Seeking Alpha: Algorithms and Techniques for Enabling Viral Marketing within Mobile Telephony Networks

Stephen Andreassend
Stanford University
sandreas@stanford.edu

Hannah Hironaka
Stanford University
hannahh@stanford.edu

Dan Thompson
Stanford University
dant1@stanford.edu

Abstract

Using the Stanford SNAP[1] tool we define and test a set of algorithms and optimizations that we consider fit for purpose for analyzing the large networks of mobile telephony network operators. It is shown that these can be implemented to identify individuals whom we refer to as *alpha subscribers*, i.e. those influential subscribers within a social network that can be used for viral marketing promotions. This allows marketers to capture the network value of these subscribers through the use of stimuli such as targeted marketing communications. We perform a series of tests to demonstrate that this is computationally possible through the use of modern computing technology, parallelized software algorithms, and optimal data structures. Of particular importance we highlight the performance and precision of the Ego Network Betweenness Centrality algorithm which offers vastly improved serial and parallel performance over the base Exact Betweenness Centrality algorithm.

Introduction

This paper proposes an optimal method of identifying alpha subscribers in large mobile telephony networks for viral marketing purposes. A set of network analysis algorithms was tested on a simulated dataset in order to draw conclusions about their suitability for a real-world usage scenario. For this analysis we used synthetic data resembling that derived from high volume SMS traffic to infer social relationships between mobile subscribers. We found that modern computing technology is fully capable of performing this task when it is combined with parallelized software algorithms and/or partitioned data structures to reduce the run-time duration of these calculations. Without these optimizations to the selected algorithms, we find that the computational cost and operational scheduling complexity are prohibitively high for this analysis to be practical.

Problem Definition

Our research builds upon previous work[1][2][3] and investigates suitable algorithms for identifying alpha subscribers for the purpose of viral marketing within a mobile telephony network.

To adapt the concept articulated by Domingos et al. [2], the alpha subscriber is defined as an on-net subscriber who will enjoy the product, has many close friends who are also on-net

[1] Stanford Network Analysis Project, http://snap.stanford.edu

subscribers, who are easily swayed, will very likely consume the product if marketed to, and has on-net contacts whose on-net contacts also have these properties. Alpha subscribers have above average connectedness compared to other subscribers. Our definition of connectedness is a subscriber who has a high degree count with high clustering or is a member of a highly clustered local clique or community of on-net subscribers.

We address the engineering challenge of analyzing large social networks in a timely and cost-effective manner. More specifically we address the following questions:

- Which algorithms are best suited in terms of their functionality and performance for identifying alpha subscribers on large social networks?
- What are the trade-offs between performance and precision for approximation algorithms such as Ego Network Betweenness Centrality?
- What are the precision trade-offs and performance gains of partitioning a graph using geographic location instead a more computationally expensive method?
- Can SNAP in its current state perform sufficiently fast to run the selected algorithms on large scale networks in a tractable amount of time?
- What improvements can be made to SNAP to reap the benefits of advanced C++ compilers which implement *automatic parallelization*?

Previous Related Work

Leskovec et al., Domingos et al. and Kvernvik et al. [1][2][3] provide useful examples about the type of information that can be extracted from large social networks. The concept of the alpha subscriber[3] is highlighted with their potential to influence other consumers defined as their network value[2]. Viral marketing is proposed as a practice which firms can adopt to generate sales by targeting consumers possessing a high network value[2]. The engineering challenge to render these networks is a recurring theme and practical measures to overcome the high computational expensiveness are offered. These include graph partitioning, parallel processing, and modified algorithms to sample data[3].

The Milgram experiment is cited as an early example of viral marketing where the routing of the message over the social network respected the geography[2]. Geography is identified as a valid means to partition graphs by localizing the social network while minimizing the number of broken or traversed edges extending out of partitions[3]. The use of parallel processing and modified algorithms are advised to efficiently extract, transform, and analyze the Big Data generated by mobile telephony networks to ensure this information is timely and cost-effective.

This partitioning process is very resource intensive but it is noted that it must only be performed once when constructing the base network[3]. An alternative to this process is to use a less resource-intensive means of partitioning the graph or allow algorithms to ignore edges expanding out of its assigned partition at the cost of affecting the quality of the results. The cell site of each subscriber is stored within each CDR and is suggested as a way for pre-partitioning to allow the main partitioning job to also run in parallel. Kvernvik et al. refer to a study that the probability of two subscribers calling each other decreases with geographical distance. This is consistent with the analysis performed by Leskovec et al. on MSN conversation data that found social networks to be a predominantly local phenomenon[1].

These investigations were performed on entry-level or out-dated computers. Given the advances in computing it is unclear whether graph partitioning is actually necessary with large networks or whether a unified graph has become feasible. In the MSN study by Leskovec et al. an 8-way server was used to run SNAP on a one month of MSN logs which resulted in a 4.5TB data set [1]. It required parallel processing to extract the data using four pipelines. Each day there was 150GB of data which required four to five hours to process. The size of the data meant that it was not feasible to perform the strength of ties analysis. The system that was constructed can be characterized as one which supports a batch reporting job rather than real-time reporting.

Mathematics - Algorithms

We examined several different notions of node centrality to measure the influence of a node in the generate network and locate alpha subscribers. We also employed several algorithms to approximate these measures with reduced time and space complexity.

Degree Centrality

The degree of a node is a measure of its influence on neighboring nodes. Node degree is defined as the number of edges adjacent to a node. A related measure is the graph's degree distribution, defined as the probability of a node in graph G having degree *k*. To calculate the degree distribution of a graph, iterate through every node in the network and count its associated edges. This algorithm runs in linear time as a function of the number of nodes in the network.

