Chapter 1

Distributed Random Walks on Dynamically Weighted Graphs

Contributed by Trenton W. Ford

1.1 Introduction

In 2014 the world saw the most massive resurgence of the Ebola virus in history. The epidemic was mainly localized to West Africa but spread to other parts of Africa, and even other countries through a myriad of transportation methods. All told, nearly 30,000 people were infected worldwide, of which 11,300 died. These numbers are terrible, but they could have easily been worse. A large part of planning against the transmission of epidemics is in modeling the dispersal of infection through physical travel networks. In 2014 this modeling helped the Centers for Disease Control (CDC) increase monitoring on select ports to slow or prohibit the possibility of widespread transmission. The goal of this paper is to investigate models of disease transmission that utilize Random Walks as their underlying kernel.

1.2 The Problem as a Graph

We can formalize the above problem as a complex, heterogeneous network with the following structure:

Node Types :

1. Airport
2. Seaport
3. Rail Station

Edge Properties :

1. Direction
2. Transition Probability ($p_1 ... p_k$)
3. Distance
4. Travel Time

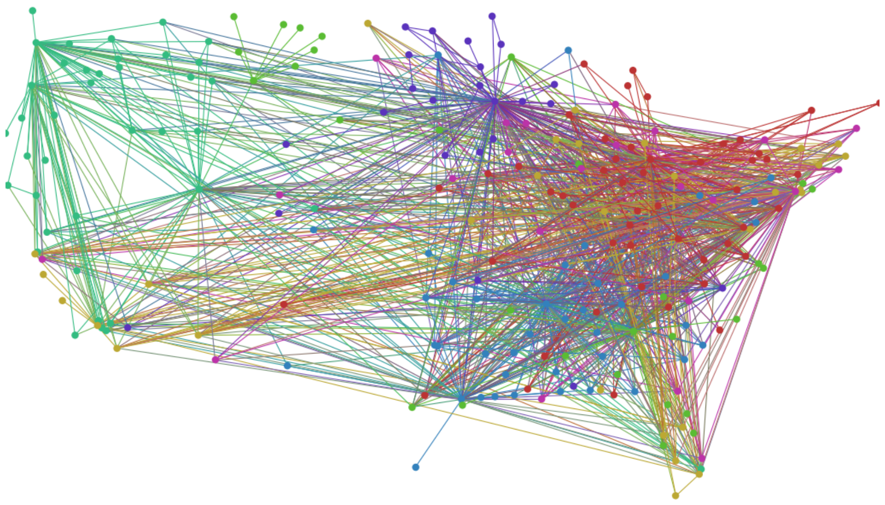

Figure 1.1: The US Domestic Flight Network[6] - Shows the US subset of the SNAP US and Canada Airport data set[4] in Table 1.1. Nodes are airports, edges are the connections between them. Edge are colored relative to the frequency of travel between incident airports.

1.3 Some Realistic Data Sets

1.3.1 Example Data Sets

Table 1.1: Real-world datasets and their sizes.

Dataset	**Approx Vertices**	**Approx Edges**
DOT Railway Data	196K	250K
SNAP Airport Data	456	71K
KNB Shipping Data	3700	15K

Transportation networks are everywhere. The SNAP Labs[1] usually maintain stable data repositories for network data, and here is no exception. The **Airline Travel Reachability Network** contains both US and Canadian airport travel nodes and edges, including metadata such as travel time, geodata, and traveling population and more.

Maritime data is more difficult to find, but there are multiple years worth of shipping data maintained by The Knowledge Network for Bio-complexity (KNB)[1], and increasing amounts of

[1] [4]https://snap.stanford.edu/data/reachability.html

data in this space is becoming open access. Railway datasets are generally maintained by the respective governments for which the railway belongs, and often the included data and accuracy is not consistent. For instance, US railway data can be found at either Data.gov, or the Department Of Transportation data page[2].The size and order of these datasets can be found in table 1.1.

