PARALLEL CONNECTIVITY ALGORITHMS

CHASE HUTTON

ABSTRACT. We provide implementations and analysis for two parallel connected component labeling algorithms using the Parlay Library. The first is based off a parallel BFS and the second makes use of the so called Low Diameter Decomposition.

CONTENTS

1. INTRODUCTION AND PRELIMINARIES

Let $G = (V, E)$ be an undirected and unweighted graph. A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path. Our problem is as follows: label each vertex in the graph with the id of it's connected component, such that two vertices are labeled the same if and only if they belong to the same connected component. Connected component labeling can be solved and implement easily in the RAM model using a breadth-first search (BFS) or depth-first search (DFS). In the PRAM model however, the problem is more complicated. Theoretically efficient (work-efficient and polylog depth) solutions based on random edge sampling and minimum spanning forest algorithms are known but are believed to be too complicated to be practical. On the other hand, previously existing practical algorithms were unable to guarantee linear work bounds. For this project, we implemented the first practical linear work algorithm for connected component labeling [4]. Additionally, we provide an efficient implementation of Low Diameter Decompositions and a naive but practical alternative solution for connected component labeling using parallel breadth-first search.

2. BFS

Suppose that the vertices of the graph assigned unique id's from 0 to $n-1$. Consider the following sequential algorithm for connected component labeling. Iterate over the vertices of the graph. If vertex i is unlabeled, label it with i and perform a breadth-first search starting at i. For each vertex encountered during this BFS, label it with i. Using a parallel implementation of BFS we have a simple parallel algorithm for connected component labeling. The pseudocode for the algorithm is given below. While this algorithm is work-efficient, it has depth $O(D \log(|V|))$ where D is the diameter of the graph. Despite the weak theoretical guarantees, Algorithm 1 performed very well against a wide range of data inputs (see section 6).

Algorithm 1

```
$\ell_i \leftarrow -1, \forall i \in V$
for $i = 0$ to $n-1$ do
    if $\ell_i = -1$ then
        $\ell_i \leftarrow i$
        BFS(i,G)
    end if
end for
```

3. Low Diameter Decomposition

Suppose $S = \bigcup_{i=1}^{k} V_i$ is a partition of G. We define the strong diameter of each V_i to be the maximum length of a shortest path using only vertices in V_i between two vertices in V_i. We define a low diameter decomposition as follows:

Definition 3.1. Given an undirected, unweighted graph $G = (V_E)$, a (β, d) decomposition is a partition of V into subsets $V_1, \dots, V_k$ such that

(1) The strong diameter of each V_i is at most d
(2) The number of edges with endpoints belonging to different subsets is at most βm

A simple sequential algorithm to compute such a decomposition can be done using ball growing. The algorithm repeatedly picks an arbitrary uncovered vertex v and grows a ball around it using BFS until the number of edges incident to the current frontier is at most a β fraction of the number of internal edges.

Lemma 3.2. *Arbitrary ball growing with termination conditions described above produces a $(\beta, O(\frac{\log(n)}{\beta}))$ decomposition.*

Proof. Let $B_r(v)$ denote the ball of radius r centered at v. That is $B_r(v)$ is the set of vertices that are within r edges of v. Additionally, let $E_r(v)$ denote the set of edges in the subgraph of $B_r(v)$. Let ℓ denote the radius of the ball centered at v given by the above sequential algorithm. Then

$$m > |E_\ell(v)| \geq (\beta+1)|E_{\ell-1}(v) \geq \cdots \geq (\beta+1)^\ell$$

And thus $\ell \leq \log_{\beta+1}(m) = O(\frac{\log(n)}{\beta})$. Since the diameter of $B_r(v)$ is at most $2 \cdot \ell$ the result follows.