PageRank

The PageRank algorithm was designed by the Stanford doctoral students who went on to found Google Inc. to run the Google search engine based upon this algorithm[9]. PageRank measures the influence of each individual node within a network and assigns it a score. The sum of all scores of all nodes in the network equals 1. The PageRank (PR) of a node u is formulated as follows:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

Where B_u is the set nodes v pointing to node u, $PR(v)$ is the PageRank of each node v, and L is number of outbound links of each node in B_u. PageRank has a time complexity of [10]:

$$\Omega\left(\frac{m^2}{ln(1/(1-e))}\right)$$

Eigenvector Centrality

Eigenvector centrality also measures the influence of a node, based on the principle that a node of great influence will be linked to by many other nodes of great influence. A node's eigenvector centrality is recursively defined as the average eigenvector centrality of its adjacent nodes, modeling the fact that not all connections are equal; a link to an influential node is more valuable. More concretely, define a eigenvector centrality r_i for node *i* to be:

$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$$

Algorithm

1. Assign each node initial *r* = 1/*n*, where *n* is the number of nodes in G.
2. Repeat until convergence:
    a. For each node *i* in graph G set eigenvector centrality at time *t*+1 equal to the sum of its neighbors' eigenvector centrality at time *t*.
    b. Normalize such that all eigenvector centralities sum to 1.

The time complexity of this algorithm is O(*n* + *m*), where *n* is the number of nodes and *m* is the number edges, for sparse graphs [3].

Betweenness Centrality

Betweenness Centrality estimates the volume of information 'passing through' a node within a network. In telecom network applications this metric can be used to identify viral marketing targets, since subscribers with a high betweenness centrality have greater control over the information passed between separate groups in the network[3]. Concretely, the betweenness centrality C_b of node *v* is defined as

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the total number of shortest paths from node *s* to *t* and $\sigma_{st}(v)$ is the number of paths that pass through *v*.

Algorithm

1. For each node *s* in G:
    a. Perform a modified BFS where at each iteration:
        i. Choose the next node *v* on the frontier of the search.
        ii. For each neighbor *w* of *v*:
            1. If *w* is found for the first time, store its distance from *s*.
            2. If *w*'s distance from *s* is equal to *v*'s distance from *s* + 1, update *w*'s running count of shortest paths passing through it by adding *v*'s count of shortest path passing through it.
            3. Store *v* in a list that keeps track of the dependencies of *w*.
        iii. Iterate through each vertex *w* in non decreasing order of distance from *s*:
            1. For each dependency *v* of *w*.
                a. Increment the stored dependency of *v* by the number of shortest path passing through *v* over the number of shortest paths passing through *w*, multiplied by 1 + the dependency of *w*.

2. If *w* does not equal *s*, increment its stored betweenness centrality by the dependency of *w*.

The above algorithm runs in O(*n***m*) time.

Ego Network Betweenness Centrality

The time and space complexity of betweenness centrality calculations makes it infeasible for large-scale telecom networks. To approximate the betweenness centrality of each node in the graph we employ ego network betweenness centrality. The original algorithm is simplified by only considering nodes which are *k* steps away from the target node during evaluation. Evaluating at one step away, this simplification reduces the time complexity to O(*n*). Although this algorithm has no mathematical guarantees for precision, it is shown to be correlated with the true betweenness centrality of a node in real world networks[3][8]. The source code can be found in Appendix A.

Graph Partitioning

Manual parallelization is another means to improve run time. All of the above algorithms can be run on a subset of the network by segmenting the original graph into several subgraphs. To maximize the performance improvement of manual parallelization we must minimize the number of edge-cuts to prevent the need for inter-process communication. There are many partitioning algorithms that achieve this end, however previous research has shown that the geographic region can proxy for optimal partitions, since links within a social network tend to respect geography[2].

Algorithm

1. Break the graph into five segments to correspond with the five geographic regions.
2. Break the links between regions (“edge-cuts”).

Data Generation

We did not have access to mobile telephony data so we generated simulated networks. Mobile telephony networks have several structural properties that we replicated in the sample data:

- They have a Double Pareto Lognormal degree distribution[1].
- They exhibit strong clustering effects[2].
- They can be partitioned easily along geographic lines. In other words, mobile telephony networks can be divided into sectors which have relatively little connectivity between each other as compared to the level of connectivity within the sectors[3].

We generated the network in three phases. First, we created five graphs with a Double Pareto Lognormal degree distribution. Second, we increased the average clustering coefficient of each graph. Third we merged the five graphs into one network with five sectors and created connections between sectors.

First, using a Matlab toolbox referenced by Ramirez-Cobo, *et al*[4], we created a Double Pareto

Lognormal distribution by creating a set of 100,000 integers ranging from 1 to 3,000 whose histogram could be fitted to a Double Pareto Lognormal distribution.

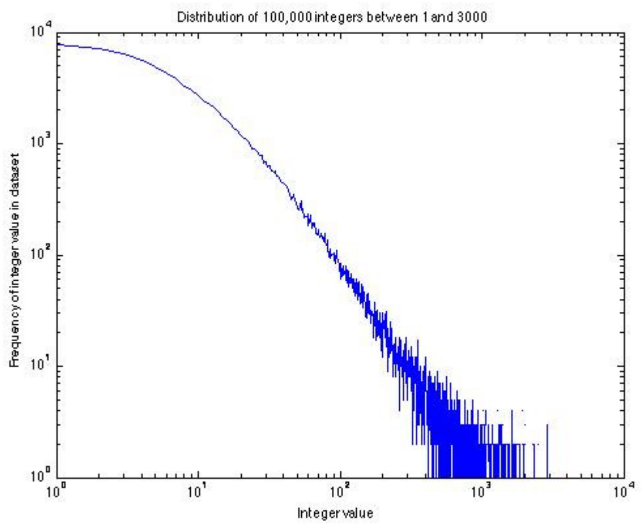

Figure 1. Degree distribution using a double pareto lognormal distribution

Using this dataset, we created a degree distribution where each of the 3,000 unique values represented a degree, and its frequency represented the number of nodes with that degree. Using SNAP's GenConfModel function, we created a graph with this degree distribution, creating a graph with 100,000 nodes and ~4,000,000 edges that had a Double Pareto Lognormal degree distribution. Repeating this process five times gave us five unique graphs with roughly similar distributions.

