Summer 8-1-2019

Distributed Edge Bundling for Large Graphs

Yves Tuyishime
University of Nebraska - Lincoln, yves.tuyishime@huskers.unl.edu

Distributed Edge Bundling for Large Graphs

by

Yves Tuyishime

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Hongfeng Yu

Lincoln, Nebraska

August, 2019

Distributed Edge Bundling for Large Graphs

Yves Tuyishime, M.S.

University of Nebraska, 2019

Adviser: Hongfeng Yu

Graphs or networks are widely used to depict the relationships between data entities in diverse scientific and engineering applications. A direct visualization (such as node-link diagram) of a graph with a large number of nodes and edges often incurs visual clutter. To address this issue, researchers have developed edge bundling algorithms that visually merge similar edges into curved bundles and can effectively reveal high-level edge patterns with reduced visual clutter. Although the existing edge bundling algorithms achieve appealing results, they are mostly designed for a single machine, and thereby the size of a graph they can handle is limited by the available memory of the machine.

To tackle large-scale graphs, a more scalable solution is to carry out edge bundling using multiple machines in a distributed environment. However, the existing edge bundling algorithms typically require the global information structure of a graph. Therefore, with a simple division of the edges of a graph, it is challenging to balance the workload and lower inter-processor communication among the processors. In this work, we select a representative edge bundling algorithm, Force-Directed Edge Bundling (FDEB), and parallelize it in a distributed environment. Particularly, to address the difficulties of partitioning and distributions of a large graph among processors, we first create a high dimensional space to represent the data distribution of a graph in FDEB. Second, we map each edge as a data point in this high dimensional space, and then partition and distribute the point cloud among processors. In this way, we can significantly reduce the data communication across processors, and ensure each processor assigned with a similar workload. We demonstrate the scalability of our distributed algorithm in our experimental study. Although we design the

distributed approach for FDEB, we expect that the parallelization methodology developed in this work can be extended to other edge bundling algorithms as well.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor and mentor Prof. Hongfeng Yu. Without his consistent encouragement and motivations, I wouldn't be able to complete this thesis. Thank you for believing in me and thank you for your time and support through my whole master's program here at UNL. Besides teaching me his enthusiastic ideas of how to approach a complex problem and turn it into many simple tasks to be solved, Prof. Yu, has helped me improve career skills that are beyond academic level by always reminding me to see things in big a picture and I am very thankful for that.

I would like to thank my committee members Prof. Lisong Xu and Prof. Chi Zhang for taking time from their busy schedule to review my work and give me constructive comments.

I would like also to thank Dr. Jieting Wu and Dr. Lina Yu who gave me access to their existing code of edge bundling and rendering part for my thesis. Without their previous work, my thesis would have been much more complex to finish.

Lastly, I would like to thank my parents especially my mum Jacqueline, my siblings, and my friends for their unconditional love and support through my whole life. Thank you for being there for me all the time.

I am grateful for the support provided by the National Science Foundation through grants IIS-1423487 and IIS-1652846, and the Department of Computer Science and Engineering at the University of Nebraska-Lincoln to make this research possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs or networks are widely used to depict the relationships between data entities in diverse scientific and engineering applications. Conventionally, graphs are visualized as node-link diagrams where vertices or nodes represent the entities while the edges or lines show the relationship between the two entities paired together. Nodes can be represented in different forms such as boxes, disks, or textual labels, while the edges are commonly represented as line segments/poly-lines, or curves in the Euclidean plane [1]. A social network is a typical example of a node-link diagram where nodes represent individual persons and edges represent the friendships among them. That is, in a social network diagram, two persons are connected together if either they know each other, or they have at least a mutual friend in common. Another node-link diagram example is a software system in which nodes represent source-code elements while the edges represent dependency relations. In case of traffic networks, nodes depict locations while edges represent the amount of traffic between the two locations [2]. However, though node-link diagrams provide an effective way to represent graphs, visualization of dense graphs consisting of a large number of nodes and edges suffers from visual clutter because the visual readability of a node-link diagram can be quickly degraded as the number of nodes and edges increase.

To address the issue of visual clutter for node-link diagrams, researchers have devel-

oped *edge bundling* algorithms for different types of graphs such as hierarchical graphs [3] and general graphs [2]. Edge bundling algorithms visually merge similar edges into curved bundles and can effectively reveal high-level edge patterns with reduced visual clutter. Although edge bundling algorithms could achieve appealing results, they are typically characterized with high computational complexities and slow speeds for large graphs. For example, to create a general algorithm applicable for all types of graphs, Holten et al. developed a Force-Directed Edge Bundling (FDEB) algorithm in which edges are considered as flexible springs that can attract each other while their node positions stay the same, and then a force-directed technique is used to calculate the bundling [2]. However, because the algorithm goes through every pair of edges, its complexity is $\mathcal{O}(n^2)$ where n is the number of edges of a graph. To speed up edge bundling algorithms, the existing efforts have mostly focused on GPU acceleration techniques to achieve interactive frame rates. However, the size of a graph that can be handled is constrained by the available memory of a single machine, and thereby the scalability is limited.

A more scalable solution is to carry out edge bundling using several machines in a distributed environment. However, the existing edge bundling algorithms require the global structure of a graph. Therefore, with a simple division of edges in a graph, it is difficult to partition and distribute a balanced workload of a large graph and lower inter-processor communication among the processors. A naive solution would incur extensive data exchange and communication among processors, which results in poor scalability.

In this work, we select the representative edge bundling algorithm, FDEB, and parallelize it in a distributed environment. Particularly, to address the difficulties of partitioning and distribution of a large graph among processors, we first create a high dimensional space to represent the data distribution of a graph in FDEB. Second, we map each edge as a data point in this high dimensional space, and then partition and distribute the point cloud among processors. In this way, we can significantly reduce the data communication across processors, and ensure each processor assigned with a similar workload. Our experimental

results demonstrate the high performance of our distributed algorithm. Overall, our method is simple and easy to understand and implement. Although we design the distributed approach for FDEB, we expect that the parallelization methodology developed in this work can be extended to other edge bundling algorithms as well.

