



Universiteit
Leiden
The Netherlands

Real-time interactive visualization of large networks on a tiled display system

Brinkmann, G.G.; Rietveld, K.F.D.; Verbeek, F.J.; Takes, F.W.

Citation

Brinkmann, G. G., Rietveld, K. F. D., Verbeek, F. J., & Takes, F. W. (2022). Real-time interactive visualization of large networks on a tiled display system. *Displays*, 73. doi:10.1016/j.displa.2022.102164

Version: Publisher's Version

License: [Creative Commons CC BY 4.0 license](https://creativecommons.org/licenses/by/4.0/)

Downloaded from: <https://hdl.handle.net/1887/3641638>

Note: To cite this publication please use the final published version (if applicable).



Real-time interactive visualization of large networks on a tiled display system[☆]

G.G. Brinkmann, K.F.D. Rietveld, F.J. Verbeek, F.W. Takes^{*}

Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands

ARTICLE INFO

MSC:

65Y20

05C85

52C30

Keywords:

Network visualization

Tiled display systems

Interactive visualization

GPU

CUDA

ABSTRACT

This paper introduces a methodology for visualizing large real-world (social) network data on a high-resolution tiled display system. Advances in network drawing algorithms enabled real-time visualization and interactive exploration of large real-world networks. However, visualization on a typical desktop monitor remains challenging due to the limited amount of screen space and ever increasing size of real-world datasets.

To solve this problem, we propose an integrated approach that employs state-of-the-art network visualization algorithms on a tiled display system consisting of multiple screens. Key to our approach is to use the machine's graphics processing units (GPUs) to their fullest extent, in order to ensure an interactive setting with real-time visualization. To realize this, we extended a recent GPU-based implementation of a force-directed graph layout algorithm to multiple GPUs and combined this with a distributed rendering approach in which each graphics card in the tiled display system renders precisely the part of the network to be displayed on the monitors attached to it.

Our evaluation of the approach on a 12-screen 25 megapixels tiled display system with three GPUs, demonstrates interactive performance at 60 frames per second for real-world networks with tens of thousands of nodes and edges. This constitutes a performance improvement of approximately 4 times over a single GPU implementation. All the software developed to implement our tiled visualization approach, including the multi-GPU network layout, rendering, display and interaction components, are made available as open-source software.

1. Introduction

In this paper we propose a framework for real-time interactive network visualization on a tiled display system, using its graphics processing units (GPUs). The framework is implemented on a 25 megapixel tiled display system, composed of twelve monitors connected to a single off-the-shelf desktop machine (see Fig. 3). We test the approach on a number of real-world datasets consisting of, e.g., networks representing social media interaction, protein interaction networks and scientific collaboration networks.

A network (or graph) describes relationships between a set of entities. We consider the most common visualization method, in which the network's entities (nodes) are drawn as dots in the plane, with lines (edges) connecting any two related nodes. Fig. 1 provides an example drawing of a small protein-protein interaction network. In the field of social network analysis [1], also referred to as network science [2], visualization is one of many approaches to understanding the structure of a network dataset. Interactive network visualization is usually a first

explorative step in understanding the data and may assist in finding outliers, important nodes, dense clusters or others patterns in the data.

The challenge in drawing a network is positioning the nodes in such a way that the resulting layout clearly depicts the structure of the network. In general, one wants to position a given node in proximity of related nodes, and at distance of unrelated nodes. Also, edge crossings, overlapping nodes and long edges should be prevented as much as possible. For humans, this becomes difficult when the size of the network exceeds a dozen of edges, which led to the development of graph layout algorithms. Although the first graph layout algorithms were designed to operate on thousands of nodes, recent algorithms scale to millions of nodes and edges [3]. Besides being a result of algorithmic advances, this is also the result of improved implementations and a focus on exploiting real-world network structure. Algorithms in particular take advantage of the fact that networks in the real-world, such as social networks, have skewed degree distributions, exhibit small-world connectivity and have a modular structure [2].

[☆] This paper was recommended for publication by Cong Bai.

^{*} Corresponding author.

E-mail address: f.w.takes@liacs.leidenuniv.nl (F.W. Takes).

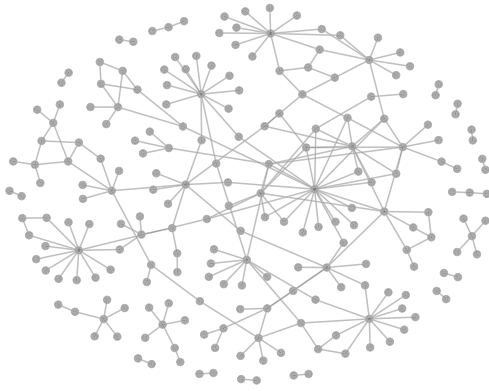


Fig. 1. Drawing of a protein–protein interaction network [5] in which the nodes represent proteins and the edges denote the interactions between proteins in a particular substance.

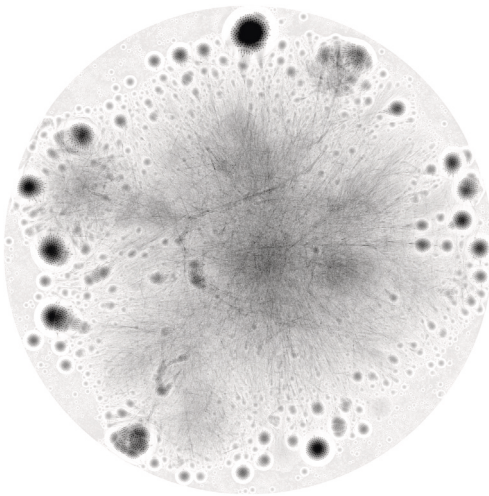


Fig. 2. Network drawing for a large network with 391,966 nodes and 1,711,968 edges [6], illustrating the excessive visual clutter caused by the small amount of available screen space.

However, visualizing very large networks on a typical desktop computer still poses challenges. Due to the relatively small amount of screen space that is available, viewing the layout of a very large network, such as the one depicted in Fig. 2, results in excessive visual clutter. An interactive presentation that allows for zooming and panning can overcome this issue. However, this reduces the number of nodes and edges that can be viewed at once, and thereby the viewer's ability to grasp the overall structure of the network. As such, applying a standard 'overview first, zoom and filter, then details on demand' information seeking principle remains challenging for visualizations of large networks [4].

To overcome this problem, the use of tiled display systems has been suggested [7,8]. Tiled display systems are composed of multiple displays, usually identical models, which are arranged as tiles to form a single large display area. For an example, see Fig. 3. Tiled display systems provide a scalable and cost-effective solution to the limited amount of screen space provided by a typical desktop computer, and have been successfully applied in diverse scenarios [9–11]. Depending on the number of displays, their resolution, and the demands of the application, the displays connect to a single computer or a distributed cluster consisting of multiple computers. In this paper we focus on the former case, in which all monitors connect to a single machine.

Earlier studies on the use of tiled display systems for network visualization relied on the system's central processing units (CPUs)

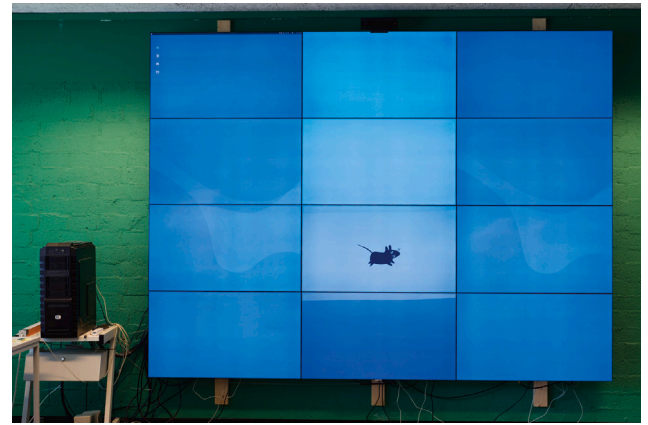


Fig. 3. 'BigEye', the tiled display system considered in this paper.

to algorithmically compute network layouts. Given the computational requirements for network visualization at high resolutions, we consider using the system's graphics processing units (GPUs) instead. Tiled display systems generally contain multiple graphics cards to be able to connect all displays to the system. On each of these graphics cards a GPU is located, which combined provide tremendous computational power. It has been shown that the GPU allows for high-performance implementations of graph layout algorithms [12–14]. Therefore we hypothesize that using the GPUs in a tiled display system allows for high-performance visualization of large-scale networks using its full resolution. More specifically, we expect to achieve real-time performance, where user's input is accounted for without visible delay, for networks with hundreds of thousands of nodes and edges. Real-time performance combined with interactivity improves the user's ability to effectively analyze a network. Besides easing data exploration (e.g., as depicted in Fig. 4), it allows users to manually steer the layout process, which can be crucial to prevent the layout algorithm from converging to a sub-optimal layout.

Our methodology considers ForceAtlas2 [15] as the graph layout algorithm to use for network visualization. The main contribution of our work is an extension of an existing GPU implementation of ForceAtlas2 in order to distribute the work over multiple GPUs. We adapt and combine this with distributed rendering and a 'tiled visualization' approach in which each GPU renders the part of the network to be displayed on the monitors attached to it. The focus of our work is on modifying the ForceAtlas2 algorithm in efficiently exploiting multiple GPUs, while guaranteeing the quality of the layout that is computed by the algorithm. This results in a system enabling real-time interactive visualization for networks with tens of thousands of nodes and edges, at a resolution of 5760×4320 pixels. To assess the resulting system's interactive capabilities, we will implement a number of interactions using a wireless control device that enables natural interactions with visualizations on the wall-sized display.

In summary, our contribution is a new methodology for visualizing large real-world networks on a tiled display system utilizing multiple GPUs. In doing so, the layout can be computed and thereafter immediately rendered, resulting in the possibility of real-time interactive visualization. Although an implementation was done on a particular 12-screen tiled display system, the approach can be used on any tiled display system equipped with one or multiple GPUs. Our implementation and demonstration videos of the framework are available at <https://liacs.leidenuniv.nl/~rietveldkfd/tiledvis>.