1.3.2 Constructing A Heterogeneous Network

Network heterogeneity refers to the fact that within a network, a vertex may represent more than one "type" of entity. For instance, if one were to construct a heterogeneous network by combining air travel and ship travel paths, nodes would be ports, but there would be a distinction in the "type" of a port that would become important metadata for the heterogeneous network. Part of the difficulty though, is that if we were to construct a heterogeneous network from the above transportation network datasets, we would find that the size and order of the network would increase significantly. For instance, if an airport terminal also contains a train terminal (not irregular), then the airport and train station nodes must either merge or copy the inbound and outbound edges of their counterpart. This sort of node merging or edge duplication activity results in quite irregular networks. This irregularity, coupled with the network's heterogeneity, makes it challenging to build network generators that sufficiently capture the depth of interactions and metadata represented in real-world data.

1.3.3 Generating Representative Datasets

Given the difficulties discussed above related to wrangling the real-world datasets, synthetically generated datasets are used in the scalability testing. Because the goal is to speed up the random walk modeling process, it is sufficient to test against potentially non-representative networks initially. Several sizes of grid networks, Erdős Renyi graphs, and complete graphs were generated for testing with both the sequential and parallel implementations. Use of synthetic datasets allows for better comparison of the results of the scalability testing and allows for more natural experiment replication the inclusion of other real-world adjacent graph generation techniques in future work.

1.4 Random Walk-A Key Graph Kernel

random walk on that graph represents unbiased movement through the graph where only structure is considered. For this reason, random walks are often used to determine if non-structural factors are influencing real observed traversal through a network. In this way, random walks serve as a traversal baseline given no prior knowledge of the graph.

Given a graph $G(V, E)$ where V is a set of vertices and E is a set of edges for which an edge $(u, v) \in E \iff u, v \in V$ if edge $(u, v) \in E$ then vertices u and v are said to be adjacent. A random walk on a graph G is a sequence S of adjacent vertices, where for any vertex in the sequence s_i, vertex s_{i+1} is chosen at by some heuristic, normally uniform random selection, from vertices adjacent to s_i.

Using the above described random walk as a baseline, we can tailor a random walk to model our problem by specifying the heuristics our walk will use to select an adjacent vertex to traverse. To test whether our heuristic appropriately models our specific application, we can attempt to replicate a previously observed real-world occurrence of our application. The closer we can get the model to approximate real-world observations the better. We can iteratively modify the model heuristics to

[2]osav-usdot.opendata.arcgis.com/

better fit observations using several methods (gradient descent, grid search, etc.) - using this simple process, a random walk model can be built to fit our application. Once a good model is trained the goal would be to use the model to make predictions about unobserved occurrences. In our specific application, we would want to use the model to make predictions about epidemic transmission for future disease outbreaks. In the disease transmission scenario described above, probabilistic random walks are used to represent individuals (infective or susceptible) moving through travel networks.

Unfortunately, in the real world, infective individuals do not just traverse through a network. As infective individuals move through a network, there are contagion characteristics to consider. Disease-specific factors that a high-fidelity model would need to consider are:

- Carrier Proportion
- Incubation Time
- Infective Duration
- Infectivity
- Transmissibility
- Mobility Index

Many of these factors require statistical models for best approximations[7]. For our sequential and enhanced implementation, we will keep our model as simple as reasonable to test benchmark our solutions as comparable as possible. More clearly stated, the statistical support that would enable consideration of these disease-specific factors are computationally expensive and rely heavily on pseudo-random sampling. Not only would this type of sampling increase the computation time of our implementations, but it would introduce a great deal of uncertainty into our results. The factors this implementation will consider are discussed in Section 1.7.

1.4.1 Computational Bottleneck

Consider the given application. If one wanted to use a random walk based algorithm to model epidemic transmission within a real-world travel network, then none of the individual data sets referenced above would be sufficient. People use many modes of transportation and often mix modes during travel. Merge any two travel networks, and it quickly becomes apparent that the number of vertices may be linear, but the number of edges and the overall complexity of the resulting network increases exponentially. One of the most powerful motivations for using random walk based algorithms is their speed, but the network sizes and complexity mean that a random walk sequence may need tens of billions of steps model a scenario, and that process repeats until the fit is good. Refining a model on a network of this scale is time prohibitive. Reducing the runtime is the aim of this enhancement.

