OPTIMIZING ROAD NETWORKS USING SHORTEST PATH ALGORITHMS: A COMPARISON OF SEQUENTIAL AND DISTRIBUTED APPROACHES

CSE 6140 Project Report

Presented by:

Shankar Vishwanath

Chandra Prakash Khatri

Presented to: Prof. David Bader

Fall 2013 - December 6th, 2013

College of Computing

Georgia Institute of Technology

Contents

Abstract

In current study, shortest path algorithms are analyzed. The distributed vertex-oriented approaches have been compared with the conventional shortest path algorithms, which are mostly edge-oriented approaches. However, the study is dealt for obtaining the shortest distance for road networks, similar concepts can be applied to any kind of network.

Introduction

Optimizing the road networks has always been the biggest challenge since the inception of road infrastructure. Finding the shortest path from one location to another location is one of the first problems of the computational researchers. We represent social networks, computer networks, protein regulation networks etc. in the forms of graphs. Road networks, which can be correlated to the graphs are analyzed using the graph algorithms. Researchers have proposed many different kinds of algorithms and approaches to find the shortest path. The shortest path problem (SPP) has been classified into three categories: The Single-Source Shortest Path (SSSP), All-Pairs Shortest Path problem (APSP) and the single-destination shortest path (SDSP). In SSSP, we compute the shortest paths from a source vertex *v* to all other vertices in the graph; in APSP we find the paths between every pair of vertices, v and v', while SDSP can be computed by reversing the arcs in directed graph and applying the SSSP.

Dijkstra has proposed the algorithm for the SSSP and still is one of the widely used in the industry due to its simplicity. This algorithm keeps the current shortest distance values from the starting vertex v to all of other vertices, once a shorter path is found by extension of directly

connected edges (e.g. distance from P to Q to R is shorter than current P to R value), the current value will be replaced with the shorter one. In the end of the algorithm, all of distance values can be guaranteed to be the shortest paths from v to all of other vertices. A limitation for the Dijkstra's algorithm is that it is applicable only to the positive edge weights. Bellman and Ford have suggested the algorithm if the edge-weights are also negative, commonly known as Bellman-Ford algorithm. To find the APSP there is Floyd-Warshall algorithm. A* algorithm was proposed to perform heuristics to speed-up the search. Furthermore, Johnson's algorithm is used to solve the SPP problem faster than the Floyd-Warshall algorithm when the graphs are sparse. Although these are standard forms of the SPP, there are many variations in solving these problems. Most of these algorithms are designed from two basic algorithms, i.e. Dijkstra's Algorihtm and Floyd- Warshall Algorithm. One such variation is by using Dijkstra's algorithm with Fibonacci heap. This algorithm was proposed by Fredman and Tarjan in 1987 and It reduced the complexity from $O(V^2)$ to $O(E+V \log V)$.

Zhan and Noon depicted that best performing implementation for solving the SSSP problem is Pallottino's graph growth algorithm when it is implemented with two queues (TQQ). Moreover, they suggested that for one-to-one and one-to-some shortest paths Dijkstra's algorithm offers some advantages, since it gets terminated when the shortest distance is obtained. The another variation of the SPP problem which we are considering in this paper is Traveling Salesman Problem (TSP), wherein the aim is of finding the shortest path that goes through every vertex exactly once, and returns to the start. Usually, the SPP can be solved in the polynomial problem for non-negative cycles; the TSP is an NP-complete problem and is considered as not to be efficiently solvable when the data is large. Furthermore, finding the longest path is also an NP-

complete problem. All the above mentioned algorithms work well when the data set is small, but because of being polynomial and NP-complete (TSP) when the data is large, which is the case of road networks where we have millions of nodes and edges (California road network~ 1,965,206 nodes, 5,533,214 edges), then these methods becomes computationally expensive.

In current study, we aim to analyze the performance and implement the shortest path problems for massive data set by applying parallelism and distributed computing. We depict parallelism for different SPP and related algorithms. Typical graph mining algorithms assume that the graph fits in the memory locally on the single disk. However all these real-life graphs are massive and doesn't follow the case. Spanning multiple Giga-bytes and heading to Tera-bytes and Peta-bytes of data. In such cases parallelism stands-out and act like an elixir. In the last decade social-graphs and web-based graphs have grown to the size of billion edges and trillion nodes, which has brought the attention of many researchers. Low cost of storage for these kinds of massive scale graphs, and recent successes in different parallel approaches have depicted immense potential. How to perform graph based computation efficiently in a distributed environment is the main purpose of these researches.