Chapter 2

Related Work

We will first give background information of node-link diagrams and present an overview of commonly used graph visualization techniques. We will then review the general methods for reducing visual clutter of graph visualization, and describe the existing edge bundling methods.

2.1 Node-link Diagram

Node-link diagram is the most intuitive graph visualization compared to other graph visual representations, such as adjacent matrix and adjacent list [1, 4, 5]. A typical node-link diagram shows how things or entities are interconnected through the use of nodes/vertices and links/edges to represent their connections as one simple network shown in Figure 2.1. This representation helps users understand the relationships between a group of entities, calculate different paths between entities, and estimate the total cost for moving from one entity to another in the case of weighted graphs. However, with a larger number of edges, a node-link diagram can become increasingly clutter due to the heavy edge overlapping and crossings, which can significantly degrade its readability for large graphs.

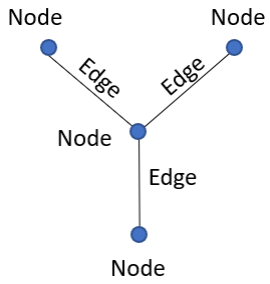


Figure 2.1: A simple node-link diagram example.

2.2 General Visual Clutter Reduction Techniques

Various techniques have been developed for the reduction of edge clutter. One way is to visualize graphs in a clustered way. Clustered graphs contain a hierarchical component in the form of a recursive clustering structure as well as non-hierarchical connections between the nodes of the clusters. For a clustered graph, the number of edges between clusters of nodes is significantly reduced compared to the number of the individual edges between the original nodes in visualization results [6–8]. Although these methods can reduce visual edge clutter, the existence of hierarchy is required and many low-level edges are fully merged into inter-cluster edges, which makes it hard to discern individual edges.

To address the problem of visual clutter for large node-link diagrams, researchers have proposed to use matrix-based representations. A matrix-based graph representation, also known as adjacency matrix, is a 2D array of size $V \times V$ where V is the number of vertices in the graph. Typically, if there is an edge between two vertices, the corresponding entry of the matrix is set to 1 or the edge weight. Van Ham suggests that if the main visualization goal is to convey edges and not nodes, matrix representations have more advantages over traditional node-link diagrams [9]. In addition, matrix-based visualizations provide stable and crisp layouts over node-link diagrams, especially for large graphs [9, 10]. Matrix representations are inherently well suited for large multilevel visualizations because of their recursive structures. Although matrix visualizations offer a clean and uncluttered layout, they are less intuitive than node-link diagrams. Therefore, their representations may not be

used as a completely effective alternative for node-link diagrams.

An alternative approach for solving the problem of visual clutter for large node-link diagrams is to use rendering and interaction techniques. It is possible to generate a node-link diagram at high resolutions while applying anti-aliasing to reduce staircase effects and draw graphs with smoother edges. We can also use alpha blending to control the transparency of an image by increasing the visibility of individual edges in areas with high edge density. Previous work [11] has shown that the edges close to one another can overlap due to high edge density. This phenomenon is known as edge congestion. To address this problem, it is possible to change graph layouts (i.e., node positions). However, this is not suitable for graphs with fixed node positions, such as airlines connections, geographic maps, and telecommunication networks. To address this issue, interaction methods can provide a user with the ability to explore certain part of the graph. For example, Wong et al. [11] propose an interactive method, named EdgeLens, to push away edges located in the user's focus of attention while preserving the locations of nodes. Furthermore, Wong et al. [12] propose Edge Plucking to pull edges from one center of attention but without changing the node positions. In other words, Edge Plucking does exactly the opposite of EdgeLens. However, in case of non-interactive graph visualization, the use of EdgeLens and Edge Plucking is not an option.

2.3 Visual Clutter Reduction using Edge Bundling

Recently, edge bundling techniques have attracted many research attentions for visual clutter reduction. Holten first proposes the use of Hierarchical Edge Bundling (HEB) for graphs that contain hierarchy [3]. This technique is based on the idea of tying/combining adjacent and close edges together in the same way we can combine electrical wires or other network cables into bundles and separate them again at the end. However, this method is not applicable for graphs that do not contain hierarchy. Later, Holten et al. [2] extend HEB and

propose Force Directed Edge Bundling (FDEB). This is an edge bundling method that uses a self-organizing approach and models edges as flexible springs that can attract each other. In contrast to HEB, neither hierarchy nor control mesh is used and the resulting bundled graphs show a significant visual clutter reduction and they are clearly visible with high-level edge patterns. It also takes care of curvature variation by minimizing it to produce smooth bundles that are easy to follow. Cui et al. propose a geometry-based edge-clustering framework which groups edges together to increase the visibility of the graph by minimizing the overall edge crossings [13]. This method relies on the use of a control mesh to guide the edge-clustering process, and then edges are forced to pass through some control points on the mesh in order to make bundles. Since this method relies on the generation of a control mesh to reduce edge clutter, the control mesh also produces bundles with considerable curvature-variation which are hard to follow. Gansner et al. [14] leverage agglomerative clustering to enhance edge bundling effects for large graphs. Selassie et al. [15] extract graph topology to improve edge bundling results.

Although edge bundling algorithms can generate visually appearing results to convey high-level graph patterns, they are typically characterized with high computational complexities and can take several to hundreds of seconds to generate bundles for large graphs. Therefore, they are not suitable for large graphs, and particular for interactive applications. To address this issue, only a few attempts have been perceived. Zhu et al. [16] propose a parallelized FDEB on the GPU (GPUFDEB) that reforms FDEB and achieves a balanced partitioning of data and calculation to suit computation on GPU. Wu et al. [17] use GPU textures to encode a graph and use GPU to carry out force-directed edge bundling. Their method makes it possible to interactively visualize graphs on web-based platforms. However, these solutions mostly leverage multithreading techniques and require the data of graphs and intermitted calculation results to be held in the GPU memory, which makes it difficult to tackle graphs that are not held in a single machine.

