Parallel Topological Sorting: Project Milestone Report

Ezra Fu (erzhengf), Robert Benjamin Osborne (rbo)
https://codeplay0314.com/parallel-topological-sorting/

Summary

Our goal for this milestone is to implement three correct topological sorting methods for Directed Acyclic Graphs (DAGs): serial, CUDA, and a Domain-Specific Language (DSL). Ensuring the correctness of these three implementations has been our primary focus up to this point. We have successfully implemented all three approaches and conducted tests to verify that they produce accurate and consistent results.

Accomplishments

Problem Definition

With a DAG as the input, our goal is to generate batches of nodes from the DAG, where nodes in each batch (excluding the first) strictly depend on nodes from the previous batch.

Input File

The first line contains an integer `n` representing the number of nodes in the DAG.

The next `n` lines describe the nodes and their dependencies: line `i` begins with an integer indicating an integer d_i, the number of dependencies of node `i` followed by d_i integers representing the dependencies of node `i`.

It is guaranteed that the nodes and dependencies form a valid DAG.

Output File

The first line contains an integer `b` indicating the number of batches.

The next `b` lines describe the batches: line `i` begins with an integer indicating k_i indicating the number of nodes in the batch, followed by k_i integers representing the nodes in that batch.

Data Generation

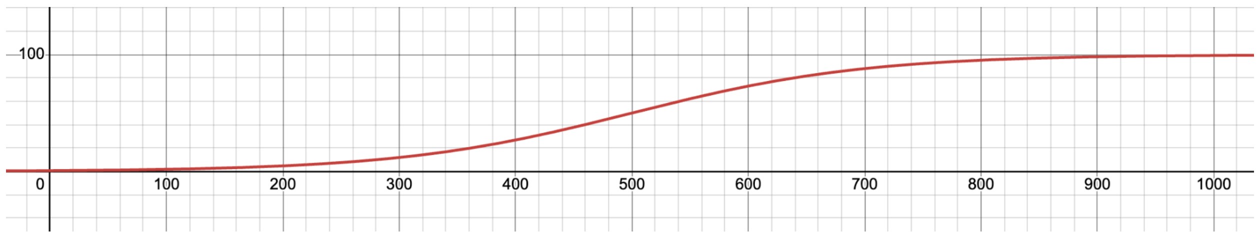
Our intent is to use a number of different algorithms to generate data for this project so that we may both test different real-world scenarios and verify program correctness. So far we have implemented three approaches:

Simple Data Generation

For each node a random number of dependencies are generated that are within a certain bounds. Similar to class labs, 4 data sets will be created with varying difficulties: easy, medium, hard and impossible. The count of nodes will remain constant across these, but the count of dependencies will double with each. To prevent a cyclic dependency, nodes may only be dependent on nodes with a lower index.

Exponential Probability Graph

Similar to the simple data generation, this data generation method seeks to generate data with the real-world property that items are often more likely to be dependent on nearby items as opposed to items that are further away. An example of this may be PHD level course requirements, which may be dependent on other PHD or master level coursework, but less likely to be dependent on earlier level undergrad courses. Below is an example of the probability weights graph when i = 1000.



Opposite Order Data Generation
Another way to generate data is to generate answers first then deduct the input. We can preset the number of nodes, the number of batches first, fit nodes into batches, and then build the dependencies between batches. In this way we can control the structure of the DAG as well as have standard input and output at the same time for testing.

Implementation

Serial
The serial implementation keeps track of the incoming degree of each node and the adjacency list representing the DAG.
The process iterates arbitrarily, with each iteration producing a batch. In each iteration, nodes with an incoming degree of 0 are included in a new batch, meaning they no longer have any dependencies. These nodes are then removed from the graph, along with the dependencies associated with them. This process continues until no nodes remain in the graph.
Finally, the batches produced are written to an output file.

