An Analysis of Graph Algorithms on GPU: Local Clustering Coefficient and Community Detection via Label Propagation

Chris Cai, Zongjing Chen, Zhuo Chen, Henry Haase

1. Introduction

Graphs, in the computer science context, can be considered one of the most important data structures ever invented purely based on how useful they are. From representing social networks to transport maps, so many structures in the world can be represented with a directed or undirected graph, so it is understandable that there would be a lot of work into developing algorithms to determine key properties of these structures. In fact, two such properties that are important to study are determining how well clustered singular nodes are as well as determining the overall community for sets of nodes in the graph. This leads to two important algorithms in studying graphs: the local clustering coefficient algorithm (LCC), which determines how well a node and its neighbors for a clique (all the neighbors being linked to each other as well), and community detection via label propagation (CDLP), which determines and labels overall communities within the graph. Given the importance of these algorithms, it is also important for them to be performant enough to run on large datasets on individual or large clusters of compute resources.

2. Background

2.1 Local Clustering Coefficient

The local clustering coefficient algorithm is one that is used to determine the connectedness a node has between its neighbors and its neighbors with each other. Effectively, it determines how much a node is part of a clique, which can be very useful in finding certain patterns and communities in many graphs (i.e. social network graphs). The algorithm for determining this clustering coefficient is actually relatively simple, and simply involves finding the degree of the node as well as the number of triangles the node is a part of. The pseudo code, give by the LDBC council (1), is given below:

```
input: graph G = (V, E)
output: array lcc storing LCC values
for all v ∈ V do
    d ← |N_in(v) ∪ N_out(v)|
    if d ≥ 2 then
        t ← 0
        for all u ∈ N_in(v) ∪ N_out(v) do
            for all w ∈ N_in(v) ∪ N_out(v) do
                if (u, w) ∈ E then                ▷ Check if edge (u, w) exists
                    t ← t + 1                     ▷ Found triangle v − u − w
                end if
            end for
        end for
        lcc[v] ← t / (d(d−1))
    else
        lcc[v] ← 0                                ▷ No triangles possible
    end if
end for
```

Figure 1. Local Clustering Coefficient Pseudocode

An example of multiple graphs with varying clustering coefficients dependent on the existence of certain edges can also be seen here:

