December 13, 2019

Abstract

Summary Of The Problem

1 Introduction

2 Literary Review

As we approach the limit of De Morgan's law, we come to an age in computing where sequential computing is limited by processing speed. Using RAM (Random Access Memory) as our base model of computation we create optimal sequential algorithms. However, in the age of big data, these sequential algorithms are not powerful enough to process all the data in reasonable time. Thus, we turn to parallel computing to where we have two models of computation which attempt to divide work. The first model assumes that all knowledge is known about the problem, then divides work base accordingly; this is known as the field of parallel computing. The second approach assumes that only local information is known about the problem, and through a series of steps, protocol definitions, and message passing an answer can be derived/approximated; this is known as the field of distributed computing. However, there are tradeoffs to both approaches like : speed, run-time, correctness, optimality. Most algorithms take a tradeoff between these two fields as computation gets further away from processors.

In this age being able to extract information form graphs becomes more prevalent in analyzation of web link graphs, social networks, and many other graphs types. One property that we can extract is graph connectivity. In graph theory graph connectivity is the minimum number of vertices, or elements that need to be removed to separate nodes into isolated subgraphs [7]. A very similar definition of this is the minimum cut of a graph where a minimum cut is defined as the minimum partition of verticies, and edges into two disjoint sets. For simple unweighted graphs David R. Krager et al [8] created a randomized sequential algorithm based on properties preserved when edge contraction is applied. The papers by David R. Krage et al [9], and Barbara Geissmann et al. [10] further improve the algorithm by adding parallelisation. The downside of parallel algorithms introduced is that they are still based on the Parallel Random Access Memory (PRAM) model which can be used to prove theoretical bounds

on algorithm. But the algorithms introduced disregarded hardware limitations where CPU cache is finite, and message passing to start up threads/processes are costly. It is noted that the paper in Barbara Geissmann et al. [10] attempts to improve cache misses by creating monotonic sequences to be read, however, the paper still relies heavily on parallelism of subroutines.

Another version of this problem comes from weighted undirected graphs. The goal is to find a cut where the sum of the edge weights are minimum. The Stoer-Wagner algorithm by Mechthid Stoer et al. [11] is a good sequential recursive algorithm for solving undirected weighted graphs with non-negative weights in $O(|V||E| + |V|^2 \log |V|)$ time. Usually the min-cut with respect to weighted undirected graphs is linked with the maximum flow problem as the minimum cut is often the bottleneck of the network [12,13].

A subfield of graph connectivity is connected components. Connected components can be divided into two categories: strongly connected components (SCC), and weakly connected components (WCC). A graph is said to be strongly connected if every vertex is reachable from every other vertex. Then a SCC is the same definition, but the maximal subgraph that keeps this property. The SCCs of a directed graph form partitions of subgraphs where they themselves are strongly connected. WCCs of a directed graph form a subgraph for every pair in the subgraph there exists a path, but not necessarily a bijective path.

One of the earliest algorithms introduced is Tarjan's algorithm which uses the properties of depth first search (DFS) to find SCC given by Robert Tarjan [16]. Later Edsger W. Dikstra came up with the path-based strong component algorithm seen in [17]. A simpler algorithm was also found after named Kosaraju's algorithm and published by Micha Sharir [17]. However, all of these algorithms abuse depth first search to find SCC, yet for parallel algorithms DFS is hard to paralyze. In fact, it was proved by John H. Reif that DFS is P-complete [19]. Therefore parallel algorithms do not focus on DFS when computing strongly connected components seen in Lisa K. Fleischer et al. [20], and Sungpack Hong et al. [15].