A more feasible and scalable way to visualize large graphs is to use multiple machines

in a distributed environment. However, it is non-trivial to partition and distribute a graph among processors for edge bundling. In the next chapter, we will revisit the representative edge bundling algorithm, FDEB. Moreover, we will illustrate the challenges in parallelizing FDEB in a distributed environment. In Chapter 4, we will propose our solution to speed up FDEB by splitting and balancing the workload among multiple processors and lowering inter-processor communication.

Chapter 3

Background

We aim to develop a distributed algorithm for FDEB [2] and make it scalable for large graphs. Figure 3.1 shows the framework of the original FDEB algorithm [2]. The key computational step of FDEB is force-directed edge bundling that converts the edges of a graph into a set of smooth curved bundles. However, for a large graph, this step requires a high computational cost. To reduce the number of edges to be bundled together, FDEB uses a preprocessing step named edge compatibility measures for accelerating the bundling step. We will revisit these two steps and illustrate the challenges in developing a distributed algorithm for FDEB.

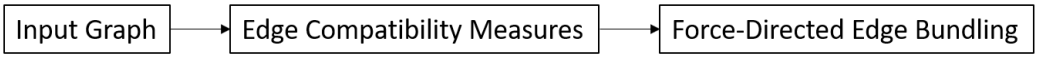


Figure 3.1: The FDEB framework.

3.1 Force-Directed Edge Bundling

The initial input for FDEB is a straight-line node-link diagram of a general graph. The graph can represent different entity relationships. For example, in the case of geographic information such as traffic between locations, the node positions are determined by geographic coordinates. The next step after acquiring the input is to perform edge bundling in

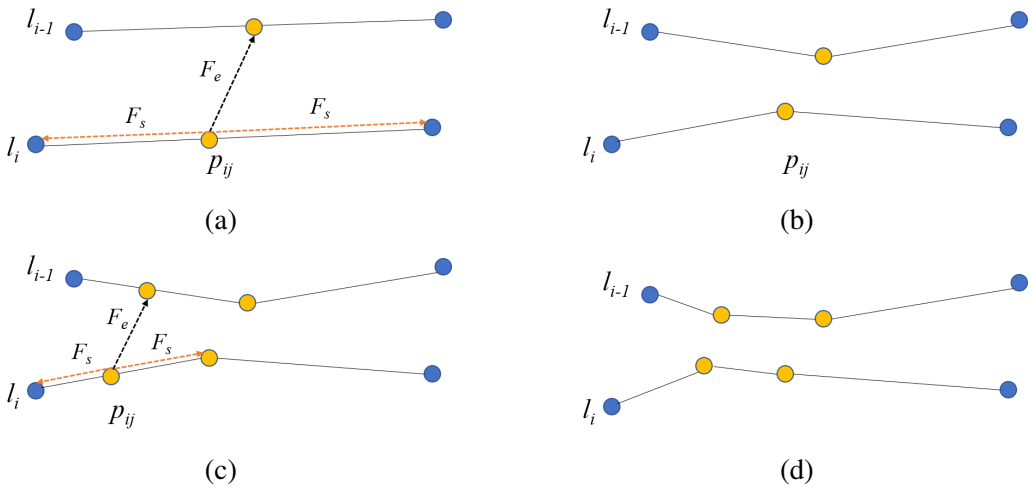


Figure 3.2: Edge subdivision and subdivision point movement in FDEB.

iterative simulations. Specifically, there are C numbers of simulation cycles, and to enable straight-line edges to change shape during bundling, for each simulation cycle, we subdivide each edge into smaller segments and then iteratively move each subdivision point to a new position after computing forces among the points.

Let C_0 be the first simulation cycle and P_0 be the initial number of subdivision points in this simulation. As shown in Figure 3.2 (a), FDEB starts with a single subdivision point (P_0 =1 and shown in orange color) and two endpoints (shown in blue color) for each edge. Then, for each edge, a linear attracting spring force F_s is used between each pair of consecutive subdivision points. For instance, in Figure 3.2 (a), at a subdivision point p_{ij} where p_{ij} is the j^{th} point on an edge l_i, we model and compute a spring force F_s as follow:

$$F_s = k_p(||p_{i(j-1)} - p_{ij}|| + ||p_{ij} - p_{i(j+1)}||) \tag{3.1}$$

where k_p is a spring constant, and $p_{i(j-1)}$ and $p_{i(j+1)}$ are the neighboring points of p_{ij} on the edge l_i. In addition, an attracting electrostatic force F_e is used between each pair of

corresponding subdivision points of a pair of interacting edges. F_e is defined as:

$$F_e = \sum_{m \in E} \frac{1}{||p_{ij} - p_{mj}||} \tag{3.2}$$

where E is a set of edges where each edge l_m interacts with l_i and p_{mj} is the corresponding subdivision point on such an interacting edge l_m, as shown in Figure 3.2 (a). During each calculation cycle of the iterative simulation, after computing both forces, the combined force $F_{p_{ij}}$ exerted on each subdivision point of each edge is calculated as the sum of the two forces. $F_{p_{ij}}$ is defined as:

$$F_{p_{ij}} = Fs + Fe \tag{3.3}$$

The position of p_{ij} is updated by moving it a small distance in the direction of $F_{p_{ij}}$. As this is an iterative process, F_e, F_s, and $F_{p_{ij}}$ are also updated according to the new position of p_{ij}. The simulation C_0 starts with an initial number of subdivision points $P_0 = 1$ for each edge and an initial step size S_0. The step size S determines the distance a point is moved at each iteration step in the direction of the combined force $F_{p_{ij}}$ that is exerted on it [2]. Furthermore, a specific number of iteration steps, I, is conducted to move the subdivision points to reach an equilibrium between forces as shown in Figure 3.2 (b).

The edge subdivision and the subdivision point movement are continued in the next simulation cycles until the simulation number is exhausted. Figure 3.2 (c) and (d) show the edge subdivision, subdivision point movement, and the equilibrium state of C_1.

The number of iteration steps during the first cycle is I_0. After performing a cycle, from the figure above, it is clear that the number of subdivision points P is doubled, while the step size S is halved before initiating the next cycle, and the number of iteration steps I per cycle is decreased by a factor R. Holten et al. [2] reported that assignments of $P_0 = 1$, $C = 6$, $I_0 = 50$, and $R = 2/3$ provide appropriate results.