Cuda
The CUDA implementation uses an adjacency matrix to represent the DAG and maintains a set of nodes that have already been batched.
In each iteration, the number of dependencies is calculated by summing the values in each row of the adjacency matrix. Nodes whose corresponding row sums are 0 are selected to form a new batch, excluding those already batched. These nodes are then added to the set of batched nodes. The adjacency matrix is updated by clearing the columns corresponding to these nodes, effectively removing them from the DAG. This process repeats until all entries in the adjacency matrix are 0.

DSL
For this project we decided to use the NetworkX library (link), which was created to improve working with graphs and networks. It supports features such as directed graphs, which enables speedy development here. This library is written in Python, which provides great usability, but potentially poor performance. To mitigate this, our team plans to implement PyPy as well as the "parallel" and CUDA backends of this library. If

performance is excessively slow, our team is considering other solutions, including up to using the Neo4j graph library.

Testing

We tested the correctness of these three implementations by comparing their outputs, generated from standard inputs, against the expected standard outputs. To ensure the implementations are reliable and executable at different scales, we generated outputs of varying sizes. The datapoint statistics are shown below:

	Small	Medium	Large	Impossible
# of nodes (n)	100	1000	4000	60000
# of batches (b)	10	50	100	1000
Density (# of edges / (n * (n - 1)))	~0.1	~0.01	~0.01	~0.001

Goals and Deliverables Review

Completed

Implementation

Initial programs have been created in all three categories: serial, CUDA and DSL. This includes:

- A serial topological sort algorithm as a performance baseline
- A CUDA-based parallel topological sort using adjacency matrix manipulation
- A DSL-based solution (via NetworkX)

Items left before presentation

Implementation

- Adjust CUDA algorithm to scale well as more nodes are added to the graph
- Experiment with DSL implementations and backends (parallel, CUDA)
- [Optional] Implement other DSL alternatives that may provide good performance (such as Neo4j)

Performance Analysis

- Benchmark both parallel implementations against the serial baseline
- Analyze speedup factors, scalability, resource utilization, and overheads

Demo for Poster Session

- Performance Graphs: Show speedup graphs and resource utilization charts to highlight efficiency gains.
- Code Walkthrough: Briefly explain key sections of the code to demonstrate optimization strategies.
- Interactive Visualization: Display real-time comparisons of the serial and parallel algorithms processing sample DAGs.

Main Concerns

- DSL implementation
    - Our current DSL's frontend is written in Python, which provides poor performance. We are still waiting to see if performance is too slow. If so, we may pivot to another solution.
    - Our team would very much like to run the DSL's CUDA functionality on the GHC machines, but greater than 2 GB storage is required (which is the limit for our AFS accounts). Our team requested an increase, for which we are waiting to hear back.
- CUDA implementation
    - The implementation requires $O(n^2)$ space which can be troublesome with DAG with a large number of nodes.
    - The operations on the adjacency matrix introduce an extra layer of computation compared to the serial implementation, resulting in significantly slower performance with the current test cases. Further optimization is needed to address these challenges.

Schedule Update

Week 1 (Nov 11 - 17)

- [x] ~~Review literature on topological sorting~~
- [x] ~~Set up development environments for C++, CUDA, and DSL tools~~
- [x] ~~Implement the standard sequential topological sort~~
- [x] ~~Validate correctness with sample DAGs and establish baseline performance metrics~~

Week 2 (Nov 18 - 24)

- [x] ~~Begin coding the CUDA-based topological sort~~
- [x] ~~Optimize kernel functions for better memory access patterns~~

Week 3 (Nov 25 - Dec 1)

- [x] ~~Select a DSL~~
- [x] ~~Implement the topological sort algorithm using the DSL~~

Week 4 (Dec 2 - 8)

- [x] ~~Test and debug issues related to correctness, concurrency, data races, and synchronization~~
- [ ] Collect data on execution time, speedup, scalability, and resource usage
- [ ] Analyze performance and efficiency
- [ ] Test implementations on real-world DAGs from applications like deep learning and rendering

Week 5 (Dec 9 - 15)

- [ ] Compile findings into the final report
- [ ] Reassess goals and ensure all deliverables are met
- [ ] Rehearse the demo and prepare for the poster session