MDPI

Article

Interactive Graph Layout of a Million Nodes

Peng Mi [1,*], Maoyuan Sun [2], Moeti Masiane [1], Yong Cao [3] and Chris North [1]

[1] Department of Computer Science, Virginia Tech, Blacksburg, VA 24060, USA; moeti@cs.vt.edu (M.M.); north@cs.vt.edu (C.N.)
[2] Department of Computer and Information Science, University of Massachusetts Dartmouth, Dartmouth, MA 02747, USA; smaoyuan@umassd.edu
[3] The Boeing Company, 3455 Airframe Dr, North Charleston, SC 29418, USA; claio0128@gmail.com
* Correspondence: mipeng@cs.vt.edu; Tel.: 540-838-5474

Academic Editor: Dr. Olga Kurasova
Version December 12, 2016 submitted to Informatics; Typeset by LaTeX using class file mdpi.cls

Abstract: Sensemaking of large graphs, specifically those with millions of nodes, is a crucial task in many fields. Automatic graph layout algorithms, augmented with real-time human-in-the-loop interaction, can potentially support sensemaking of large graphs. However, designing interactive algorithms to achieve this is challenging. In this paper, we tackle the scalability problem of interactive layout of large graphs, and contribute a new GPU-based force-directed layout algorithm that exploits graph topology. This algorithm can interactively layout graphs with millions of nodes, and support real-time interaction to explore alternative graph layouts. Users can directly manipulate the layout of vertices in a force-directed fashion. The complexity of traditional repulsive force computation is reduced by approximating calculations based on the hierarchical structure of multi-level clustered graphs. We evaluate the algorithm performance, and demonstrate human-in-the-loop layout in two sensemaking case studies. Moreover, we summarize lessons learned for designing interactive large graph layout algorithms on the GPU.

Keywords: graph layout, big data, visual graph exploration, human-in-the-loop analytics, GPU

1. Introduction

Graphs are commonly used to depict complex relations among objects. Graph drawing offers solutions to geometrically represent graphs, with the intention of improving their readability. This supports applications and analysis in various domains, such as social network analysis (e.g., [1] and [2]), cyber security (e.g., [5]), and intelligence analysis (e.g., [4]).

However, with the increasing size and complexity of graph data, the performance of drawing large graphs, especially those with millions of nodes, is still a significant challenge. Furthermore, user interaction with the graph layout becomes increasingly more critical to support sensemaking tasks (e.g., [2] and [6]) in the presence of large graph data. Many graph drawing algorithms have been proposed in the last few decades. However, many of them ([7] and [8]) focus on relatively small graphs (e.g., graphs with hundreds of nodes) or certain types of graphs (e.g., trees or planar graphs). Existing large graph layout algorithms emphasize static graph layout results either based on their structures [9] or semantic meanings [10]. They focus on generating static and visually pleasing results, but their layouts are constrained by their predefined aims. Thus, these algorithms are limited in their support for interactive sensemaking tasks with large graphs.

In this paper, we argue that *enabling direct manipulation and real-time interaction are essential to support knowledge discovery and sensemaking activities with large graphs*. Static graph layouts may not meet the need to reveal multiple perspectives of a graph. If there is not a clear definition of the best graph layout, users will need to explore the graph and interact with the layout to find

meaningful information. In some cases, users interactive feedback can alter graph layout properties, such as modifying edge weights [3] or expanding subgraphs [2], leading to the need for incremental layouts that dynamically respond to users' sensemaking activities. We highlight human interactions on a large graph layout, which support user exploration of unexpected knowledge from large graphs.

Thus, we are not overly concerned about the initial layout time, which could be preprocessed. Rather, we are focusing on the interactive update process that occurs when users are interacting with the large graph after it has already been initially displayed. We therefore stress that the performance of the interactive graph layout computation should be optimized for single iteration speed, rather than total batch completion speed, for the purpose of handling real-time feedback and human intervention that can interrupt or interplay with layout computation.

To this end, we focus on optimizing a force-direct algorithm based on the spring-electrical model [11]. This model depicts the graph drawing problem as a physical system, where the spring-like attractive forces are generated by each edge, and each charged node repels others via electrical force. Since it iteratively calculates the position of each node, this algorithm has the inherent ability to dynamically incorporate user interactions into the ongoing automatic layout process. During the iterations of the algorithm, users can flexibly select, drag, and pin nodes to modify the layout results. However, this algorithm does not scale well with respect to the size of graphs, and it suffers from poor performance even on moderate-sized graphs.

Similar to the N-body problem, the bottleneck of this algorithm lies in the repulsive force calculation [12]. Many solutions have been proposed to replace computing the exact repulsive forces between all pairs of nodes with some approximated calculations. A common heuristic is that if two nodes are far away from each other, the repulsive force between them can be ignored or approximated. Thus, a spatial indexing structure needs to be built and updated to describe the spatial relationships among nodes in each iteration. For example, Godiyal et al. [13] proposed a method combining CPU and GPU to construct a balance k-d tree, while Burtscher et al. [14] gave a GPU based octree implementation. However, building and traversing hierarchical structures can be computationally expensive for large graphs.

In this paper, we contribute a new algorithmic solution to speed up repulsive force calculation, which can enable *human-in-the-loop* layout updates of large graphs. We calculate approximate repulsive forces based on the graph structure, rather than on the spatial distribution of nodes. The benefit of our algorithm, compared to previous accelerated methods, is that we avoid building, traversing, and updating a spatial indexing structure. To further improve performance, we design a novel parallel force-directed graph layout algorithm that utilizes the massive computational power of GPUs to achieve real-time frame-rates that support human interaction. Moreover, our algorithm can be integrated into the multi-level graph layout paradigm, which relieves the local minimum problem, to generate visually pleasing results.

As a proof of concept, we demonstrate our algorithm with a few simple types of human interaction, such as dragging nodes, without loss of generality. Our method can be utilized in visual analytics systems to support many different kinds of dynamic interactions with large graph layouts.

2. Related Work

In this section, we review large graph layout algorithms, and focus on force-directed algorithms and GPU based methods.

2.1. Large Graph Layout

Hachul et al. [9] survey existing solutions to large graph layout considering two aspects: aesthetics and performance. They suggest that certain force-directed layout algorithms can generate pleasing layouts for most tested graphs. Several recent papers, including [13], [15] and [16], present advanced large graph layout techniques based on force-directed algorithms (e.g.,[11], [17], [18] and [19]).

Another strategy to layout large graphs is based on linear algebra, rather than physical simulation, such as [20] and [21]. However, these algorithms are designed to work on a few specific types of graphs. In addition, linear algebra based methods cannot be easily integrated with human interactions.

Beyond these, a variety of other large graph drawing algorithms have been investigated. For example, Muelder et al. [23] present a treemap based approach; Wong et al. [24] use a space-filling fractal curve to layout graph nodes; and Khoury et al. [25] propose a layout algorithm that simplifies matrix computations based on linear-algebraic properties. However, these works focus on generating static visual representations of large graphs, so it is difficult to interactively change their layouts for analytical purposes (e.g., pin some interesting nodes and drag certain parts of the graph).

2.2. Force-Directed Algorithms

Force-directed algorithms are commonly used to support visual analysis of graphs, because they are conceptually simple and able to produce aesthetically pleasing layouts. These algorithms are also called energy-based methods, since they seek to minimize the net force on all vertices. Many practical algorithms have been proposed (see [26] for details).

The spring-electrical model [11] is a popular force-directed layout algorithm, which generates graph layouts based on two types of forces: attractive force and repulsive force. This algorithm cannot handle large graphs well. The complexity of calculating repulsive force is $O(N^2)$, where N denotes number of nodes in the graph. For a graph with a million nodes, each iteration of repulsive force calculation requires tera-scale computations. This is beyond the processing power of a typical consumer desktop CPU. To improve performance, a natural approach is to compute approximated forces with some heuristic methods, instead of striving for the optimal solution. Fruchterman et al. [11] propose a grid-variant algorithm that accelerates repulsive force computation by splitting nodes into grids. Based on the Barnes-Hut Tree [27], a tree structure is used to speed up repulsive force calculation by grouping distant nodes as a super node ([16] and [28]). Moreover, [13] and [29] propose the Fast Multiple Method (FMM [12]) to accelerate repulsive force computation.