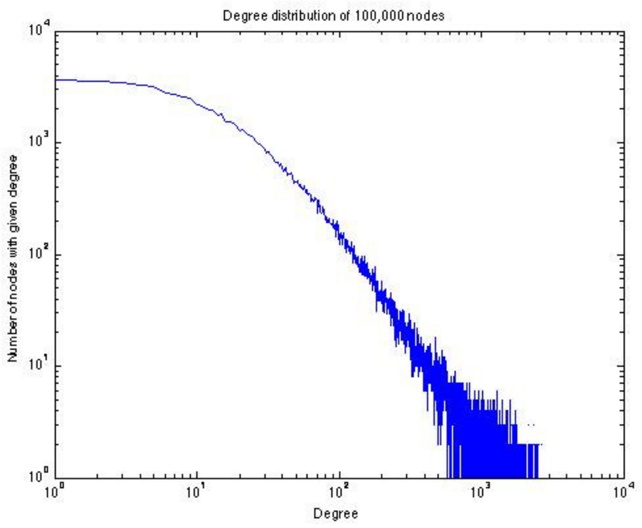

Figure 2. Degree distribution of 100,000 nodes

The generated graphs had an average clustering coefficient of ~0.045, which is lower than the ~0.137 found by Leskovec and Horvitz in massive online messaging networks[1]. We increased the clustering by randomly creating edges between nodes with similar NodeIDs in each graph. Increasing the average clustering coefficient to approach 0.14 required relatively few additional edges, and so each graph still had a degree distribution best fit by a Double Pareto Lognormal distribution.

Then, we merged the five graphs together and randomly created edges, weighting the probability of receiving a new edge toward nodes with higher degrees. This preferential attachment created interconnectivity among the sectors without substantially impacting the degree distribution within each sector.

Ultimately, we were able to successfully create a network that is easily partitioned into sectors, that has a Double Pareto Lognormal degree distribution (and so do each of the sectors), and that has a high average clustering coefficient. The network consists of 20 million nodes clustered into five regions which each consist of four million nodes. This represents the size of a mobile telephony network which may be operating in a state such as California or in a country the size of Spain.

Automatic Parallelization of SNAP

We investigated the use of the Oracle Solaris Studio[6] (OSS) developer tool which is available for Solaris and Linux. This provides a C++ compiler which implements automatic parallelization by finding and optimizing loop. It also offers OpenMP support to parallelize code with explicit instructions in compiler hints. This enables applications to utilize servers with multiple processors, threads, and cores without having to redesign for multithreading. This has the potential to greatly improve performance of SNAP for analyzing very large networks. In addition to parallelization OSS includes a number of optimization features to improve the run-time performance of code. It supports 64-bit compilation to allow applications to access large data sets in memory and on disk.

The following extract from an Oracle white paper[7] provides an overview of parallelization capabilities of OSS:
"*Automatic Parallelization is a feature available on the Oracle Solaris Studio compilers. Through an option, the user activates this loop-based mechanism in the compiler to identify those loops that can be executed in parallel. In case the dependence analysis proves it is safe to do so, the compiler generates the parallel code. All the user needs to do is to set an environment variable to specify the number of threads prior to running the application.*

OpenMP is a de-facto standard to explicitly implement parallelism. Like Automatic Parallelization, it is suitable for multicore and bigger types of shared memory systems. It is a directive based model, augmented with run time functions and environment variables. The Oracle Solaris Studio compilers fully support OpenMP, as well as additional features to assist with the development of applications using this programming model."

To use OSS requires that the GNU C++ pragma hints be modified to comply with the OSS pragma syntax. These minor changes allowed for SNAP to be easily ported to another compiler. OSS has a G++ compatibility mode which ensures that all SNAP code is accepted barring one reference to a Microsoft Visual Studio UI object.

What we found from compiling SNAP with OSS is that automatic parallelization found very few opportunities for parallelization. The SNAP code makes frequent use of “while” loops constructs that have multiple exits. The OSS compiler rejects these. These loops need to be redesigned to allow for automatic parallelization. On the evidence of this experience the ease of achieving compiler-based automatic parallelization appears to be overstated.

However the OpenMP support in OSS was able to be used for the ego network betweenness centrality algorithm. OpenMP allows for greater control with its fork and join model to manage the workflow. This was achieved by parallelizing a high-level loop with compiler hints to specify which parts of the code to run in parallel or serial and which variables are shared or private. As we shall see in the results this approach delivered outstanding results.

Main Results

Test Environment Setup

We ran our tests on resources made available to us in the Oracle Performance Engineering Lab in Santa Clara, CA. The test server consisted of:

- Oracle SUN X3-2B blade server, 2x Intel E5-2658 2.1GHz Xeon CPUs (32 vCPUs), 256GB RAM DDR3-1066MHz, 4x 600GB SAS HDD 10,000 RPM
- Oracle Enterprise Linux 6.2
- Oracle Solaris Studio 12.3
- Stanford SNAP 1.10 library

This hardware and software stack provides a reasonable approximation of the performance levels that SNAP can realize with modern computing technology. It has significantly more compute capacity than that used for some of the related research done in this area which dates back several years. It is acknowledged that there are server models currently available and in future with more CPU sockets and or faster processors which may result in higher performance than what we measured. As a disclaimer our results are not intended as a statement of the product performance of the Intel or Oracle products and are not to be construed as being indicative of the performance of said products.