Parallelization and Distributed Computing

Generally, there are two categories of graph-based algorithms, vertex-oriented and edge-oriented, Most of the previously mentioned algorithms are edge-oriented ones. For distributed computing it is preferred to use vertex-oriented computing, Google. Therefore, in the current study for distributed computation, vertex-oriented approaches have been adopted. In 2004, Dean et. al. and Aggarwal et. al. came up with a programming framework called Map-reduce, which is used to deal with huge amounts of unstructured data in a massively parallel way. Map-reduce provides

many advantages: (i) functional-programming, which provides just two functions called map and reduce to execute a task (ii) the programmer can track the data distribution, load-balancing and replication during execution of a task.

In this paper we have performed our analysis using Apache Hadoop and Apache Giraph, which are both open source frameworks which implements Map-reduce and are meant for analyzing large-scale data. There are several benefits of Hadoop such as fault tolerance, low maintenance and is very powerful for large-scale graphs. However, the Hadoop project is still young and immature. The weaknesses of Hadoop have manifested themselves mainly in the areas of Resource Scheduling , Single Point Failure etc. Hadoop implicitly handles the communication from node to node there by making it reliable. It is very robust when concerned with the data congestion issues. In the case of a node failure it restarts the tasks on other processing machines. Hadoop has some disadvantages like the input data elements cannot be updated. It does not provide any security model or safe guards. As it consumes some time to start the map reduce tasks it may not show optimal performance with a small quantity of data processed over a small number of nodes.

Apache Giraph is built on top of Hadoop. The difference between Giraph and Hadoop is that it is optimized only for large-sclae graphs. Giraph, is also a distributed and fault-tolerant system that adopts the Bulk Synchronous Parallel programming model. After Google introduced the Pregel System, which follows a vertex-centric approach, adopted from Bulk Synchronous Parallel (BSP) model. BSP model synchronizes the distributed computation by Super-step. In one Super-step, each machine performs independent computation using values which are stored in its local memory. After computation, it exchanges information with other machines and waits until all of

machines have finished the communication. A user-defined function for each vertex is computed in super-step by the framework in parallel. The user-defined function specifies what kind of the function is to be performed for a single vertex V and a sing super-step S. The function read messages that are sent to *V* in super-step *S-1*, send messages to other vertices that are received at super-step *S+1*, and modify the state of *V* and its outgoing edges. Messages are typically sent along outgoing edges, but you can send a message to any vertex with a known identifier.

A salient feature of Giraph is its simple, yet effective, programming model. A Giraph programmer has to think like a "Vertex". In other words, the programmer implements a method, called `compute()`, that is invoked for each visited (active) vertex. The Compute method performs the following tasks: (I) receives messages sent to the vertex in the previous superstep, (ii) computes using the messages, and the vertex and outgoing edge values, which may result in modifications to the values, (iii) may send messages to other vertices. The Compute method does not have direct access to the values of other vertices and their outgoing edges. Inter-vertex communication occurs by sending messages.

One such library available for parallelizing graph algorithms functionalities is CGMgraph. There are two drawbacks of CGMgraph; it is not an infrastructure/framework to let programmer design the algorithms themselves, since it tried to give out well implemented parallel graph algorithms..

Microsoft has also involved in researches of large scale graph mining and processing. Surfer is alarge graph processing engine implementing Map-reduce, designed to execute in the cloud.

Test Configurations

Hardware

Hadoop was installed in a pseudo distributed mode on a normal laptop with processor speed of 2.4 Ghz, 8 GB RAM, Intel i3 dual core system. Other algorithms included Giraph implementation and the Fibonacci heap implementation of shortest path algorithm were done on the same configuration.

Software

OS: Linux Mint 15

Software : Hadoop, Giraph, Eclipse, Java

Implementation

Dijkstra's Algorithm using Fibonacci Heap

We implemented Dijkstra's shortest path algorithm using a minimum priority queue.

We add all the vertices/nodes to the priority and then start computing shortest path by extracting minimum element from the priority queue.

The Fibonacci heap implementation is asymptotically the fastest known shortest path algorithm for non-negative digraphs.

