each node depends on the input of all its neighbors in forward inference, and the gradient of each node is also sent to all its neighbors in backward propagation. This makes GCN resemble the GraphRats task in Assignment 3 and 4 of this course, in terms of computational dependency, but the graph dataset commonly used for machine learning is much more unstructured than the grid-and-hub graph in GraphRats. Therefore, work balance could be a more pronounced issue when running GCN on real-world datasets.

Besides work balance, memory access is also an issue. The input of each GCN is the feature vectors of all neighbors of one node. In a large graph, the neighbors' feature vectors are basically random access in main memory, or even worse, in disks or networks, as explained in Section 3.3. We need find a way, if possible, to utilize cache more smartly.

3.2 Benchmarking

The reference implementation released by authors of the GCN paper was written in TensorFlow, which is both explicitly random (e.g. dropout and random initialization) and implicitly unpredictable (e.g. tricks in TensorFlow to accelerate computation and the nature of stochastic gradient descent). Even with exact model hyper-parameter, we cannot get the same model every time.

Meanwhile, some platform-specific and undocumented optimization of mathematical operations in TensorFlow will make our implementation harder to compete with it. This means we may need some optimization that is not related to parallel computing to beat the baseline.

3.3 Bounded Memory

GCN requires a memory that is linear to the number of edges. Some datasets that may be used for benchmark may be too large to fit in memory. For a large network graph (possibly as large as millions of or billions of nodes) will probably deplete memory. In this case, we may need a graceful way of splitting data (e.g. utilizing sparsity) or use stable storage.

Fully resolving this problem is beyond the scope of this project and even this course. We will take this challenge into consideration, but won't put too much emphasis on this until our basic goals are achieved.

4 Goals and Deliverables

We have two-fold goals. Once the basic goals are achieved would we try for the additional goals.

Basic Goals:

1. Implement a single-threaded GCN from scratch in C++ that are computationally equivalent to the reference TensorFlow implementation. This version of codes should be able to produce similar[2] prediction accuracy on multiple datasets.
2. Implement a parallel version of GCN using OpenMP.
3. Benchmark on parallel version. We expect to see that the multi-threaded implementation runs significantly faster than the single-threaded version, and analyze the scaling behavior of the program.
4. Analyze the performance on different types of graphs to understand how the graph types affect parallelism.

Additional goals (we expect to achieve 1-2 of these goals if basic goals are achieved early):

1. Achieve comparable or higher performance than the reference TensorFlow implementation.
2. Achieve near-linear parallel performance.
3. Rewrite a version using asynchronous parameter update.
4. Rewrite a distributed implementation of GCN using MPI.

5 Discussion

5.1 Relationship between Other Graph Computing Systems

Graph computing has become a hot topic in system research. There exists lots of graph computing framework out there. To name a few, GraphLab from Low et al. (2014), Distributed GraphLab from Low et al. (2012), PowerGraph from Gonzalez et al. (2012), Pregel from Malewicz et al. (2010).

The focus of these framework is to provide a domain specific framework and primitives for graph processing. They provide a set of APIs for programmers to build general purpose graph processing algorithms. They also focus a lot on fault tolerance and scalability. Our project just implement a single algorithm, i.e. GCN, and don't support algorithm modification. Moreover, we don't put efforts on fault tolerance and scalability, but fine-tuned performance instead.

5.2 Asynchronous or Synchronous Updates

Asynchronous or synchronous updates are critical design decision in graph processing systems. In the original implementation, the authors uses batch gradient descent method, which is inherently synchronous. Dean et al. (2012), on the other hand, showed that asynchronous SGD works very well for training deep networks, and can provide more parallelism.

Therefore, we put a synchronous implementation into the basic goals and asynchronous implementation into the additional goals.

[2]Exact reproduction is impossible due to reasons stated in Section 3.2

6 Schedule

Our tentative schedule is shown in Table 1.

Time	Task
April 10 - April 19 (Ckpt 1 Due)	1. Study original paper and baseline implementation in TensorFlow. 2. Implement a single-threaded C++ version of the algorithm.
April 19 - April 26 (Ckpt 2 Due)	1. Parallelize the program using OpenMP. 2. Benchmark with original TensorFlow implementation.
April 26 - May 1 (Int'l Workers' Day)	If the parallelized version has satisfactory performance, we will try to implement a distributed version using MPI and asynchronous parameter update. Otherwise, we will continue to optimize performance.
May 1 - May 6 (Report Due)	Prepare final report and poster

Table 1: Tentative Schedule

7 Resources

7.1 Original Paper and Codes from the Authors

We will primarily implement GCN based on "Semi-Supervised Classification with Graph Convolutional Networks" from Kipf and Welling (2016).

An out-of-the-box TensorFlow implementation of GCN was also released by its authors[3], along with some datasets used in their experiments. This is Python code and calls TensorFlow APIs. Therefore, we cannot directly reuse as a starter code. That being said, we can still learn about details of the model from it. Moreover, It also serves as a performance baseline for our implementation, although running faster than it can be very difficult, since TensorFlow has been heavily optimized.

7.2 Math Libraries

Some math operations, such as sparse matrix multiplication, can be optimized even in a single-threaded program without any parallization. We should utilize some wrapped libraries, for example Intel Math Kernel Library[4], if we want to compete with heavily-optimized TensorFlow.

7.3 Datasets for Benchmarking

Some of datasets in original paper are already accessible. Figure 2 shows datasets that was tested in it. Among them, the first three citation networks are released by authors in preprocessed format, and we will use them as benchmark datasets.

[3]`https://github.com/tkipf/gcn`
[4]`https://software.intel.com/en-us/mkl`

Dataset	Type	Nodes	Edges	Classes	Features	Label rate
Citeseer	Citation network	3,327	4,732	6	3,703	0.036
Cora	Citation network	2,708	5,429	7	1,433	0.052
Pubmed	Citation network	19,717	44,338	3	500	0.003
NELL	Knowledge graph	65,755	266,144	210	5,414	0.001

Figure 2: Benchmark datasets, from Table 1 of Kipf and Welling (2016)

Besides the original datasets, there are plenty of graphs of similar types that are available. We found a good resource ”Stanford Large Network Dataset Collection” [5], which collects almost 100 graph datasets. We will also choose several newer datasets after reproducing the results of original datasets.

We may also use some randomly generated graphs with various properties to benchmark the performance of different graphs.

8 Platform

Due to the sparse and non-local nature of graphs, GCN does not benefit as much from GPU as other neural networks, and running on GPU may be less cost-efficient than on CPU in practice. Therefore, we decided to only implement GCN on CPU.

We plan to use C++ and OpenMP to implement multithreaded GCN, and use GHC machines for performance tests. If time permits, we may also implement distributed GCN using MPI, and test it on Latedays.

References

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[5] `http://snap.stanford.edu/data/index.html`

Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. 2002.