Baseline Performance of Algorithms

These tests are conducted on the global network of 20 million mobile subscribers with 29,692,610 edges between them. The exact betweenness algorithm was not used for these benchmark tests because it does not scale well enough to handle a large network of 20 million nodes.

Performance Run-Times

Metric	**Run-Time (Seconds)**
Degree Centrality	5s
Ego Network Betweenness Centrality	27,150s
PageRank	201s
Eigenvector Centrality	584s

Table 1. Baseline performance run-times

Data Quality of Ego Network vs Betweenness Centrality

As it is infeasible to compute exact betweenness on the 20,000,000 node synthesized telephony network, we performed the comparison on smaller graphs. We generated two sample networks of 1,000 and 100,0000 random nodes using preferential attachment to compare the performance of the exact betweenness centrality algorithm with the ego network centrality algorithm specified in [3].

We found that ego network centrality proxies well for exact betweenness scores. Table 2 shows a strong correlation between ego and exact betweenness in both generated networks. A node's ego and exact betweenness rank relative to other nodes within the network are also highly correlated. Figure 3 plots exact and ego betweenness side by side demonstrating a visual correlation, although the magnitude of exact betweenness is consistently greater.

The performance of the algorithm is also greatly improved compared with the exact form of the algorithm. This improvement in conjunction with minimal loss to precision leads us to the conclusion that ego network centrality is practical and reasonably accurate to use for the purposes for finding alpha subscribers, while exact betweenness centrality is impractical for any scaled usage.

Correlation	**1,000 Nodes**	**100,000 Nodes**
Betweenness Centrality Score	.987	.981
Betweenness Centrality Rank	.886	.845

Table 2. Exact and Ego Network Betweenness Centrality correlations on random graphs

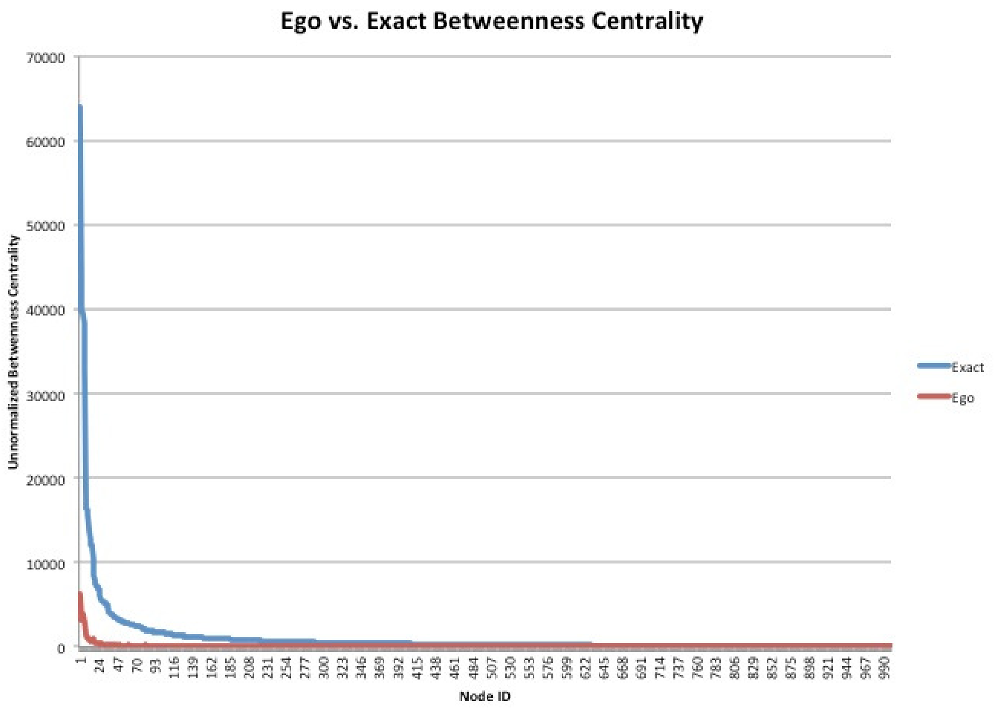

Figure 3. Comparison of Exact vs Ego Network Betweenness Centrality

Automatic Parallelization via Compiler Optimizations

TopicFitter Reference Tests

These tests provide an indication of the scalability that is possible with compiler-based parallelization using SNAP. TopicFitter is a tool provided by Stanford to benchmark SNAP performance. It makes use of OpenMP to force the compiler to parallelize specific loops rather than relying upon the automatic parallelization we attempt to employ in this study. Figure 4 demonstrates the near-linear scalability potential of OpenMP on our test server using TopicFitter. The increase from 16 threads to 32 threads experiences diminishing returns as the 32 CPU core server becomes 100% loaded. This scalability behavior represents the ideal pattern that we want to replicate using SNAP in our set of algorithms. What these results show is that parallelization offers the potential for huge performance gains.