In the general case of random walks the computation time can be reduced using an "embarrassingly parallel" variant of the algorithm that runs separate walks on different copies of the same graph using many different compute nodes simultaneously. When the individual compute nodes reach a stopping point, their results are gathered and merged into a final product. Unfortunately, in the epidemic transmission specific random walk an embarrassingly parallel implementation does not exist. Because our random walker carriers information that could change the underlying network means that the graph is not guaranteed to be consistent across the individual compute nodes.

Considering these facts the obvious option is to do away with different compute nodes operating on copies of the graph and have them work on the same copy simultaneously. This method introduces the problems of compute node memory consistency and coherence. Fortunately, there is another strategy.

1.4.2 Proposed Solution

This research proposes to parallelize a random walk based epidemic transmission model across many compute nodes by partitioning the original network and distributing said partitions to the memory associated with individual compute nodes. Each compute node will be responsible for operations relating to the vertices within the partition of the graph allocated to it, and the outgoing edges associated with those vertices. This method still introduces the need for compute nodes to communicate when changes to the network are made that could effect graph partitions owned by other compute nodes. A further enhancement relates to the partitioning strategy for the original network. Because communication between compute nodes is expected, and communication and synchronization create a bottleneck in parallel computations, it is imperative to minimize the need to compute nodes to communicate.

Recognizing that communication and synchronization happen between computer nodes when changes propagate through an edge for which the source node and the target node are owned by different compute nodes, an optimal solution would be to partition such that all adjacent vertices are in the same partitions and thus owned by the same compute nodes. Unfortunately, given that we must choose a number of partitions equal to the number of compute nodes, and the size of the partitions must fit in the memory of a compute node, and the fact that ideally, we would like a workload distribution that is uniform across all compute nodes - the process of finding a perfectly optimal partitioning scheme itself becomes computationally prohibitive. Although at this time the implementation does not include this feature, a reasonable approximation of the optimal partitioning schema may be found through a k-way Spectral Clustering method. The current implementation will uniformly distribute the network vertices arbitrarily to compute nodes. The effects of the unmitigated communication overhead are seen in the results.

$$\begin{tikzcd}
\boxed{1} \arrow[r, leftrightarrow, dashed] \arrow[d, leftrightarrow, dashed] & \boxed{4} \arrow[d, leftrightarrow, dashed] \\
\boxed{2} \arrow[r, leftrightarrow, dashed] & \boxed{3}
\end{tikzcd}$$

Figure 1.2: Kekler's 2x2 Grid Graph For Epidemic Simulation [3]

1.5 Prior and Related Work

Most work done in the epidemic modeling application space focuses on improving the accuracy of models, and not directly on the question of parallelization. This actual epidemic random walk model implementation takes design cues from Douglas Kelker's work on a grid graph based epidemic transmission modeling algorithm, as well as Draief and Ganesh's rehashing of the algorithm using non-grid structured networks [2] [3].

As it relates to parallelization, this method takes cues from many parallel graph algorithm implementations. The parallelization strategy is so well known, the chosen implementation library (PBGL) provides much of the partitioning and communication faculties that the implementation requires natively. The strategy for the granular implementation derives significantly from Tim Weninger's vertex-centric distributed computation approach which lays out a framework from parallel algorithm design [5].

1.6 Experimental Configuration

The experimental configurations were chosen to minimize the differences between the sequential and parallel implementations' code, compilers, and hardware.

1.6.0.1 Hardware

CPU Specifications:

Architecture: x86_64	CPU MHz: 2400.028
CPU(s): 60	L1d cache: 64K
On-line CPU(s) list: 0-59	L1i cache: 64K
Thread(s) per core: 1	L2 cache: 512K

Storage Specifications: Available Onboard RAM 127GB

1.6.0.2 Software

For reduced computational overhead, the implementation was written with C++11. Graphing packages named The Boost Graph Library (BGL), and Parallel Boost Graph Library (PBGL) were used to maintain graph structures and provide stubs for running algorithms on the stored graphs. PBGL contains added functionality that allows for distributed graph storage and algorithms. A useful result of using similarly written packages to implement both the sequential and parallel algorithms is for the simplicity of utilizing the sequential code from BGL to PBGL.

PBGL facilitates communication between the distributed compute nodes, and uses MPI for this purpose. OpenMPI was chosen as the compiler for both the sequential and parallel implementations.