In addition to the performance issue, another problem with the spring-electrical model is local minimum configurations of a large graph layout, particularly when randomizing the initial layout. A multi-level approach can overcome this limitation ([16], [30] and [31]). A sequence of successive smaller graphs are generated by graph coarsening and partitioning techniques to simplify the topology. Global optimal layout is obtained from a small graph, which is then used as a starting layout for the next level, until the finest graph layout has been achieved. Bartel et al. [32] summarize the multi-level layout paradigm in three phases: coarsening, placement, and single level layout. They conclude that there is no clear winning combination, since different coarsening, placement and layout algorithms have different attributes.

2.3. Graph Layout on the GPU

GPUs were designed for videogames and graphics. The remarkable advances in performance and programmability make GPUs popular for general purpose computation [22]. GPUs have been shown to produce rapid speed-ups in even straightforward implementations of the repulsive force calculations in force-directed graph layout [33] [34]. Godiyal et al. [13] present a parallel FMM algorithm on the GPU, and use a K-D tree structure for describing nodes' spatial distribution. Yunis et al. [29] extend a parallel FMM algorithm to multiple GPUs. Moreover, Tikhonova et al. [35] propose a scalable parallel force-directed layout algorithm in distributed computing environments. Frishman et al. [15] apply the multi-level method to GPU based graph layout algorithms.

In this paper, we aim at applying the spring-electrical model based force-directed layout algorithm to large graphs, wherein users can interact with graph layouts to perceive and explore meaningful information. To our knowledge, existing acceleration techniques cannot achieve interactive performance on very large graphs (specifically those with millions of nodes) to support

real-time exploratory interactions. Furthermore, existing work focuses on evaluating performance based on total convergence time, whereas we are more interested in individual iteration time so as to support human interaction during the algorithm run. We contribute a GPU based approximated force-directed layout algorithm, which achieves real-time human interactions on large graphs.

3. Algorithm Outline

In this section, we address the challenges of the spring-electrical model for large graph layout from two key perspectives: performance and result quality. We propose a new approximation algorithm for repulsive force, which intertwines with the multi-level approach.

3.1. Approximation of The Repulsive Force

Both Barnes-Hut Tree [27] and FMM [12] methods calculate approximated repulsive force based on node distribution, and a spatial indexing structure is built and updated in each iteration. To avoid these stages, we propose to use graph topology to approximately calculate repulsive forces. If two nodes belong to two different graph clusters, they are “far away" from each other, and we use their inter-cluster repulsive force for approximation. In total, repulsive forces have two components: *internal-electric-force* and *external-electric-force*. The *internal-electric-force* refers to the repulsive force between pairs of nodes within the same cluster, and the *external-electric-force* is the inter-cluster repulsive force. The *internal-electric-force* is used to obtain the local structure of a graph, while the *external-electric-force* is used for capturing the overview of a graph.

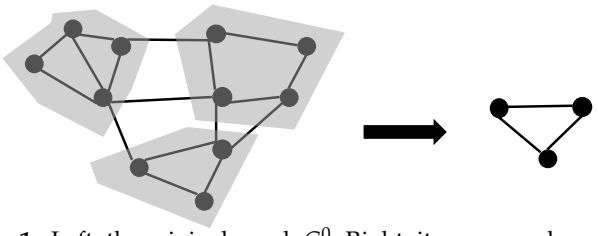

Figure 1. Left: the original graph G^0. Right: its coarsened graph G^1.

Given an undirected graph $G = G^0 = (V, E)$, we first use a multi-level coarsening or clustering algorithm to generate serially successive coarser graphs $(G^1, G^2, \cdots)$, where each (super)node in the next upper level represents a cluster of nodes in its lower level. Figure 1 shows a two-level graph, and the original graph is coarsened based on its topology structure. In the picture, the left graph G^0 is a lower level graph, and the right graph G^1 is a upper level graph. To compute the repulsive forces on all nodes, we begin at the coarsest graph and work our way down. Each node’s total repulsive force in level i with graph G^i is computed as the sum of the *external-electric-forces* inherited from graph G^{i+1} plus the sum of the *internal-electric-forces* within its parent cluster. Thus, we calculate the *internal-electric-forces* at each level of the graph and pass the total repulsive forces down to the next level as *external-electric-forces* until we reach the finest level of the graph G^0. This strategy accumulates the approximated total repulsive forces for each node in a logarithmic hierarchical fashion. Finally, we add the attractive force to the approximated repulsive force of the finest level graph and update each node’s position.

Therefore, the complexity of computing the repulsive forces is the complexity of computing *internal-electric-forces* for a single cluster times the total number of clusters processed. Assuming that the total number of nodes is N, a graph G^0 is evenly partitioned into clusters with P nodes. We have N/P number of clusters in the finest level graph G^0, which equals the number of nodes in the next upper level graph G^1. Graph G^1, then has N/P^2 clusters. In summary, the total number of clusters is $N/P + N/P^2 + N/P^3 + \cdots = N/(P-1)$. The complexity of *internal-electric-forces* is $O(P^2)$ for a single cluster. So the total repulsive force complexity is $O(NP)$. Adding the attractive force, the total complexity is $O(NP + E) \approx O(N + E)$, where E is the number of edges in the graph, and $N >> P$.

However, a graph cannot always be evenly partitioned based on its topology, such as in cases involving scale-free networks where the node degree distribution follows a power law. Different sizes of graph clusters are generated when we apply graph partition algorithms to these kinds of graphs. As a result, the value of P varies for different clusters. We may constrain the size of P to generate graph clusters with the same size. However, we cannot generate the graph structure well if we evenly partition it. In our work, the value of P is depended on the graph type, and we have no constrained value of P. Thus, the actual performance is highly impacted by graph type, and we discuss these details in Section 5.2.

3.2. Multi-Level Approach

Beyond the expensive force computation, another challenge of force-directed layout algorithms is the potential sub-optimal result for large graphs. Various multi-level approaches have been proposed to address this challenge. Bartel et al. [32] present a survey of graph clustering algorithms based on graph topology (e.g., edge collapse, independent set merger, solar merger, etc.). Frishman et al. [15] propose a spectral based graph partitioning algorithm, and Muelder et al. [23] discuss a modularity based hierarchy cluster algorithm. We propose our above approximated repulsive force computation algorithm to weave into the multi-level paradigm for big graph layout.

In this paper, we adopt the solar merger algorithm to build multi-level graphs. The solar merger is introduced by Hachul et al. [36] in their fast multiple multi-level method (FM^3), where each sub-graph is simulated as a solar system. Each node of a sub-graph is classified as sun, planet or moon. The solar system collapses a sub-graph into the sun node of the next level graph. Since a sun node is always the center of a sub-graph, it can represent all nodes within its sub-group for repulsive force computation.

In the placement stage, we keep the positions of all parent nodes and initialize their child nodes along a circle, the center of which is their parent node. This design is based on the solar system where child nodes are either planet nodes or moon nodes. After this, we apply our approximated force-directed layout method by traversing graphs from the top level to the bottom. In summary, the three steps of our multi-level graph layout algorithm are as follows:

- ***Coarsening***: We use the solar merger [36] to generate a sequence of graphs $G^1, G^2, \ldots, G^{coarsest}$, where the maximum number of nodes in the coarsest is 50. Note that this hierarchical structure is strictly preprocessed, and does not need to be updated over time.
- ***Placement***: To initialize the layout for the next level graph, we keep the sun node's position and assign its child nodes along a circle around it.
- ***Layout***: We use our approximated force-directed layout algorithm to update graph layout of each level. We parallelize our algorithm to achieve real-time interactions for graphs with millions of nodes.

The multi-level scheme is used for obtaining high quality results, and reducing the number of iterations. However, the performance of a single iteration for force computation is crucial for human interaction with large graph layout. The GPU based approximated force-directed algorithm is the key to satisfy the demand of interactivity. In other words, any graph layout algorithm can be applied to obtain the initial pre-processed graph layout, and then our interactive GPU graph layout algorithm is deployed for dynamically updating graph layout on the fly. In summary, we generate a sequence of coarsened graphs in the CPU, and then apply the node placement and layout algorithms on the GPU to speed up performance.

4. GPU Implementation

This section describes how the GPU is utilized to accelerate the approximated force-directed layout algorithm. We begin our discussion with a GPU friendly data structure for the multi-level graphs, and then we discuss our GPU parallel force-directed algorithm design.

4.1. Data Storage

There are two common types of data structures used to represent a graph: adjacency matrix and adjacency list. Compared with the adjacency list, the adjacency matrix suffers scalability issues for large graphs. Thus, we use a modified adjacency list, which is similar to the compressed sparse row (CSR) format used by [13]. We extend the modified adjacency list to organize multi-level graphs.