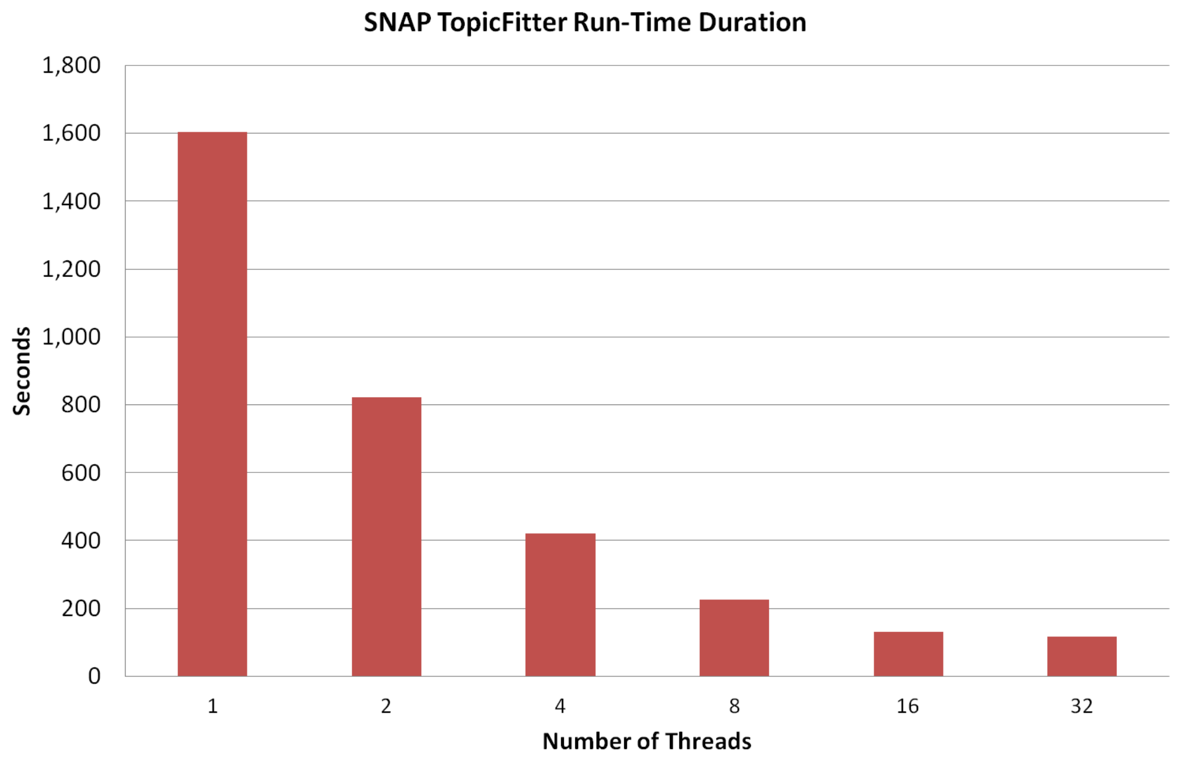

Figure 4. TopicFitter run-times (10 iterations and 1,000,000 objective function terms)

Although it is beyond the scope of this project, TopicFitter was compiled and tested with OSS where it was found to run 3X faster than with the GNU C++ compiler. This gain can be explained by the various OSS compiler performance optimizations rather than by additional parallelization. Hence OSS is the compiler of choice for our tests.

Performance Run-Times

Our testing in this area focused on the ego network betweenness centrality algorithm. This is the most computationally expensive algorithm in our test set and the most likely to benefit from parallelization. We found none of the algorithms were able to be automatically parallelized by OSS at compilation time. This was because the design of the ”while” loop constructs that are frequently used in SNAP were rejected by OSS because they specify multiple exit criteria.

For the ego network betweenness centrality algorithm we modified our serialized version of the algorithm to run with OpenMP. We did this by providing compiler hints to specify which variables use private or shared memory, and which parts of the algorithm can be run in parallel or serial to avoid data races. The source code can be found in Appendix A. During each test the CPU utilization of each thread was 100% and the server CPU utilization corresponded to the number of threads being used. For example with 16 threads the aggregate server CPU utilization was 50% busy. This indicates that the parallelization is very efficiently executed with little inter-process communication overhead or serialization bottlenecks which can cause a degradation. Figure 5 shows the results as this algorithm is scaled from one to 32 threads.

