15-418 Project Proposal: Parallel Low Diameter Decompositions in Graphs

Jacob Imola (jimola), Sidhanth Mohanty (smohant1)

April 10, 2017

1 Summary

We are implementing a parallel algorithm from a paper of Gary Miller, Richard Peng and Shen Chen Xu to perform low diameter decompositions in graphs.

2 Background

A very fundamental problem that captures several other problems we might want to solve is given a $n \times n$ matrix A and a vector b, solve for x in $Ax = b$, and Gaussian elimination is the method we are taught to solve these systems of equations, which has a runtime of $O(n^3)$. There are more sophisticated techniques too, but they all run in super-quadratic runtime. What if we are promised that A has a special structure, that is, A is "symmetric diagonally dominant" (SDD)? And additionally, what if we only care about approximating x and not actually completely solving for it? Can we do better? The answer is affirmative because of an important gamechanger in the landscape of algorithm design called Fast SDD solvers. Among matrices that satisfy the SDD property, important ones that always come up in practice are the Laplacian of a graph. For a Laplacian L, a lot of graph problems we care about (like max flow, electric current flows in a graph etc) can be solved by solving $Lx = b$ for an appropriately chosen b. There are algorithms that let us solve $Lx = b$ in near linear time in the number of nonzero entries in the matrix. Practical SDD solvers would indeed change how we solve a lot of problems in practice. Naturally, a follow up question is ask is "how can one parallelize a SDD solver?". A key ingredient in algorithms for solving $Lx = b$ is to perform what is called a 'low diameter decomposition' of the vertices in a graph. A low diameter decomposition put vertices into clusters so that the diameters of the clusters are small, and the number of edges between clusters is small too. Thus, making this ingredient highly parallel would give parallel algorithms for solving SDD systems. In our project, we hope to move one step closer towards practical, parallel SDD solvers by coming up with a practically feasible implementation of the key step of low diameter decomposition.

3 The Challenge

The standard sequential algorithm works in the following way: you start at a vertex v, and start a BFS from v: stop the BFS if the number of edges in the next round of BFS is less than some threshold, and put all the visited vertices in a cluster, remove them and recurse on the remaining graph. The obvious parallelization of this algorithm is to parallelize the BFS step in the creation of each cluster. This is a simple algorithm with work only linear in the number of edges (and with a low constant factor). However, it isn't obvious how to parallelize generating multiple clusters at once and that seems like a sequential bottleneck. The paper by Miller, Peng and Xu get around this bottleneck: still, the algorithm gets more complicated, and in terms of work done, it has a larger constant factor than the standard sequential algorithm, so it isn't immediately obvious that a straightforward implementation of their improved algorithm would actually beat the standard algorithm limitedly parallelized in the trivial way. The goal of our project will be to implement both the algorithm of Miller-Peng-Xu, the sequential algorithm, and the trivial parallelization of the sequential algorithm, try to optimize both a lot, and decide which one is a practically better algorithm by looking at the speedups they obtain against the low work sequential algorithm. The main challenge will be in optimizing the Miller-Peng-Xu algorithm, and understanding how to efficiently implement some parts whose details are not completely clear from the paper (which is fair since it is a CS theory paper).

4 Resources

We are starting the codebase from scratch. The paper by Miller-Peng-Xu describes the algorithm and we would have to understand and internalize the paper well. We were planning to use the `latedays` cluster and the multicore CPUs there, and we plan to achieve parallelization using `OpenMP`.

5 Goal and Deliverables

- Implement the baseline sequential algorithm (call the runtime as T_1).
- Implement the trivial parallelization of the sequential algorithm (call the runtime as T_2) (this is the baseline that we compare the parallel version of Miller-Peng-Xu against).
- Implement the algorithm in the paper of Miller-Peng-Xu (call the runtime of the single threaded version as T_3 and the multithreaded version as T_4).
- We generate 7 large connected graphs with more than 10 million edges (possibly 100 million): one of them would be $G(n, \frac{1}{2})$ on 5000 vertices, two others would be $G(n, p)$ for other values smaller values of p (to get sparser graphs with more clusters and larger diameters), one would be a large tree, one would be a large grid graph (somehow simulating an image where adjacent pixels are connected).
- As a sanity check to make sure that we parallelized Miller-Peng-Xu correctly, we compare the ratios $\frac{T_4}{T_3}$ against $\frac{T_2}{T_1}$ on multiple graphs. We expect that the former ratio will be smaller than the latter ratio on a lot of the graphs because Miller-Peng-Xu has much better span. This is a sanity check for us to proceed with optimizing both implementations.
- We compare T_4 against T_2 to determine which parallel implementation is better on each of the graphs. It is possible that even though Miller-Peng-Xu is more parallel, it does more work and hence the speedup doesn't make T_4 better than T_2. It may also be possible that Miller-Peng-Xu is better on very sparse graphs and one might want to use the naively parallelized version of the baseline for dense graphs.
- To make sure that we aren't cheating by using an inefficient baseline, we compare our baseline against the hybrid BFS implementation of Assignment 3 and make sure we are within some factor of it's speedup on similar graphs.

- As a stretch goal, if we finish all the described goals faster than we anticipate, we also plan to write up an implementation of finding low-stretch spanning trees in a graph using our parallel algorithm for Low Diameter Decompositions: the details of this can be found in Lecture 5 of Professor Anupam Gupta's class 15-850 (Advanced Algorithms) this semester.
- As an even more ambitious goal, if we can finish all the previous mentioned goals, we plan to use our implementation in a parallelized version of the SDD solver from the captivating breakthrough paper by Cohen, Kyng, Miller, Pachocki, Peng, Rao, Xu called Solving SDD systems in nearly $m\sqrt{\log n}$ time.

6 Platform

`C++` with `OpenMP` because the parallelization is mostly doing different independent iterations of a `for` loop in parallel.

7 Schedule

- Week of April 10: Figure out how to generate the graphs that we will test on and get a sequential baseline version running and try to match the performance of hybrid BFS on similar graphs from Assignment 3.
- Week of April 17: Parallelize the sequential baseline and try to match the performance of hybrid BFS on similar graphs from Assignment 3 using the same number of cores. Start implementation of the algorithm of Miller-Peng-Xu
- Week of April 24: Finish implementing the algorithm of Miller-Peng-Xu.
- Week of May 1: Try to optimize harder and get better speedups on both. Start working on parallel implementation of finding low stretch spanning trees based on the Miller-Peng-Xu algorithm.
- Week of May 8: Finish work on finding low stretch spanning trees.

Even though we claim we are implementing low stretch spanning trees, we are giving ourselves one week buffer time to accomodate for unexpected things such as it being hard to match hybrid BFS implementation, or not receiving good speedups on the parallel version of Miller-Peng-Xu against the sequential version (which means we have to fix what we are doing). In the case we use this buffer time, we won't implement low stretch spanning trees.