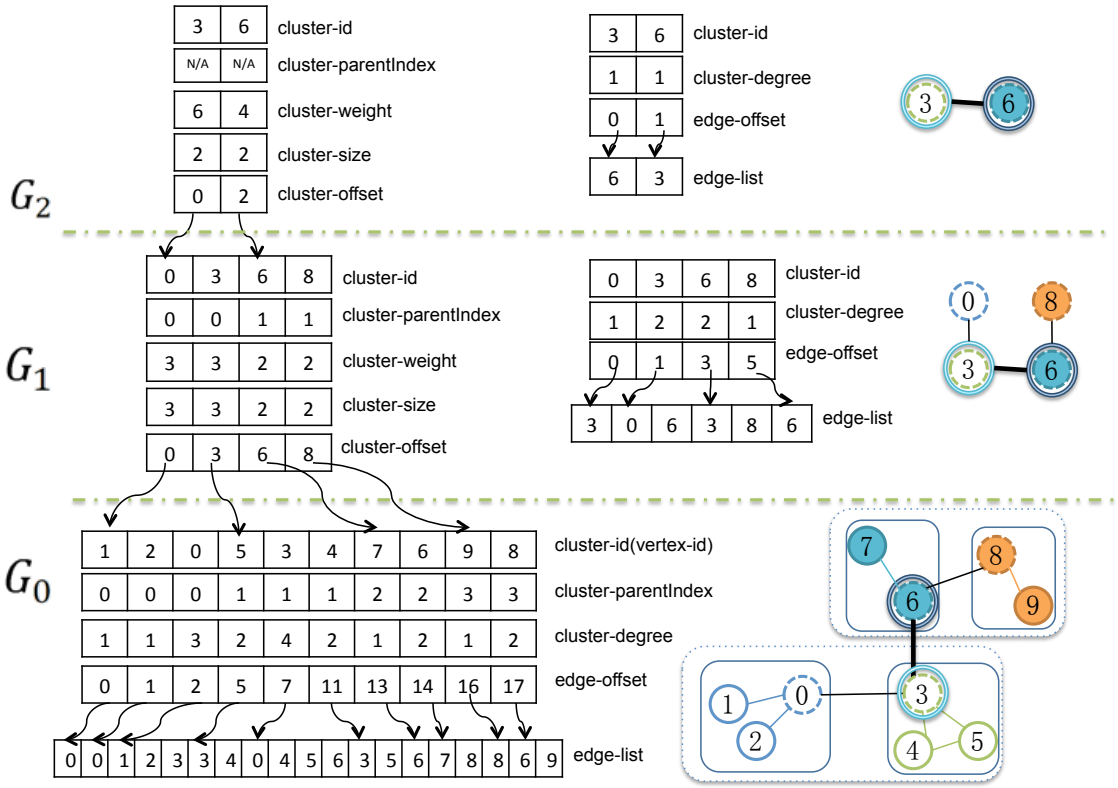

Figure 2. An example of the GPU memory organization of three-levels of graphs. Node-link diagrams (right) show the corresponding level graphs. Several data arrays are stored in GPU memory to represent these graphs. In this figure, G_0 is the original graph, which is partitioned into four sub-graphs. G_1 is the coarsened graph of the original graph, and is partitioned into two sub-graphs. G_2 is the coarsest graph.

Figure 2 shows the GPU data structure for a three-level graph. Except for the finest level graph, there are two groups of data arrays used to describe graph clustering and edge connections at each level. The first group includes *cluster-id*, *cluster-parentIndex*, *cluster-weight*, *cluster-size*, *cluster-offset*, which are used to describe the organization of the clusters of graph nodes:

- *cluster-id* stores the nodes of the current level graph, which are the sun nodes from the lower level graph.
- *cluster-parentIndex* provides the indexes of the sun nodes for the next level graph.
- *cluster-weight* is the number of nodes in the finest level graph that are descendent members of current cluster.
- *cluster-size* is the number of nodes collapsed into this node from its next lower level graph.
- *cluster-offset* is the actual index of the nodes in the sub-graph.

The second group of arrays represents the edge connections of each level graph, including *cluster-degree*, *edge-offset* and *edge-list*:

- *cluster-degree* is the number of edges that other clusters connect to the current cluster.
- *edge-offset* stores the beginning index of the adjacency edge list.
- *edge-list* is the adjacency edge list of the current level graph.

In addition to these data arrays, $Vert_{pos}$ holds the positions of all the nodes in the graph; F_{att}, F_{rep} and $Vert_{displacement}$ respectively store the attractive forces, repulsive forces and node displacements

of each layout iteration. Thus the total memory usage is linear with the total number of nodes and edges. Memory usage is constrained by the original graph, not the value of P.

4.2. GPU Kernels

Our parallel algorithm includes three GPU kernels, shown in Figure 3. Each thread in the *attractive forces* kernel and *updated position* kernel is responsible for one node of the selected level graph, which calculates its attractive force and updates its position based on its total force.

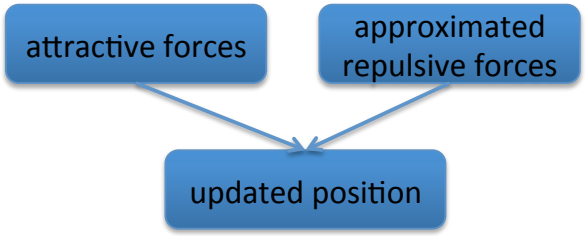

Figure 3. The GPU computation flowchart for a single level graph layout. The kernels of *attractive forces* and *approximated repulsive forces* are described in Algorithm 1 and Algorithm 2.

Algorithm 1: Attractive Force Kernel

Input : $level$, user-chosen graph level
$cluster\text{-}id$, an array of node IDs
$cluster\text{-}degree$, an array of node degrees
$edge\text{-}offset$, an array of node offsets
$edge\text{-}list$, an array of edge list
Output: F_{att}, an array of attractive forces

```
1 for i = 0 to size(cluster-id[level]) − 1 do in parallel
2     id = cluster-id_level [i];
3     degree = cluster-degree_level [i];
4     start = edge-offset_level [i];
5     for j from 0 to degree-1 do
6         adjacent-id = edge-list_level [start + j];
7         F_att(id) += AttForce(id, adjacent-id);
8     end
9 end
```

Algorithm 1 shows the attractive force computation, where its inputs are data arrays with edge information. The kernel of *approximated repulsive forces* is complex. It calculates the approximated repulsive force based on the data arrays with graph clustering information. Algorithm 2 shows the details: the while loop in Line 3 describes the top-down graph traversal to compute approximate repulsive force. In each level graph, there are two parts: computing the *internal-electric-forces* and passing the repulsive forces to the lower level graph as *external-electric-forces*. The first part includes Lines 3-31, with Lines 7-15 describing the special case of the top level graph layout. The second part in Lines 33-44 is concerned with bestowing the repulsive forces from one level graph to a lower level.

4.3. Workload Imbalance

Based on our parallel algorithm design, the thread workload imbalance problem occurs when the sizes of graph clusters vary significantly. Threads handling smaller graph clusters have less work to do compared to threads handling larger graph clusters. To solve this problem, we can evenly partition a graph into sub graphs. In this case, the workload is evenly distributed among threads (e.g., [15]). However, we may lose some graph topology knowledge when clustering the graph this way, which causes the force-directed layout to converge poorly.