The content of this paper is structured as follows. In Section 2 we first discuss the preliminaries of this study. As such, we discuss definitions for concepts used throughout the paper, graph layout algorithms, the GPU as a platform for general purpose computation, as well as the

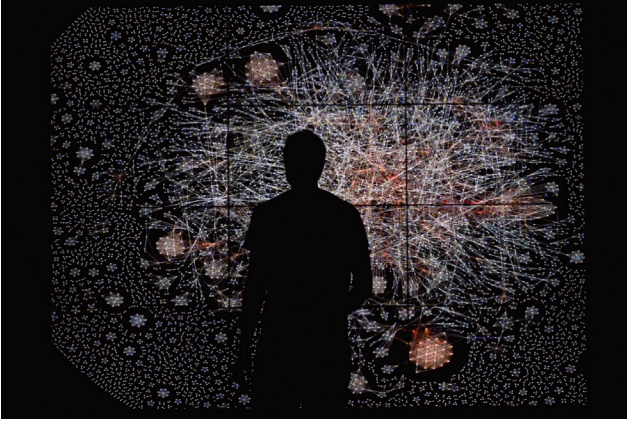


Fig. 4. Impression of a user analyzing a large network on BigEye.

ForceAtlas2 algorithm. Related work and existing approaches to network visualization on tiled display systems are discussed in Section 3. Our ‘tiled visualization’ approach to achieve real-time visualization on a tiled display system addressing the network visualization steps, distributed ForceAtlas2 layout and rendering, and the combination with ‘tiled visualization’, is explained in Section 4. Experiments to evaluate the performance of various aspects of the approach are outlined in Section 5. Finally, Section 6 concludes the paper and provides suggestions for future work.

2. Preliminaries

In this section, we review relevant concepts related to networks (Section 2.1), graph layout algorithms (Section 2.2), the architecture and potential benefits of graphics processing units (Section 2.3), and an existing GPU implementation of the ForceAtlas2 algorithm (Section 2.4).

2.1. Networks

In this work we do not make a distinction between networks and graphs, and hence use the terms interchangeably. We define a *network* or *graph* $G = (V, E)$, as a set of *vertices*, or *nodes*, together with a set of *edges*, or *links*. The set of edges that relate vertices, denoted E , is defined such that $E = \{\{u, v\} : u, v \in V \wedge u \neq v\}$. This means that we are dealing with *undirected* graphs; for the purpose of our visualization method the directionality of a graph can be ignored. Also, we do not consider networks with self-loops. Instead of $\{u, v\} \in E$, we can also write that node v is *adjacent* to node u , or that u and v are *neighbors*.

Below, we discuss several common and well-known topological properties of graphs. The *density* of a graph is defined as the ratio between the number of edges and the maximum number of edges, given the number of nodes. The density of an undirected graph equals $2|E|/(|V|(|V| - 1))$. The *degree* of node u is defined as the number of nodes it links to, i.e., $\deg(u) = |\{v \in V : \{u, v\} \in E\}|$. The *average degree* of a graph is the average over all node degrees.

Two nodes $u, v \in V$ are *connected* if a *path* between them exists. A length- n path between nodes u and v consists of a sequence of nodes $(x_0, x_1, \dots, x_n)$ such that $x_0 = u$, $x_n = v$ and $\forall i, 0 \leq i < n : \{x_i, x_{i+1}\} \in E$. The *shortest path* between nodes u and v is defined as the path with the shortest length that connects nodes u and v , and its length is referred to as the *distance* between these nodes, denoted $d(u, v)$. If no path between u and v exists, i.e. they are not connected, $d(u, v) = \infty$. A graph is *connected* if all node pairs are connected via a path.

Given a graph $G = (V, E)$, we define the subgraph induced by the set of nodes $V' \subseteq V$ to be the graph $G' = (V', E')$, such that $\{u, v\} \in E' \Leftrightarrow$

$u, v \in V' \wedge \{u, v\} \in E$. The *connected components* of a graph are its maximal connected subgraphs. These components are maximal in the sense that no nodes can be added to them without breaking the property that the subgraph is connected. Since we do not consider the directionality of edges for the purpose of our visualizations, we do not need to distinguish between strongly connected components (SCCs) that account for directionality and weakly connected components (WCCs) that ignore edge direction.

2.2. Graph layout algorithms

To reveal the structure of a graph, it is commonly drawn in a plane with vertices represented as points, and edges as straight lines between neighbors, as shown in Fig. 1. As discussed in Section 1, the challenge in drawing a graph is assigning a position $\mathbf{p}_v \in \mathbb{R}^2$ to each $v \in V$ such that a ‘readable’ layout emerges. In a readable layout related nodes are generally positioned in spatial proximity of each other, whereas unrelated nodes are at distance of each other. Also, overlapping nodes and edge crossings should be avoided. Most importantly, the layout should reflect the structure of the network.

Whilst determining readable layouts for networks with less than a dozen edges is feasible for a human, it is desirable to use a computer and a graph layout algorithm as the network size increases. A graph layout algorithm takes as input the graph $G = (V, E)$, potentially with additional information on nodes and edges, and computes for each $v \in V$ a position $\mathbf{p}_v \in \mathbb{R}^2$, such that a readable layout emerges. From the different types of graph layout algorithms that have been conceived [3,16], we choose to focus on force-directed graph layout algorithms. In Section 2.4 we describe how these algorithms work and why they are especially suitable for our goal of realizing interactive network visualization.

2.3. Graphics processing units (GPUs)

For the present study we use graphics processing units (GPUs) for network visualization, i.e., to implement the layout algorithm and the network renderer that generates network drawing from the layouts. In contrast to the CPU architecture, that is optimized to achieve low latency on a wide range of computational problems, the GPU is optimized for high throughput on (highly) data-parallel problems [17]. This is partly due to the relatively large number of functional units that the GPU features, which generally come at the expense of omitting the advanced, latency-reducing, features employed by the CPU such as speculative execution. Although the design of the GPU has its origins in computer graphics applications, which involve the processing of many independent pixels and geometrical primitives, the architecture has increasingly been used in high-performance implementations of general purpose computations [18]. Whereas general purpose computation on GPUs (GPGPU) initially required formulating problems using graphics APIs such as OpenGL [19], dedicated GPGPU frameworks such as CUDA [20] and OpenCL [21] have since been released.

Given the large number of independent, per-node, operations used in graph layout algorithms, the GPU has enabled high-performance implementations [12,22,23] of these algorithms as well. Implementing the layout algorithm on the GPU also enables ‘in situ’ visualization, where the GPU datastructures are shared between the network layout and drawing algorithms. The performance improvements this brings, can be especially important for interactive applications. When using NVIDIA graphics cards, the CUDA-OpenGL interoperability API [24] can be used to implement an in situ visualization approach.

Since we use the NVIDIA compute unified device architecture (CUDA) platform [20] to implement our multi-GPU network layout algorithm, which is discussed in Section 4.4, we use this section to provide some background information on CUDA and the architecture of NVIDIA GPUs.

CUDA enables programmers to implement general purpose computations on the highly parallel architecture provided by NVIDIA GPUs. For our study we use CUDA C, which is an extension to C programming language. Using CUDA C, programmers can specify *kernels*, which are functions to be executed in parallel on the GPU using many *threads* of execution. Although threads execute the same kernel, they generally operate on different data elements, thus parallelizing a computation. CUDA threads are organized by the programmer into a *grid* of *thread blocks*. When launching a grid for execution, the blocks contained therein are assigned to different GPU *multiprocessors*, which subsequently subdivides the thread block into *warps* of 32 threads. The threads in a warp concurrently execute the same instruction, through a *single instruction multiple threads* (SIMT) architecture and the multiprocessor is equipped with dedicated hardware to interleave the execution of different warps in order to hide latency. To achieve optimal hardware utilization, it is important that the threads in a warp share execution paths, since diverged threads in a warp cannot execute concurrently. Also, to effectively utilize the memory bandwidth provided by the GPU, threads in a warp should access consecutive ranges of aligned memory. This allows for multiple memory accesses to be *coalesced* into single transactions, which greatly improves the utilization of memory bandwidth.

2.4. GPU implementation of ForceAtlas2

We selected ForceAtlas2 [15] as graph layout algorithm for a number of reasons. First, it is designed for interactive applications, hence its origins in the well-known Gephi [25] network analysis software. Second, the iterative approach, which corresponds to a time-continuous process, allows for the entire layout process to be visualized. As such users can comprehend the layout procedure, and better understand the effect the different parameters have. The force-directed method also accounts for interaction with the layout process, allowing users to manually steer it to improve layout quality. Since ForceAtlas2 largely follows the general force-directed approach to graph layout, our results have the potential to apply to other force-directed graph layout algorithms as well. An open source implementation of the ForceAtlas2 algorithm for the GPU was introduced in previous work [12].

Like other force-directed graph layout algorithms, ForceAtlas2 approaches the graph layout problem by considering the graph layout to be a physical system. Nodes are represented by physical bodies, and the layout process corresponds to a simulation of their interactions over time. An iterative procedure repeatedly displaces each node in the network in accordance to the resultant force acting on it. Repulsive forces between all node-pairs serve to move unrelated nodes away from each other, whereas the attractive forces between neighboring nodes should cause related nodes to move towards each other. A 'gravitational' force towards the origin of the layout ensures that all components of the graph remain in proximity of each other.

A direct implementation of ForceAtlas2 would result in evaluating all node-pairs to calculate the repulsive force acting on each node, yielding a computational complexity in $\Theta(|V|^2)$. Naturally this does not scale well to large networks. Repulsive forces between all node pairs are thus approximated using the Barnes–Hut algorithm [26], which employs a quadtree representation of the graph layout to approximate the pair-wise force interactions. As a consequence, the computational complexity of the repulsive force computation is reduced from $\Theta(|V|^2)$ to $\Theta(|V| \log |V|)$. Since the computational complexity of computing attractive forces is in $\Theta(|E|)$, and since gravitational forces can be applied in $\Theta(|V|)$ time, the computational complexity of one iteration of the ForceAtlas2 Algorithm is then in $\Theta(|V| \log |V| + |E|)$.