- Boost Graph Library 1.68
- Parallel Boost Graph Library 1.68
- openMPI 3.3.1

1.7 A Sequential Algorithm

One of the major advantages of the random walk process is that it is easy to understand in the uniformly weighted, non dynamic case. The algorithm is relatively simple. The input is a graph $G(V, E)$ where the edges $e \in E$ have associated weights, or transition probabilities w_i. If we consider a simple case where we select one starting vertex say u, and wish to perform a random step we would consider all adjacent vertices, say $x_1 \ldots x_k$, connected to u over edges $e_1 \ldots e_k$, with associated weights $w_1 \ldots x_k$. Before taking a step, the weights of the adjacent vertices must be normalized as follows:

$$p_i = \frac{w_i}{\sum_1^k w_i}$$

Where each of the p_i will represent the probability of traversing across a given edge e_i. Given the normalization process, we are guaranteed that the probability of all adjacent edges will sum to 1, so we can use a standard uniform random number generator to sample from the edges. At that point we may successfully take a step. We repeat that process until we reach a stopping criteria which, in the case of our epidemic transmission model is reaching a homogeneous status for.

When we apply this simple random walk model to our epidemic transmission model, we start by re-imagining our random walk process. Conceptualize the nodes in our graph as arrival and departure locations in a transportation network. The edges indicate arrival and departure locations for which there exists a direct travel method between. The introduction of a small set of parameters, as discussed in Section 1 are needed to capture the disease-specific factors. We introduce variables mirroring those used in Kelker's work[3] published in 1973 that used a simple grid model and a minimal set of transition parameters to model the spread of measles and ferret distemperment. Put just, the model used a 2x2 grid of vertices within which four infective individuals were located at time zero. At each successive time step each has an equal opportunity of staying in place or moving to one of the adjacent nodes. Let this probability be λ. If an individual moved to a vertex that contains uninfected or susceptible individuals, there is a probability p that each may become infected. The paper also introduced a variable μ to represent the probability that at each time-step a person recovers or changes from an infective state to a susceptible state.

At each time-step, a vertex maintains a current state and future state. During a time-step, if a vertex transmits a shift in population, either infective or susceptible, to another vertex it must send the new population to the target vertex's future state. During the same time-step, the target vertex will not be aware of future changes and will process the infection, recovery, and movement stages using only current state information. Once all vertices have activated during the time-step, the time-step ends by updating the current state of all vertices with future state population changes.

As discussed in Section 1, this implementation uses considers a subset of the actual parameter space that would produce an optimal model for computational purposes, but the parts of the implementation that can benefit from being parallelized should not change as the complexity of parameter sampling increases.

Unlike Kelker's algorithm, our implementation is not limited to a 2x2 grid. We also improve accuracy by λ, μ, and p be sampled from distributions instead of being variables set based on expert opinion or solely on prior data. For complexity analysis, the parameters will be sampled from a uniform distribution.

1.7.1 Complexity

An analysis of the operations necessary for the completion of one time-step of the algorithm has complexity $O(|V| + |E|)$. In each time-step, every vertex must activate ($O(|V|)$) and each vertex may move a portion of its population to all adjacent vertices ($O(|E|)$). The difficulty in estimating the complexity of the remainder of the algorithm is found in the fact that the stopping criteria for the while loop are based on reaching a homogeneous infection state. The determinants of reaching such a state include variables being sampled at random from uniform distributions.

The necessary analysis of the expected number of time-steps until infection state convergence is currently outside of the scope of the research.

1.8 A Reference Sequential Implementation

For the sequential representation, our implementation will roughly copy that put forth by Kelker in 1973[3]. Modifications were to increase the accuracy of the model by allowing the three variables

Algorithm 1 Random Walk With A Purpose

```
Require: Graph $G(V, E)$
Ensure: Boolean Infective State
     Let $p, \lambda, \mu$ be $RVs$
 1: v0 = random(vertex, G);
 2: $\forall v \in V$ let $loc[v] = (init\_pop, init\_ipop, init\_spop)$:
 3: $\forall v \in V$ let $floc[v] = (0, 0, 0)$:
 4: while (True) do
 5:     for $v \in V$ do
 6:         infect(loc[v]);
 7:         recover(floc[v]);
 8:         move(floc[v]);
 9:     end for
10:     for $v \in V$ do
11:         update(loc[v]+=floc[v])                ▷ If no update for any v, break;
12:     end for
13: end while
14: total_pop = sum(loc[:](0))
15: if (sum(loc[:](1)) == total_pop) then
16:     return True
17: end if
18: return False
```

(λ, μ, and p) to be samples from non-uniform distributions. We also allow graphs that are not grids. Kelker's simulation model was the groundwork for more recent epidemic transmission simulations, but the complexity that they add to the random walk process makes it difficult to do meaningful complexity analysis of the algorithms.