Algorithm 2: Approximated Repulsive Forces

Input : $level$, user-chosen graph level
$cluster\text{-}id$, an array of nodes IDs
$cluster\text{-}weight$, an array of cluster weights
$cluster\text{-}parentIndex$, an array of parent indexes
$cluster\text{-}offset$, an array of node cluster offsets
$cluster\text{-}size$, an array of node cluster sizes
$Rep(id1, id2)$, a subroutine to compute repulsive forces of two nodes
Output: F_{rep}, an array of repulsive forces

```
curLevel = TotalLevel −1
//inter-cluster-forces (3-31)
while curLevel ≥ level do
    size = size( cluster-id_curLevel) − 1
    if curLevel = TotalLevel −1 then
        //all nodes belong to one cluster in top level
        for i = 0 to size do in parallel
            id1 = cluster-id_curLevel [i]
            w1 = cluster-weight_curLevel [i]
            for j = 0 to size do
                id2 = cluster-id_curLevel [j]
                w2 = cluster-weight_curLevel [j]
                F_rep(id1) += Rep(id1, id2) ∗ w1 ∗ w2
            end
        end
    else
        for i = 0 to size do in parallel
            id1 = cluster-id_curLevel [i]
            w1 = cluster-weight_curLevel [i]
            index = cluster-parentIndex_curLevel [i]
            //find the nodes belonging to one cluster
            start = cluster-offset_curLevel+1 [index]
            size = cluster-size_curLevel+1 [index]
            end = start+size − 1
            for j = start to end do
                id2 = cluster-id_curLevel [j]
                w2 = cluster-weight_curLevel [j]
                F_rep(id1) += Rep(id1, id2) ∗ w1 ∗ w2
            end
        end
    end
    //pass forces to next level, until the finest level
    if curLevel > 0 then
        for i = 0 to size do in parallel
            id = cluster-id_curLevel [i]
            start = cluster-offset_curLevel [i]
            size = cluster-size_curLevel [i]
            end = start+size − 1
            for j = start to end do
                childrenID = cluster-id_curLevel−1 [j]
                F_rep(childrenID) = F_rep(id)
            end
        end
    end
    curLevel -= 1
end
```

To deal with the workload imbalance problem, Chen et al. [42] propose a dynamic load balancing method to effectively exploit the GPU concurrency. Hong et al. [43] present a warp-centric method that considers the detailed GPU architecture (a basic group of threads that share instructions on the GPU is called a warp) to achieve performance gains. However, both of them introduce extra work for thread assignment.

In next section, we show the performance of the imbalance workload problem. We propose using a threshold of the maximum cluster size for generating graph clusters, to avoid outliers. We plan to consider other solutions to the workload imbalance problem to improve performance in future work.

5. Results and Discussion

We tested our algorithm on a desktop computer running Windows 7 Enterprise, which was equipped with an Intel i7 processor and an NVIDIA GeForce GTX 680 graphics card programmed with CUDA 7.5. We choose FM^3 [36], a classic force-directed layout algorithm widely used for large graph layout ([9], [15] and [35]), as a baseline to evaluate the performance and visual results of our algorithm.

In total, we picked five different graphs as five test cases for our algorithm, including one artificial graph generated by ourselves and other well-known graphs. Table 1 shows the summary of these five graphs.

Table 1. Summary of the five tested graphs

Graph	Number of Vertices	Number of Edges
crack	10,240	30,380
finan512	74,752	261,120
web-Stanford	255,265	1,941,926
grid-mesh	1,000,000	1,998,000
roadNet-TX	1,379,917	3,843,320

5.1. Visual Assessment

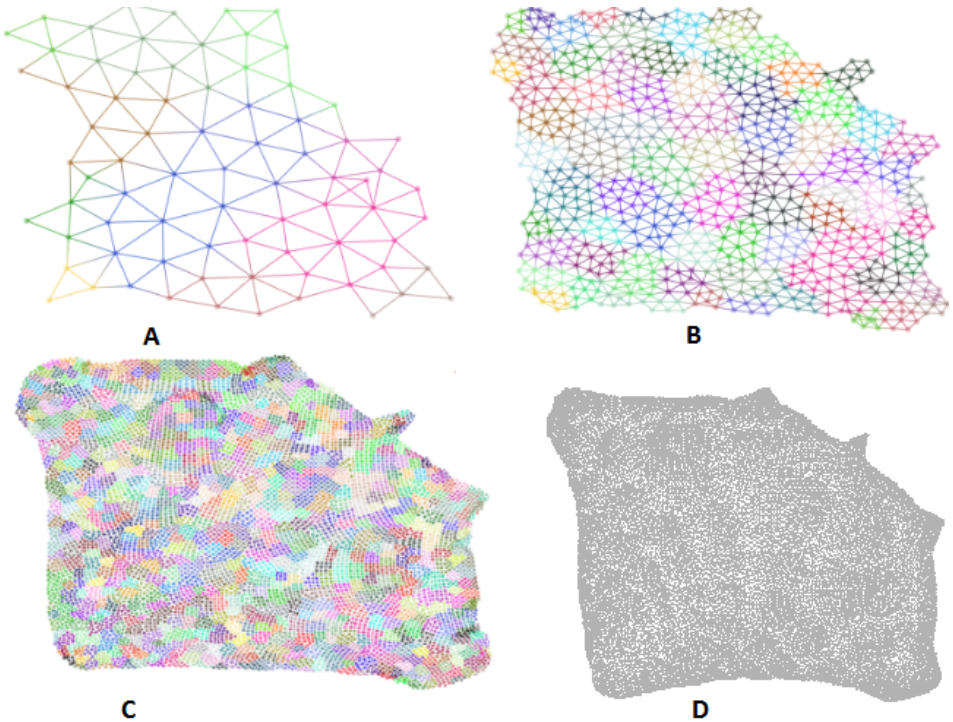

Figure 4. The *crack* layout results. A, B, C are three levels of coarsened graphs generated by our algorithm (graph nodes colored by cluster membership). D is the result from FM^3 implemented by ODGF [38].

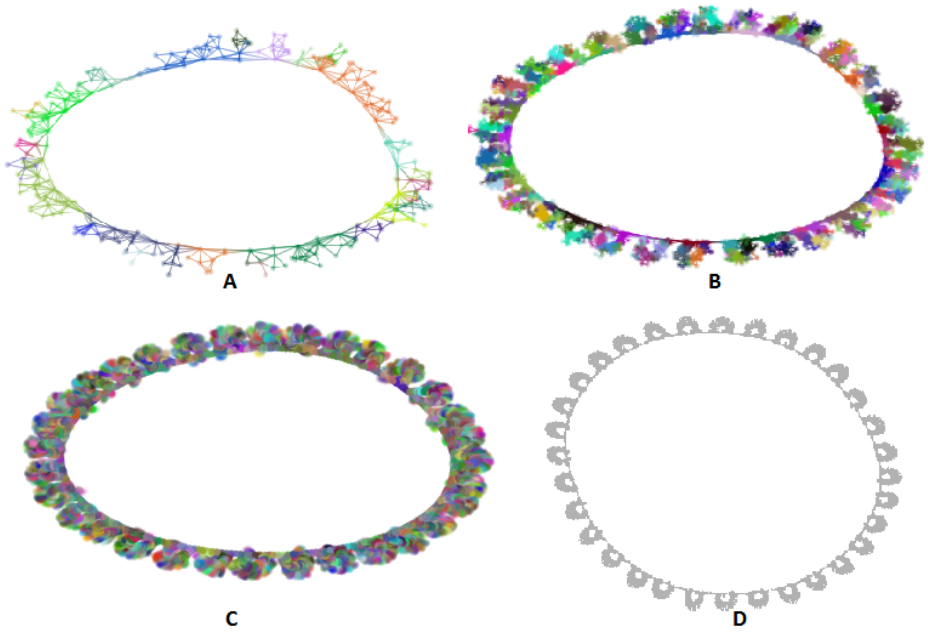

Figure 5. The *finan512* layout results. A, B, C are three levels of coarsened graphs generated by our algorithm (graph nodes colored by cluster membership). D is the result from FM^3 implemented by ODGF [38].