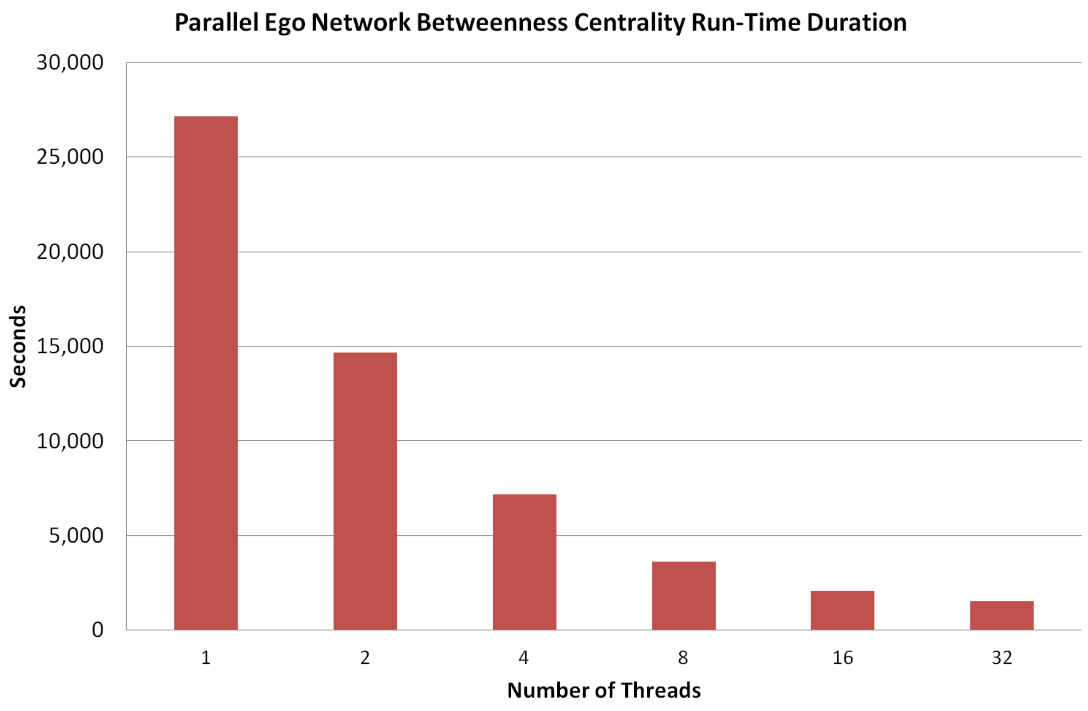

Figure 5. Parallel Ego Network Betweenness Centrality algorithm on 20 million nodes

Data Quality

We were able to confirm that the parallelized ego network betweenness centrality algorithm produces the exact same scores as the serialized version for each node.

Manual Parallelization via Graph Partitioning

These tests involved running the baseline serialized algorithms on five separate graph partitions concurrently on the same server. This allows for parallel processing to reduce the run-time duration where the Linux scheduler runs each task on separate CPU cores. Conceptually it is possible to distribute the workload on separate servers.

Coarse partitioning of the graph was used to split the network into separate regions corresponding to the hypothetical geographical cluster of mobile subscribers. This follows the discussion where mobile subscriber networks tend to follow geography and is therefore a suitable basis for partitioning. Although there are more advanced techniques to partition graphs, we used a coarse method to implement the edge-cuts to split the global graph into the five clustered regions. Each node is assigned to one region only. The drawback of doing this is that it introduces a margin of error into the scores for each node. We therefore provide a comparison of the impact on data quality along with the run-time durations.

Performance Run-Times

Each region is represented by one graph partition with one dedicated worker thread assigned to it.

Metric	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5
Node Count	3,999,337	3,999,709	3,999,911	3,999,909	3,999,759
Edge Count	9,461,488	5,861,555	3,813,080	3,763,082	5,379,472
Degree Centrality	1s	1s	1s	1s	1s
Ego Network Betweenness Centrality	13,270	5,012	1,622s	1,453s	4,375
PageRank	78s	47s	21s	22s	43s
Eigenvector Centrality	41s	31s	31s	33s	31s

Table 3. Thread run-times of manual parallelization via graph partitioning

This is a crude method of partitioning the workload and it will cause imbalances. In practice each graph partition corresponding to a specific geography will not be the same size. For instance a very large city such as Los Angeles will have a larger graph and corresponding higher run-time than for example a small city like Santa Barbara. Hence implementing this design in practice may require a complex workload scheduling regime to distribute the workload to avoid backlogs.

It is noted that the use of manual and compiler-based parallelization are not mutually exclusive. It is possible to partition the graph and process each partition in parallel using compiler-based parallelization. The use of parallelized algorithms running in parallel may mandate the workload to be distributed across multiple servers to avoid processor contention.

Data Quality

In Table 4 and Table 5 the results for the baseline scenario are compared with the scores of the manual parallelization of five graph partitions. These figures confirm the expectation that partitioning with edge-cuts affects the metrics collected. Overall the differences from the error we introduced error are minor and are unlikely to have a material impact on the effectiveness of using this information to identify alpha subscribers. This confirms that manual parallelization produces results which can be used for this purpose.

Metric	Baseline	Combined Graphs
Number of nodes	19,998,625	19,998,625
Number of edges	29,692,610	28,278,677
Average Degree	2.9695	2.8281
Average PageRank	0.5558	0.6057
Average Eigenvector	0.6656	0.8493

Ego Betweenness sum	6.1286e+09	5.5493e+09
Average Ego Betweenness	306.4510	277.4845

Table 4. Comparison of combined manual parallelization results with baseline global network

Metric	Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
Number of nodes	3,999,337	3,999,709	3,999,911	3,999,909	3,999,759
Number of edges	9,461,488	5,861,555	3,813,080	3,763,082	5,379,472
Average degree	2.8281	2.9310	1.9066	1.8816	2.6899
Average PageRank	0.7089	0.6669	0.5076	0.4967	0.6486
Average Eigenvector	0.7492	0.7492	0.9337	0.9369	0.8776
Ego Betweenness sum	2.6266e+09	1.1247e+09	4.3714e+08	3.9299e+08	9.6785e+08
Average Ego Betweenness	277.4845	281.1948	109.2874	98.2503	241.9783

Table 5. Comparison of individual manual parallelization results with baseline global network

Interpretation of Results

These results confirm our expectations. The performance of serialized algorithms with a high time complexity is impractical for analyzing large networks such as mobile telephone networks. In particular the Exact Betweenness Centrality algorithm scales very poorly due to its time complexity. In its place the approximated Ego Network Betweeness Centrality performance has proven to be a suitable alternative to the exact form of the algorithm. It offers significantly better performance and the data it produces gives a sufficient indication of a node’s betweenness score relative to other nodes. By design it cannot give the same absolute betweenness score as the exact algorithm but for this applied usage scenario it is adequate for identifying alpha subscribers in mobile telephony networks.

Our parallelization of the Ego Network Betweenness Centrality algorithm using OpenMP was found to have near-linear scalability as the number of compute threads are increased. It provides the ability to fully utilize the compute capacity of systems with SMP, multi-core, and multi-threaded processors. Thus it is able to dramatically reduce the run-time without sacrificing any accuracy.