3.2 Edge Compatibility Measures

We can see that the complexity of force-directed edge bundling in each simulation cycle depends on the size of E in Equation 3.2. In the worst case, the size of E can be the total edge number n when every edge interacts with all the other edges, leading to a complexity of $\mathcal{O}(n^2)$ for processing all edges in each iteration of one simulation cycle. Given C simulation cycles and I iteration steps per cycle, the total complexity is $C \cdot I \cdot \mathcal{O}(n^2)$

However, in practice, it is rare that an edge interacts with all the other edges. To avoid a high computation cost, the size of E is controlled by four edge compatibility measures, angle, scale, position, and visibility. That is, only the edges with the compatibility measures greater than a threshold are considered as interacting edges. Thus, the original FDEB algorithm involves a preprocessing step for generating edge comparability measures.

Angle compatibility states that edges that are almost perpendicular should not be bundled together. FDEB calculates the angle α between two edges l_i and l_j, and the value of angle compatibility $C_\alpha(l_i, l_j) \in [0,1]$ as

$$C_\alpha(l_i, l_j) = |cos(\alpha)| \tag{3.4}$$

$C_\alpha(l_i, l_j)$ is close to zero if both edges are almost perpendicular and it is equal to 1 if both edges are parallel.

Scale compatibility states that edges that differ considerably in length should not be bundled together either, as doing so might result in noticeable stretching and curving of short edges to accommodate to the shape of long edges. The scale compatibility $C_s(l_i, l_j)$ for two edges l_i and l_j as

$$C_s(l_i, l_j) = \frac{2}{l_{avg} \cdot min(|l_i|, |l_j|) + max(|l_i|, |l_j|)/l_{avg}}, \tag{3.5}$$

where $l_{avg} = (|l_i| + |l_j|)/2$. Similarly, the value of scale compatibility is 1 if two edges

have equal length and approaches 0 if the ratio between the longest and the shortest edge approaches ∞.

Furthermore, position compatibility states that edges that are far apart should not be bundled together either. The position compatibility $C_p(l_i, l_j) \in [0,1]$ is defined as

$$C_p(l_i, l_j) = \frac{l_{avg}}{(l_{avg} + \|l_{i_m} - l_{j_m}\|)}, \tag{3.6}$$

where l_{i_m} and l_{j_m} are the midpoints of two edges l_i and l_j. That is, the position compatibility is equals to 1 if the middle points of two edges coincide, and it is equal to zero if the distance between them approaches ∞.

Lastly, visibility compatibility $C_v(l_i, l_j)$ states that it is possible that edges are parallel, equal in length, and close together, but should nevertheless have a fairly low bundling-compatibility. The total edge compatibility $C_e(l_i, l_j) \in [0,1]$ between two edges l_i and l_j is defined as

$$C_e(l_i, l_j) = C_\alpha(l_i, l_j) \cdot C_s(l_i, l_j) \cdot C_p(l_i, l_j) \cdot C_v(l_i, l_j). \tag{3.7}$$

If $C_e(l_i, l_j)$ is greater than a user-defined threshold r, two edge l_i and l_j are considered as interacting edges for edge bundling calculations.

The usage of edge compatibility $C_e(l_i, l_j)$ can significantly improve the performance without compromising the edge bundling results. For example, even for $r = 0.05$, 50% - 75% of the total edge pairs can be dropped for edge bundling calculations.

3.3 Challenges for Distributed Edge Bundling

It remains a critical demand to develop distributed edge bundling algorithms for large graphs mainly because of two reasons.

First, although edge compatibility measures can significantly reduce the number of pair-wise edge interactions, the complexity of force-directed edge bundling remains $C \cdot I \cdot \mathcal{O}(n^2)$.

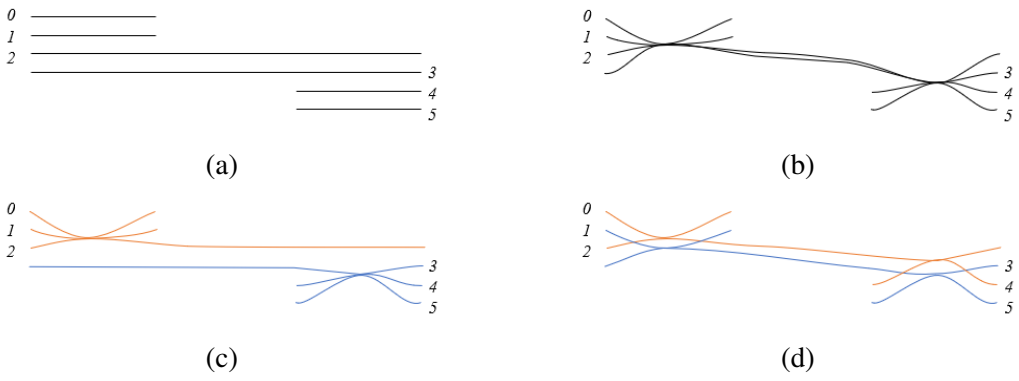


Figure 3.3: Challenges for distributed edge bundling. Given a simple graph in (a), (b) shows the correct edge bundling results. We partition and distribute the graph among two processors in different ways as shown in (c) and (d). If each processor only considers its local data to generate partial edge bundling results, the final aggregated results in (c) and (d) are not correct.

In addition, the step of edge compatibility measures requires to compute the compatibility values between every pair of edges, and thus its complexity is $\mathcal{O}(n^2)$ for n edges. A feasible way is to use multiple computing units to accelerate the computation of both edge compatibility measures and force-directed edge bundling.

Second, although multithreading based techniques or devices (e.g., GPUs) are effective for performance acceleration, their capability is limited by their available memory size. It is difficult to use a single machine to hold large graphs in a scalable fashion.

Therefore, one feasible way is to use multiple machines or processors to process large graphs in a distributed manner. However, it is challenging to partition and distribute a graph among distributed processors due to the entire graph information needed in the step of edge compatibility measures, which can be illustrated using a simple example in Figure 3.3.