Algorithm 1 presents pseudocode for ForceAtlas2, which describes the procedure discussed in this paragraph more formally. Note that the many per-node force calculations during an iteration of the algorithm are independent and can all be performed in parallel. This property is used to develop a highly parallel GPU implementation of this algorithm

Algorithm 1 Pseudocode for a simplified version of the ForceAtlas2 graph layout algorithm, adapted from [12]. Full details on ForceAtlas2 can be found in [15].

Input: Graph $G = (V, E)$, it_{max} (number of layout iterations), k_g (gravitational force scalar) and k_r (repulsive force scalar), θ (Barnes–Hut accuracy).

Output: For each $v \in V$, a position $\mathbf{p}_v \in \mathbb{R}^2$.

```

1:  $global\_speed \leftarrow 1.0$ 
2: for all  $v \in V$  do
3:    $\mathbf{p}_v \leftarrow \text{RANDOM}()$ 
4:    $\mathbf{f}_v \leftarrow (0.0, 0.0)^T$ 
5:    $\mathbf{f}'_v \leftarrow (0.0, 0.0)^T$ 
6: end for

7: for  $i = 1 \rightarrow it_{max}$  do
8:   BH.BUILD()
9:   for all  $v \in V$  do
10:     $\mathbf{f}_v \leftarrow \mathbf{f}_v - (k_g * (deg(v) + 1) * \mathbf{p}_v)$ 
11:     $\mathbf{f}_v \leftarrow \mathbf{f}_v + (k_r * \text{BH.FORCE\_AT}(\mathbf{p}_v, deg(v), \theta))$ 
12:    for all  $w \in \text{NEIGHBORS}(v)$  do
13:       $\mathbf{f}_v \leftarrow \mathbf{f}_v + (\mathbf{p}_w - \mathbf{p}_v)$ 
14:    end for
15:  end for
16:   $global\_speed \leftarrow \text{UPDATEGLOBALSPEED}()$ 
17:  for all  $v \in V$  do
18:     $\mathbf{p}_v \leftarrow \text{LOCAL\_SPEED}(v) * \mathbf{f}_v$ 
19:     $\mathbf{f}'_v \leftarrow \mathbf{f}_v$ 
20:     $\mathbf{f}_v \leftarrow (0.0, 0.0)^T$ 
21:  end for
22: end for

23: function LOCAL_SPEED( $v$ )
24:   return  $\frac{global\_speed}{1.0 + global\_speed * \sqrt{SWING(v)}}$ 
25: end function

26: function SWING( $v$ )
27:   return  $|\mathbf{f}_v - \mathbf{f}'_v|$ 
28: end function

```

that has the potential to scale to significantly larger networks than serial implementations. In this paper, we further extend the open source CUDA implementation of ForceAtlas2 that we published previously [12]. The CUDA implementation of the Barnes–Hut algorithm used by this ForceAtlas2 implementation is taken from [27]. For further details on these implementations, we refer reader to the corresponding papers.

3. Previous work

In this section we discuss previous work on the use of tiled display systems for network visualization, and compare these works to the approach taken for the present study.

Mueller et al. [8] present a network visualization approach for distributed (tiled display) systems, that uses MPI [28] for parallel computation and Chromium [29] for distributed rendering and display. A distributed force-directed graph layout algorithm is derived, based on the classical algorithm by Fruchterman and Reingold [30], by considering how different approaches to data- and work-distribution affect performance and the layout quality. Realizing performance levels suitable for interactivity is a clear objective of the authors. The authors evaluate their system on an eight monitor tiled display system, connected to an eight node cluster, both in terms of its scalability when using increasing numbers of processors and in terms of the 'costs' resulting from displaying the layouts. Randomly generated networks are used for the evaluation. For networks with 2000 nodes the results show

an average performance improvement of 10× when scaling from one to four processors, after which performance improvements saturate. Using eight processors allowed for randomly generated networks with 8000 nodes, and up to (approximately) 80,000 edges, to be visualized at frame rates between 2 to 5 frames per second.

Chae [7,31] presents distributed algorithms for network visualization, which are evaluated on a 200 megapixel cluster-based tiled display system composed of 50 monitors. In the development of the algorithms, techniques are considered to reduce the number of edges crossing between nodes in the cluster, but also to prevent nodes in the layout from being positioned on the bezels of monitors. A modified k-means clustering algorithm is evaluated in order to reduce the running time of the layout algorithm. The algorithms are evaluated both in terms of their scalability and the layout qualities that result from them.

Jingai et al. [32] report their intermediary results on adapting the open source Gephi [25] network analysis and visualization software to run on a cluster-based tiled display system via a commercial middleware. Since the middleware supports the OpenGL graphics library, that Gephi uses for network rendering, few changes to Gephi were required. The authors report their system was successful at visualizing a protein–protein interaction network with 2361 nodes and 7182 edges, with all labels visible, and that the performance of the system should be assessed in future studies.

Gu et al. [33] present an approach for the interactive visual analysis of image collections by means of a compound ‘iGraph’ representing the relationships between the images and keywords in the collection. To visualize the iGraph, a force-directed graph layout algorithm based on the classical algorithm by Fruchterman and Reingold [30] is first used to compute a layout for the backbone of iGraph, which consists of images and keywords that are representative for the collection. This layout is then further refined ‘on demand’ as the user navigates around the iGraph. The system allows for the interactive filtering and comparison of data in the iGraph, and uses an approach akin to collaborative filtering to recommend interesting data to users for further exploration. The approach is evaluated using two datasets on an eight node cluster using graphs with thousands of nodes and millions of edges. Besides, a method for using the cluster to visualize results on a 50 megapixel tiled display system is discussed. The GPUs in the system are used for data pre-processing, whereas the CPUs are used for the graph layout process. Finally, a user evaluation is discussed.

Similar to the studies discussed in this section, we focus on using force-directed graph layout algorithms for our visualization system. However, we do not consider significant modifications to the algorithms to improve their operation in the context of the tiled display system. We rather focus on improvements in their implementation. Most importantly, we consider using the GPUs in the tiled display system for their implementation rather than the CPU. Also, we do not focus on a tiled display system which distributes the visualization over multiple computers (nodes), but on a single computer with 12 monitors connected to three graphics cards. We consider the multiple GPUs for improved processing power only, so to distribute the computational load, without distributing the data between the GPUs to obtain an increased memory capacity. Our previous work [12] does not suggest that the memory capacity of current systems bounds the size of networks that can be visualized interactively. In contrast to the studies we discussed in this section, we use a wider collection of large real-world networks for the evaluation of our system. Similar to the study of Mueller et al. we focus on realizing an interactive system. However, we aim to avoid any perceivable delay between user input and corresponding updates on the monitors, since this would degrade the extent to which users experience interactivity. We choose to focus on obtaining a frame rate of 60 Hz., matching the refresh rate of the monitors in the tiled display system.

4. Approach

Our goal is to enable the visualization of large networks at high resolutions and on large screen areas, while still allowing a user to interactively explore the network through panning and zooming. The resolution and screen area that we would like to achieve require the use of more monitors than can be connected to a single graphics card. As a result, it is required to use multiple graphics cards. This introduces performance and synchronization challenges, which significantly complicate the implementation compared to a typical desktop system. We discuss these challenges and present a ‘tiled visualization approach’ to overcome them in this section. In Section 5 the effectiveness of the proposed solutions will be evaluated.

We first briefly describe the target hardware used in this study in Section 4.1. Next, in Section 4.2, we propose a GPU accelerated network renderer as an extension of the GPU implementation of ForceAtlas2 that was described in Section 2.4. It enables high resolution visualizations of results of the layout process. To then be able to depict the visualization on a tiled display, Section 4.3 addresses performance and synchronization issues with full screen visualizations spanning multiple graphics cards (GPUs) in a Linux-based system. In Section 4.4 we describe a distributed implementation of the layout algorithm and network renderer, which exploits the multiple GPUs that are available in the system to further improve the performance.

We describe the integration of the aforementioned distributed GPU implementation of ForceAtlas2 and network renderer with the tiled visualization approach in Section 4.5. Finally, we address interaction with the wall-sized display using the Nintendo WiiMote in Section 4.6.

4.1. Target hardware

In this paper, we focus specifically on off-the-shelf components, which allows this setup to be easily replicated. Our target system consists of a single computer equipped with an Intel Core i7 processor and three NVIDIA GeForce GTX660 graphics cards to which 12 monitors with a resolution of 1920×1080 pixels are connected. The software installation consists of Ubuntu 16.04.3 LTS with the NVIDIA proprietary graphics driver. Throughout the paper, we will refer to this system as ‘BigEye’ (see Figs. 3 and 4). Full details on this system can be found in Section 5.1.

4.2. GPU accelerated network rendering

The ForceAtlas2 algorithm computes a graph layout, which consists of positions in the two-dimensional plane for each node of a graph. In order to visualize such a computed layout, the nodes and edges must be rendered in a framebuffer of pixels. In this section, we propose a GPU accelerated network renderer as an extension to the GPU implementation of ForceAtlas2.

In order to implement GPU acceleration, the network renderer is implemented using OpenGL 4.1 (Core Profile) [19]. CUDA and OpenGL code is typically controlled and executed separately. This implies that the resulting node positions of the CUDA implementation of the ForceAtlas2 layout algorithm must be transferred from the GPU back to the CPU memory and in order to initialize the OpenGL accelerated network renderer, these same node positions need to be wrapped in an OpenGL datastructure and uploaded to the GPU memory again. This can be seen in Fig. 5(a), in which the copy operations have been highlighted in red. The transfer of data to/from the GPU is subject to PCIe bandwidth and latency constraints and these frequent transfers can prohibit us from achieving real-time visualization performance for larger networks.