At first glance the above sequential algorithm seems inherently sequential as we only start growing a new ball once the current ball finishes. Miller, Peng, and Xu [3] propose an elegant solution to this limitation. As with the sequential algorithm it involves growing balls around vertices, but in this case in parallel. It's obvious that we can't start growing balls around each vertex at the same time as this would put every vertex in its own cluster. Miller, Peng, and Xu get around this by picking start times that are randomly shifted based on the exponential distribution. The pseudocode for the algorithm is given below. They show that such a procedure produces a $(\beta, O(\frac{\log(n)}{\beta}))$ decomposition in $O(m)$ work and $O(\frac{\log^2(n)}{\beta})$ depth with high probability.

Algorithm 2 Parallel Partition Algorithm

```
1: IN PARALLEL each vertex $u$ picks $\delta_u$ independently from an exponential distribution with mean $1/\beta$.
2: IN PARALLEL compute $\delta_{\max} = \max\{\delta_u \mid u \in V\}$
3: Perform PARALLEL BFS, with vertex $u$ starting when the vertex at the head of the queue has distance
   more than $\delta_{\max} - \delta_u$.
4: IN PARALLEL Assign each vertex $u$ to point of origin of the shortest path that reached it in the BFS.
```

We provided an implementation of low diameter decomposition as described in [2] using CMU's Parlay Library [1]. In order to increase efficiency, we make use of parlay's implementation of parallel block-delayed sequences (see [5]).

4. Contraction

Given a graph $G = (V, E)$ and a labeling of V that represent a low diameter decomposition of G, we would like to form a contracted graph G_c such that each cluster in G is a node in G_c and all duplicate edges and self loops generated during this contraction are removed. Furthermore, if a cluster becomes a singular node in G_c (i.e. no adjacencies), then this cluster represents a connected component of the original graph and should no longer be processed. Thus we need to relabel the nodes of G_c and exclude any singular nodes. We give an implementation similar to [2]. The main challenge for implementing this algorithm was efficiently updating the labeling of V. Our implementation involved a lot of overhead to achieve this which caused our connectivity algorithm's overall performance to suffer (see source files for more detailed implementation notes).

5. Work-Efficient Connected Component Labeling

Combining our contract primitive with the low diameter decomposition gives rise to the following algorithm for connected component labeling:

Algorithm 3 Connectivity

```
Input: $G = (V, E), \beta$
  $L \leftarrow LDD(G, \beta)$
  $G_c(V_c, E_c) \leftarrow Contract(G, L)$
  if $|E_c| = 0$ then
      return $L$
  end if
  $L' \leftarrow Connectivity(G_c, \beta)$
  $L'' \leftarrow \emptyset$
  parfor $v \in V$
      $L''[v] \leftarrow L'[L[v]]$
  return $L''$
```

At a high-level the algorithm repeatability clusters and contracts the graph until we arrive at a forest where each node represents a connected component in the original graph. Then, as we walk out of the recursion, we update the vertex labeling by assigning each vertex to the label assigned to its cluster in the recursive call.

6. Experimental Setup and Analysis

We benchmarked three algorithms for connected component labeling using Amazon Elastic Compute Cloud. For our AMI, we used an Ubuntu Server 24.04 LTS (HVM) SSD Volume Type with a 64-bit (x86) architecture. In all experiments we set $\beta = 0.2$. We were able to achieve significant speed-ups across all graph inputs. As mentioned above, the overhead required to maintain labels between contract steps caused performance issues. Namely, connectivity (Algorithm 3) only outperformed the naive BFS_CC (Algorithm 1) implementation for the rMAT graph using 32 cores. Despite it's inferior theoretical guarantees, BFS_CC benefited from it's minimal overhead. It may be possible to reduce the overhead required by our connectivity algorithm through a more refined implementation. Additionally, as the number of cores increased, the gap between BFS_CC shrunk dramatically. This was expected our connectivity algorithm can exploit a higher degree of parallelism than BFS_CC. Another point of interest was the relative performance of our connectivity and connectivityND. ConnectivityND, is an implementation of our connectivity algorithm without the use of parallel block delayed sequences. Connectivity seemed to slightly outperform connectivityND for most configurations with a decent gap on rMAT and usa-road using 4 cores. Interestingly the gap was more profound on configurations using 1 or 4 cores (See section 8 for detailed tables and figure).