Given an input graph in Figure 3.3 (a), we can clearly see that the edges l_0 - l_3 are the interacting edges on the left side, and the edges l_2 - l_5 are the interacting edges on the right side. Figure 3.3 (b) is the corresponding edge bundling results.

Assume that we partition and distribute the edges among two processors using different strategies. In Figure 3.3 (c), we divide the edges according to their indexes, such that the

edges l_0 - l_2 (colored in orange) are assigned to one processor, and l_3 - l_5 (colored in blue) are assigned to another processor. If each processor only bundles its local edges, the global edge bundling result is not accurate. As shown in Figure 3.3 (c), l_3 is not bundled with l_0 - l_2 on the left side, as they are on different processors. Similarly, l_2 is not bundled correctly with l_3 - l_5 on the right side. In Figure 3.3 (d), we distribute the edges in a round-robin fashion, and see more bundles are generated due to a lack of global information at each processor.

To address this issue, one intuition is to exchange the edge information among the processors. However, given the original FDEB algorithm, each processor needs to go through all the edges to determine the interacting edges with its local edges using edge compatibility measures. For example, in Figure 3.3 (c), the processor assigned with l_0 - l_2 needs to scan all the edges to determine that l_3 is the interacting edges with l_0 - l_2, but l_4 and l_5 are not. This implies that the entire graph needs to be duplicated on each processor, which is not feasible for large graphs.

Chapter 4

Distributed Edge Bundling

It is non-trivial to partition and distribute a graph among processors to generate its correct edge bundling result. We characterize the FDEB algorithm and tackle the challenges by introducing a novel method for graph partitioning and distribution.

4.1 Basic Idea

The original FDEB algorithm needs to scan each pair of edges to determine their compatibility values. We find that it is not necessary to apply this pairwise operation for all edges if we have a distance measure for all edges.

Let us consider angle compatibility first. Given an edge l with its start point p and its endpoint q, we can easily gain its normalized vector $v = (q-p)/||q-p||$, which can be mapped to a point on a unit circle. In this way, the angle compatibility between any two edges can be estimated as the arc length between two corresponding points on the unit circle in a 2D space. Figure 4.1 shows a simple example with five edges. We can see that the angles of l_2, l_5, and l_4 are close to zero, and thereby their corresponding points are close to the x axis of the 2D space. Meanwhile, the points of l_1 and l_3 are close to each other, but are relatively far away for the other three points in the 2D space. Given this 2D representation, we can easily compute the distance between any pair of points.

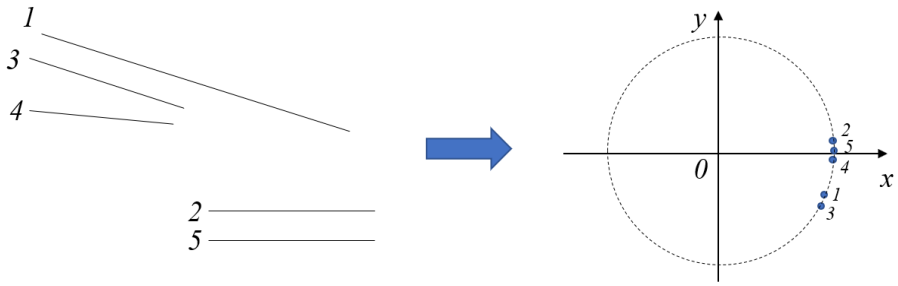


Figure 4.1: Map the edges into a 2D space according to their angles.

More importantly, it facilitates us to partition the points according to their distribution. For example, Figure 4.1 clearly shows two distinct groups with respect to their angles.

4.2 High-Dimensional Representation

We can extend this idea to create a space for each of the edge compatibility measure. For scale compatibility, we can create a 1D space and map each edge as a point in this 1D space according to its scale or length. Figure 4.2 shows the mapping of the simple example with five edges. We can see that the scale of l_1 is considerably larger than the other four edges. We can easily partition the points into two groups according to their scales in this 1D space.

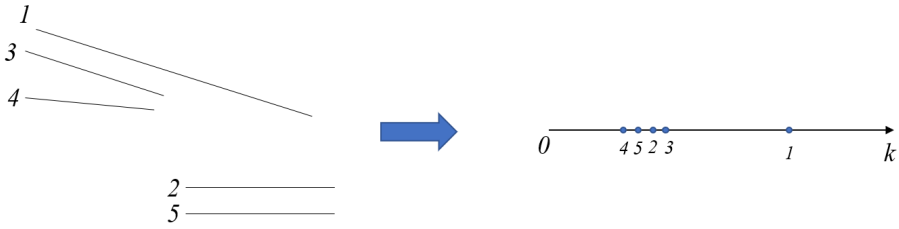


Figure 4.2: Map the edges into a 1D space according to their scales.

For position compatibility, we create a 2D space and map each edge as a point in this 2D space according to the positions of its middle point. Figure 4.3 shows the results of the simple example. In this space, we can easily measure the distance with respect to the positions between any two edges.

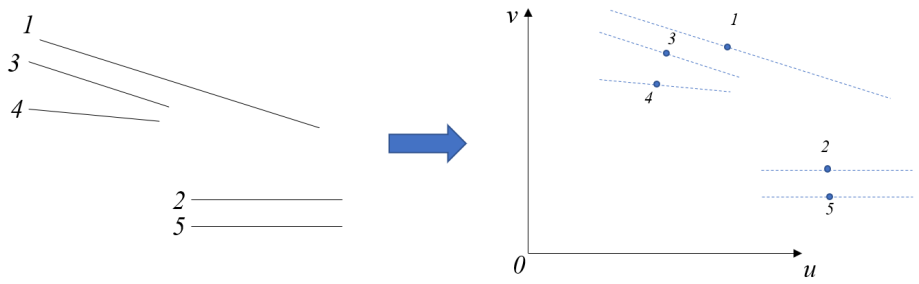


Figure 4.3: Map the edges into a 2D space according to their positions.

