Technische Universität München

Parallel Programming (TUM) SS2021 Final Project

Examiner: Prof. Martin Schulz
Project Authors: Shubham Khatri, Maximilian Stadler
Teaching Assistants: Bengisu Elis, Vincent Bode

Project 01 – GCN

General Guidelines and Rules

- Your work needs to be submitted through the LRZ GitLab. You will be granted access to your repositories at the kickoff.
- You will submit your solutions in three steps with three different due dates. The individual tasks and their due dates may be found below.
- There are separate repositories under your GitLab group for each task in this document, namely OMP, MPI, Hybrid, Bonus and Documentation. Please make sure to submit your solutions to the corresponding repositories.
- Keep in mind that only the master branches of each repository will be considered as a valid submission and graded. You are expected to submit a single solution file in the repositories that contains your solution code with the best speed-up you have achieved until the deadline. Make sure this single file on the master branch contains the solution that you want to have graded.
- In case you use or adapt code from a work other than yours (internet tutorials, public repositories etc.) cite the source in a code comment. Doing otherwise will be considered as plagiarism and will be taken into account when grading your code.
- For your project presentation, further instructions are provided in the presentation template.
- **Note:** You must not not change the algorithm itself but perform the optimization and parallelization on the algorithm described above.
- To be considered a valid submission, your code should compile on the submission server. Therefore, be careful with the libraries used in your code. In case of unexpected compilation errors, contact your responsible tutor.
- You will be assigned a responsible tutor. Your tutor is available for answering questions and to help resolve any issues you might encounter.
- Write a brief README for each repository which explains the changes or additions in your code other than performance improvements. If you implemented new functions and data structures in your solution, explain these in your README.

Algorithm : Graph Convolutional Neural Networks

Neural networks have been knowingly applied quite successfully in domains like image processing and natural language processing. Another successful application can be found in the field of graph learning. A graph $\mathcal{G}$ is defined by a set of N vertices $\mathcal{V}$ (also referred to as nodes) which are connected through edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Thus, an edge $e_{u \to v} \in \mathcal{E}$ if nodes u and v are connected. The neighborhood $\mathcal{N}(v)$ of each node v is defined as the set of nodes that are connected to v, i.e. $\mathcal{N}(v) = \{u \in \mathcal{V} : \exists e_{u \to v} \in \mathcal{E}\}$. For aggregation steps, we usually consider the neighborhood of a node including the node itself, i.e. $\mathcal{N}(v) \cup v$. In attributed graphs, each node $u \in \mathcal{V}$ is represented by its attributes (also referred to as features) $\mathbf{x}^{(u)} \in \mathbb{R}^D$. Common examples are citation networks, where each vertex is a paper represented by its title or abstract while the citations are the links (or edges) to other papers. In this example, one typically wants to classify papers into research domains based on only a few known ground-truth labels. More formally, in this setting of node classification, we want to assign each node u a class $\hat{y}^{(u)} \in \{1, ..., C\}$. For training and evaluating algorithms, one usually also has access to the ground-truth class-labels for a subset of vertices, i.e. $y^{(u)}$ for some nodes $u \in \mathcal{U} \subseteq \mathcal{V}$. A model also should predict probability scores indicating how likely those predictions are, i.e. $\hat{p}_c = P(y = c)$. Based on these probability scores, the actual prediction is obtained as $\hat{y}^{(u)} = \arg\max_c \hat{p}_c^{(u)}$. Remark: the code contains a bug that uses an inverse normalization factor, i.e. $\sqrt{d_u d_w}$ (in red) is used instead of $1/\sqrt{d_u d_w}$ (in green). Please parallelize the operations described in the code (i.e. using the red normalization factor).

A quite common and simple model designed for classifying nodes is the Graph Convolutional Neural Network (GCN)[1]. With trainable weights $\mathbf{W}^{(1)} \in \mathbb{R}^{D \times H}$, $\mathbf{W}^{(2)} \in \mathbb{R}^{H \times C}$, and bias terms $\mathbf{b}^{(1)} \in \mathbb{R}^H$, $\mathbf{b}^{(2)} \in \mathbb{R}^C$, the final prediction is obtained in the following way:

$$
\begin{aligned}
h_i^{(u)} &= ReLU\left(b_i^{(1)} + \sum_{w \in \mathcal{N}(u) \cup u} \left[\frac{1}{\sqrt{d_u d_w}}\right] \sqrt{d_u d_w} \left(\sum_j^D x_j^{(w)} W_{ji}^{(1)}\right)\right) \\
z_i^{(v)} &= b_i^{(2)} + \sum_{u \in \mathcal{N}(v) \cup v} \left[\frac{1}{\sqrt{d_v d_u}}\right] \sqrt{d_v d_u} \left(\sum_j^H h^{(u)}{}_j W_{ji}^{(2)}\right) \\
\hat{\mathbf{y}}^{(v)} &= softmax\left(\mathbf{z}^{(v)}\right)
\end{aligned}
\tag{1}
$$

where

$$
ReLU(x) = \max(x, 0) \qquad softmax\left(\mathbf{v}\right)_i = \frac{e^{v_i}}{\sum_k e^{v_k}} \tag{2}
$$

This simple model transforms the input in a linear transformation into a hidden space, aggregates representations from neighbors, and applies a non-linear activation function afterwards. A second layer does the same with the final non-linear transformation being the softmax-function that transforms the logits $\mathbf{z}$ into probability scores that sum up to 1. While the transformation is applied for each node independently, you see see that the aggregation step entangles the nodes. We can measure how well the models performs with the accuracy

$$
ACC = \frac{1}{N} \sum_{u \in \mathcal{V}} \mathbf{1}\left[\hat{y}^{(u)} = y^{(u)}\right] \tag{3}
$$

i.e. the fraction of nodes for which the predicted label $\hat{y}^{(u)}$ corresponds to the ground-truth label $y^{(u)}$. Also note that when only interested in the prediction, we have $\hat{y}^{(u)} = \arg\max_c \hat{p}_c^{(u)} = \arg\max_c z_c^{(u)}$ since the softmax is a monotonous transformation.