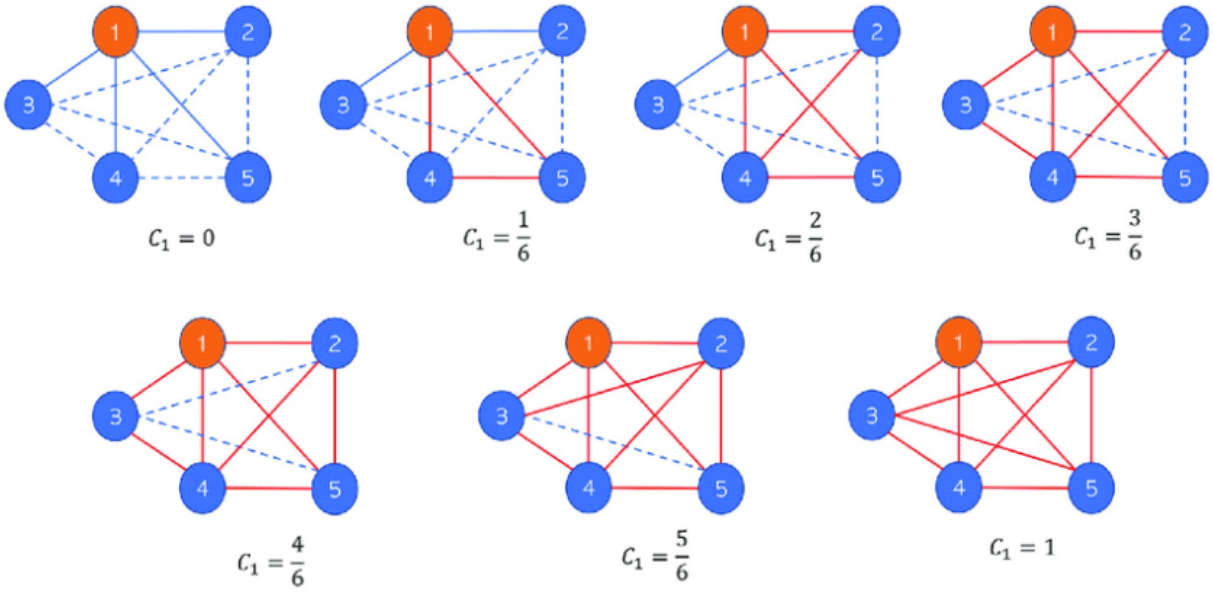

Figure 2. Local Clustering Coefficient Example

2.2 Community Detection via Label Propagation

Community detection via label propagation is an iterative algorithm that labels nodes based on their “community”, and then updates their label based on a quorum of all the nodes' neighbors. At the first iteration, every node is given a unique label (usually the node’s index), and then each neighbor queries all its neighbor to find the the most common label, and then relabel’s itself as that label (“lower” value labels are often used as the tiebreaker). This often means by the second iteration, every node is labeled as the smallest index of its neighbors, and then further iterations will continue to update labels. The pseudocode for this algorithm, provided by the LDBC council (1), is given below:

input: graph $G = (V, E)$, integer *max_iterations*
output: array *labels* storing vertex communities

```
1:  for all v ∈ V do
2:      labels[v] ← v
3:  end for
4:  for i = 1, ..., max_iterations do
5:      for all v ∈ V do
6:          C ← CREATE_HISTOGRAM()
7:          for all u ∈ N_in(v) do
8:              C.ADD(labels[u])
9:          end for
10:         for all u ∈ N_out(v) do
11:             C.ADD(labels[u])
12:         end for
13:         freq ← C.GET_MAXIMUM_FREQUENCY( )        ▷ Find maximum frequency of labels
14:         candidates ← C.GET_LABELS_FOR_FREQUENCY(freq)   ▷ Find labels with max. frequency
15:         new_labels[v] ← MIN(candidates)          ▷ Select smallest label
16:     end for
17:     labels ← new_labels
18: end for
```

Figure 3. Community Detection via Label Propagation Pseudocode

We can also see how the labels propagate by the given example below. As a note, due to the fact that every iteration is based on a quorum of its neighbors, highly connected graphs can result in oscillatory patterns that do not fully converge. For this reason, we

only run the algorithm for a set number of iterations, at which point we are confident that the graph has mostly converged.