The total compatibility of two edges is the multiplication of the individual compatibility measure. This inspires us that we can create a high-dimensional space by combining the point distribution in the space of each compatibility measure. We can then map each edge as a data point in this high-dimensional space, and measure their distance as their total compatibility among any pair of edges. Given the space of each compatibility measure, we can obtain a 5D space. It is difficult to directly visualize this 5D space. We illustrate it using the 3D space combining the 2D angle space and the 1D scale space. As shown in Figure 4.4, by only considering the angle and the scale, we can see that l_1 is noticeably different from the other edges in this 3D space. If we further consider the 2D space of positions, we obtain a 5D space where the distance between l_2 and l_5 will remain small, and the distance among l_1, l_3, and l_4 will become smaller.

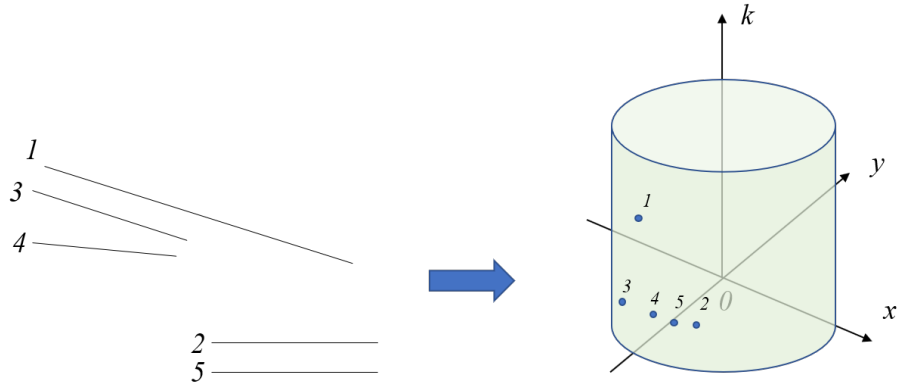


Figure 4.4: Map the edges into a 3D space combining the angle and scale spaces.

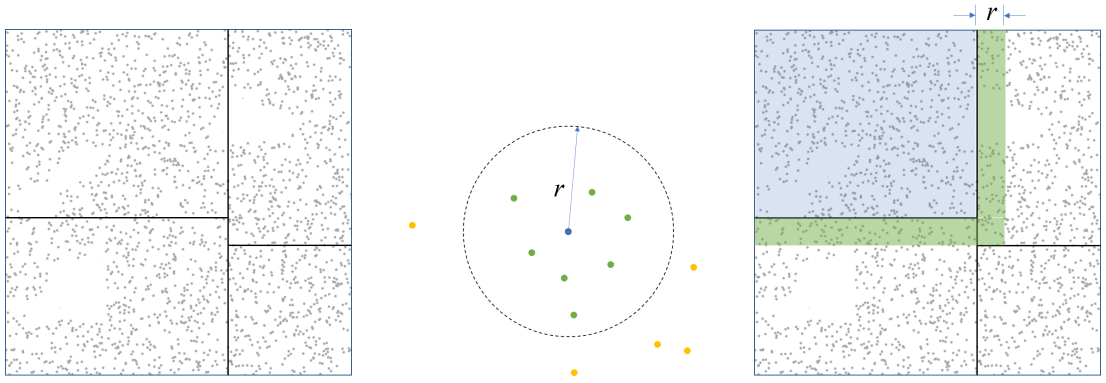


Figure 4.5: High-dimensional partitioning.

4.3 High-Dimensional Partitioning

By mapping the edges into the high-dimensional space, we can generate a high-dimensional point cloud whose distribution along each dimensional represents the distribution with respect to one compatibility measure. An intuitive idea is to partition the point cloud and distribute them among processors. Figure 4.5 (a) shows a 2D point cloud that is partitioned into four blocks using a K-D tree. We assume that the distance d between an interior point p of a block and the boundary of the block is larger than the compatibility threshold r. In this way, we can ensure that the point p cannot interact with any point in other blocks. This is because the distance between p and any point in other blocks must be larger than r.

However, we cannot ensure this property for a boundary point of a block. As shown in Figure 4.5 (b), a point interacts with other points within a radius of r. Therefore, it is possible that a boundary point can have its interact point in a foreign block. In this way, the points in the foreign block need to be visited when computing force-directed edge bundling, incurring data communication.

To address this issue, we enlarge each partition with a ghost area. As shown in Figure 4.5 (c), for a partition (colored in blue), we also include the points in an area (colored in green) surrounding the partition. The width of the green area is r. The points within

the blue and green regions are assigned to one processor. We note that the processor only bundles the edges corresponding to the points in the blue region. However, as we also provide the points in the ghost area with the processor, we can ensure that the processor does not need to require any points from the other processors, and thus ensure to minimize the communication cost among processors.

Note that we did not handle visibility compatibility in the current work due to the difficulty in designing its high-dimensional representation. However, this will not affect the accuracy of our data partitioning. This is because by introducing additional dimensions, we can gain finer partitioning results. Thus, the current partitioning can be regarded as a coarser result with a larger bound, and does not compromise the distance measure with respect to visibility compatibility.

4.4 Distributed FDEB

Given our high-dimensional representation and its associated partitioning method, we can carry out FDEB in a distributed environment by increasing the performance and avoiding data duplication among the processors. Figure 4.6 shows the framework of our distributed FDEB algorithm.

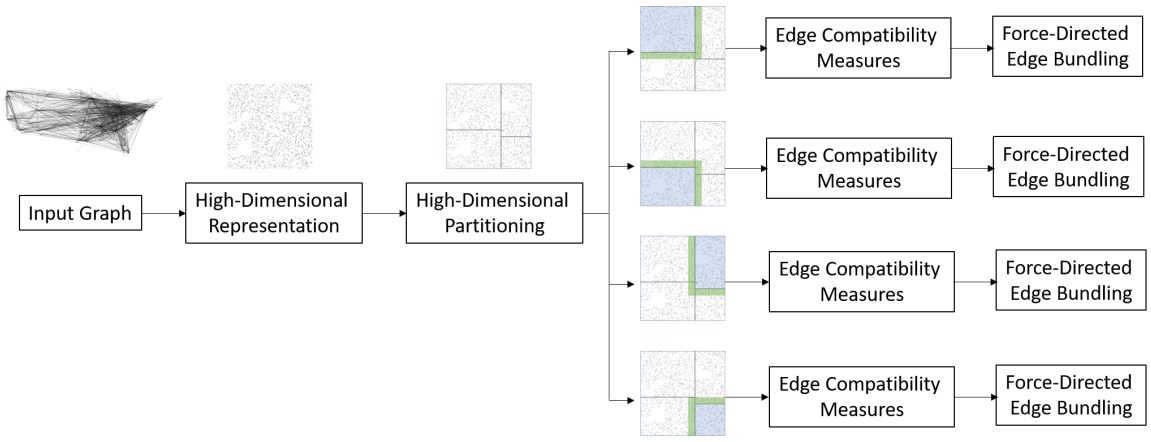


Figure 4.6: Our distributed FDEB framework.