There is another version of Connected Components in undirected graphs. Unlike the directed version, if there is a path, then that path is a SCC since edges are bidirectional. Therefore the undirected version of SCC focuses on finding disjoint subgraphs in the original graph, since each disjoint subgraph is itself a SCC of the original graph. One may also relate this to finding WCC in a directed graph, since a path represents a bijective edge in undirected graphs. However, by doing so the WCC may contain a SCC in the original graph. A simple sequential algorithm can be done to find this where you take a DFS, or a breadth first search (BFS). Once you have exhausted your search, you select a non-traversed vertex to find the next SCC [21]. There have also been algorithms for dynamically changing graphs researched by Yossi Shiloach et al. [22].

As graphs increase in size, so does the amount of processing power required. In the field of SCC on undirected graphs there have been several subfields in which parallelization has taken affect. The first parallel algorithms dealing with SCC on undirected graphs deal with spanning forests [23,24,25,26,27,28,29,30,31,32,33,34,35,36,37]. The algorithms by Uzi Vishkin et al. [36,4], and Awerbuch et al. [37] work by combining vertices into trees, such that a constant number of vertices are deleted every iteration, but does not guarantee a constant fraction of edges. Thus they require $O(|E|log|V|)$ work. The random algorithm by Reif, and Phillips [?] work by contracting vertices in the same component, which guarantees a constant number of vertices every iteration, however, it again does not guarantee a constant fraction of edges. Thus the expected work is $O(|E|log|V|)$.

Work-efficient polylogarithmic-depth parallel connectivity algorithms have been designed in theory [38,39,40,41,42,43]. These algorithms are based on random edge sampling, or filtering edges. However, these have complicated structures which may be impractical to implement in practice.

Min-Label Propagation

BFS-CC

The algorithm which we will be focusing on is based on Shilovach-Vishkin[4] which has $O(\log n)$ parrallism, an uses a maximum of $2|E| + |V|$ processors. Shilovach-Vishkin represent their CC as a forest where each tree represents a connected component. They have two key phases *hook*, and *shortcut*. *Hook* is reponsible for connecting processed components together where it hooks new nodes with their respective component, or connectes two componnets together given the current knoldge know of the graph. They prove that this process results in tree that have height at most $\lfloor \log_{3/2} n \rfloor + 2$. The *shortcut* phase results in a compression of the trees in the forest such that they have depth at most one.

3 Statment of Problem

The key problem with Vishkin-Shilovach is that work was done on a per-node basis, such that there may be duplicate reads to the same edge. Furthermore, if there is a large component in the graph, then the algorithm would read all edges that exist within that component. Thus, if we have a large complete graph, then all $|E| = |V|^2$ edges in this complete graph would be traversed, when only $|V|$ edges are needed inorder to determine a CC.

A solution to this is to sample for the largest component in the graph, and skip any node not processed in this component. This algorithm was proposed by Michael Sutton et al.[1] where they attempt to do superlinear work on the

number of edges in the graph, such that no edges are visited twice. The algorith that they present is called *afforest*, which is similar in nature to the algorithm presented by Shilovach-Vishkin. We will be performing a review on *afforest* guaging performace, and filling in missing proofs not fully described in the original paper.

- state research question : to analyze the performance of afforest
- why question is answered
- discussion why this question is worthwhile