To mitigate this, we use the CUDA-OpenGL interop [24] capability to map an OpenGL datastructure into CUDA memory, such that the CUDA implementation of ForceAtlas2 can store the node positions directly in an OpenGL datastructure. We represent the network data in

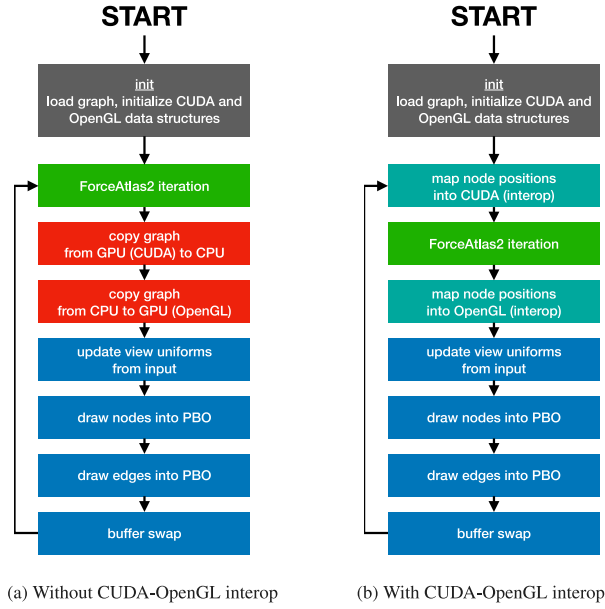


Fig. 5. Display loop used for our network visualizer. Green boxes correspond to calls to the CUDA API, blue boxes correspond to calls to the OpenGL API.

OpenGL as follows. All node positions are stored in an OpenGL buffer object, which is indexed by node identifiers (node IDs) that range from 0 to $|V|$. Both the CUDA implementation of ForceAtlas2, as well as the OpenGL network renderer, operate directly on this OpenGL buffer. Additional node attributes can be represented in additional buffers, or can be interleaved with the data on nodes' positions. Such data could be accounted for in different stages of the OpenGL pipeline, and as such could be reflected in the appearance of nodes. For example, this would allow nodes to be sized or colored based on a property, such as their degree. The network's edges are also stored in an OpenGL buffer object, and are represented by their respective source- and target-node IDs, which are stored consecutively in the buffer of edges. This data is uploaded only once to OpenGL, and is not shared with CUDA, given that it remains static throughout the layout process. The resulting display loop is depicted in Fig. 5(b), where the expensive copy operations have been eliminated.

The network's nodes are drawn by generating OpenGL point primitives using node positions from the described buffer object. The square shapes that result from this operation are transformed into circles using a fragment-shader as described in [34]. Edges are drawn as OpenGL line primitives, whose starting- and end-points are obtained by indexing into the buffer of node positions using the source- and target-node identifiers stored in the buffer representing the network's edges. In order to enable user navigation, a vertex shader contains uniforms representing the current view. Using the vertex shader, all vertices are translated according to these uniforms, which are updated in correspondence to the user's interactions.

4.3. Tiled visualization

In this subsection we describe how to display the network drawings resulting from the renderer presented in the preceding section at the full resolution available on a wall-sized display comprising of multiple monitors. This visualization method is also referred to as the use of tiled display systems for visualization; for a more general overview we refer the reader to [35].

Displaying a framebuffer to multiple monitors connected to a single graphics card is already provided by various graphics card drivers. In our particular system, we make use of the most recent version (390.42)

of the proprietary NVIDIA driver that was available for the graphics cards in our system at the time of implementation. Using the NVIDIA driver up to four display devices connected to a single graphics card can be addressed as a single uniform display device.

To span the visualizations beyond the monitors connected to a single graphics card, we consider the extensions to the *X Window System* [36], the windowing system adopted by most Linux distributions, that enable this. For example, the *Xinerama* extension, allows applications to address all physical display devices connected to the system through a single X screen, instead of a distinct X screen per display device. Besides *Xinerama*, the *X Resize, Rotate and Reflect* extension for the X Windows System (RandR) also provides multi-monitor support for systems using multiple graphics cards, since version 1.4 [37]. Unfortunately version 1.4 of RandR is not fully supported by latest version of the proprietary NVIDIA driver for the graphics cards in our system [38] at the time of implementation. Also, using *Xinerama* presents severe performance and synchronization issues.

We will discuss these performance and synchronization issues, and how our 'tiled visualization' approach overcomes them, in Sections 4.3.1 and 4.3.2 respectively.

4.3.1. Displaying framebuffers spanning multiple graphics cards

A main problem with the *Xinerama* approach when combined with the use of OpenGL for rendering, is that addressing individual GPUs is no longer possible, and that one relies on the NVIDIA driver to (eventually) distribute the OpenGL command stream between the GPUs. With regard to the NVIDIA implementation of OpenGL it appears in practice, to the best of our knowledge, that one GPU is made responsible of performing all OpenGL rendering, after which the resulting framebuffers are pushed to the other GPUs. As a consequence, the other GPUs do not contribute to the rendering and essentially sit idle, and, secondly, the transfer of resulting framebuffers to the other GPUs are subject to PCI Express (PCIe) latency and bandwidth limitations.

To address these challenges, we propose and implement a 'tiled visualization approach' in which each GPU in the tiled display system renders the part of the visualization to be displayed on the monitors directly attached to it. This is achieved by creating a separate X screen for each GPU present in the system. Each GPU visualizes and displays the data residing on it, without transferring computed data nor rendered framebuffers to other parts of the system. This significantly reduces the amount of PCIe bandwidth utilization for each GPU. This manifests itself in particular for increasing image resolutions, where order of magnitude performance differences are seen compared to CPU and *Xinerama* GPU implementations that do require framebuffers to be transferred to different GPUs. Additional benefits of this approach are that by individually addressing the different GPUs, we can potentially reduce tearing artifacts between different monitors (see Section 4.3.2), and that the computational power provided by all GPUs can be employed to scale to the high resolution of the tiled display system (see Section 4.4).

Implementation details. We use the existing support at the level of the graphics-card driver, in our case the proprietary NVIDIA driver, to provide for each graphics card a single framebuffer and X screen that spans the monitors connected to that card. As such we end up with one framebuffer per graphics card, in our case each addressing a column of monitors in the tiled display system.

A separate CPU thread is created for each of the GPUs, which will create the corresponding OpenGL context and execute the display loop described in Section 4.2 to update the contents of the distributed framebuffers. Each framebuffer is stored in an OpenGL Pixel Buffer Object (PBO) and is displayed on the monitors connected to a certain GPU by setting the PBO as the source of an OpenGL texture that is drawn to a window spanning all monitors connected to that particular GPU. This window is configured to be a double-buffered drawable. As such, all drawing occurs to an invisible back-buffer and the contents of

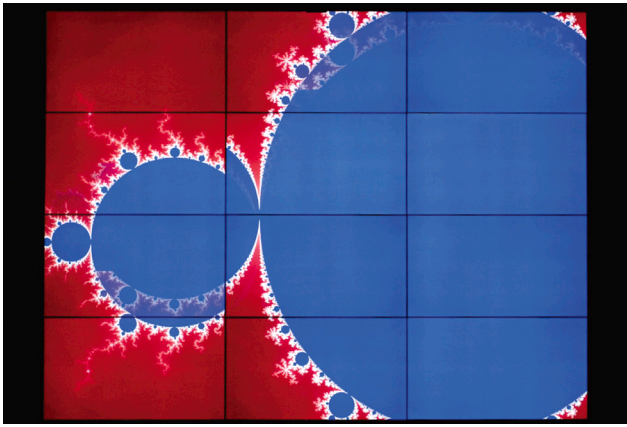


Fig. 6. Example of tearing artifacts when using Xinerama.

this buffer are only displayed once its contents are exchanged with the visible front-buffer through a bufferswap. This bufferswap completes the display loop, which then proceeds with its next iteration. Each GPU will be operating on its own OpenGL (and CUDA) datastructures.

In order to successfully implement the proposed approach, it is crucial to choose the appropriate X client library. The multi-threading support of Xlib [39] was found to be insufficient.¹ Therefore, we turned to the 'X protocol C-language Binding' (XCB) [40] library, which did not present these issues. Using XCB, it was possible to address the X server from all threads through a single connection. Moreover, for a coherent visualization without tearing artifacts, the threads are required to be synchronized, which will be discussed next.

4.3.2. Monitor and input synchronization

To ensure a coherent image on the tiled display system, updates to the monitors in the system need to synchronize to both their vertical refresh, as well as to updates to the other monitors in the system. If this is neglected, moving imagery can result in discontinuities, that appear as horizontal or vertical tearing artifacts between past and current input to the display. An example of such vertical tearing artifacts can be seen in Fig. 6. For our study we do not consider synchronizing updates to the monitors with their vertical refresh, but rather focus on synchronizing updates between the sets of monitors connected to different graphics cards. We deem the latter to be of greatest importance in ensuring a coherent image across the different monitors.

As discussed in the preceding section, the framebuffer for our display system is distributed between the different GPUs, and a separate display loop is executed for each GPU in a distinct CPU thread. We will refer to these threads as display threads. These display threads are executed in parallel. Updates to the different GPUs that are performed by these separate display threads have to be synchronized to ensure a coherent image across the wall-sized display.

Next to the display threads, the system foresees in a separate event thread that handles user input such as keyboard and mouse events. These events control the current view-state, i.e. the part of the network that should be visualized on the tiled display. Updates to the view-state from the event thread can occur interleaved with reads by the display threads, during the generation of a single video frame. For example, imagine that for a given frame the display thread controlling GPU 2 first reads the view state. Next, the view-state is updated from the event

thread, after which the display thread controlling GPU 1 read the view-state. If this occurs the different display threads no longer agree on the part of the network that should be visualized, each working on a different version of the view-state. This causes discontinuities in the visualization, similar to tearing.

Implementation details. To synchronize updates to monitors that connect to different GPUs, we synchronize the OpenGL bufferswap across the different CPU threads executing the display loops in parallel. A thread-barrier is used to block each thread from advancing until all threads are ready to perform the bufferswap. However, by default the NVIDIA driver synchronizes the OpenGL bufferswap operation for a given GPU to the vertical refresh of one of the monitors attached to it [41]. If the monitors selected for this across different GPUs are not synchronized in terms of their vertical refresh, this can introduce delays between the bufferswaps on different GPUs. These delays can take up to 16.67 ms, assuming a refresh at rate of 60 Hz. Hence we disable the synchronization between bufferswap and vertical refresh via the `EXT_swap_control` [42] extension to OpenGL. The image corruption introduced by this is relatively minor. Besides, it potentially allows for significant improvements in the framerate that is achieved, given that the bufferswap operation no longer waits for the next refresh of one of the monitors.