First, given an input graph, we first map each edge into a point in our 5D space where each subspace encodes the data distribution with respect to each individual compatibility measure. The distance between any two points in this high-dimensional space is proportional to the compatibility measure between the two corresponding edges.

Second, we partition the point cloud using the K-D tree that can ensure that each partition has a similar number of points. The number of partitions is equal to the number of processors. For simplicity, we assume that the number of processors is an exact power-of-two. In this case, if we assign the points within a partition to a processor, the processors can have a similar number of edges to be bundled. To avoid data exchange among the processors, we create a ghost area for each partition. The width of the ghost area is equal to the compatibility threshold r. The points with the ghost area are also assigned to the processor. In this way, the processor can always find the corresponding interacting edges of its local data without fetching data from any remote processors, thus minimizing the data communication cost.

Third, each processor computes edge compatibility measures for its local data. With our high-dimensional partitioning, we ensure that a processor can find the pairwise interacting edges within its local data.

Fourth, each processor conducts force-directed edge bundling based its local edge compatibility measures. We note that each processor does not bundle the edges within the ghost area, but use these edges to bundle its local edges. The edges within the ghost area are bundled in remote processors. In this way, although the edges in the ghost area may be different, the total number of local bundled edges is approximately equal for each processor. This ensures the balanced workload among the processors.

Finally, the partial edge bundling results generated at each processor are then aggregated. This is simply implemented using distributed gather operations (e.g., MPI_Gather).

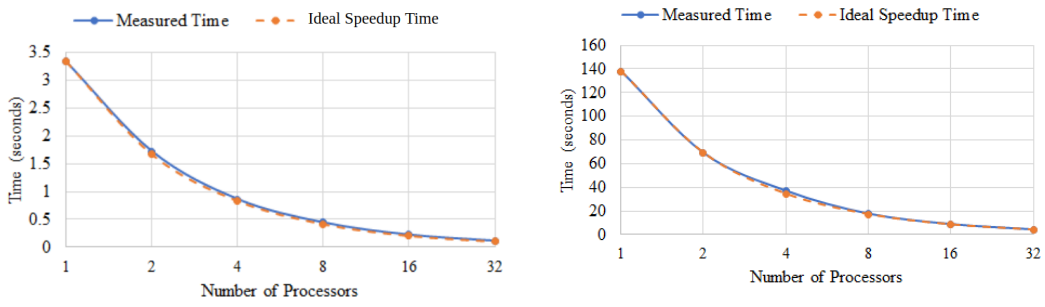
Our distributed FDEB can achieve a balanced workload, and minimize data communication, leading to scalable performance for bundling large graphs.

Chapter 5

Results

We have evaluated the performance of our distributed FDEB. We implemented our algorithm using MPI and C++. We experiment with our algorithm on Crane, a supercomputer operated by the Holland Computing Center at the University of Nebraska-Lincoln. Each node of the Crane partition used in our experiments has two Intel Xeon E5-2670 2.60GHz CPUs with 16 cores per node. We used a US airline graph (2100 edges) and a US migration graph (9780 edges) to design and conduct our scalability experiment.

Figure 5.1 shows the performance of distributed FDEB carried out on the processors.



(a) (b)

Figure 5.1: Speedup of our scalability experiment to bundle (a) the US airline graph and (b) the US migration graph. In each plot, the horizontal axis presents the number of processors, and the vertical axis represents the running time in seconds. In each plot, we show the measured time and ideal speedup time.

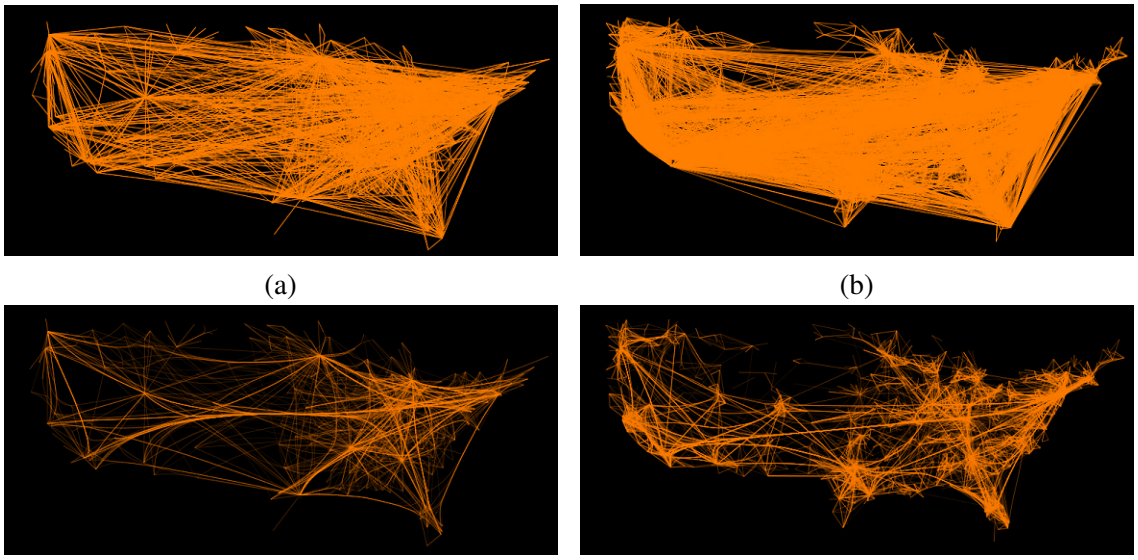


Figure 5.2: Visualization results. (a) and (b) show the node-link diagrams of the US airline graph and the US migration graph, respectively. (c) and (d) show the edge bundling results of the US airline graph and the US migration graph, respectively.

