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

Jacob Imola (jimola), Sidhanth Mohanty (smohant1)

May 12, 2017

1 Summary

We implemented an algorithm from a paper of Gary Miller, Richard Peng and Shen Xu [?] to perform a low diameter decomposition in graphs in parallel using `C++` and `OpenMP`. Depending on the structure of the graph, we achieved somewhere between a 2-3$\times$ when using all the cores on a single machine in `latedays` compared to the single threaded implementation of this algorithm. We also implemented a work efficient simple sequential algorithm, and unfortunately, on some sparse graphs, the sequential algorithm is more efficient.

2 Background

A low diameter decomposition of a graph is a partition of the vertices of the graph into clusters such that the distance between any two vertices inside a single cluster is small and the number of edges between any two clusters is small compared to the total number of internal edges. Formally, an (α, β) LDD of the graph is a decomposition of V into disjoint $V_1, V_2, \ldots, V_t$ such that for any $v_1, v_2 \in V_i$, $d(v_1, v_2) \leq \alpha$ and the number of edges leaving V_i is at least a β fraction of the number of internal edges in V_i.

The baseline sequential algorithm to find a $(\frac{\log n}{\beta}, \beta)$-LDD works in the following way: pick a vertex and start a BFS and stop when the fraction of edges leaving the current component is less than β(number of internal edges). Put all the visited vertices in a cluster and recurse on the remaining graph.

The reason this algorithm is not nicely parallelizable is that picking clusters is an inherently sequential process. Miller-Peng-Xu [?] propose an algorithm to bypass this limitation, which we describe below:

1. Assign a value $\delta_u \sim \mathrm{Exp}(\beta)$ to each vertex u (a parallel step). This value is the amount of "head start" that each vertex gets.
2. Compute $\delta_{\max} = \max\{\delta_u | u \in V\}$, and set the start times of vertex u as $\delta_{\max} - \delta_u$.
3. Do a single BFS, where each vertex in the frontier belongs to a cluster and set any neighbors in the next frontier to the same cluster. Each round of BFS takes 1 second, and a vertex v with start time S joins the frontier at round $\lfloor S \rfloor$ with its own cluster if it hasn't already been visited. We store $S - \lfloor S \rfloor$ as a tie-breaker; if a vertex u in the next frontier hit by multiple vertices in the current frontier, u belongs to the smallest cluster. It is not hard to see that a vertex will belong to the cluster of the vertex that has the minimum distance minus head start.

Steps 1 and 2 are extremely parallelizable. Step 3 involves a single BFS, so it's parallelization depends on the topology of the graph. For each vertex, we store the cluster it belongs to and its minimum tiebreaker. Multiple BFS's happening in parallel causes multiple clusters to form at once and is the key step in achieving parallelism. We note that for smaller values of β the performance of the parallel algorithm worsens but the sequential algorithm runs in exactly the same amount of time. Also, the parallel algorithm is more complicated and hence a sequential version of it is slower than the sequential baseline that we have.

3 The Approach

We used `C++` with `OpenMP` and optimized our code on the `latedays` machine.

- In the first step of assigning a value to each vertex, we run a `#pragma omp parallel for` loop to have multiple threads assign values to different vertices at the same time.
- Although the max can be computed using a reduce, we do it sequentially since the advantage of reduce is noticable only in a large number of cores.
- Constructing the new frontier from the old frontier can be done in parallel by having different threads retrieve a list of neighbors of different vertices in the frontier, then these different lists are merged to get the new frontier. We were initially using a standard `OpenMP #pragma omp parallel for` loop and then later added dynamic scheduling, and we noticed a speedup.

We spent some time optimizing our simple sequential baseline to make sure that our parallel implementation was being compared against a fair test harness. In our initial sequential version, we were adding all neighbors to a tentative frontier (with possible duplicates because we didn't want to assign vertices to clusters before making sure that there were enough edges leaving the current cluster) and then going through this fully created list to count the number of edges and process the new frontier (without duplicates). Running this code on dense graphs completely killed performance so we modified our code to not store duplicates. We were also storing 2 arrays whose ith element was accessed at similar times, and then got a minor speedup with better cache locality by having one array of length $2n$ where even indices store one array and odd indices store the other array.

We didn't use any existing pieces of code and started from scratch.

4 Results

¡insert bunch of results here, and graphs¿

There was still more scope for parallelism and all the CPU cores did not exhaust the opportunities for parallelism. Indeed, the strength of the parallel algorithm was that it had very good span and hence would benefit from more cores, and hence doing the project using CUDA on a Nvidia GPU instead of using `OpenMP` might have been a better idea.