This inter-GPU synchronization approach leaves room for further improvement, since it only synchronizes the initiation of the bufferswap processes for different graphics cards on the CPU, rather than the actual bufferswap operations which occur on the graphics cards. For our target system, it appeared impossible to enforce a synchronized bufferswap at the level of the graphics card. The NVIDIA provided `NV_swap_group` [43] OpenGL extension that accounts for this, was not available on the consumer-grade GeForce graphics cards we use.

The thread-barrier is also used to solve the synchronization of the view-state. Similar to the framebuffers, two copies of the view-state are maintained. The event thread only accesses one copy, the 'event-view-state', which it updates continuously based on user input. The other copy, the 'display-view-state', is used by all display threads to determine what needs to be drawn to the display. A single display thread copies the event-view-state to the display-view-state, once all display threads have reached the thread-barrier. Display threads are released from the barrier only after this copy completes, guaranteeing that all display threads used the same view-state.

4.4. Distributing network visualization over multiple GPUs

In initial experiments we found that the majority of the network visualization time is spent on ForceAtlas2 layout computation and on drawing the resulting layout. Given that we have multiple GPUs available in our target system due to the fact that these are required to attach the large number of monitors, we can improve the performance by extending the ForceAtlas2 GPU implementation and network renderer to distribute the work over multiple GPUs. As a consequence of the improved performance, we can scale our network visualization approach to larger networks.

4.4.1. Distributed ForceAtlas2

Our objective in modifying ForceAtlas2 to use multiple GPUs is to exploit the increased processing power provided by multiple GPUs, rather than using the increased amount memory that is available. This, given that the former limits the size of networks that can be visualized interactively in real-time. For example, the email-EUAll network, with 224,832 nodes and 339,924 edges, requires at most 100 MiB of memory, which is approximately 5% of the memory available on a single NVIDIA GTX660 graphics card. As such, we allocate a copy of all datastructures on each GPU. Besides simplifying our implementation, duplicating data across different GPUs also reduces the need for communication between GPUs, potentially improving performance. Only

¹ Although we explicitly enabled multi-threading support through the `XInitThreads` function, we observed that a call to `XNextEvent`, to retrieve events from the event-queue, blocked all display threads. This could not be resolved by using a distinct connection to the X server for each of the threads.

the partial results computed by each GPU need to be transferred to the other GPUs.

Initial experiments prior to modification of the ForceAtlas2 GPU implementation showed that force approximation using the Barnes–Hut algorithm accounts for approximately 80% of the running time, on average (see Section 5.3.2 for a validation of the workload composition). Hence, we initially focused on utilizing multiple GPUs to improve the performance of the force approximation component of ForceAtlas2. In order to do so, we adapted the existing force approximation code [27] that is used in our ForceAtlas2 implementation, to run on multiple GPUs.

The distributed ForceAtlas2 algorithm can be seen in Algorithm 2. In this algorithm, the repulsive force approximation computation was distributed between GPUs by partitioning the network's nodes into equally sized parts, and assigning a different part to each GPU. After each GPU has sorted all nodes according to their spatial proximity, each GPU will perform the repulsive force approximation computation for its assigned part. By doing so, nodes assigned to the same streaming multiprocessor on a given GPU are in spatial proximity of each other.

As described by the authors of the force approximation code [27], this is important to achieve good performance. Note that our work distribution does not assign the nodes displayed by a given GPU to that same GPU for repulsive force approximation. Although this would allow for true in situ visualization, it would pose a number of problems. For example, if nodes move between monitors connected to different GPUs, either as a result of user interaction or as the result of layout process, they would need to be assigned to a different GPU. Especially during the initial iterations of the layout algorithm, which involves large displacements for most nodes, this would result in significant amounts of inter-GPU communication. Besides, distributing nodes to different GPUs based on their location in the layout would not always allow for a balanced computational load between the different GPUs.

After approximating the repulsive forces in a distributed manner, the results are reduced to a master GPU. The global speed update and node displacements are performed on the master GPU, after which the new node positions are broadcasted to the other GPUs, such that each GPU again holds an up-to-date copy of all data. Given that from our initial experiments followed that speed-update and node displacement comprise a negligible fraction of the total iteration duration, we did not consider distributing them between the GPUs to focus our effort on distributing the parts of the computation forming the scalability bottleneck.

4.4.2. Distributed network rendering

Rendering network drawings comprises approximately 46% of the visualization time, of which 45% is due to the time spent on drawing the network's edges, see also Section 4.2. As such, we also distributed the network renderer across multiple, which comprises the blue boxes in Fig. 5(b). As for the distributed ForceAtlas2 implementation, all datastructures are duplicated on the different GPUs.

To distribute the render work across GPUs in general, the layout space was partitioned into equally sized columns, and each GPU was assigned a column for which to render the network drawing. As the first step, on each GPU the view uniform is updated which causes vertices to be translated to reflect the partitioning of the layout space. Secondly, each GPU will issue commands to render all nodes and edges. Due to the updated view uniform, each GPU will discard vertices which are part of the layout space assigned to other GPUs, via vertex clipping in the OpenGL pipeline. Finally, each GPU will perform a buffer swap. Summarizing, distribution is achieved by issuing commands for all nodes and edges to each GPU, and relying on the OpenGL pipeline to discard unnecessary work. This can also be seen in Fig. 7. In the case of BigEye three parts are used that correspond to three columns of monitors, where each column of monitors is attached to a single GPU.

4.5. Network visualization on a tiled display system

To realize interactive network visualization on BigEye, we combine the multi-GPU network layout and rendering implementation, described in Section 4.4, with the tiled visualization approach described in Section 4.3. The resulting architecture is shown in Fig. 7. Three display loops, that run on different CPU threads, each address one of the GPUs in the system. The X window and GL context for each of the display loops is setup using XCB. For each GPU, the display loop repeatedly advances the layout algorithm. The CUDA implementation of the layout algorithm outputs the node positions directly into an OpenGL datastructure using CUDA-OpenGL interop. Subsequently, the part of the layout to be displayed on connected monitors is rendered and displayed. An additional synchronization point between the threads is required, besides the thread-barrier that synchronizes the bufferswap.

The reduction step is introduced due to the work distribution used in the ForceAtlas2 implementation, as is detailed in Algorithm 2. Strictly seen the computation of the layout does now no longer happen fully 'in situ'. As will be described in Section 5.3.2 the performance benefits of using multiple GPUs outweighs the time required to exchange the results, which comprises a negligible part of the total duration of the display loop.

4.6. Interaction

In order to be able to test and assess the interactive capabilities of our network visualization system we implemented a number of interactions. All interactions were controllable by means of standard X keyboard and mouse events. Besides simplifying development, this provides an interface that many input devices can target. Basic navigation, i.e. panning and zooming, was implemented by means of mouse dragging and mouse scrolling, respectively, as well as via the keyboard's arrow-keys and the +/- keys. Next, we enabled control over a number of the network layout algorithm's parameters, as well as parameters for the network renderer, through the keyboard. As such users can adapt the scalars used for repulsive and gravitational forces during the layout process, and adjust the opacity of nodes and edges. The network renderer can also be set to color nodes based on their degree. Finally, we added two special interaction modes:

1. *Local repulse mode*, causing the mouse pointer to act as a repulsive force. Nodes will move away from the pointer, with a displacement proportional to their distance to it and a keyboard-controlled scalar. Using the local repulse mode, it is possible to destabilize the layout locally, potentially causing it to converge to a different configuration.
2. *Local heat mode*, causing the local speed to increase for nodes near the mouse pointer. This increases the distance by which nodes around the pointer are displaced. We expect it can also aid to further develop certain parts of the layout that may have converged to a sub-optimal layout.

Fig. 8 depicts our "BigEye" system. A standard keyboard and mouse do not prove a natural form of interaction with such a wall-sized display. Therefore, we evaluated the use of a Nintendo Wii remote as input device for our system. The WiiMote connects to the system via Bluetooth and reports on button presses, its acceleration along three dimensions and on its position in relation to infrared light sources that it tracks using an embedded camera. The latter allows the position of the WiiMote to be determined in relation to a 'sensor bar', which contains a number of infrared light sources. In our system the sensor bar is mounted underneath the middle column of displays.

The open source XWiiMote [44] software was used to convert WiiMote data into keyboard and mouse events to be processed by our system. We configured the X input driver provided by XWiiMote such that the mouse pointer followed the location to which the WiiMote was