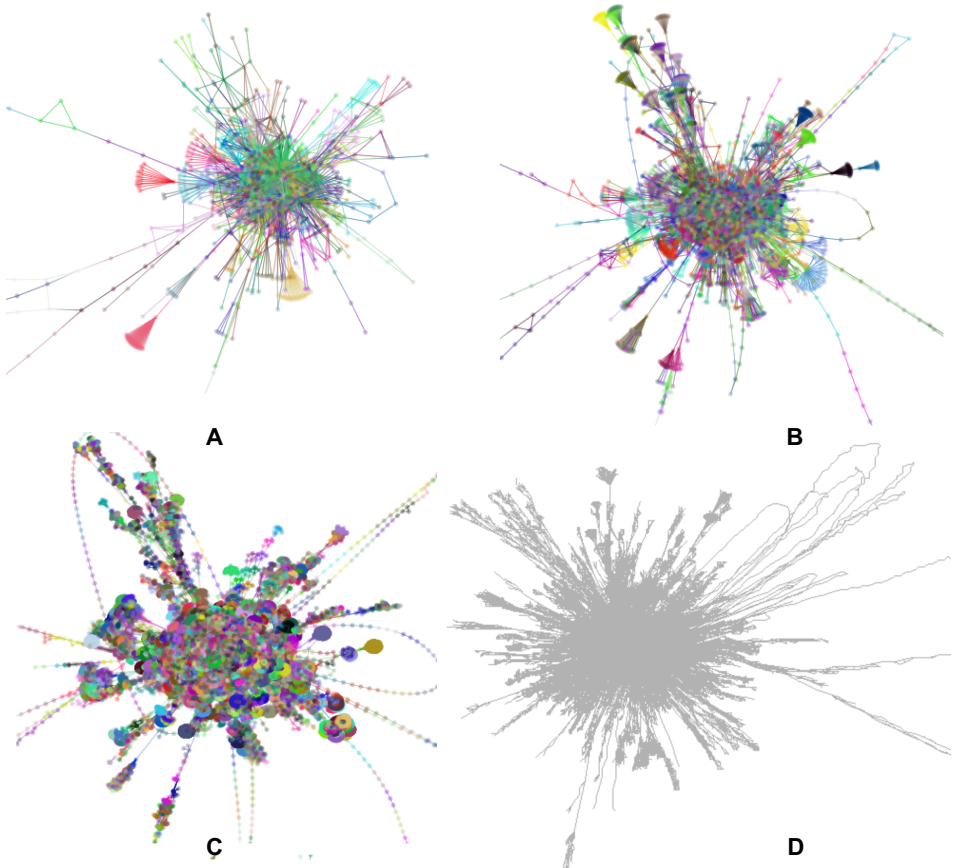

Figure 6. The *web-Stanford* layout results. A, B, C are three levels of coarsened graphs generated by our algorithm (graph nodes colored by cluster membership). D is the result from FM^3 implemented by ODGF [38].

Making a fair, reasonable and comprehensive evaluation of visual representations of a graph is difficult because the assessment could be highly subjective (e.g., when considering the aesthetics aspect). Most previous large graph layout algorithms attempt to generate visually pleasing graph layouts but ignore the capability for human interactions to adjust layout results. Alternatively, *we emphasize the importance of enabling interactive capabilities to improve visually pleasing layouts* . This can potentially help to generate more meaningful layouts for a large graph, especially from a human perception and sensemaking insight perspective.

Figures 4, 5 and 6 show results of our multi-level algorithm (200 iterations per level without human interactions) and FM^3 on graphs *crack*, *finan512* and *web-Stanford*. The visual results generated by our method can achieve at least the same visual quality as those from FM^3.

5.2. Performance Analysis

Termination conditions of iterative force computation impact the quality and performance of graph layouts. Previous works (e.g., [13], [15]) focus on generating static pleasing layouts of large graphs, and they select a fixed number of iterations to halt the force computation. However, simply using a fixed number of iterations as a termination condition cannot guarantee that the computed layouts always satisfy user requirements, especially considering the fact that different users may focus on different aspects to evaluate graph layouts. For example, Endert et al. [37] argue for "human is the loop" sensemaking, in which analytical algorithms such as graph layouts are fundamentally centered around human sensemaking interactions to support exploratory knowledge discovery. Thus, it is important to ***support user interaction with graph layouts*** when sensemaking on large graphs.

The support for dynamic interaction with large graph layouts is a key design goal in this paper. Thus, the average computation time for one iteration, rather than the total amount of time for computing a converged layout, is used as a meaningful measurement to evaluate the performance of our algorithm. This one-iteration oriented computation time is calculated by averaging the time of all iterations needed in our algorithm to achieve the finest level graph. We run both our algorithm and FM^3 (implemented by ODGF [38]) at least 20 times to collect enough (statistically meaningful) data for comparison. In addition, we consider rendering time because it directly impacts the interactiveness of the actual visual layouts presented to users.

The graph layout computation follows the multi-level graph structure. It starts from the top level graph, then expands the repulsive forces into its children level. When users drag a node, the algorithm updates each nodes positions from the top level graph to the finest level graph. If a dragged node belongs to a top level graph, its changed layout expands to its lower level graphs. If the dragged node only belongs to the finest level graph, the changed layout does not need to be expanded.

Table 2. Performance results (milliseconds per iteration) on each tested graph.

Performance	crack	finan512	web-Stanford	grid-mesh	roadNet-TX
FM^3 on CPU	63.900	630.000	2581.399	7814.100	11484.799
Our approximated force calculation on CPU	6.799	83.738	337.049	476.548	473.564
GPU Kernel Attractive-Force	0.315	0.660	38.177	1.287	1.767
GPU Kernel Approximated Repulsive-Force	2.099	3.887	28.184	14.484	17.819
GPU Kernel Updated-Position	0.007	0.008	0.015	0.013	0.011
GPU Others	0.302	0.383	0.728	1.296	1.727
GPU Total	2.732	4.938	67.104	17.080	21.324
Rendering (graph nodes)	0.809	1.288	2.252	4.958	9.727

Table 2 summarizes the measured performance metrics of each tested graph. Based on the first two rows, it is clear that the CPU version of our algorithm, on average, runs at least 7 times faster than FM^3. For graphs with millions of nodes, the performance of our algorithm remains relatively stable, compared to that of FM^3. The worst performance of our algorithm (GPU Version) is much

smaller than 100ms [39], which verifies that the performance of our algorithm can support real-time user interactions for graphs with millions of nodes.

By comparing columns 4 and 5, we find that the performance of our algorithm (GPU version) for mesh-like graphs (*grid-mesh*) is better than that for the small-world graphs (*web-Stanford*). The reason for this is that we can get evenly partitioned graph clusters based on the solar merger for mesh-like graphs. However, the node degree of small world graphs is unevenly distributed, following the power law distribution. Therefore, the sizes of sub-graphs follow the power law distribution when applying graph clustering algorithms (e.g., Figure 7 shows the sub-graph distribution of *web-Stanford*). As a result, the GPU threads for *internal-electric-force* computation suffer the workload imbalance problem.

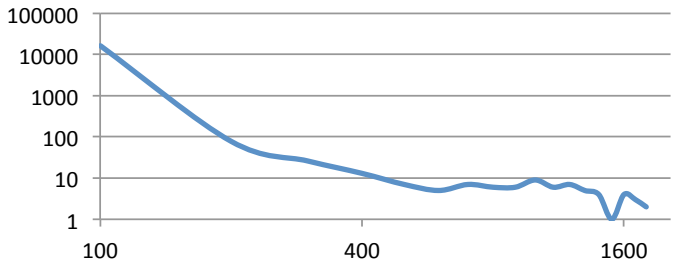

Figure 7. The sub-graph distribution of *web-Stanford*. The X-axis is the number of nodes, the Y-axis is the number of sub graphs containing that many nodes. It follows a power law distribution.

To address this problem, a graph coarsening or partition algorithm that can evenly partition a graph is a better choice (e.g., the spectral graph partition algorithm [15]). However, evenly partitioned graphs may not generate the optimal graph topology structure as well. For example, the spectral graph partition algorithm [15] can evenly divide a graph into sub groups based on a graph's spectrum. But this algorithm loses the graph's topology, which may cause difficulties for the force-directed layout algorithm to converge into a global minimal configuration. To balance the trade-offs between performance and quality of layouts, following the discussion in Section 4.3, we use a threshold of maximum cluster size P to avoid generating giant clusters, which alleviates the workload imbalance problem. Another possible solution is to use the stochastic force-directed layout algorithm discussed in [40]. For giant clusters, the size of which are larger than the threshold, we can randomly pick neighbor nodes from the cluster for the *internal-electric-force* computation. In fact, workload imbalance is a classical problem for GPU programming, such as GPU based ray tracing [44] and GPU based graph traversal [45]. We will investigate the thread scheduling and job management methods from these previous works to improve performance in future work.

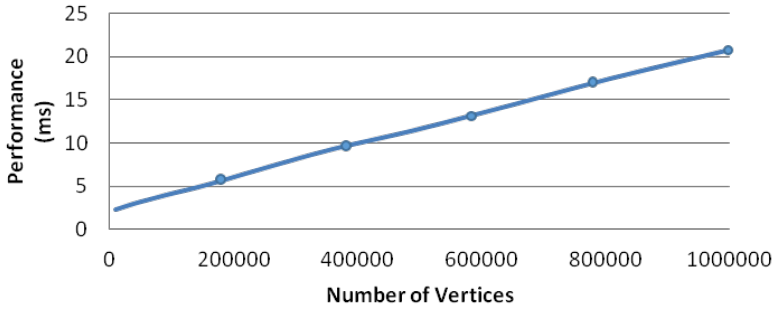

Figure 8. The performance of grid-mesh layouts. The X-axis is the number of vertices, and the Y-axis is the performance time. The average P of each cluster is 9, based on the solar merger algorithm.