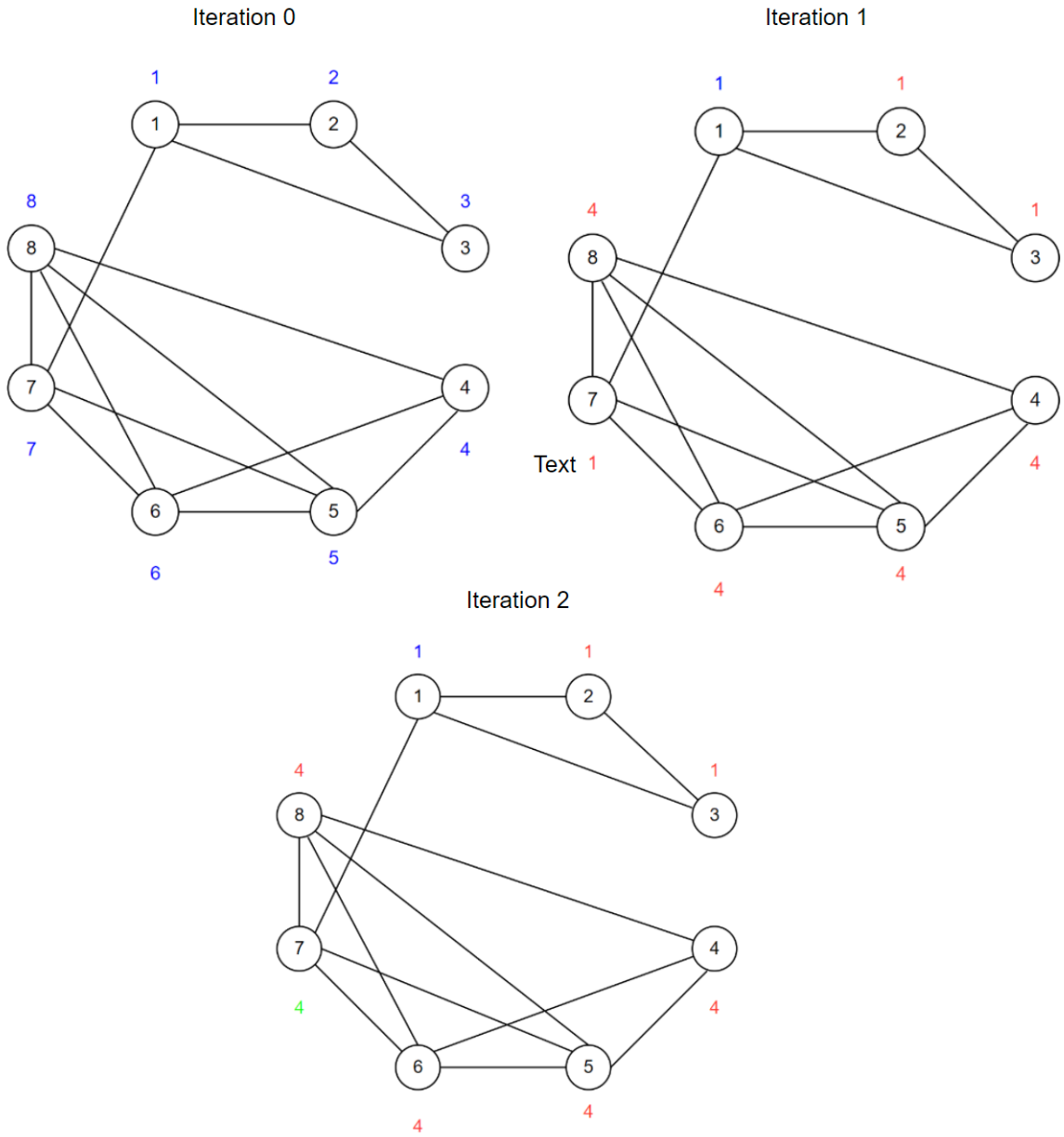

Figure 4. Community Detection via Label Propagation Example

3. Baselines

Since we worked on two graph algorithms in this project, two baselines are discussed here respectively.

3.1 CDLP baseline

The CDLP baseline is a striaght-forward and intuitive implementation of the CDLP algorithm using gather approach, which could be summarized as these few steps:

1. Pre-allocate global memory for holding each node's neighbor list.
2. One thread collects all neighbors's labels of a node into the neighbor list.
3. The same thread creates a histogram of the labels of the same node. The process of generating the histogram is by first sorting the labels then counting each label's frequency.
4. Set the current node's label to that with the highest frequency.
5. Proceed to the new iteration.

This is the baseline which both the gather optimizations and scatter optimizations will compare against. Additionally, a CPU baseline is coded as well to assess the speed-up of GPU to CPU, which follows the pseudo-code established by the LDBC benchmark, as illustrated in figure.1.

Both baselines were implemented from scratch while making use of Pangolin for graph representations. Existing codes we found on Github were either too complicated or nested deeply within a different graphing system that are hard to reference. Thanks to Pangolin, we needed a little extra work to make the baseline code running in CUDA. The time taken for Pangolin to generate the graph representation is recorded separately from the actual computations.