We also adopt Kelker's stopping criteria, in that we run the model until all of the individuals reach a homogeneous state. That is to say that either all people are infective, or all people are susceptible. Reasonable thresholds can be used instead of absolute convergence, but for the parameters that we set, absolute convergence should always be possible.

In both sequential and parallel implementations, the same algorithm will be employed. The BGL and PBGL DFS visitor will be used to activate each vertex during a time-step and during each timestep. The visitor and the methods it utilizes can be found in Listings 1.1-1.5.

Listing 1.1: DFS Visitor

```
class DFS :
public boost::default_dfs_visitor
{
public:
    // caled when each vertex
    // is first reached in
    // DFS
  void discover_vertex
  (Vertex v,
  const Graph& g)
  const
  {
    auto vp = G.properties(v);
```

```
      vp.recover();
      vp.infect();
      move(v, g, vp);
      G.properties(v) = vp;
    }
    // called when all vertices
    // have been seen using
    // dfs
    void vertex_covered
    (Vertex v,
    const Graph& g)
    const
    {
      auto vp = G.properties(v);
      vp.update();
      G.properties(v) = vp;
    }
  }
```

Listing 1.2: Recover Method

```
    void recover()
  {
    std::random_device rd;
      std::mt19937 gen(rd());
    std::uniform_real_distribution<>
       dis(0.0, 1.0);
    for (uint i = 0; i < cs.ipop; i++)
    {
      // Random Recovery
      if (dis(gen) < mu)
      {
        cs.spop += 1;
        cs.ipop -= 1;
      }
    }
    cs.pop = cs.spop + cs.ipop;
  }
```

Listing 1.3: Update Method

```
  void update()
  {
    cs.pop += fs.pop;
    cs.ipop += fs.ipop;
    cs.spop += fs.spop;
    set_future_values(0,0,0);
  }
```

Listing 1.4: Recover Method

```
void recover()
{
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_real_distribution<>
     dis(0.0, 1.0);
  for (uint i = 0; i < cs.ipop; i++)
  {
    // Random Recovery
    if (dis(gen) < mu)
    {
      cs.spop += 1;
      cs.ipop -= 1;
    }
  }
  cs.pop = cs.spop + cs.ipop;
}
```

Listing 1.5: Infect Method

```

void infect()
  {
    std::random_device rd;
      std::mt19937 gen(rd());
    std::uniform_real_distribution<>
      dis(0.0, 1.0);

    for (uint i = 0; i < cs.ipop; i++)
    {
      for (uint j = 0; j < cs.spop; j++)
      {
        if (dis(gen) < p)
        {

          cs.spop -= 1;
          cs.ipop += 1;
        }
      }
    }
    cs.pop = cs.spop + current_step.ipop;
  }
```

1.9 Sequential Scaling Results

Do to time constraints, the only graphs that were tested thoroughly were complete graphs. Complete graphs should provide a worst-case scenario in the epidemic transmission application space. That is to say that every location is connected to all locations directly via a transportation mode. The results of the sequential scaling are quite clear using the complete graphs.

Figure 1.3 shows the results of 5 iterations of complete graphs of size 2^4 to 2^{14} vertices. Table 1.2 shows the relevant subset of the raw data statistics. It is clear from that data and from curve fitting equation that the runtime scales quadratic with number of vertices. The variance of the times increases similarly, but this is likely due to the non-deterministic part (parameter sampling of λ, μ, and p) and is expected. To better understand the change in variance, ANOVA should be performed.