7. References

[1] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. "ParlayLib - A toolkit for parallel algorithms on shared-memory multicore machines". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2020, pp. 507–509.

[2] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. "Theoretically efficient parallel graph algorithms can be fast and scalable". In: *ACM Transactions on Parallel Computing (TOPC)* 8.1 (2021), pp. 1–70.

[3] Gary L Miller, Richard Peng, and Shen Chen Xu. "Parallel graph decompositions using random shifts". In: *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2013, pp. 196–203.

[4] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. "A simple and practical linear-work parallel algorithm for connectivity". In: *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2014, pp. 143–153.

[5] S. Westrick et al. "Parallel block-delayed sequences". In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2022. DOI: 10.1145/3503221.3508434.

8. Figures and Tables

TABLE 1. Execution times (in seconds) for parallel computation experiments on AWS EC2, rounded to three decimal places. Times exceeding 300 seconds are noted as aborted (marked > 300).

Graph	1 vCPU	4 vCPUs	16 vCPUs	32 vCPUs
rMAT				
BFS_CC	> 300	18.159	7.521	5.984
connectivity	> 300	35.942	11.100	5.101
connectivityND	> 300	45.864	10.427	5.515
youtube				
BFS_CC	2.764	0.768	0.238	0.157
connectivity	12.237	3.388	0.981	0.578
connectivityND	15.367	3.699	1.035	0.517
com-orkut				
BFS_CC	> 300	9.654	2.416	1.234
connectivity	> 300	21.296	5.663	2.892
connectivityND	> 300	22.539	5.928	2.964
LiveJournal				
BFS_CC	> 300	5.133	1.440	0.837
connectivity	> 300	17.331	5.541	3.302
connectivityND	> 300	18.831	5.208	2.657
usa-road				
BFS_CC	> 300	14.038	6.279	5.001
connectivity	> 300	65.140	19.980	10.515
connectivityND	> 300	75.668	22.817	11.594

Graph Dataset	Num. Vertices	Num. Edges	CC
LiveJournal	4,847,571	85,702,474	1876
com-Orkut	3,072,627	234,370,166	187
youtube	1,138,499	5,980,886	930
usa-road	23,947,348	57,708,624	2
rMAT	8,388,608	199,729,542	667,942

TABLE 2. Graph inputs, including vertices and edges. CC is the number of connected components of the graph. All graphs are undirected

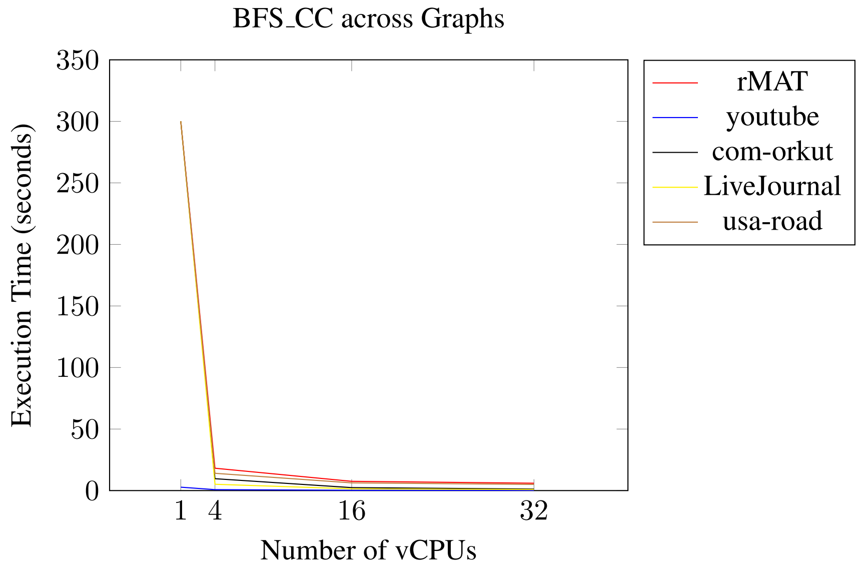

FIGURE 1. Performance of BFS_CC on various graphs across different core configurations.