3.2 LCC baseline

For the Local Clustering Coefficient (LCC) kernel, our baseline kernel was based off the triangle counting lab, as, for undirected graphs, the LCC algorithm could utilize triangle counts to determine the number of existing edges between a node's neighbors. Since we had implemented triangle counting in a previous lab, we were able to make use of

the code with some minor changes to calculate the triangle counts per node and the number of neighbors for each node. To compute the actual coefficient, we created a second kernel that took the triangle counts and neighbor counts per node and computed the LCC using the following formula.

$$C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)}.$$

Because we worked off of the existing lab code, we did not need to perform any extra work to convert our code to CUDA. To perform operations on the graphs (like converting it to COO), we utilized pangolin for similar reasons as in CDLP.

4. Optimizations and Results

4.1 CDLP Optimizations

The team's initial thoughts on how to optimize the CDLP baseline diverged at an early stage of the project. As a result of this divergence, two versions of the optimizations were proposed which we decided to implement both and combine the best of both worlds. The main divergence is on whether to keep using gather or to switch to scatter approach. Optimizations from both approaches will be discussed separately here.

4.1.1 CDLP gather optimizations

The gather optimizations preserved the same logics as illustrated in the baseline. However, having identified key weaknesses of the baseline implementation, these optimizations were proposed to mitigate them.

First optimization is to switch to a better sorting algorithm during histogram construction. The initial baseline used simple selection sort + hybrid quicksort as the

sorting algorithm. We decided to switch to a hybrid of selection sort + thrust::sort as thrust::sort is much more sophisticated and is your go-to GPU sorting algorithm for large cardinalities. A threshold is also exhaustively tested to decide when to switch between both algorithms.

Second optimization is to enable dynamic parallelism. Specifically, we enabled DP for the histogram construction. The original baseline suffers greatly from graph imbalance, and because the graph takes input from both the in and out nodes of a graph, there's no way to reorganize the graph such that it becomes less imbalanced. For one of the graphs we tested, *graph500-scale18-ef16_adj*, each node's degree varies from 10-10^5, resulting in a total computation time of 2000ms while other graphs generally take 200ms. To address this, first we enabled the DP version of thrust::sort. The frequency finding is also rewritten in a DP fashion. To find the most frequent item, we first do a binary search of the middle element of the sorted array to find the index where copies of this element start. We then split the array into two at this index to guarantee that all elements of the same labels are grouped together. We then iteratively call the splitting function using DP on these new arrays, until they are smaller than a given size or the maximum DP depth is reached. For each of the arrays, we do a sequential scan to find the label with the highest frequency and return the label with frequency. The parent call will receive these two frequencies from the two child calls and compare them to decide which label to return.

The final optimization is block-based write-back. Even though the reading of labels for all threads within a block is random and very hard to optimize using shared-memory, when they write their final labels back to the output they are well organized as each thread's assigned node index is calculated using thread index. Based on this observation, we could coalesce the inputs such that each block processes several chunks of continuous nodes, aggregating their results in shared memory before collectively writing them back to global memory.

With all three optimizations enabled, we are able to compare the baseline performance with that of the optimized gather.