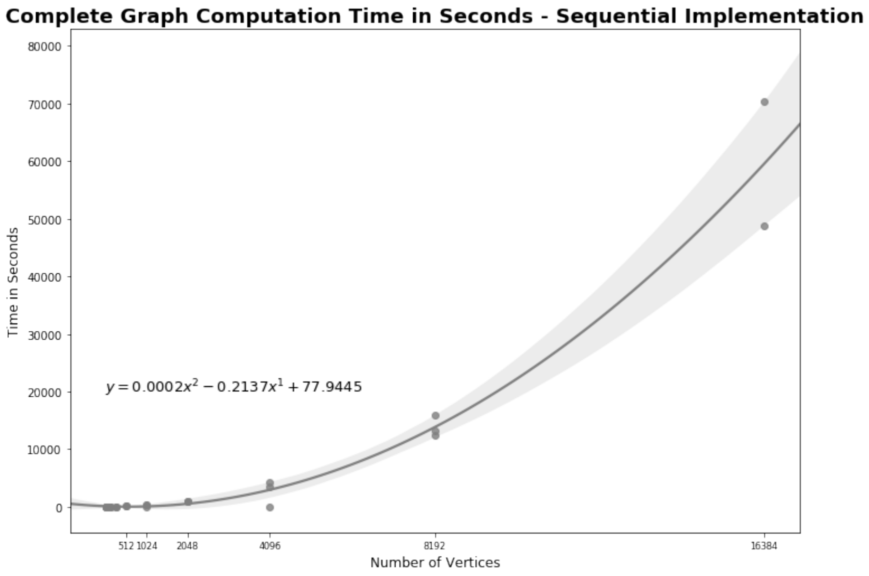

Figure 1.3: Sequential Implementation Scaling Results - Complete Graph

Table 1.2: Sequential Implementation Scaling Results

Number of Vertices	**Number of Edges**	**Mean Runtime (s)**	**Max Runtime (s)**
16	120	1.36217	1.884534
64	2016	6.859021	8.149915
256	32640	16.635931	48.824964
1024	523776	215.699807	322.510716
2048	2096128	952.982129	970.69758
4096	8386560	2616.989533	4313.243031
8192	33550336	13841.35421	15934.91333
16384	134209536	59523.70404	70260.42398

1.10 An Enhanced Algorithm

Part of the initial design consideration of this implementation was to limit rewriting the sequential algorithm. Because the libraries used for both the sequential and parallel implementations are closely related, rewriting the body of the sequential algorithm is not necessary. The changes that need to be made to parallelize the algorithm happen in the implementation specifications.

1.11 A Reference Enhanced Implementation

The parallel implementation uses The Parallel Boost Graph Libraries to handle storage, distribution, and communications. Given PBGL's kinship to the library that the sequential algorithm was written, only changes to the constructs surrounding the algorithm are necessary. The only change of note that is required is shown in the following code snippets:

Listing 1.6: Sequential Graph Type

```
    typedef adjacency_list<
        vecS,
        vecS,
        bidirectionalS,
        property<vertex_properties_t,
        VERTEXPROPERTIES>,
        property<edge_properties_t,
        EDGEPROPERTIES>
    > Graph;
```

Listing 1.7: Parallel Graph Type

```
    typedef adjacency_list<
        vecS,
        distributedS<mpi_process_group,
        vecS>,
        bidirectionalS,
        property<vertex_properties_t,
        VERTEXPROPERTIES>,
        property<edge_properties_t,
        EDGEPROPERTIES>
    > Graph;
```

For all other implementation details, the code snippets in the sequential scaling section hold for the parallel solution as well.

1.12 Enhanced Scaling Results

The parallel implementation did not perform as expected. The implementation had poorer runtimes than the sequential implementation, but the order of runtime scaling was still quadratic. Experiment run times did not allow for strong scaling analysis, and all results were run using the *mpirun* command using four of the available CPUs as computational units.