The Fibonacci heap allows insertion and find min in O(1) time and takes O(log n) for delete node and delete min. It has a better amortized running time than other heap implementations of priority queue.

Pseudocode

```
Dijkstra(source: vertex, target: vertex)
    Visited = empty vertex set
    toVisit = empty priority queue
    Insert: source → toVisit with priority 0
    Insert: all other vertices → toVisit with priority INF
    while: toVisit → !empty
        extract (v,d) ← toVisit
        if v = target then return d
        add n to Visited
        for each arc from vertex v to other v' with distance d':
            if v' is not visited then add v' to toVisit with priority d+ d'
        done
        #return No Path
```

Complexity

The Fibonacci heap implementation of Dijsktra has time complexity of ***O(E+V log E)***.

This is an improvement from the binary heap implementation which has a complexity of ***O(ElogV)*** and the naïve implementation which takes $O(n^2)$ time.

For dense graphs the number of **edges E** is comparable to $\mathbf{n^2}$ in which case the Fibonacci heap has better time performance than binary heap implementation.

Hence in order to test shortest path algorithm on large scale graphs we chose the Fibonacci heap implementation.

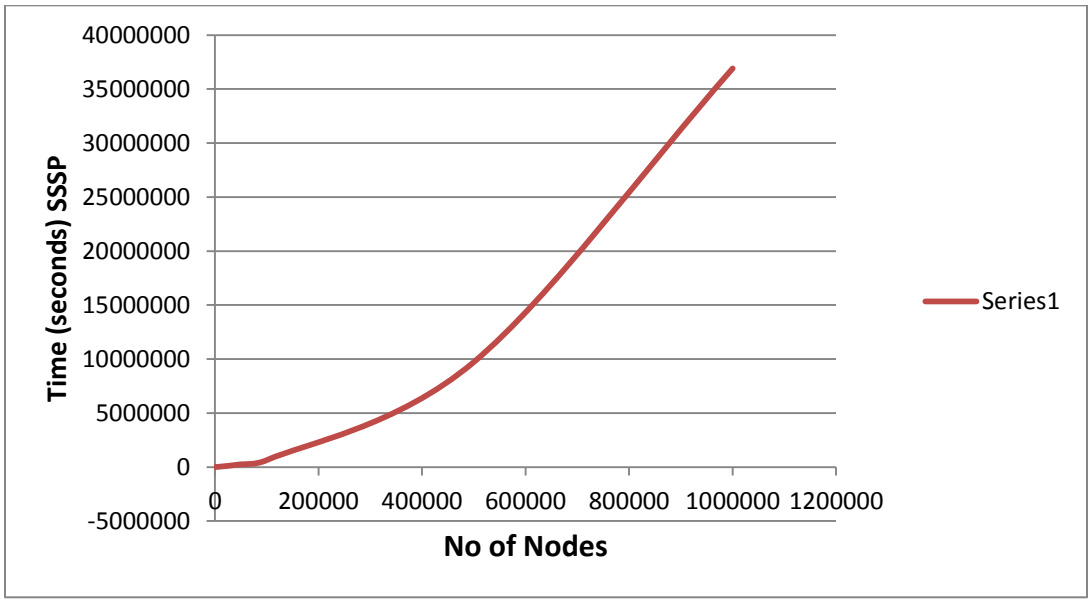
Observation

Our primary objective is to compare shortest path algorithms on large scale California Road Network Graph. Since both Bellman Ford and Fibonacci heap implementation of Dijkstra's algorithm have similar times, we calculated a single source single destination shortest path and

approximated it to Single Source Shorstest Path (SSSP), by taking a number of readings for the same graph and taking the worst case time and multiplying it by the number of nodes.

1 Time vs No of Nodes - SSSP Sequential

No of Nodes	CPU Clock Ticks	Time in seconds	Time (sec) - SSSP	Minutes	Hours	Days	Years!!
50	3409	3.409	170.45	2.840833			
200	3415	3.415	683	11.38333			
500	3327	3.327	1663.5	27.725			
1000	3300	3.3	3300	55			
5000	3498	3.498	17490	291.5			
10000	3673	3.673	36730	612.1667	10.20278		
50000	5078	5.078	253900	4231.667	70.52778	2.938657	
100000	6416	6.416	641600	10693.33	178.2222	7.425926	
500000	19445	19.445	9722500	162041.7	2700.694	112.5289	
1000000	36910	36.91	36910000	615166.7	10252.78	427.1991	1.170408



1 Time vs No of Nodes

Analysis

It is quite obvious that even with the implementation of a Fibonacci heap beyond a certain number of nodes calculating the shortest path sequentially is unfeasible.