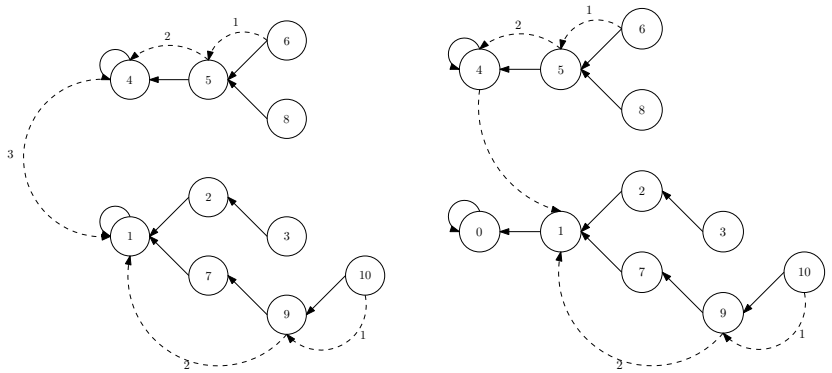
Afforest

Similar to Shilovach-Vishkin, afforest's underlying data structre is a forest. In this forest each tree is represented as a connected component found thus far in the view of the processed graph. For all $v \in V$, v has a unique identifier. We denote the symbol $\pi(v)$ to represent the parent of v, such that we hold an invariant inwhich $\pi(v) \leq v$. We note that if v is the root then $\pi(v) = v$.

The basis of afforest is contingent on two operations : link, and compress. Link is seen in Algorithm 1. Link ensures that two verticies will be in the same component tree. Link searches the tree(s) for a root with the larger identifier. Upon finding the root with a larger identifier link attempts a Compare and Swap (CAS) opeartion inorder to connect the larger tree to the smallar identifier. The smallar identifier does not nessarily need to be the root of the tree. We can see that our invarient still holds.

Algorithm 1 Link Operation

```
1: procedure LINK(edge(u, v) , π)
2:     p1 ← π(u)
3:     p2 ← π(v)
4:     while p1 ≠ p2 do
5:         h ← max{p1, p2}
6:         h ← min{p1, p2}
7:         if compare_and_swap(π(h), h, l) then return
8:         p1 ← π(π(h))
9:         p1 ← π(l)
```



(a) Case where CAS happens at both roots

(b) Case where CAS happens at one root

Figure 1: Link operation traversing up component trees linking edge(6,10)

In figure 1, we see the opeation of adding the edge with $u = 6$, and $v = 10$. We see that in both cases we traverse up to the largest root 4, and we connect the tree rooted at 4, to the tree rooted at 1. It can be seen that not matter where we connect root 4, there is always a path from from every element in the tree rooted at 4 to the new component tree root 1.

Lemma 1. If the endpoints of link u, and v are within the same component tree, then the ancestors of the component tree do not change. Since a CAS operation is only triggered when the largest node is the root, and since our invarient proves that the root is the smallest value in the tree, the CAS opeartion is never triggured. ■

Lemma 2. If the endpoints of link u, and v are not within the same component tree, then the link operaiton will ensure that both trees are merged into one component tree. There are three possible cases that we need to conver to prove this.

1. If the processors' h remains the root, and CAS succeds, then we know that h, and l are directly connected. Resulting in one connected component

2. If another processor connecets h to the tree, then we apply lemma 1. This results in one connected component.

3. If another processor connecets h to l' such that l' does not live in the tree of u, nor the tree of v. Then $h \neq \pi(h)$. We then repeat searching up the tree, and we end up with the same problem as decribed in Lemma 2. Since there are a finite number of vertecies, then there are also a finite number of components. Therefore the operation will complete results in one connected component containing both u and v.

■

Lemma 3. If we apply link to all edges, then the resulting component tree represents the CC of the original graph. $\forall edges(u, v)$, u and v will be in the same component since we apply link to all edges. Then by Lemmas' 2, and 3 all edge pairs will be in the same component.

By defenition a CC is connected if there exists a path between all pairs in the component. If we take one such path $x_1, x_2, ...x_{n-1}, x_n$ then link would be applied to : $(x_1, x_2), ..., (x_{n-1}, x_n)$. Which means all the nodes in the path will be in the component tree. If we take take the set of all paths and convert it to the set of all edges, then every vertex that is an element of a CC C_i, is also an element of a unique component tree coresponding C_i. Since we apply link to all edges, then we cover the set of all possible paths. Therefore the a resulting component tree represents a connected component in the original graph. ■