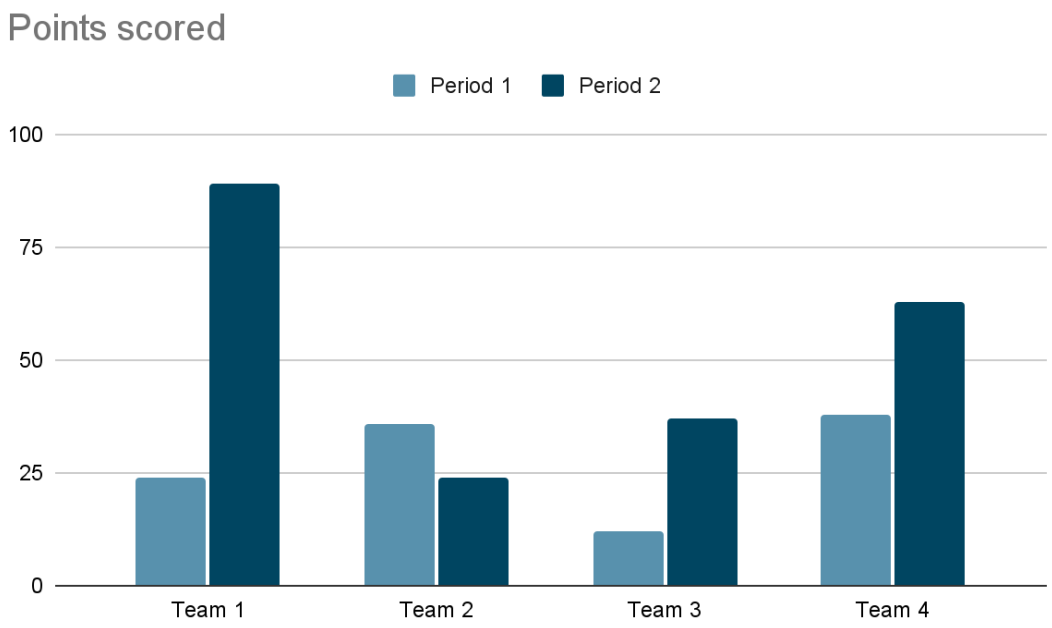

Figure 5. Performance results between the optimized CDLP gather approach (light blue) and the baseline CDLP approach (dark blue) on multiple graph sets.

4.1.2 CDLP Scatter Optimizations

A potential other set of optimizations that we considered and attempted to implement for CDLP was a scatter approach. This approach revolved around the idea that instead of a node querying its neighbors for their label, a node (and thus, a thread) would be responsible for adding its own label to the histogram of all its neighbors, and then once synchronized, would find the most frequent label in its own histogram and update its label. The motivation for this approach was that if we can transform the problem into a scatter problem, the data could much more easily be separated across multiple compute resources (such as different GPUs or even different machines).

The first challenge of this approach was that the graph had to be “transposed”, or in other words, all the edges of the graph had to be flipped so that each node knows which nodes connect to it. Initially, we tried to do this by actually transposing the COO edge matrix, however after identifying that this can actually be a complicated step, we instead decided to inverse the edges on importing the graph, so that almost no actually

processing time was added compared to the baseline as both have to read the graphs off the disk.

The second challenge was dealing with the large amount of global memory accesses that this approach used, due to the fact that the data could not be easily transitioned to shared memory (with a lack of locality between nodes and their neighbors in memory). Initial implementations of the scatter approach showed that almost 80% of the kernel computation time was spent accessing global memory. Though we were not able to fix the locality of neighbor's histograms and bring them into shared memory for a particular node, we did attempt to implement a sparse version of the histogram, that only adds buckets for nodes that have values. However, while the sparse histogram heavily decreased the memory footprint of the algorithm, it only increased the number of global memory accesses as you have to search for a label within the histogram before adding a label.

Overall, due to the constraints given on memory accesses, the scatter approach did not perform better than the gather optimizations and even performed worse than the baseline in some cases.