From the graph it can be seen that the as the no of nodes and so the edges of the graph increased the CPU runtime too grew exponentially.

Hadoop – MapReduce : Shortest Path Implementation

The implementation of SSSP on Hadoop-MapReduce was done in two parts viz. 1. finding complete road network data and creating an adjacency list which our program accepted and 2. running our algorithm on Hadoop's map reduce framework.

While public road network data was available freely, they were mostly in the form of edge lists. Moreover most of these datasets had whitespaces and blank lines randomly inserted which meant that they frequently caused errors in our program. We wrote a program on java to remove whitespaces, blank lines and create an adjacency list off the road network edge list.

We used google multi-map library in creating the adjacency list for the undirected graph.

Pseudocode

Hadoop Driver: receive info on Mapper, Reducer class & i/p o/p files

Mapper → takes line and breaks into keys and values

Reducer → combines keys and keeps only the lowest value associated with each key

If: Reducer !combine → pass new <K,V> pair

Mapper Class

D ← N.distance
Emit(nodeID, N)

```
        forAll nodeID m in AdjList
                Emit(nodeID m, D+1)
Reducer Class
        Dmin ← INF
        M ← null
        forAll: D in number[D1,2…N]
                if isnodeD then
                        M ← D
                Else
                        Dmin ← D
        M.Distance ← Dmin
        Emit(nodeID m, node M)
```

The intuition behind all this rather simple. At the first step all the nodes except the sources node have distance INF, the source node has distance 0. The algo works by taking each node and looking at its adjacency list. It then tells the connected nodes its current distance from the source + 1. The next node then will update its own distance. We keep repeating this process until the minimum distances for all of the nodes stop changing. This means that the algorithm has reached convergence and most likely a shortest path has been found.

Complexity

The time complexity of this implementation depends on the number of map and reduce tasks. If there are M Mappers with t threads for each map and time Ti to initialize a single mapper then total time to start all map tasks will be M*Ti. And if Nd is the no of distances of each node computed and td is the time take by it, then total time is Nd*td.

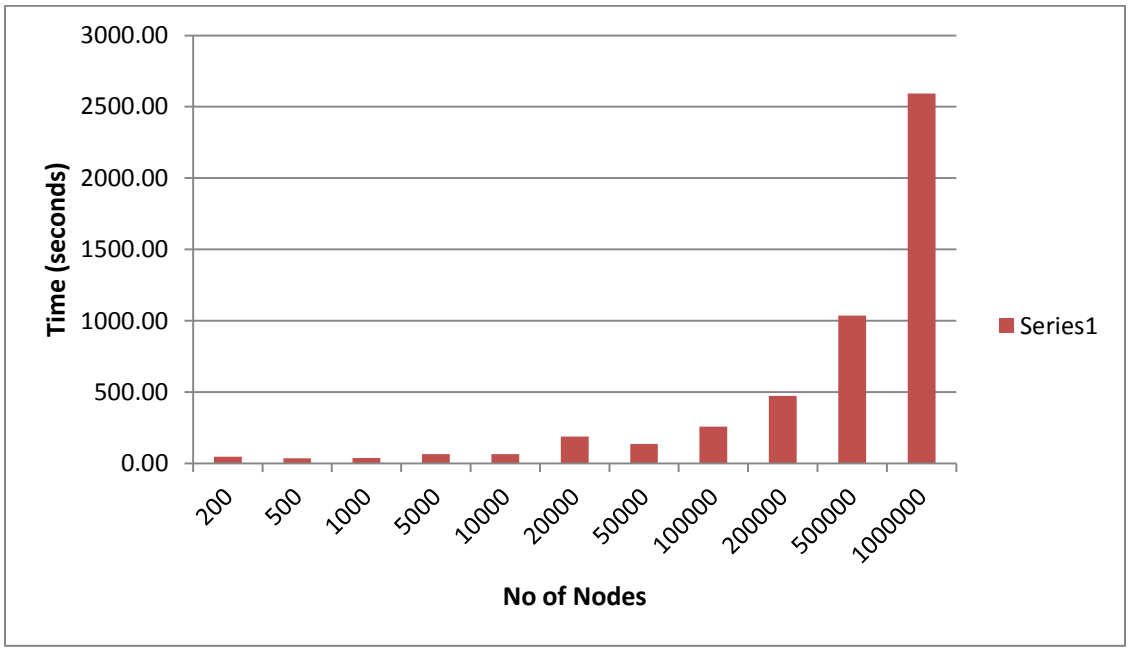
Hence total time for Map Phase is **O(MTi + Nd*td).**