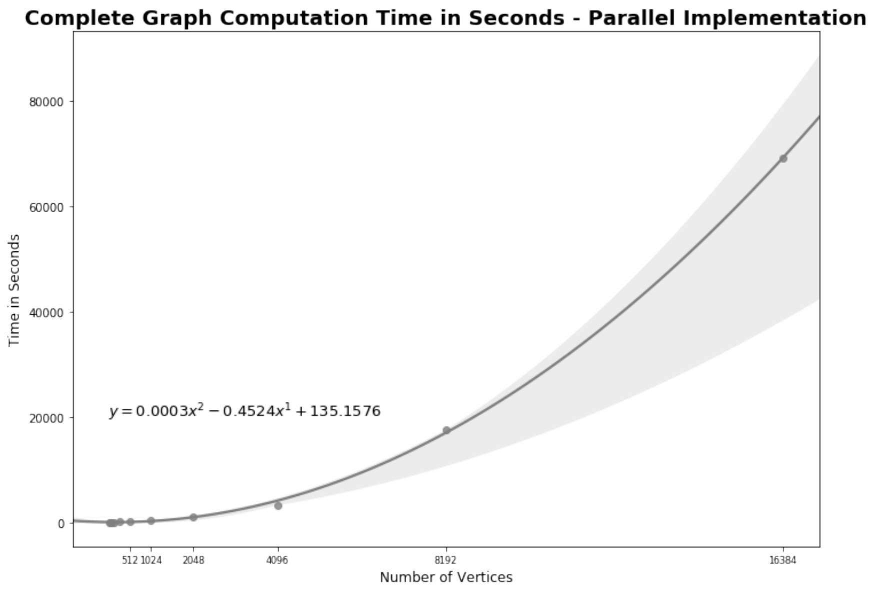

Figure 1.4: Sequential Implementation Scaling Results - Complete Graph

1.13 Conclusion

The parallel implementation was slower in this case. A likely culprit is the use of the depth-first search stub built into PBGL and BGL. If the DFS driver in PBGL acts as the sequential version does, then a parallelized DFS will still have a complexity of $O(|V|+|E|)$. Considering that, along with the added overhead of distributing the graph structure and maintaining synchronization

Table 1.3: Parallel Implementation Scaling Results

Number of Vertices	Number of Edges	Mean Runtime (s)	Max Runtime (s)
16	120	1.07757	1.731991
64	2016	7.625351	12.473645
256	32640	31.407889	57.236048
1,024	523776	289.612513	432.16106
2,048	2096128	1,089.112612	1,347.682374
4,096	8386560	3,166.591334	4,955.366348
8,192	33550336	17,548.35421	21,586.49425
16,384	134209536	69,144.12156	75,481.65595

explains the consistently poorer performance of the parallel implementation. A modification to this implementation that would overcome that potentiality is to convert the implementation to use a "for all vertices" implementation.

In Section 1.9 the results' variance was discussed briefly. Even once the modifications are made to the implementation, analyzing the parameter to runtime variance would be useful for understanding the non-deterministic component of the transmission algorithm.

Once the current implementation produces stable results, increasing the model complexity through the addition of more of the parameters discussed in Section 1 would be useful in determining if any of the factors introduce dependencies that would no longer allow this parallelization strategy.

1.14 Response to Reviews

1. **The key metrics was not clear**.
   I addressed this by rewriting the introduction and kernel sections to better underscore the fact that the key metric in this case is runtime.

2. **No related work**.
   I've added a related work section.

3. **Unclear whether the complexity is related to the model or random walk algorithm**.
   I've attempted to clarify that the part of the time complexity that can be controlled is related to the distribution of the algorithm and the communication that must take place during computation. The parameter sampling and complexity of the work that must be done to determine infection and movement will depend on the model.

The feedback regarding the paper was insightful. The responses helped tease out the most meaningful components of the research, and actually influenced the direction of the implementation.

Bibliography

[1] John Potapenko Kenneth Casey Kellee Koenig et al. Benjamin Halpern, Melanie Frazier. Knowledge network for biocomplexity, 2015.

[2] Moez Draief and Ayalvadi Ganesh. A random walk model for infection on graphs: Spread of epidemics & rumours with mobile agents. In *Discrete Event Dynamic Systems: Theory and Applications*, 2011.

[3] Douglas Kelker. A Random Walk Epidemic Simulation. *Journal of the American Statistical Association*, 68(344):821–823, 1973.

[4] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.

[5] Robert Ryan McCune, Tim Weninger, and Gregory R. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for distributed graph processing. *CoRR*, abs/1507.04405, 2015.

[6] Elijah Meeks. Visualization of network distance, Nov 2011.

[7] Henrik Salje, Derek A.T. Cummings, and Justin Lessler. Estimating infectious disease transmission distances using the overall distribution of cases. *Epidemics*, 2016.