We have tested the scalability of our algorithm for mesh-like graphs. We generate successive grid-meshes, which can be evenly partitioned by the solar merger. Thus, the repulsive force computation of mesh-like graphs avoids the workload imbalance problem. Figure 8 shows the

performance of our algorithm (GPU version) for these grid-meshes. It is clear that our algorithm can scale to mesh graphs with millions of nodes while also guaranteeing real-time user interactions.

It is difficult to compare our GPU version with other GPU implementations. Godiyal et al. [13] provide the total time to generate appealing graph layouts without giving the number of iterations. Besides the GPU based force computation, their algorithm spends extra time on tree structure building (CPU) and CPU-GPU data movement. Frishman et al. [15] report that their algorithm is 2-4 times faster than FM^3, and their GPU implementation can achieve times 5.5 faster than their CPU version. However, their algorithm takes more time for each iteration computation. Even though our algorithm needs a greater number of iterations to achieve similar results (200 iterations per level compared with Frishman's 50 iterations per level), we can achieve real-time frame-rate for interactive large graph layouts.

We use the OpenGL VBOs (Vertex Buffer Objects) to efficiently render graphs. This avoids data movement between the CPU and the GPU. The last row of Table 2 shows the rendering performance. To visualize a million nodes takes about 5 milliseconds. Based on our GPU implementation, we can directly map the GPU memory to OpenGL VBOs without any CPU-GPU memory communication. This implementation keeps all graph data in the GPU memory, which fully utilizes the GPU resources for efficient computing and rendering.

5.3. Discussion

The evaluation of graph layout algorithms is challenging. Force-directed algorithms based on stress models [19] use the stress error functions to quantify layout quality and terminate the computation [41]. These algorithms need to store pairwise node distances, so they do not scale well for graphs with millions of nodes. Our method (spring-electric model) is based on adjacency lists, which can be applied to big graphs. However, termination conditions (e.g., the energy threshold and local minimum) of spring-electric model still need further investigation. Our work attempts to enable users to interrupt the computation and modify the layout results, so we focus on a single iteration of force calculation in the performance analysis.

One limitation of our algorithm is that it only considers the scalability of the repulsive force computation. For dense graphs, the number of edges may be linear with squared number of nodes. In this case, computing attractive forces is a bottleneck. Compared with less dense graphs, our algorithm may not handle this well.

6. Usage Scenarios

In this section, we present two case studies to demonstrate how users can interact with a large graph for sensemaking tasks using our system. We first demonstrate that a user can explore a large graph structure based on the multi-level paradigm. Then we show that a user can interact with the graph layout to gain new insights.

6.1. Visual Exploration of Web Networks

Suppose that Elijah is a network analyst and he attempts to analyze the structure of the *web-Stanford* graph. He first processes this graph using the solar merger to generate four abstracted levels, which leads to a total of five level graphs. He explores these graphs in a top-down order. When he is satisfied with the layout of one level graph, after making some tweaks to the layout, he then expands and explores the next level layout. Figure 9 shows the five levels of graph layout resulting from this process.

By comparing the levels of graph layout, Elijah notices that two specific sub-graphs show different evolutions. These two sub-graphs are circled in red and blue in Figure 9. At first, the basic structures of these two sub-graphs are in the shape of a triangle. After Elijah moves to the next level graph, the sub-graph circled in red changes to a clique-like structure, while the sub-graph circled in blue changes to a tree-like structure. In the finest level graph, the former becomes a compact cluster

but the latter still remains tree-like. The difference in structure evolutions of the two sub-graphs catches Elijah's attention, especially the first sub-graph. He wants to better understand why the structure of the first sub-graph changes more, so he decides to zoom in to see more details for further exploration.

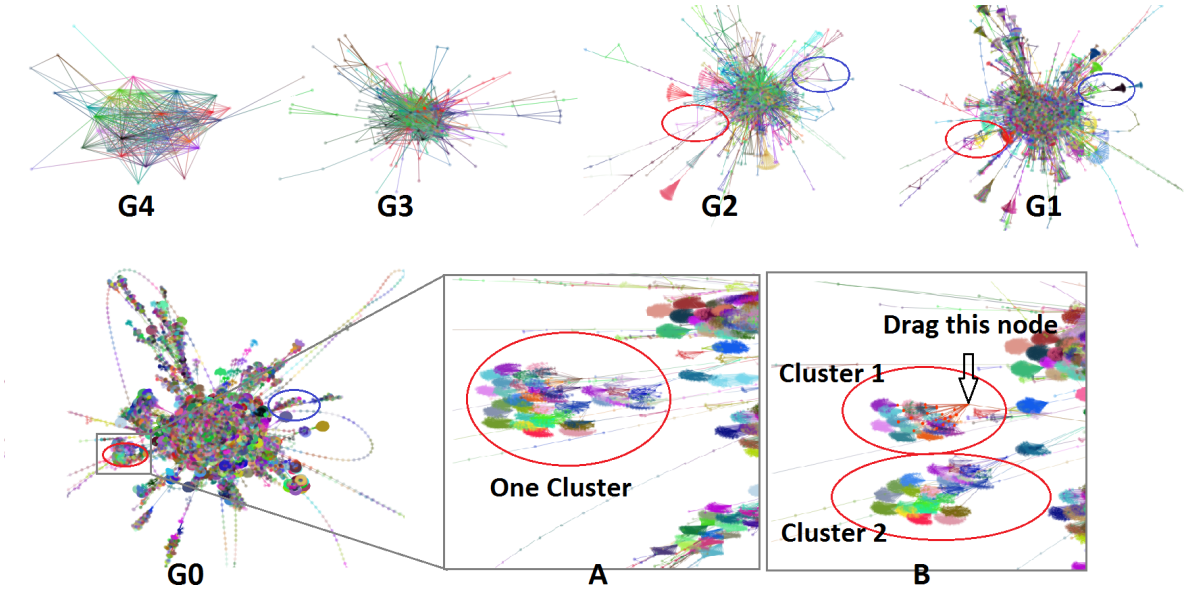

Figure 9. Five level graph layout of *web-Stanford* data (graph nodes colored by cluster membership). From coarse to fine, the graph levels are G_4,G_3,G_2,G_1,G_0. The red and blue circles are highlighted sub-graphs. Figure *A* shows the detailed layout when the user zooms into the red circle. Figure *B* shows that one cluster separates into two clusters when the user drags a hub node.

Figure 9(A) shows the detailed layout result. This sub-graph consists of multiple small clusters that overlap each other. Elijah finds a hub node and drags it a little bit to see what happens. It turns out that several small clusters dynamically follow the dragged one, as shown in Figure 9(B). After this, he quickly realizes that this sub-graph can actually be separated into two clusters.

In summary, with our proposed technique, users can inspect the evolution of large graph layouts by exploring different level layouts. With the capability of dragging nodes and modifying the layout in real time, our algorithm enables users to flexibly reorganize the layout and even assist in improving large graph layouts. This approach combines human cognition with computational power (e.g., GPU graph layout algorithms) to solve difficult problems in sensemaking of large graphs.

6.2. *Visual Exploration of Co-authorship Networks*

In this case study, we present a scenario of an analyst making sense of a co-authorship network. Suppose Grace is a social network analyst. She is interested in how research scientists collaborate with each other. Thus, she downloads publications from DBLP [46], parses them and generates a co-authorship network (515,103 nodes and 1,856,690 edges). At first, she obtains a graph layout based on FM^3 as shown in Figure 10(T0). The visualization is a dense and intermingled network. The core of the network contains most of the nodes and edges, which has a nontrivial structure. Unfortunately, the research communities cannot be visually separated using FM^3. Thus, she switches to using our algorithm to interact with the graph layout.

First, Grace identifies research communities by their central actors (hub node of a subgraph). She is interested in a research community with *Jean-Daniel Fekete* (JDF) as a central actor. To separate the JDF community from the core network, she drags this central actor away from the core in Figure 10(T1). The nodes belonging to this community move towards the central actor. By observing the nodes' movement, shown in Figures 10(T2 and T3), she easily identifies that the nodes are

separated into two groups. One group of nodes moves slowly and blends into the core network, while the other group of nodes moves faster and they are separated well from the core. She examines the details of each node, and finds that the first group consists primarily of senior researchers, while the other consists of junior researchers.