Similarly the time for Reducer phase will be of order **O(Rti + Nr*tr).** Referring to a paper from ***RIT,NY*** we can say that the time complexity of the shortest path algorithm on mapreduce is on an average $\mathbf{O(n*(\log n)^2/MT) + O(n*(\log n)^2/RT)}$ where M and R are number of mapper and reducer tasks and T is the time taken.

Observations

No of Nodes	Time (seconds)
200	47.36
500	35.77
1000	37.37
5000	65.57
10000	65.88
20000	189.28
50000	136.32
100000	257.55
200000	474.04
500000	1036.94
1000000	2591.87

2 Hadoop Nodes vs Time



2 Hadoop : No of Nodes vs Time

Analysis

It is clear from the table and graph above that Hadoop provides a much faster implementation of the SSSP algorithm compared to the sequential implementation.

We believe if the number of clusters working in parallel are increased then the computation will become faster up to a certain limit. While the hadoop algorithm expands on all reached or visited nodes at every step, it is still faster than Dijkstra which expands to only the current cheapest node. This is because of the distribution.

Disadvantage: While moving toward convergence a lot of the data remains unchanged yet it is necessary to reprocess and reload data for each iteration. This results in wastage of I/O operations, network and power resources. Hadoop is more suited for processing data from static and dynamic databases like Hive and Pig where large data updates and changes have to be tracked, than for machine learning and graph problems.

Apache Giraph : Shortest Path Implementation

In Giraph each superstep ie iteration the framework starts the user defined function on each vertex in parallel. This consists of a series of master/worker execution steps where the master node assigns partitions to workers, coordinates synchronization and collects statuses. A worker sends, receives and assigns messages with other vertices. The rest of the working of Giraph is similar to MapReduce implementation of SSSP where the program terminates if the distance for all nodes decreases no further ie. when messages become inactive.

Pseudocode

```
Set: minDist → 0 for source & INF for others
While(more.messages)
        If(msg.get < min.msg)
                min.Dist = msg.get()
        done
```

```
done
if(min.Dist < current.getvalue)
        setValue = min.Dist
        for edges: allEdges
                sendMsg(edge,mind.Dist + current.DistValue)
        else
                stop
done
```