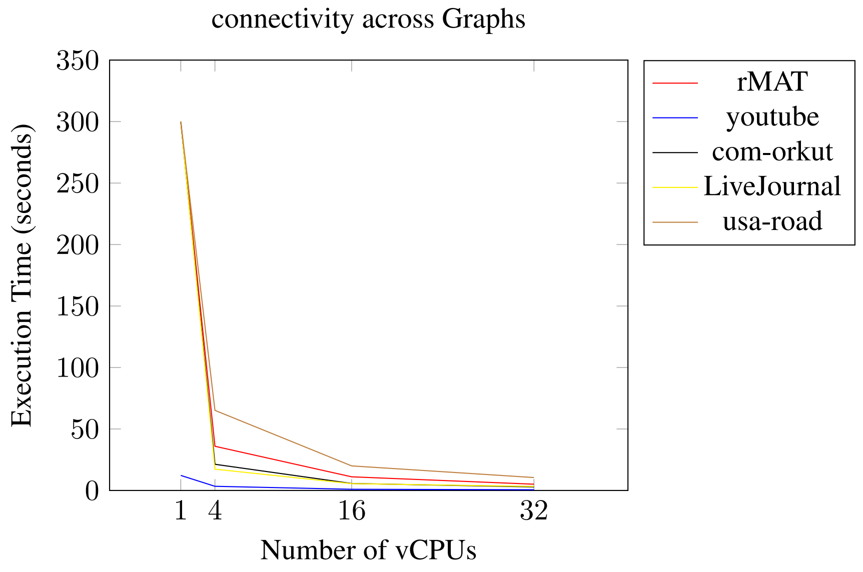

FIGURE 2. Performance of connectivity on various graphs across different core configurations.

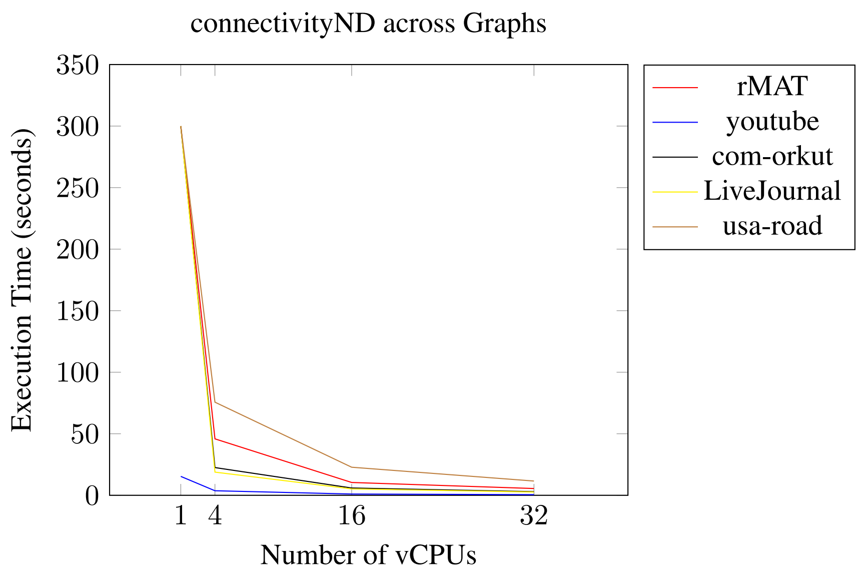

FIGURE 3. Performance of connectivityND on various graphs across different core configurations.