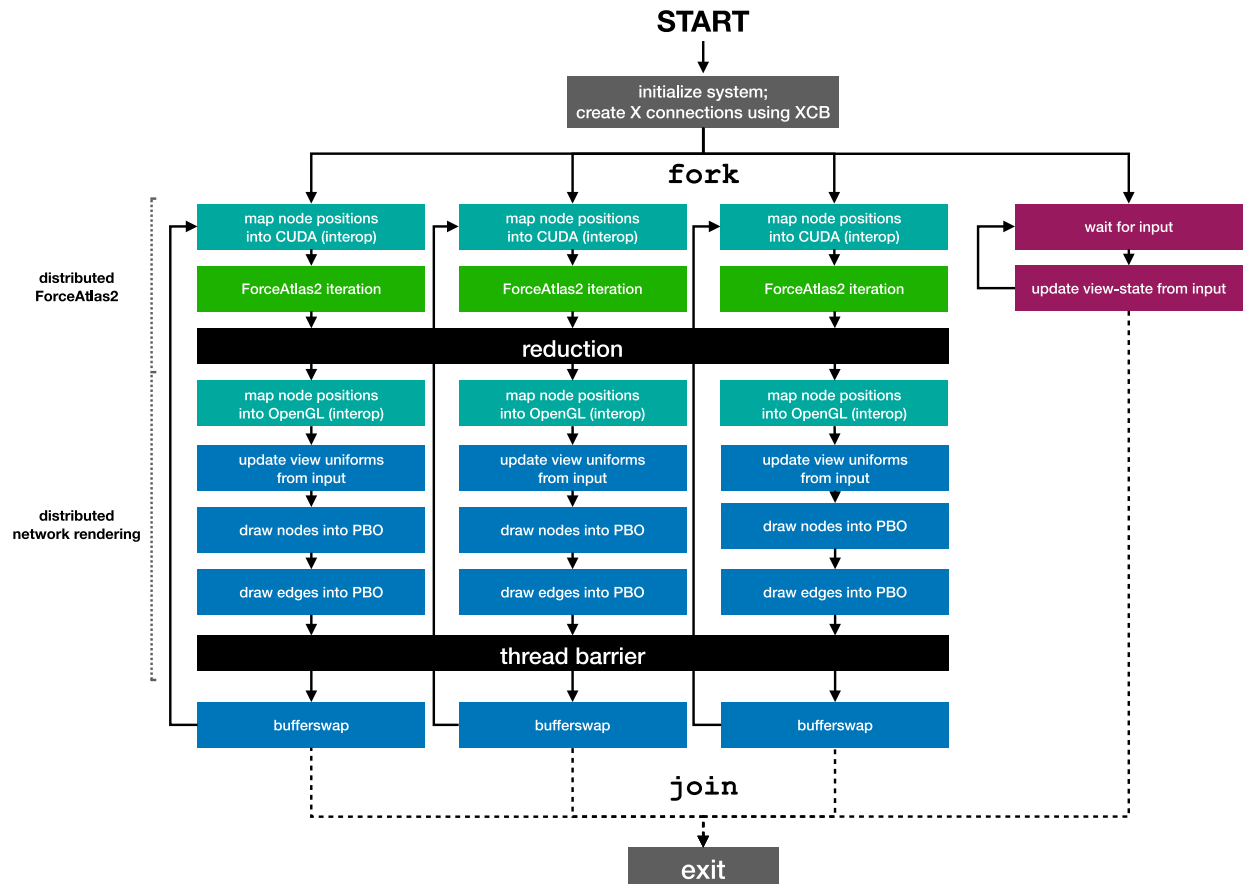


Fig. 7. Network visualization on BigEye using the tiled visualization approach, implemented using multiple threads. Green boxes correspond to calls to the CUDA API, blue boxes to calls to the OpenGL API. Event-processing commands are shown in purple. Dotted lines are branches taken when the application should exit.

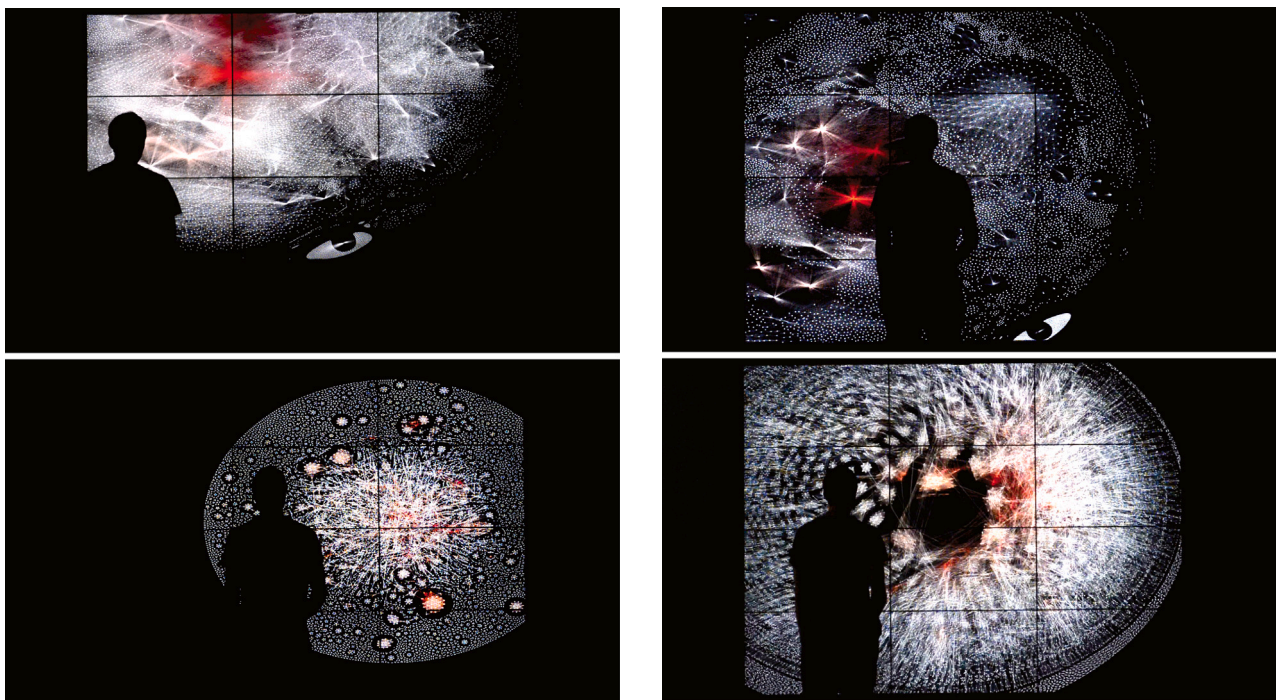


Fig. 8. The resulting network visualization application running for two different networks on BigEye. Bottom-right image depicts the local repulsion mode.

Algorithm 2 Pseudocode for a distributed version of the ForceAtlas2 graph layout algorithm shown in Algorithm 1.

Input: Graph $G = (V, E)$, n (number of GPUs), it_{max} (number of layout iterations), k_g (gravitational force scalar), k_r (repulsive force scalar), θ (Barnes–Hut accuracy)
Output: For each $v \in V$, a position $\mathbf{p}_v \in \mathbb{R}^2$.

```

1:  $global\_speed \leftarrow 1.0$  ▷ Initialize variables
2: for all  $v \in V$  do
3:    $\mathbf{p}_v \leftarrow \text{RANDOM}()$ 
4:    $\mathbf{f}_v \leftarrow (0.0, 0.0)^\top$  ▷ Net force on node  $v$ 
5:    $\mathbf{f}'_v \leftarrow (0.0, 0.0)^\top$  ▷  $\mathbf{f}'_v$  is  $\mathbf{f}_v$  of preceding iteration
6: end for
7: for  $j = 1 \rightarrow n$  do ▷ Copy graph to all GPUs
8:   Transfer  $(V, E), \mathbf{p}, \mathbf{f}, \mathbf{f}'$  from  $CPU$  to  $GPU_j$ 
9: end for
▷ Start layout process
10: for  $i = 1 \rightarrow it_{max}$  do
11:   parfor  $j = 1 \rightarrow n$  do ▷ Perform on all GPUs in parallel
12:     BH.BUILD( ) on  $GPU_j$  ▷ (Re)build Barnes–Hut tree on all GPUs
13:     parfor  $m = 1 \rightarrow |V|$  do
14:        $\mathbf{f}_{v_m} \leftarrow \mathbf{f}_{v_m} - (k_g * (deg(v_m) + 1) * \mathbf{p}_{v_m})$  ▷ (Strong) Gravity
15:     end parfor
16:     Sort  $V$  based on spatial proximity.
17:     parfor  $m = ((j - 1) \times \lfloor |V|/n \rfloor) \rightarrow \max(j \times \lfloor |V|/n \rfloor, |V|)$  do
18:        $\mathbf{f}_{v_m} \leftarrow \mathbf{f}_{v_m} + (k_r * \text{BH.FORCE\_AT}(\mathbf{p}_{v_m}, deg(v_m), \theta))$  ▷ Repulsion
19:     end parfor
20:     parfor  $m = 1 \rightarrow |V|$  do
21:       parfor  $w \in \text{NEIGHBORS}(v_m)$  do
22:          $\mathbf{f}_{v_m} \leftarrow \mathbf{f}_{v_m} + (\mathbf{p}_w - \mathbf{p}_{v_m})$  ▷ Attraction
23:       end parfor
24:     end parfor
25:   end parfor
26:   for  $j = 2 \rightarrow n$  do ▷ Reduction: Aggregate results to master
27:     Transfer  $\{\mathbf{f}_{v_m} \mid m \in [(j - 1) \times \lfloor |V|/n \rfloor, \max(j \times \lfloor |V|/n \rfloor, |V|)]\}$  from  $GPU_j$  to  $GPU_1$ 
28:   end for
29:   Update  $\mathbf{f}$  on  $GPU_1$  with  $\mathbf{f}_v$ 
▷ Update speed and displacement on master GPU
30:    $global\_speed \leftarrow \text{UPDATEGLOBALSPEED}()$ 
31:   parfor  $v \in V$  do
32:      $\mathbf{p}_v \leftarrow \text{LOCAL\_SPEED}(v) * \mathbf{f}_v$  ▷ Displacement
33:      $\mathbf{f}'_v \leftarrow \mathbf{f}_v$ 
34:      $\mathbf{f}_v \leftarrow (0.0, 0.0)^\top$ 
35:   end parfor
36:   for  $j = 2 \rightarrow n$  do ▷ Broadcast node positions to all GPUs
37:     Transfer  $\mathbf{p}, \mathbf{f}, \mathbf{f}'$  from  $GPU_1$  to  $GPU_j$ 
38:   end for
39: end for

```

pointed, and mapped the left mouse button to the button on the back of the WiiMote, which is controlled using the index finger. Buttons on the top of WiiMote were configured to control zooming, the opacity of nodes and edges, and the two interaction modes described in the preceding paragraph.

5. Experiments

In order to evaluate the effectiveness of our distributed implementation of the network visualization code and the tiled visualization approach proposed in Section 4, we conducted a number of experiments. After describing the experimental setup in Section 5.1, we first evaluate the effectiveness of the tiled visualization approach and in situ rendering in Section 5.2. Then, in Section 5.3 we evaluate the performance and scalability of the developed distributed ForceAtlas2 and network renderer implementations. Finally in Section 5.4 we assess the complete system in which the distributed visualization code and tiled visualization approach are combined. Findings regarding interaction are listed in Section 5.5. Finally, in Section 5.6 we relate to previous work and consider directions for future research.