[1] Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.

Implementation : Short Documentation

You will be given a straight-forward sequential implementation of the algorithm described above in *"gcn_sequential.cp*

- For your convenience, you will find a *DEBUG* flag at the beginning of the file. If this flag is set, you will get a runtime measurement which acts as a proxy to the measurement on the submission server on your local machine.
- You can further modify the computed problem through command line arguments with the following signature: *"gcn_sequential DATASET INITIALIZATION-NO [SEED]"*. (only if DEBUG is set to 1)
- For testing purposes on your local machine, we will provide you with some real-world datasets with different characteristics (*CORA*, *CITESEER*, *PUBMED*, *COAUTHORCS*) and corresponding sets of weights which have been obtained by training the above model with different random initialization seeds $(1, \ldots, 5)$. If you specify a *INITIALIZATION-NO* smaller than 0, you will create a randomly initialized problem based on the *SEED* passed to the program (which otherwise is not needed, if no seed is passed it defaults to 42).
- To use those datasets, you first have to extract the corresponding *.zip* archives. You can find those as *01_gcn/data/DATASET.zip* and you should extract them into so the resulting directory is *01_gcn/data/-DATASET* for each *DATASET* you want to test your code on (with *01_gcn* being the root directory of this project).
- For evaluation on the submission server, please ensure that *DEBUG* is set to 0 or that the corresponding lines of code are removed. In this case, your code will be evaluated for the *COAUTHORCS* dataset with a prompt specifying the seed for the random initialization (similar to the homework assignments). Note that if you also want to use this case on your local machine, you don't have to pass command line arguments to the program. Note that to be able to do this on your local machine (not on the server), you have to extract the *COAUTHORCS* dataset at least.
- The correctness of your code is evaluated by comparing the accuracy obtained from your implementation with the accuracy of this classification problem obtained from a reference solution.
- Utility code is provided in the files *"Node.cpp"*, *"Node.hpp"*, *"Model.cpp"*, and *"Model.hpp"*. You are not expected to change code in those files. However, feel free to do so.

Tasks

1. **General:**

   This task includes preliminary optimisations on the sequential code without parallelising. You are not expected to submit a separate file for this task but by optimising the sequential version provided, this task aims to help you achieve better speed up for following parallelisation tasks.

   Go through the problem and analyze the given sequential code. Where do you see improvements through parallelization? By looking at the problem and the code while keeping memory layouts of buffers in mind, you should identify one obvious change to the sequential code. Changing the corresponding parts should already give you a visible speedup without parallelization. **Remark:** Using the improved sequential code you should expect a speedup between 10 and 15 (at least) on a 6-core machine for parallel implementations.

2. **OMP:** *Due 15.06.2021 23:59, Submission file "gcn_omp.cpp"*

   Parallelize the computations using OMP. Parallelize as much as possible as long as it does not harm performance. You do not have to parallelize the process of reading the input. Explain which parts of the code you parallelize and why it might be better than other options. Show your work.

3. **MPI:** *Due ~~22.06.2021 23:59~~ 24.06.2021 23:59, Submission file "gcn_mpi.cpp"*

   Assume (hypothetically) that you have to run your code on a cluster without shared memory access and only one core per compute node. Parallelize the computations using MPI. Parallelize at least the two linear transformations (i.e. the parts involving the weight matrices $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}$. Explain which parts of the code you parallelize and why it might be better than other options. Show your work. Remark: When reading the input from the file system, you should assume that those files may only be found on RANK 0, i.e. you may only read files on RANK 0.

   Remark: Your MPI solution should be agnostic to the number of MPI processes used when executing the code. In other words the solution should be executable by any number of MPI processes.

4. **Hybrid (MPI + OMP):** *Due 29.06.2021 23:59, Submission file "gcn_hybrid.cpp"*

   Assume (hypothetically) that you have to run your code on a cluster with multiple compute nodes without shared memory access but several cores per compute node with these cores sharing memory access locally. How can you leverage this architecture in a hybrid approach using MPI and OMP? Parallelize at least the two linear transformations (i.e. the parts involving the weight matrices $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}$. Modify the code from the MPI assignment accordingly. Explain which parts of the code you parallelize and why it might be better than other options. Show your work.

5. **Bonus (Optional):** *Due 29.06.2021 23:59, Submission file "gcn_bonus.cpp"*

   Given your hybrid implementation (MPI + OMP), can you observe any speedup using one of the following of your choosing: SIMD intrinsics, OMP GPU offloading , CUDA or HIP ? Explain how and provide necessary details.

   Note: Implement this using intrinsic operations and not OMP SIMD.

6. **Presentation Slides:** *Due 06.07.2021 23:59, Submission file "PPSS21_final_project.pdf "*

   Use the template provided in the Documentation repository for your presentation slides. Submit your presentation into the Documentation repository.