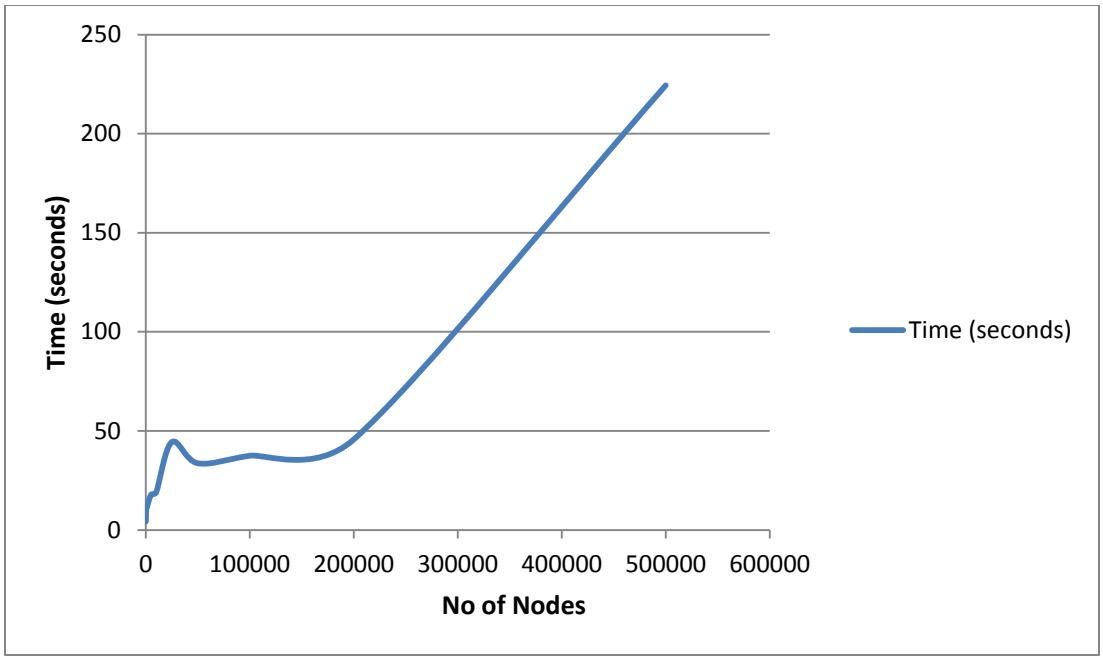
Complexity

Giraph is a framework built on top of Hadoop and is optimized especially for graph analysis.

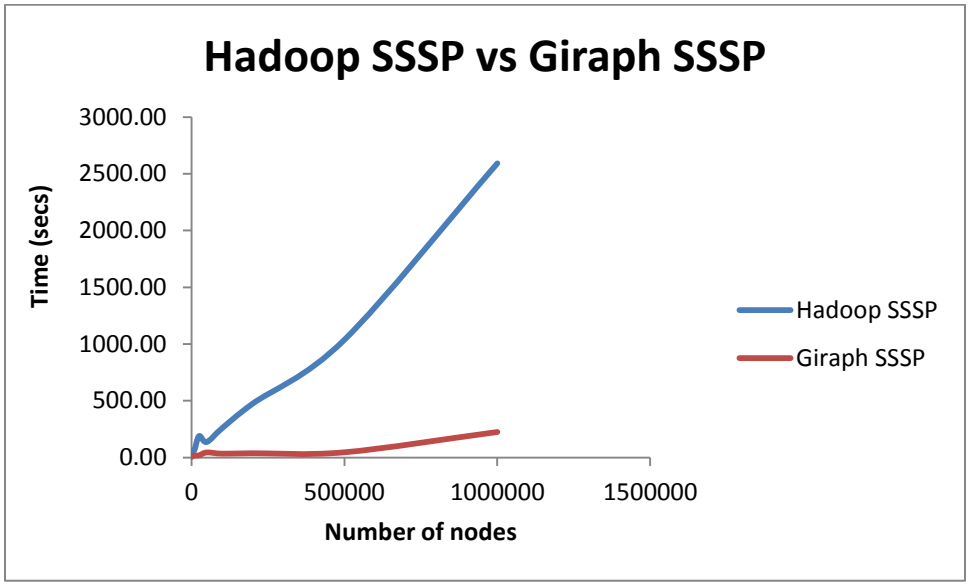
Observation

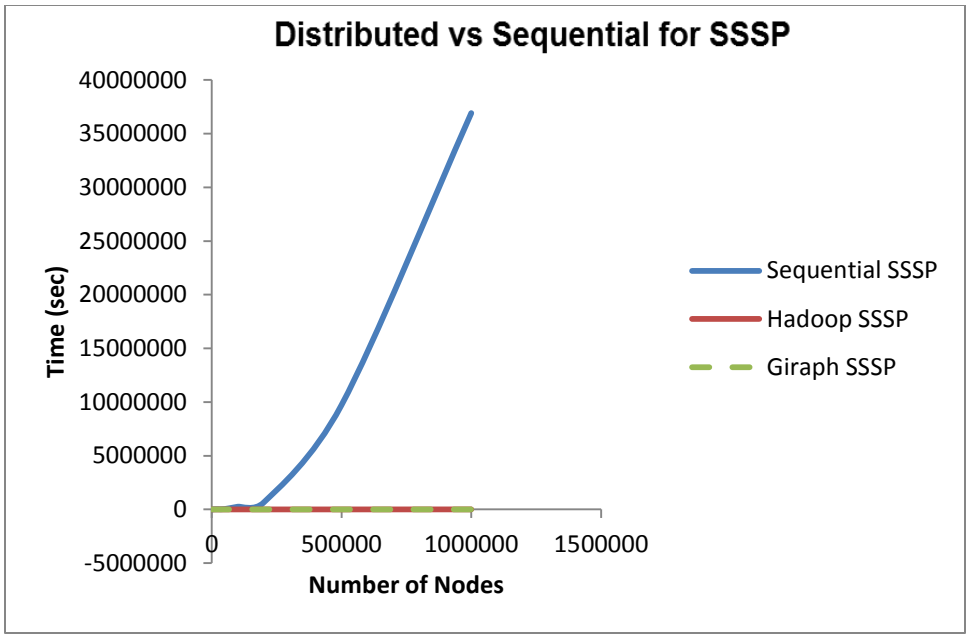
No of Nodes	Time (seconds)
50	4.3
200	7.4
500	11.55
1000	11.49
5000	17.83
10000	19.07
25000	44.37
50000	33.69
100000	37.49
200000	45.71
500000	224.39