5.1. Experimental setup

Our experimental setup consists of the ‘BigEye’ tiled display system, which is our target system for this study and is depicted in Fig. 3, consists of twelve 47" Philips BDL4777XL monitors with a resolution of 1920×1080 pixels, that are arranged in a 3×4 grid. The four monitors in each column are connected to a single NVIDIA GeForce GTX660 graphics card via DVI-I, DVI-D, HDMI and DisplayPort connections. Each graphics card is equipped with 1999MiB of GDDR5 memory. An MSI Big Bang-Marshall (MS-7670) motherboard hosts the three graphics cards, which are connected to it using the PCI Express v2 bus, with 8 PCIe lanes available to each card. Note that the graphics cards are not interconnected via other means, such as an NVIDIA SLI bridge.

The NVIDIA GeForce GTX660 GPU implements the NVIDIA Kepler microarchitecture and is composed of 960 CUDA cores. The base clock frequency is 980 MHz and the chip can attain a peak clock frequency of 1098 MHz. The system is equipped with an Intel Core i7-2600K CPU, which consists of 4 cores (8 threads), 8 MiB shared L3 cache and is clocked at 3.4 GHz with a boost clock of 3.8 GHz. Further, 16 GiB of DDR (1333 MHz) memory is available.

The network datasets used in our experiments stem from well-known sources commonly used in network science, in particular the

Table 1

Obtained framerates (fps) for Mandelbrot visualization on BigEye, using both the single-GPU Xinerama baseline and the three-GPU tiled approach.

	Baseline	Tiled approach (speedup)
Worst case	4.4	25.2 (5.7×)
Average case	6.5	47.9 (7.3×)

KONECT [5] (<http://konect.cc>) and SNAP [45] (<https://snap.stanford.edu/data>) repositories. The datasets span a range of different scientific domains, and include for example social networks, scientific collaboration networks, road networks, protein interaction networks and communication networks. For each network used, we list basic topological properties (as discussed in Section 2.1) in Table 2. All properties were computed using NetworkX (version 2.1) [46]. These properties demonstrate how we experiment with networks of varying size in terms of number of nodes and edges, as well as varying sparseness measured through the average degree. In addition, we analyze a number of extremely large network datasets (with up to 3 million nodes and 117 million edges) in a specific set of scalability experiments in Sections 5.3.2 and 5.3.3, for which properties are listed in Table A.5. The software installation consists of Ubuntu 16.04.3 LTS, which is configured to use the Xfce Desktop Environment and a regular X implementation. Version 9.2.148 of the CUDA toolkit is installed, and the proprietary NVIDIA driver (version 396.37) is used.

For our experiments of the multi-GPU implementation's scalability, we use a dedicated machine that is equipped with six NVIDIA GeForce GTX 980Ti GPUs. These GPUs are based on the NVIDIA Maxwell microarchitecture and each comprise 2816 CUDA cores running at a default base block frequency of 1000 MHz. The software installation of this system consists of CentOS 7.5, version 396.26 of the proprietary NVIDIA driver and version 9.2.88 of the CUDA Toolkit.

5.2. Effectiveness of tiled visualization

In this section, we evaluate our implementation of the tiled visualization approach described in Section 4.3 in comparison to a standard visualization implementation that runs across all monitors of the BigEye system by means of Xinerama. When Xinerama is used, one GPU generates the entire visualization which is subsequently transferred over the PCIe bus for display on the monitors connected to other GPUs. We therefore hypothesize that our tiled visualization, should reduce PCIe bus traffic and improve framerates. Additionally, we expect our efforts to achieve synchronization, as described in Section 4.3.2 to achieve a coherent image across all monitors without tearing artifacts occurring as a result of moving imagery. In order to perform these comparisons we use an application that computes and visualizes Mandelbrot [47] on the GPU, that we have modified to implement our tiled approach. Mandelbrot is a straightforward embarrassingly parallel application, which allows us to evaluate the tiled visualization approach without external influences.

Regarding improvement in framerate, we consider both the worst- and average-case framerates achieved by both implementations. The worst-case performance is determined by viewing a part of the Mandelbrot set that requires iteration up to n_{max} for all pixels. We approximate the average-case framerate by displaying the whole Mandelbrot set at a particular setting. Note that we disable synchronization between the OpenGL bufferswap and the vertical refresh of the monitors for both implementations. Table 1 provides our results.

As the results demonstrate, the tiled visualization approach improves performance significantly. We observe an average performance improvement of 6.5× between the two cases. The average case framerate of 48 fps enables more responsiveness than the framerate of 7 fps achieved using the baseline Xinerama implementation. These performance improvements are in part due to using all GPUs in the

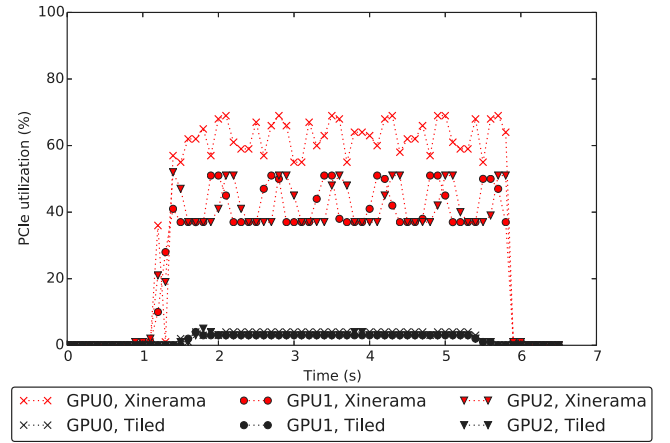


Fig. 9. PCIe utilization for each GPU during two runs of the Mandelbrot visualization, once using Xinerama and once using the tiled visualization approach.

system to compute the Mandelbrot drawings, without requiring inter-GPU communication over the PCIe bus to display resulting drawings on all monitors. To determine whether the inter-GPU communication has indeed reduced, we compare the PCIe utilization of each GPU for both implementations. We obtain this metric using `nvidia-settings` [48] which reports it as part of the `GPUUtilization` attribute. The results are depicted in Fig. 9. As can be seen in the figure, in the case of Xinerama all GPUs show a PCIe utilization between 40 and 60%. Our tiled visualization approach reduces this to 5% for all GPUs, signifying that the inter-GPU communication has almost been fully eliminated and as such contributing to the improved framerate.

To evaluate whether our approach prevents tearing artifacts between different monitors, we synchronize the calls that initiate the OpenGL bufferswap for each of the GPUs. As discussed, this approach leaves room for improvement since it only synchronizes the initiation of the bufferswap process on the CPU rather than the actual bufferswap on the GPU. Still, our approach significantly reduces tearing artifacts compared to the implementation using Xinerama. When operating at high framerates, only slight tearing artifacts between different monitors could be observed. Tearing artifacts can be more significant when framerates were reduced. The latter is an additional motivation to disable the synchronization of the OpenGL bufferswap and the vertical refresh of selected monitors.

5.3. Distributed network visualization using multiple GPUs

To assess the effectiveness of the distributed ForceAtlas2 GPU implementation and distributed network renderer described in Section 4.4, we first establish the baseline performance for the GPU implementation of ForceAtlas2, as described in Section 2.4, and the GPU accelerated network rendered, described in Section 4.2. After that, we analyze the workload composition of the algorithm when run on a single GTX660 GPU compared to on two GTX660 GPUs. Subsequently, we describe a number of scaling experiments on the dedicated multi-GPU machine equipped with six 980Ti GPUs. The aim of these experiments is to establish to what extent the distributed algorithms can improve the performance when more GPUs are utilized. Because the peak performance of the GPUs in this system is significantly greater than the peak performance of the GPUs installed in BigEye, we do also consider a set of larger networks for these experiments.

5.3.1. Baseline GPU accelerated layout and rendering performance

The baseline experiments were conducted on BigEye using a single NVIDIA GTX660 graphics card. We use a range of real-world networks for our experiments, which are further detailed in Appendix A. We

Table 2

Network datasets, their topological properties (number of nodes, edges and average degree) and running time on a single GTX660 GPU (the ForceAtlas2 layout (t_{fa2}), drawing nodes (t_{rn}), drawing edges (t_{re}), and total running time (t_{total})). The final two columns denote the fraction of layout time spent in the Force Approximation step for single GPU and for dual GPU runs. Times are in milliseconds, and averaged over the first 500 iterations. Standard deviations are denoted between parentheses and the dashed line indicates the real-time threshold. Network data was obtained from KONECT [5] and SNAP [45] (see Section 5.1).