The performance of the Degree Centrality, PageRank, and Eigenvector Centrality algorithms proved to have adequate performance in their serialized form for the usage scenario. Degree Centrality has the lowest time complexity but offers limited information about the social network surrounding each node. PageRank and Eigenvector Centrality provide a more robust measure but have a higher time complexity. On very large networks it may be useful to implement parallelization of these algorithms, particularly if these networks are changing their structure frequently. Parallelization will require code changes in SNAP due to the inability of the compiler to automatically parallelize these functions. It is also noted that nodes with a high betweenness centrality score tend to have a high PageRank score. This suggests that PageRank with its lower time complexity is sufficient to measure influence without using a betweenness centrality measure.

Manual parallelization of the algorithms running on graph partitions proved to be a less elegant but effective means of reducing the run-time of the algorithms we tested. This technique approximately halved the total run-time of the baseline serialized algorithms on the global network. Given the available compute capacity of the server it has a less efficient resource utilization than automatic parallelization. In addition the size of graph partitions based upon geography can never be equal therefore balancing the workload distribution across processors or servers is complex. The use of graph partitions with forced edge-cuts introduces a margin of error to all of the algorithms tested. However for this applied usage scenario they deviation from the true values is acceptable.

Conclusions

From this study we are able to conclude that it is computationally feasible to analyze the large social network of mobile telephony operators using modern computing technology and optimized software algorithms. We have identified a set of algorithms which are fit for the purpose of identifying alpha subscribers in terms of their potential influence on other mobile subscribers for viral marketing purposes. The use of these techniques may extend beyond the realms of marketing. For instance a potential unintended usage of the techniques described could be adopted by oppressive regimes to use mobile telephony data to identify the most influential activists to disconnect them from the network to limit the flow of information among the general population.

The use of PageRank is intuitive and serves as a useful measure to identify these subscribers with mobile telephony networks, much like Google uses it for measuring the influence of web pages on the Internet. It can be combined with Degree Centrality, Ego Network Betweenness Centrality and Eigenvector Centrality to support this identification process and track the movement of these metrics over time. Being able to efficiently re-compute these metrics on a regular basis is fully realizable and would allow marketers to track the effectiveness and impact of their viral marketing activities.

Our performance results showed that compiler-based parallelization offers the most efficient use of computational resources without sacrificing accuracy. And it allows for the operating system scheduler to take over the complexity of balancing the workload. Manual parallelization via graph partitioning can provide additional performance benefits on top of this when the computational needs require it. We believe that the trade-off between performance and accuracy introduced by manual parallelization is not a major concern for the applied usage scenario of the information. However with the widespread availability of SMP, multi-core, and multi-threaded servers in cloud computing environments, algorithms should be designed firstly to take advantage of this processor architecture with compiler-based parallelization.

Recommendations

We offer the following recommendations based upon this study:

- Make use of intelligent compilers and advanced features to optimize the serial and parallel execution of code. This may require maintaining separate code branches if multiple compilers are supported by the algorithms.
- Design for parallelization of code to take advantage of the benefits of compiler-based parallelization. Attempting to parallelize the code after the algorithms are designed and implemented is unlikely to result in any gains from automatic parallelization.
- Avoid the complexity of manual parallelization when possible. Reserve its usage for when compiler-based parallelization is not possible or a single server has insufficient computational capacity.
- Consider for inclusion into the SNAP code base the serialized and parallelized Ego Network Betweenness Centrality algorithms to allow users of the tool to dramatically improve the performance of betweenness centrality calculations on large networks.
- Marketing intelligence can be enhanced by increasing the frequency of batch processing jobs to calculate the metrics covered in this paper. This gives the marketer a better understanding of the network behavior that they seek to influence and the evolution of this behavior. The significant parallelization gains we have demonstrated make this more feasible from the perspectives of both operational scheduling and computational cost.

Areas for Further Research

The success of the use of OpenMP could potentially be used to parallelize the Exact Betweenness Centrality algorithm within the SNAP tool to improve its performance. This may only be an academic exercise as we have shown that the ego network serves as a reasonable approximation base of the global network for commercial purposes.

Investigating the impact of removing the multiple exits within the loop constructs frequently used in SNAP may allow for low-level automatic parallelization using a compiler. This may safely improve the performance of many SNAP algorithms on large networks. In addition investigating the performance limits of compiler-based parallelization on the next generation of servers which have many times more CPU cores per server than what we tested on. It may be that the algorithms need to be further redesigned to overcome a greater amount of parallelization overhead.

We have specified a set of algorithms that are fit for the purpose of identifying alpha subscribers for viral marketing purposes within mobile telephony networks. What we have not investigated is how to design and implement promotions to actually capture the viral network value and which types of promotions are most effective. One of the papers we built our research upon[2] offers some abstract examples for using price discounts on products or services to customers. This leads to a larger question about how to use social analytics to *monetize mobile* through advertising. At the time of writing this is a marketing practice area that is yet to be completely mastered by the major social networks and search engines.

Acknowledgements

We would like to acknowledge the Oracle Corporation for making available computer resources for our testing work. And Rok Sosic and Jure Leskovic of the School of Engineering at Stanford University for their guidance for using the SNAP tool.

References

[1] J. Leskovec and E. Horvitz (2007). *Planetary-Scale Views on an Instant-Messaging Network*, Carnegie Mellon University and Microsoft Research.

[2] P. Domingos and M. Richardson (2000). *Mining the Network Value of Customers*, University of Washington.

[3] T. Kvernvik and F. Hildorsson (2009). *Scalability for Social Network Analysis Algorithms*, Ericsson Research and Uppsala University.