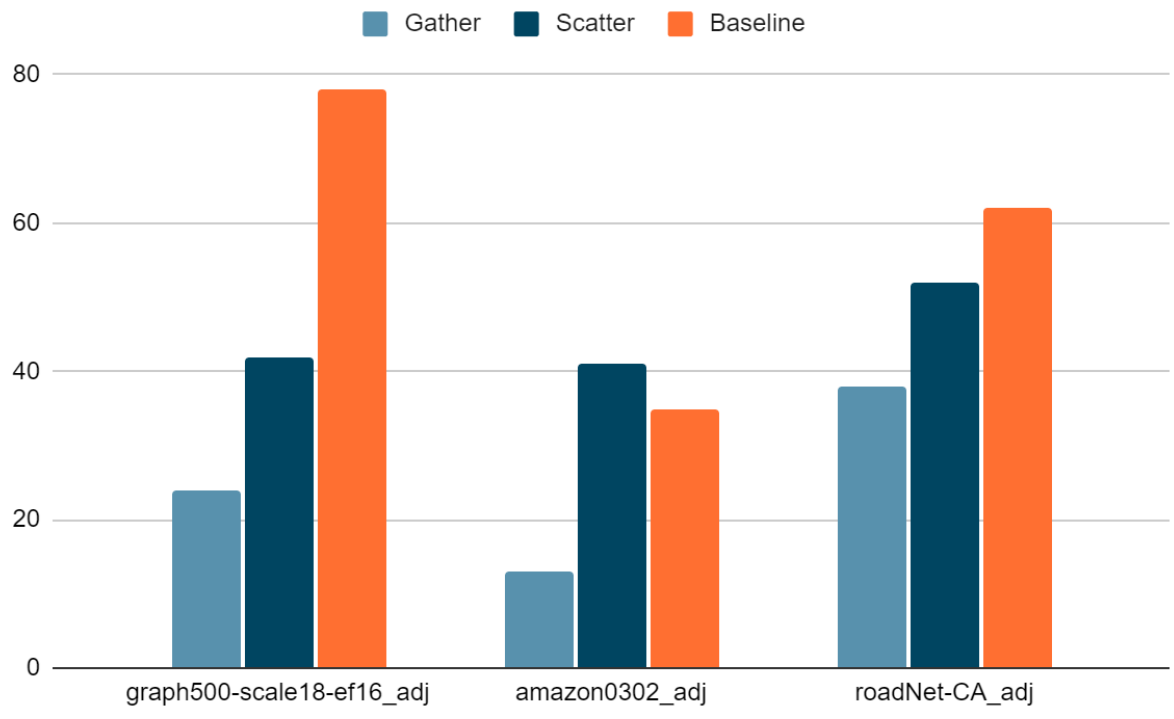

Figure 6. Performance of CDLP scatter approach vs. gather approach and baseline

Again, the original motivation for this approach was to try on multiple compute resources, but due to the poor performance even relative to the baseline, we were not able to test performance when having access to multiple resources.

4.2 LCC Optimizations

To optimize our baseline code, we initially planned to add dynamic parallelism when determining the union of each node's neighbor lists. We would launch threads for each of the elements in the smaller neighbor list with each thread performing binary search on the larger neighbor list. However, when trying to implement this, we had problems with running the dynamic parallelism and the code not executing properly. We then tried to launch only two threads to limit the number of resources being used by the GPU and split the neighbor list between the two child threads, but the issues with getting the code to run continued.

```
if (tend - ts >= 2500 && (tend - ts) / (end - start) >= 1.25) {
  int mid = (start + end) / 2;
  binary_search_kernel<<<1, 1>>>(start, mid, ts, tend, nodeNum, dstNode, edgeDst, triangleCounts);
  binary_search_kernel<<<1, 1>>>(mid, end, ts, tend, nodeNum, dstNode, edgeDst, triangleCounts);

} else {
  if (uDiff > vDiff && uDiff >= 64 && uDiff / vDiff >= 6) {
      // One node may have many edges, use atomic add
      x = binary_search_and_add(edgeDst, triangleCounts, vPtr, vEnd, uPtr, uEnd);
      // atomicAdd(&triangleCounts[nodeNum], x);
  } else if (vDiff > uDiff && vDiff >= 64 && vDiff / uDiff >= 6) {
      x = binary_search_and_add(edgeDst, triangleCounts, vPtr, vEnd, uPtr, uEnd);
      // atomicAdd(&triangleCounts[nodeNum], x);
  } else {
      x = linear_search_and_add(edgeDst, triangleCounts, vPtr, vEnd, uPtr, uEnd);
      // atomicAdd(&triangleCounts[nodeNum], x);
  }
  atomicAdd(&triangleCounts[nodeNum], x);
  atomicAdd(&triangleCounts[dstNode], x);
```

Figure 7. Code used for dynamic parallelism

We decided to launch four threads for each edge and divide the neighbor lists among the threads based on thread index. If the neighbor list had less than four elements, the threads without elements would be left idle.