Each processor independently calculates edge compatibility measures and conducts force-directed edge bundling for its assigned local edges. Thanks to our high-dimensional representation and partitioning approach, there are no communications required among the processors, and each processor has a balanced workload. Figure 5.1 conveys that our distributed algorithm achieves almost ideal speedup, and the parallel efficiencies are 94% and 97.6% (32 processors vs. 1 processor) for the US airline graph and the US migration graph, respectively.

We also measure the performance results for generating high-dimensional representation and high-dimensional K-D tree partitioning. As shown in Table 5.1, it took about 0.06 seconds to map the edges of the US airline graph to a point cloud, and 0.06 seconds to partition the point cloud. The corresponding times of the two steps are 1.91 seconds and 0.14 seconds of the US migration graph. Compared to the performance gain in Figure 5.1, the cost of these steps is relatively neglectable.

Figure 5.2 shows the visualization results of the US airline graph and the US migration

Table 5.1: Performance results for generating high-dimensional representation and high-dimensional partitioning

Dataset	Representation Generation Time (seconds)	Partitioning Time (seconds)
US Airline	0.06	0.06
US Migration	1.91	0.14

graph. It is hard to perceive clear patterns from the node-link diagrams of these two datasets in Figure 5.2 (a) and (b) due to the visual clutter problem. Figure 5.2 (c) and (d) show the edge bundling results of these two datasets, which significantly reduce the visual clutter by bundling similar edges and revealing high-level patterns. These results are close to the ones generated by the original FDEB on a single machine. However, our method can generate these results in a more scalable fashion.

Chapter 6

Conclusion

We have presented a distributed algorithm for conducting FDEB using multiple processors. Our algorithm first transforms the edges of a graph into a high-dimensional point cloud representation, and then partition and distribute the points among the processors in the high-dimensional space. Our new data representation and partitioning scheme makes it possible to balance workload and minimize communication among processors. Our performance evaluation shows our distributed FDEB algorithm achieves nearly ideal speedup. In addition, the cost for generating high-dimensional representation and partitioning is comparably neglectable. Our solution has significantly improved the scalability of FDEB.

In the current stage of this work, we show the effectiveness of our solution with the preliminary results using the US airline graph and the US migration graph. The sizes of these two graphs are comparably marginal. We will use larger graphs in our next development and experimental studies to gain a deeper understanding of the impact of data partitioning and distribution schemes on conducting edge bundling in a distributed environment. In the future, we also would like to incorporate visibility comparability in our framework. We plan to extend our method on time-varying graphs to reduce the high-dimensional representation generation and partitioning cost for each time step. In addition, although we design the distributed approach for FDEB, we expect that the parallelization methodology

developed in this work can be possibly extended to other edge bundling algorithms as well.

References

[1] I. Herman, G. Melancon, and M. S. Marshall, "Graph visualization and navigation in information visualization: A survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, pp. 24–43, Jan. 2000.

[2] D. Holten and J. J. Van Wijk, "Force-directed edge bundling for graph visualization," *Computer graphics forum*, vol. 28, no. 3, pp. 983–990, 2009.

[3] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on visualization and computer graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[4] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. van Wijk, J.-D. Fekete, and D. Fellner, "Visual analysis of large graphs: State-of-the-art and future research challenges," *Computer Graphics Forum*, vol. 30, pp. 1719–1749, Sept. 2011.

[5] C. Vehlow, F. Beck, and D. Weiskopf, "The state of the art in visualizing group structures in graphs.," in *EuroVis (STARs)*, pp. 21–40, 2015.

[6] M. Kaufmann and D. Wagner, *Drawing Graphs: Methods and Models*. Springer, June 2003.

[7] Q. Feng, *Algorithms for drawing clustered graphs*. Citeseer, 1997.

[8] P. Eades, Q. Feng, X. Lin, and H. Nagamochi, "Straight-line drawing algorithms for hierarchical graphs and clustered graphs," *Algorithmica*, vol. 44, no. 1, pp. 1–32, 2006.

[9] F. Van Ham, "Using multilevel call matrices in large software projects," in *IEEE Symposium on Information Visualization 2003*, pp. 227–232, IEEE, 2003.

[10] M. Ghoniem, J.-D. Fekete, and P. Castagliola, "A comparison of the readability of graphs using node-link and matrix-based representations," in *IEEE Symposium on Information Visualization 2004*, pp. 17–24, IEEE, 2004.

[11] N. Wong, S. Carpendale, and S. Greenberg, "Edgelens: An interactive method for managing edge congestion in graphs," in *IEEE Symposium on Information Visualization 2003*, pp. 51–58, IEEE, 2003.

[12] N. Wong and S. Carpendale, "Supporting interactive graph exploration using edge plucking," in *Visualization and Data Analysis 2007*, vol. 6495, p. 649508, International Society for Optics and Photonics, 2007.

[13] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li, "Geometry-based edge clustering for graph visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1277–1284, 2008.

[14] E. R. Gansner, Y. Hu, S. North, and C. Scheidegger, "Multilevel agglomerative edge bundling for visualizing large graphs," in *2011 IEEE Pacific Visualization Symposium*, pp. 187–194, IEEE, 2011.

[15] D. Selassie, B. Heller, and J. Heer, "Divided edge bundling for directional network data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2354–2363, 2011.

[16] D. Zhu, K. Wu, D. Guo, and Y. Chen, "Parallelized Force-Directed Edge Bundling on the GPU," in *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science*, pp. 52–56, Oct. 2012.

[17] J. Wu, L. Yu, and H. Yu, "Texture-based edge bundling: A web-based approach for interactively visualizing large graphs," in *2015 IEEE International Conference on Big Data (Big Data)*, pp. 2501–2508, Oct. 2015.