3 Giraph: : No of Nodes vs Time



3 : No of Nodes vs Time





Conclusion

1. Speedup is the ratio of the execution time of sequential process to the execution time in parallel. We can clearly see from our results that we have got a speedup of greater than '1' in comparison to the sequential implementation vis-à-vis Giraph and Hadoop.
2. We were unable to test performance of the graph analysis by varying the number of cores due to limitations of hardware. But it is expected that a larger number of cores will help in faster executions.
3. Avg Speedup (Giraph) = 253900/33.69 = 7536.36
4. Avg Speedup (Hadoop) = 253900/136.32 = ~1862
5. That Parallelization using Giraph is a better option for large-scale graphs than Hadoop. The message passing feature and entirely vertex oriented approach makes the computation of large scale graphs faster.

6. However, Giraph update functions are initiated by messages and can only access the data in the message, limiting what can be expressed.
7. Unlike Hadoop, Giraph depends on graph partitioning to minimize communication and ensure work balance

Bottlenecks

One of the bottlenecks we observed and highlighted before was that the buffer size of Hadoop was not big enough to handle values emitted by the mapper and hence it used to spillover to the disk. This meant that additional time was wasted in communication and File I/O operations. Increasing the buffer size should solve this.

REFERENCES

1. Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics* 16: 87–90. MR 0102435
2. BSP: Leslie G. Valiant. A bridging model for parallel computation. In Communications of the ACM, Volume 33 Issue 8, Aug. 1990.
3. CGMGRAPH/CGMLIB:Albert Chan and Frank Dehne. CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. In Intl. J. of High Performance Computing Applications 19(1), 2005, page 81-97.
4. Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. "Single-Source Shortest Paths and All-Pairs Shortest Paths". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. 580–642. ISBN 0-262-03293-7.
5. Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik* 1: 269–271. doi:10.1007/BF01386390
6. Fredman, Michael Lawrence; Tarjan, Robert E. (1984). "Fibonacci heaps and their uses in improved network optimization algorithms". 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346. doi:10.1109/SFCS.1984.715934. ISBN 0-8186-0591-X.
7. Google, Map-reduce: http://www.google.com.mx/patents/US20130024412 HDFS. The Apache Software Foundation. http://hadoop.apache.org/hdfs/, retrieved in Jan.
8. Leyzorek, M.; Gray, R. S.; Johnson, A. A.; Ladew, W. C.; Meaker, S. R., Jr.; Petry, R. M.; Seitz, R. N. (1957). *Investigation of Model Techniques — First Annual Report — 6 June 1956 — 1 July 1957 — A Study of Model Techniques for Communication Systems*. Cleveland, Ohio: Case Institute of Technology.

9. Mapreduce: R. Lammel. Google's mapreduce programming model – revisited. ¨ Science of Computer Programming, 70:1–30, 2008. 5

10. Microsoft Serfer: Rishan Chen, Xuetian Weng, Bingsheng He and Mao Yang. Large Graph Processing in the Cloud. In SIGMOD, 2010

11. Pallottino, S. (1984) Shortest-Path Methods: Complexity, Interrelations and New Propositions. *Networks*, 14, 257-267.

12. Pregel: Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In SIGMOD, 2010

13. PIG. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In SIGMOD '08, pages 1099–1110, 2008. 5

14. Resource Scheduling, Hadoop: Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu and Raghotham Murthy.

15. Single Point Failure, Hadoop:Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design. The Apache Software Foundation. 2007

16. Zhan, F. B., and Noon, C. E. (1996) Shortest Path Algorithms: An Evaluation Using Real Road Networks. *Transportation Science*