- how does afforest work
- what is the underlying data-structure for afforest (forest)
- compare and swap operation (what it does, pros, cons)
- link operation
- Case A : in a connected component
- Case B : not in a connected component
- any modification does not modify previous ancestors (point out that this can become a bottle neck to the algorithm) - compress operation
- show that any compress operation results in a degree 1 tree
- any race condition results in a shorter path
- expected height of the tree
- union find analysis (incremental randomization)
- probability that it will deviate
- further improvement : parallel find head of linked list
- d-regular graphs
- techniques that they use to sample (edge sample vs neighbour sample)
- problem with generalization of
- Why we want to sample our graph
- How the lemma above kinda explains our goal
- display algorithm and explain each step - finding most frequent element (proof) (not well explained in paper)
- explain algorithm complexity :
- worst case
- expected case
- best case
- improvements on the algorithm (use a find linked list head)

Implementation Details

- how is the forest represented
- how is the graph stored
- how do we choose our sample

4 Experimental Evaluation

- running time - speed up compared with no random sampling - [implement thread distribution running time]

5 Improvements

- how do we reduce the number of CAS operations (since they are expensize) - on method could be to go up the entire tree (if two items are in the same component then we don't change a path) - downside (we need to traverse more of the tree) - find a fast parallel pointer jumping algorithm for multiple lists - downside (if we did this we are looking for a path of size logn, in a pool of lists of size n)

6 Conclusions

7 Bibliography

[1]https://ieeexplore-ieee-org.proxy.library.carleton.ca/document/8425156/references#refere
[2]https://ieeexplore-ieee-org.proxy.library.carleton.ca/document/6877288/references#refere
[3]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=2733089
[4]https://www-sciencedirect-com.proxy.library.carleton.ca/science/article/pii/019667748290
[5]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=2612692
[6]https://link.springer.com/chapter/10.1007/3-540-45591-4_68
[7]http://diestel-graph-theory.com/index.html
[8]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=234534
[9]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=331608
[10]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=3210393
[11]https://dl.acm.org/citation.cfm?id=263872
[12]https://link.springer.com/chapter/10.1007/978-0-8176-4842-8_15
[13]https://dl.acm.org/citation.cfm?id=1942094
[14]https://dl.acm.org/citation.cfm?doid=362248.362272
[15]https://dl.acm.org/citation.cfm?id=2503246
[16]https://ieeexplore.ieee.org/document/4569669
[17]https://dl.acm.org/citation.cfm?id=550359
[18]https://www.sciencedirect.com/science/article/pii/0898122181900080
[19]https://www.sciencedirect.com/science/article/pii/0020019085900249

[20]https://dl.acm.org/citation.cfm?id=663154
[21]https://dl.acm.org/citation.cfm?id=362272
[22]https://dl.acm.org/citation.cfm?id=322235
[23]https://www.sciencedirect.com/science/article/pii/0020019082901314
[24]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=739745
[25]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=305744
[26]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=254195
[27]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=182530
[28]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=359141
[29]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=214077
[30]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=106163
[31]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=203622
[32]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=358650
[33]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=72952
[34]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=586952
[35]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=1398907
[36]https://www.sciencedirect.com/science/article/pii/0166218X84900192
[37]B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and
[38]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=237563
[39]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=123143
[40]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=247524
[41]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=377897
[42]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=586952
[43]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=686427
[44]S. Hambrusch and L. TeWinkel. A study of connected component labeling algorithms on the
[45]S. Goddard, S. Kumar, and J. F. Prins. Connected components algorithms for mesh-connect
[46]T.-S. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finc
[47]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=1196220
[48]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=1079430
[49]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=2358680
[50]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=2312018
[51]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=746654
[53]A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on c
[54]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=231641
[55]E. Caceres, H. Mongelli, C. Nishibe, and S. W. Song. Experimental results of a coarse-g
[56]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=1869229
[57]J. Soman, K. Kishore, and P. J. Narayanan. A fast GPU algorithm for graph connectivity.
[58]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=2192614
[59]https://dl.acm.org/citation.cfm?id=2059488