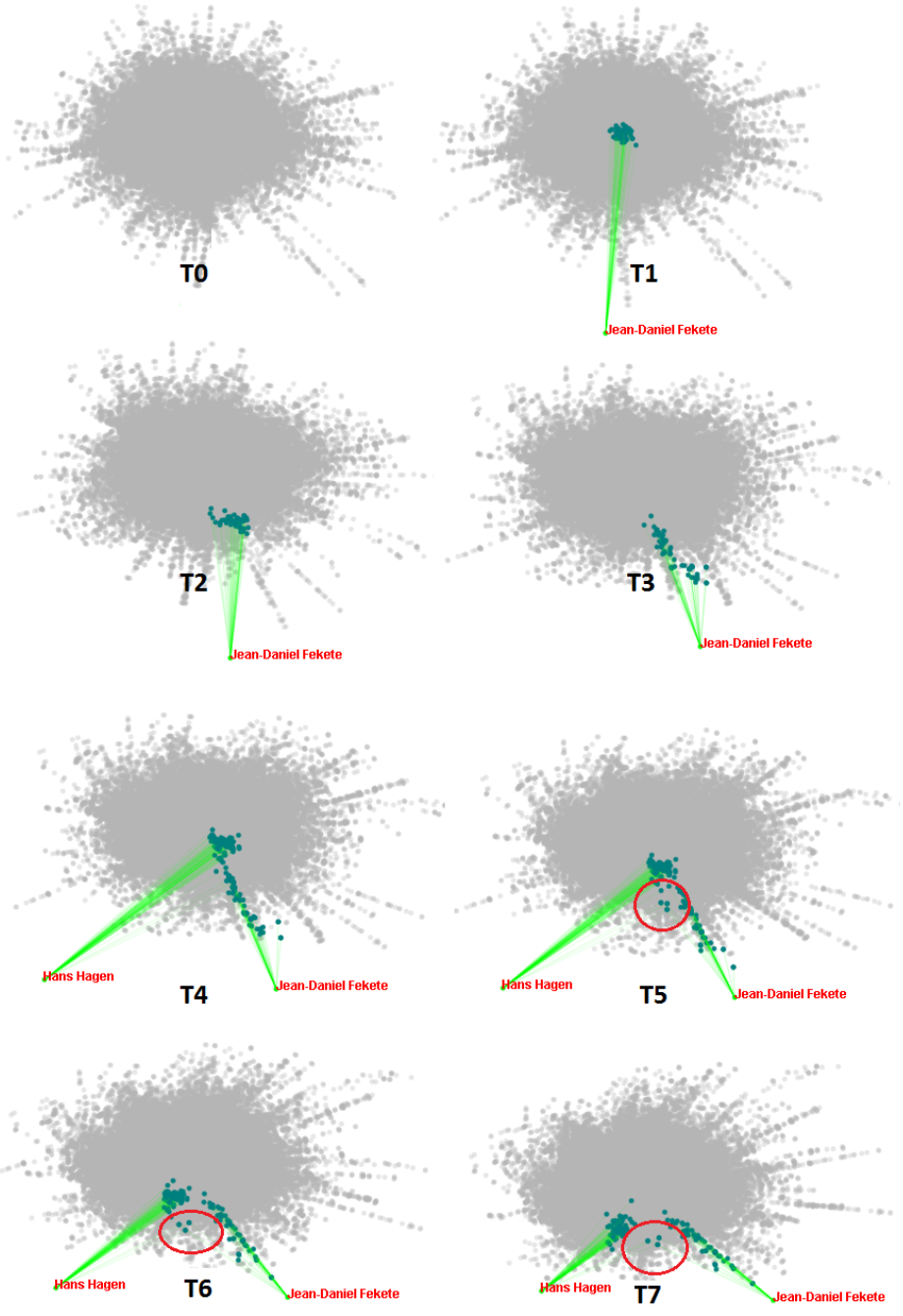

Figure 10. The graph layout of the DBLP co-authorship network, showing a series of interactions. T0 shows the initial graph layout using the FM^3 algorithm. The remaining figures show the very similar, but interactive, layout using our algorithm. In T1, *Jean-Daniel Fekete* is dragged away from the network center (the user selected community is automatically highlighted in green). T2 and T3 are snapshots of the resulting dynamic layout movement. In T4, *Hans Hagen* is dragged away from the network center. T5, T6, and T7 show resulting node movement.

Grace hypothesizes that some of the senior researchers may belong to multiple research communities. If they were tied to other large communities, this would explain why their nodes were

being held back from moving toward JDF. To verify this, she drags *Hans Hagen* (HH) away from the network core, as in Figure 10(T4). Then she observes node movement, and finds that three senior researchers collaborate with both JDF and HH, as circled in Figure 10(T5, T6 and T7). Evidently, JDF collaborates with many junior researchers, while HH emphasizes senior researchers.

Compared with other approaches, Grace gains more useful insights about the large graph by interacting with the graph layout using our algorithm. She easily separates research communities from a large nontrivial network while analyzing its structure. She also finds researchers connecting multiple communities.

7. Lessons Learned

In this paper, we seek to enable human-in-the-loop layout of large graphs. Based on this key design consideration, we contribute a topology based method to accelerate a force-directed graph layout algorithm that can support real-time exploration of large graphs. We parallelize this method by using the powerful resources of the GPU. We summarize two major lessons learned about designing accelerated algorithms to interactively layout large graphs as follows.

Taking advantage of parallelization. Performance is a key concern for designing algorithms to handle big data such as large graphs. Parallelization is one option to accelerate algorithms. The GPU has been praised for its significant performance improvement and programmability for general purpose computation. Thus, researchers can rethink existing algorithms to utilize the GPU for *parallelization*.

We design parallel algorithms by considering two important aspects: *data structures* and *independent instructions*. We use modified adjacency lists to organize multi-level graphs, and separate our force-directed algorithm into different kernels to reduce its dependency.

Light computation per iteration. Figure 11 illustrates the flow chart of our interactive large graph layout system, where force computation is a potential bottleneck. Previous graph layout algorithms cannot support dynamic real-time human-in-the-loop interaction because of their heavy computation per iteration. To support sensemaking of large graphs, we emphasize *light computation per iteration*. Our topology based method requires less computation, since we save in the calculation of approximated forces. The rapid iteration enables humans to be involved in the graph layout process along with the automatic computation, which can lead to customized layouts that make more sense for individual users and tasks.

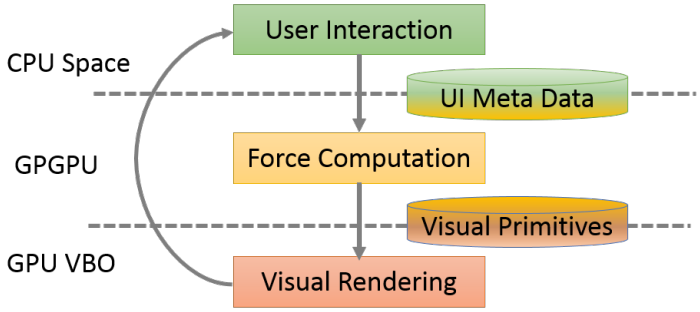

Figure 11. The flow chart of the iteration cycle of our large graph layout system. User interactions are inserted into iterative graph layout algorithm. During each iteration of the force directed layout algorithm, user interaction data are passed to the GPU to compute forces, and then visual primitives are updated based on the force calculations. To support interactivity, these iterations must be as light-weight as possible.

As is shown in Figure 11, with our approach, humans are involved in the iterative graph layout process. For example, they can modify and halt the computation iterations in advance to reduce the total time to layout large graphs. Previous algorithms use a fixed number of iterations for layout generation to guarantee a converged result, which may waste some iterations and increase the total run time. Striving for the best solution to halt layout computation may introduce more memory and

computation requirements that may increase total time to layout a large graph. In such cases, light computation per iteration is a good design choice to obtain better and meaningful layout results in a short time by involving human interaction.

However, light computation per iteration may need more iterations to get a converged graph layout in some cases, so it cannot always guarantee a decrease in total run time. Thus, the design of large graph layout algorithms often involves a trade-off between the *total run time* and the *per-iteration run time* performance. In this work, we favor per-iteration run time via light computation, to enable interactive layout.

8. Conclusion

To enable human interaction with large graph layout, we contribute a fast force-directed layout algorithm and a detailed GPU implementation. Our contribution contains two key novelties. First, we calculate the approximated repulsive force based on the topology of the graph, instead of the spatial distribution of its nodes, which avoids building, traversing, and updating a spatial indexing data structure. We use a multi-level clustering approach, carefully coordinated with a top-down approximate force computation. Second, we parallelize the algorithm in a GPU implementation, which includes the design of the GPU kernels and the GPU data structure memory layout that support our top-down approximation method. Taken together, our contributions enable light-weight computation per algorithm iteration, which increases the interactive frame-rate of the layout.