[4] M. Seshadri, S. Machiraju, A. Sridharan, J. Bolot, C. Faloutsos, J. Leskovec (2008). *Mobile Call Graphs: Beyond Power-Law and Lognormal Distributions*. http://cs.stanford.edu/people/jure/pubs/dpln-kdd08.pdf.

[5] P. Ramirez-Cobo, R. E. Lillo, S. Wilson, M. P. Wiper (2010). *Bayesian Inference for Double Pareto Lognormal Queues.* The Annals of Applied Statistics. http://projecteuclid.org/DPubS/Repository/1.0/Disseminate?view=body&id=pdfview_1&handle=euclid.aoas/1287409385.

[6] Oracle Corporation (2012). *Oracle Solaris Studio Features.* http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index-jsp-138069.html

[7] R. Van der Pas and B. Mittal, (2010). *Parallel Programming with Oracle Developer Tools*, Oracle Corporation. http://www.oracle.com/technetwork/systems/parallel-programming-oracle-develop-149971.pdf

[8] M. Everett and S. Borgatti, (2005). *Ego Network Betweenness*. Social Networks 27 31-38. http://www.analytictech.com/borgatti/papers/egobet.pdf

[9] L. Page, S. Brin, R. Motwani, and T. Winograd (1999). *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford InfoLab.

[10] B. Bahmani, A. Chowdhury, A. Goel (2010). *Fast Incremental and Personalized PageRank*. http://arxiv.org/abs/1006.2880

Appendix A

Source code for the Serial Ego Network Betweenness Centrality algorithm.

```
void egoBetweenness(PUNGraph &UGraph, TIntFltH &BtwH, int EgoRank) {
    TIntFltH EgoBtwH;
    PUNGraph EgoNet;
    TIntV EgoNodes, HopEgoNodes;

    // Iterate through each node in the network.
#pragma omp parallel for
    for (TUNGraph::TNodeI NI = UGraph->BegNI(); NI < UGraph->EndNI(); NI++) {

        // Create a vector of nodes in ego network.
        EgoNodes.Clr();
        EgoNodes.Add(NI.GetId());
        for (int n = 0; n < NI.GetDeg(); n++) {
              EgoNodes.Add(NI.GetNbrNId(n));
        }

        // Add nodes that are at least two hops away. We shouldn’t bother with this.
        for (int i=2; i<=EgoRank; ++i) {
            TSnap::GetNodesAtHop(UGraph,NI.GetId(), i, HopEgoNodes, false);
            EgoNodes.AddV(HopEgoNodes);
        }

        // Create ego network.
        EgoNet = TSnap::GetSubGraph(UGraph, EgoNodes);

        // Calculate and store ego betweenness of none.
        TSnap::GetBetweennessCentr(EgoNet, EgoBtwH, 1.0);
        BtwH.AddDat(NI.GetId(), EgoBtwH.GetDat(NI.GetId()));
    }
}
```

Source code for the Parallel Ego Network Betweenness Centrality algorithm.

```
#include "Snap.h"
#include <stdio.h>
#include "stdafx.h"
#include <stdlib.h>

#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_num_threads() 1
#endif
```

```
// OpenMP function
int get_num_threads()
{
  int nthreads;

  #pragma omp parallel
  {
    #pragma omp single nowait
    {nthreads = omp_get_num_threads();}
  } // End of parallel region
  return(nthreads);
}

#define CHUNKSIZE 1 /*defines the chunk size as 1 contiguous iteration*/
void egoBetweenness(PUNGraph &UGraph, TIntFltH &BtwH, int EgoRank) {

       // build a vector of nodes to iterate through
       TIntV nodesV;
       for (TUNGraph::TNodeI NI = UGraph->BegNI(); NI < UGraph->EndNI(); NI++) {
         nodesV.Add( NI.GetId() );
       }

    // preallocate
       BtwH.Gen(nodesV.Len());

       // declare vars
       int j;

    printf("Number of threads activated: %d\n", get_num_threads());

       // use this section to tell the compiler which variables are shared and private
       #pragma omp parallel default(none) \
              shared(j,BtwH,EgoRank,UGraph,nodesV)
       {
      #pragma omp for schedule(dynamic, CHUNKSIZE)
         for (j=0; j<nodesV.Len(); j++) {
           // Iterate through each node in the network (run this loop in parallel)

           // Shared variable declarations
           TIntV EgoNodes, HopEgoNodes;
        TIntFltH EgoBtwH;

              TUNGraph::TNodeI NI = UGraph->GetNI( nodesV.GetVal(j) );
              TInt NID = NI.GetId();
        // Create a vector of nodes in ego network.
        EgoNodes.Clr();
        EgoNodes.Add( NID );
        for (int n = 0; n < NI.GetDeg(); n++) {
                EgoNodes.Add(NI.GetNbrNId(n));
        }

        // Add nodes that are at least two hops away. We shouldn't bother with this.
        for (int i=2; i<=EgoRank; ++i) {
          TSnap::GetNodesAtHop(UGraph,NID, i, HopEgoNodes, false);
          EgoNodes.AddV(HopEgoNodes);
        }

        // Create ego network.
        PUNGraph EgoNet = TSnap::GetSubGraph(UGraph, EgoNodes);

        // Calculate and store ego betweenness of none - this is the part that runs in
serial
        TSnap::GetBetweennessCentr(EgoNet, EgoBtwH, 1.0);
              //CallBetweennessCentr(EgoNet, EgoBtwH, 1.0);

              // force this part to be done by only one thread to avoid a data race on
the external variable
        #pragma omp critical
        {
                     BtwH.AddDat( NID, EgoBtwH.GetDat( NI.GetId() ) );
              } // End of critical region

         } // End of parallel region
       }
}
```