5 List of work by each student

- **Jacob:** Implemented the Miller-Peng-Xu algorithm and parallelized it in OpenMP, also optimized and improved the sequential baseline written by Sidhanth. Also wrote the setup

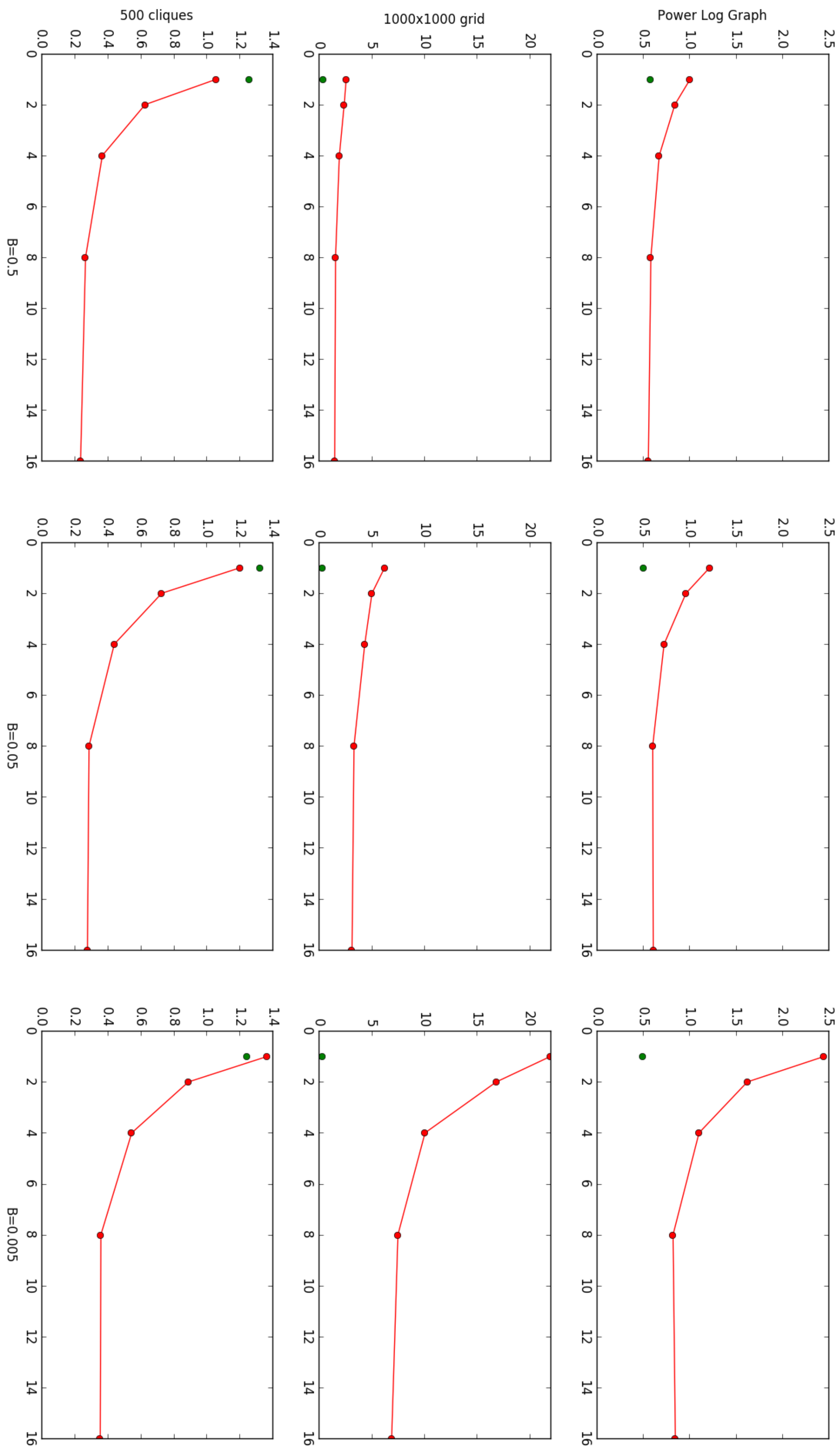

Figure 1: Speedup plots for 3 different graphs and 3 different values of β.

code in `main.cpp`.

- **Sidhanth:** Wrote the sequential baseline and performed some optimizations on the MPX algorithm.