We evaluated our method, which generates visually pleasing graph layouts for five benchmark graphs, and provides fast iteration performance that supports real-time user interactions on large graphs. The results indicate that our algorithm enables real-time interaction with graphs of over a million nodes. We present two case studies to demonstrate that our algorithm can support users in exploring large graphs and dynamically refining layouts. Finally, we summarize lessons learned from this work for designing algorithms that enable real-time human interaction with large graphs. We hope that these lessons will further inform future algorithm design, to enable a broad variety of user interactions in support of sensemaking tasks on very large graphs.

Acknowledgments: This research was partially supported by NSF grant IIS-1447416.

References

1. Heer J.; Boyd D. Vizster: Visualizing online social networks, IEEE Symposium on Information Visualization, 2005, pp. 32-39.
2. Henry N.; Fekete JD,; McGuffin MJ. NodeTrix: a hybrid visualization of social networks, IEEE TVCG, 2007, 13(6), pp. 1302-1309.
3. Endert A.; Fiaux P.; North C. Semantic interaction for visual text analytics, ACM Conference on Human Factors in Computing Systems, 2012, pp. 473-482.
4. Sun M.; Mi P.; North C.; Ramakrishnan N. BiSet: Semantic Edge Bundling with Biclusters for Sensemaking, IEEE TVCG, 2016 Jan 31, 22(1), pp. 310-319.
5. Zhang H.; Sun M.; Yao DD.; North C. Visualizing Traffic Causality for Analyzing Network Anomalies, ACM International Workshop on Security and Privacy Analytics, 2015, pp. 37-42.
6. Sun M.; Bradel L.; North CL.; Ramakrishnan N. The role of interactive biclusters in sensemaking. ACM Conference on Human Factors in Computing Systems, 2014, pp. 1559-1562.
7. Kaufmann M.; Wagner D. Drawing Graphs: Methods and Models, Springer, 2011.
8. Battista GD.; Eades P.; Tamassia R.; Tollis IG. Graph drawing: algorithms for the visualization of graphs, Prentice Hall PTR, 1998.
9. Hachul S.; Jünger M. An experimental comparison of fast algorithms for drawing general large graphs, Springer Berlin Heidelberg, 2006, pp. 235-250.
10. Shneiderman B.; Aris A. Network visualization by semantic substrates, IEEE TVCG,2006, 12(5):733-40.
11. Fruchterman TM.; Reingold EM. Graph drawing by force-directed placement, Software Practice and Experience, 1991, 21(11):1129-64.

12. Greengard L.; Rokhlin V. A fast algorithm for particle simulations. Journal of computational physics, 1987. 73(2):325-48.
13. Godiyal A.; Hoberock J.; Garland M.; Hart JC. Rapid multipole graph drawing on the GPU, Springer Berlin Heidelberg. 2009, pp. 90-101.
14. Burtscher M.;, Pingali K. An efficient CUDA implementation of the tree-based barnes hut N-body algorithm. GPU computing Gems Emerald edition. 2011 Jan, pp. 13-75.
15. Frishman Y.; Tal A. Multi-level graph layout on the GPU, IEEE TVCG, 2007 Nov, 13(6).
16. Hu Y. Algorithms for visualizing large networks. Combinatorial Scientific Computing, 2011 Sep, 5(3), pp. 180-186.
17. Davidson R.; Harel D. Drawing graphs nicely using simulated annealing, ACM Transactions on Graphics, 1996 Oct 1, 15(4), pp. 301-331.
18. Eades P. A heuristics for graph drawing, Congressus numerantium, 1984, pp. 146-160.
19. Kamada T.; Kawai S. An algorithm for drawing general undirected graphs, Information processing letters, 1989 Apr 12; 31(1), pp. 7-15.
20. Harel D.; Koren Y. Graph drawing by high-dimensional embedding, Springer Berlin Heidelberg, 2002, pp. 207-219.
21. Koren Y.; Carmel L.; Harel D. ACE: A fast multiscale eigenvectors computation for drawing huge graphs, IEEE Symposium on Information Visualization, 2002, pp. 137-144.
22. Mi P.; Sun M.; Masiane M.; Cao Y.; North C. AVIST: A GPU-Centric Design for Visual Exploration of Large Multidimensional Datasets, Informatics, 2016, pp. 40018.
23. Muelder C.; Ma KL. A treemap based method for rapid layout of large graphs, PacificVIS, 2008, pp. 231-238.
24. Wong PC.; Foote H.; Mackey P.; Chin G.; Huang Z.; Thomas J. A space-filling visualization technique for multivariate small-world graphs, IEEE TVCG, 2012, 18(5):797-809.
25. Khoury M.; Hu Y.; Krishnan S.; Scheidegger C. Drawing Large Graphs by Low Rank Stress Majorization, Computer Graphics Forum, 2012, Vol. 31, pp. 975-984.
26. Kobourov SG. Spring embedders and force directed graph drawing algorithm. arXiv preprint, 2012.
27. Barnes J.; Hut P. A hierarchical O (N log N) force-calculation algorithm, Nature, 1986, pp. 446-449.
28. Quigley A.; Eades P. FADE: Graph drawing, clustering, and visual abstraction, Springer Berlin Heidelberg, 2011, pp. 197-210.
29. Yunis E.; Yokota R.; Ahmadia A. Scalable force directed graph layout algorithms using fast multipole methods, Parallel and Distributed Computing, 2012, pp. 180-187.
30. Gajer P.; Kobourov SG. GRIP: Graph drawing with intelligent placement, Springer Berlin Heidelberg, 2002, pp. 222-228.
31. Walshaw C. A multilevel algorithm for force-directed graph drawing, Springer Berlin Heidelberg, 2001, pp. 171-182.
32. Bartel G.; Gutwenger C.; Klein K. An experimental evaluation of multilevel layout methods, Springer Berlin Heidelberg, 2011, pp. 80-91.
33. Auber D.; Chriricota Y. Improved efficiency of spring embedders: Taking advantage of GPU programming. Visualization, Imaging, and Image Processing, 2007, pp. 169-175.
34. Jezowicz T.; Kudelka M.; Platos J.; Snasel V. Visualization of large graphs using GPU computing, Intelligent Networking and Collaborative Systems, 2013, pp. 662-667.
35. Tikhonova A.; Ma KL. A scalable parallel force-directed graph layout algorithm, EGPGV, 2008, pp. 25-32.
36. Hachul S.; Jünger M. Drawing large graphs with a potential-field-based multilevel algorithm, Graph drawing, Springer Berlin Heidelberg, 2005, pp. 285-295.
37. Endert A.; Hossain MS.; Ramakrishnan N.; North C.; Fiaux P.; Andrews C. The human is the loop: new directions for visual analytics, Journal of intelligent information systems, 2014, 43(3), pp. 411-435.
38. Chimani M.; Gutwenger C.; Jünger M.; Klau GW.; Klein K.; Mutzel P. The open graph drawing framework (OGDF), Handbook of Graph Drawing and Visualization, 2011, pp. 543-569.
39. Liu Z.; Heer J. The effects of interactive latency on exploratory visual analysis, IEEE TVCG, 2014, 20(12):2122-31.
40. Chalmers M. A linear iteration time layout algorithm for visualizing high-dimensional data, Visualization Proceedings, 1996, pp. 127-131

41. Ingram S.; Munzner T.; Olano M. Glimmer: Multilevel MDS on the GPU, IEEE TVCG, 2009, 15(2), pp. 249-261.
42. Chen L.; Villa O.; Krishnamoorthy S.; Gao G. Dynamic load balancing on single-and multi-GPU systems, Parallel & Distributed Processing, 2010, pp. 1-12.
43. Hong S.; Kim SK.; Oguntebi T.; Olukotun K. Accelerating CUDA Graph Algorithms at Maximum Warp, SIGPLAN Not., 2011, 46(8), pp. 267-276.
44. Aila T.; Laine S. Understanding the Efficiency of Ray Traversal on GPUs, Proceedings of the Conference on High Performance Graphics, 2009, pp. 145-149.
45. Merrill D.; Garland M.; Grimshaw A. High-Performance and Scalable GPU Graph Traversal, ACM Transactions on Parallel Computing, 2009, pp. 1-30.
46. The Computer Science Bibliography Website, http://dblp.uni-trier.de/ (accessed on 30-11-2016).