We evaluated the performance of utilizing multiple threads per edge against the baseline (see graph). Notably, the optimized kernel performs much better on the

graph500-scale18-ef16_adj graph when compared to the baseline, averaging around 65 milliseconds on RAI as opposed to the baseline kernel averaging around 190 milliseconds on RAI. This result is expected because, for this graph, the neighbor lists were generally larger than the other two graphs. For the other two graphs, the kernel performs slightly better on the amazon0302_adj graph (9.55 ms vs 10.11 ms) and slightly worse on the roadNet-CA_adj graph (42.94 ms vs 41.45 ms).

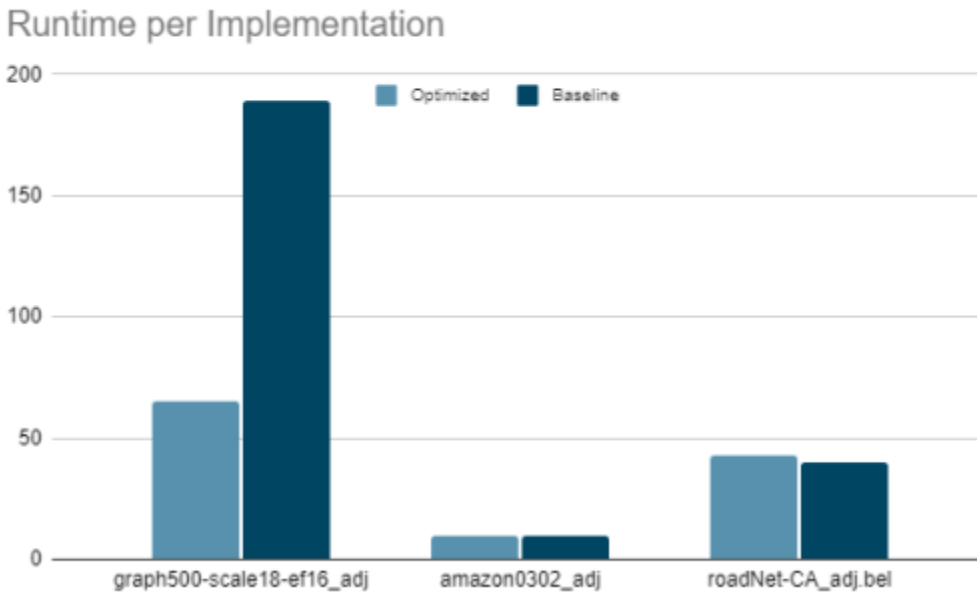

Figure 8. Performance of optimized LCC algorithm vs. baseline

5. Conclusions

Our LCC algorithm has definite improvements over the baseline for graphs with more connections between neighbors, as runtime was drastically reduced for graph graph500-scale18-ef16_adj graph. However, the load balancing issues for graphs with less connections between nodes can lead to worse performance than the baseline. For the real world use cases of LCC, graphs with lots of edges (social networks, computer networks, etc.) are more common than those with less edges. As such, we believe that these optimizations show positive results.

For CDLP, for the most part the gather approach optimizations performed better than the baseline by significant margins, while the scatter approach lagged behind due

to the overall increase in global memory accesses. Both approaches still suffer from an inability to use shared memory due to lack of locality in memory between nodes and their neighbors, especially in large and well connected graphs where the whole graph representation or even entire node's neighbor list may not be able to sit in shared memory. So while we believe the gather optimizations have value and show positive results, we think there may be a lot of work still to do resolving the locality issue in memory.

6. References

1. *Iosup, Alexandru. “The LDBC Graphalytics Benchmark.” The Linked Data Benchmark Council (LDBC), 2023, https://ldbcouncil.org/ldbc_graphalytics_docs/graphalytics_spec.pdf. Accessed 2023.*