Network	Nodes	Edges	Avg. deg.	t_{fa2}	t_{rn}	t_{re}	t_{total}	t_{FA} single GPU	t_{FA} two GPUs
ppi_dip_swiss	3,766	11,922	6.3	14.16 (0.93)	0.52 (0.06)	22.35 (7.67)	37.03	1.24 (76.08%)	1.17 (74.32%)
petster	1,788	12,475	14.0	3.84 (0.32)	0.21 (0.03)	2.90 (1.38)	6.95	0.91 (77.41%)	0.87 (76.25%)
CA-GrQc	4,158	13,422	6.5	1.79 (0.09)	0.13 (0.03)	1.92 (0.62)	3.85	1.26 (75.95%)	1.19 (74.18%)
PGPgiantcompo	10,680	24,316	4.6	24.26 (1.12)	0.78 (0.08)	42.26 (5.57)	67.30	2.80 (77.33%)	1.48 (63.29%)
ca-HepTh	8,638	24,806	5.7	3.68 (0.19)	0.19 (0.03)	4.44 (1.17)	8.31	2.83 (80.06%)	1.49 (66.68%)
dip	19,928	41,202	4.1	7.85 (0.36)	0.32 (0.04)	8.28 (2.14)	16.45	6.20 (81.98%)	3.38 (70.36%)
Newman-Cond_mat	22,015	58,578	5.3	9.64 (0.23)	0.34 (0.05)	16.14 (2.19)	26.12	7.96 (84.59%)	4.06 (72.95%)
ca-CondMat	21,363	91,286	8.5	8.28 (0.39)	0.33 (0.04)	15.85 (4.44)	24.46	6.55 (80.99%)	3.34 (67.70%)
wiki-Vote	7,066	100,735	28.5	3.33 (0.21)	0.17 (0.03)	32.77 (4.10)	36.26	2.38 (74.94%)	1.36 (62.35%)
ca-HepPh	11,204	117,619	21.0	4.64 (0.19)	0.21 (0.02)	26.27 (5.06)	31.13	3.45 (76.93%)	1.87 (63.71%)
ppi	37,333	135,618	7.3	1.82 (0.05)	0.13 (0.03)	3.67 (0.44)	5.62	11.20 (80.44%)	5.92 (67.62%)
p2p-Gnutella31	62,561	147,877	4.7	1.34 (0.05)	0.11 (0.03)	3.99 (0.42)	5.44	19.39 (81.47%)	9.80 (68.05%)
GoogleNw	15,763	148,585	18.9	6.50 (0.35)	0.26 (0.03)	42.64 (10.11)	49.40	4.88 (78.86%)	2.94 (68.63%)
email-Enron	33,696	180,811	10.7	12.56 (1.13)	0.46 (0.04)	44.89 (9.98)	57.91	9.62 (78.72%)	4.98 (64.77%)
ca-AstroPh	17,903	196,972	22.0	7.22 (0.25)	0.29 (0.03)	43.51 (7.70)	51.02	5.50 (78.34%)	3.28 (67.56%)
Brightkite	56,739	212,945	7.5	22.80 (1.22)	0.75 (0.07)	44.64 (9.65)	68.18	18.20 (81.41%)	9.73 (69.18%)
email-EuAll	224,832	339,924	3.0	100.6 (25.76)	2.73 (0.11)	87.69 (37.80)	191.06	86.19 (83.79%)	42.85 (71.49%)
Cit-HepTh	27,400	352,021	25.7	11.60 (0.40)	0.41 (0.05)	74.53 (16.44)	86.54	9.06 (79.70%)	5.13 (68.17%)
soc-Epinions1	75,877	405,738	10.7	30.90 (1.87)	0.93 (0.09)	123.24 (19.09)	155.08	24.37 (80.00%)	12.34 (66.15%)
Cit-HepPh	34,401	420,784	24.5	14.26 (0.55)	0.49 (0.05)	63.14 (19.03)	77.88	11.40 (81.44%)	5.86 (68.47%)
soc-Slashdot0902	82,168	504,230	12.3	35.62 (1.40)	1.02 (0.08)	182.22 (12.45)	218.86	27.94 (79.62%)	14.58 (66.31%)

measured the average iteration duration over the first 500 iterations of the display loop (Fig. 5(b)), setting $f_r = 80$, $f_g = 1$ and $\theta = 1$ with the gravity setting of ForceAtlas2 set to ‘strong’.² Drawings were rendered at a resolution of 5760×4320 pixels, matching the resolution of the BigEye tiled display system we use for this study. Note that we do not display drawings, but only consider the time required to generate them. Table 2 presents the results, in addition to common topological properties of the network datasets, as discussed in Section 2.1.

The results in Table 2 demonstrate that the majority of the time in the display loop is spent on computing the layout (t_{fa2}) and on drawing the edges (t_{re}). The time required to draw the nodes (t_{rn}) is small, taking only a few percent of the total iteration time for most networks. We did not expect the edge drawing time to constitute a significant part of the total visualization time. For future implementations of the network renderer we would consider the use of OpenGL triangle primitives to draw the edges, instead of using the provided line primitives. Given the prevalence of triangle drawing in most 3D computer graphics applications, we hypothesize that this primitive might yield better performance than the line primitive.

5.3.2. Workload composition of distributed ForceAtlas2

As Table 2 displayed, a significant amount of the time is spent on computing the layout. In order to characterize the workload composition of the layout computation, we measured the running times for individual components of the layout algorithm, using the NVIDIA nvprof profiler tool [49], for a number of datasets. The results for the baseline single-GPU implementation demonstrate that force approximation using the Barnes–Hut algorithm accounts for approximately 80% of the running time, as displayed in the second to last column of Table 2. Time spent on the other components is listed in Table B.6, demonstrating how speed-update and node displacement comprise a small fraction of the total iteration duration and were therefore not considered for distribution over multiple GPUs.

To determine how the multi-GPU implementation of ForceAtlas2 affects the fraction of time spent in different components of the algorithm, the last column of Table 2 presents the results for the multi-GPU implementation running on two of the GPUs in BigEye. As can be

seen, although the time spent in the force approximation component decreased from 80% to 68% on average, compared to the single GPU results, it still comprises the majority of the running time for a single iteration of ForceAtlas2.

5.3.3. Scalability of distributed ForceAtlas2 implementation

In our experiments we measure the speedup of the average iteration duration, over the first 500 iterations of the layout algorithm, using up to six GPUs. We use the same algorithm settings as before: $f_r = 80$, $f_g = 1$, $\theta = 1$ and set the gravity setting of ForceAtlas2 to ‘strong’. Fig. 13 depicts the speedups we observed, as well as the speedup that would result from a linear speedup in the force approximation component that we distribute across multiple GPUs. The latter is derived from the average fraction of time spent on force approximation, across the different networks we evaluate, and denoted as ‘Linear Speedup of 82.91%’.

As our results in Fig. 13 depict, the speedup of ForceAtlas2 we obtained for the different networks generally follows the trend corresponding to the speedup that would follow from a linear speedup of the force approximation component. The speedup gained by using additional GPUs diminishes as the number of GPUs increases, given that parts of ForceAtlas2 other than the multi-GPU force approximation start to dominate the layout time. Although this limits the scalability of our approach, this effect is less severe for our intended use-case with three GPUs. The obtained speedups generally increase with network size, which can be explained as the result of an increasing degree of parallelism being available to the system. Still, the ‘wiki’ and ‘orkut’ networks (see Appendix A for detail), the two largest we consider used for our experiments, yield small speedups given their size. We attribute this to the relatively large number of edges in these networks, i.e. their higher average degree, compared to the other networks. Indeed, as the single-GPU evaluation in Fig. 10 shows, the ‘wiki’ and ‘orkut’ networks spend a larger proportion of the layout time on computing attractive forces between neighboring nodes. Given that only the repulsive force calculation is distributed between different GPUs, the speedup we obtain will be less for networks that spend more time in other components.

5.3.4. Scalability of distributed network rendering

We evaluate the scalability of our distributed rendering approach using the same machine used to evaluate the distributed implementation of ForceAtlas2. The same set of networks is used, and we again

² The gravity setting is a ForceAtlas2-specific setting that allows the user to adapt the force-model employed by the algorithm. For more details we refer the reader to [15].

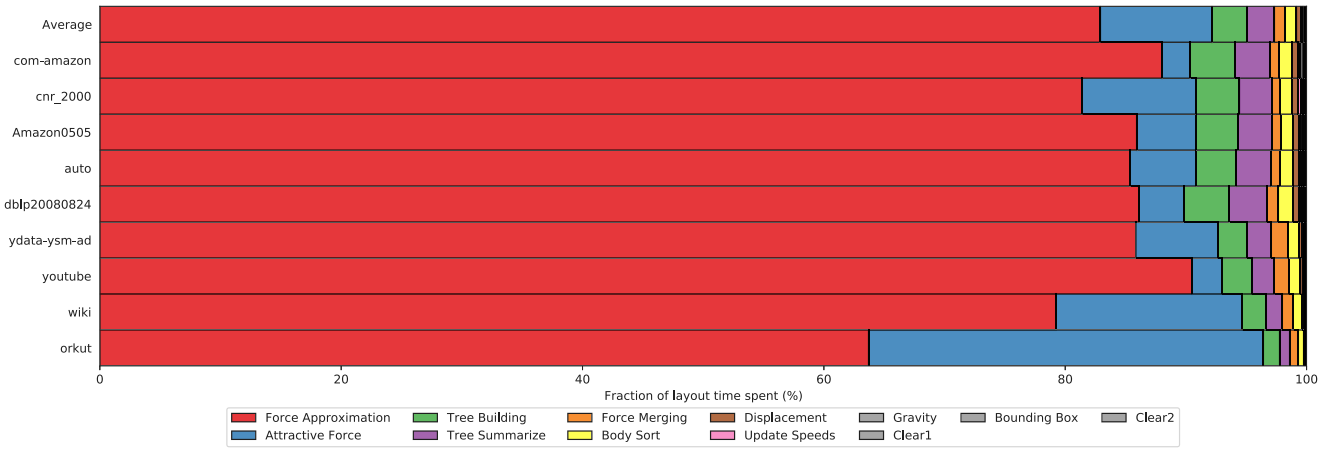


Fig. 10. Fraction of the time spent in different components of ForceAtlas2 for the single GPU implementation. Derived from average running times for the different components over the first 500 iterations. Colors in the legend below the figure each denote phases of Algorithm 2 as explained in Section 4.4.

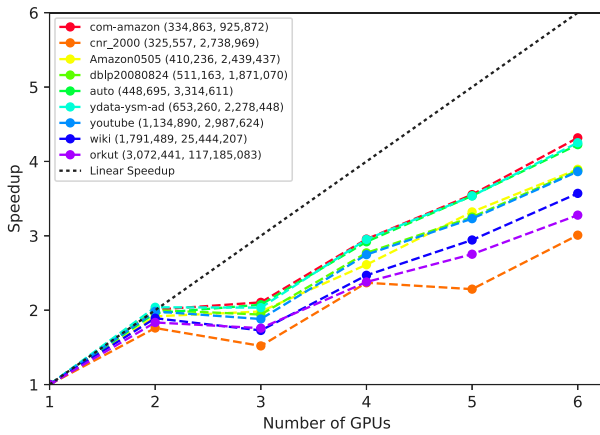


Fig. 11. Performance scaling of our multi-GPU network renderer, on a machine equipped with six NVIDIA GeForce GTX980Ti GPUs. Legend denotes network name and $(|V|, |E|)$.

determine the speedup for the average iteration duration over the first 500 layout steps, setting $f_r = 80$, $f_a = 1$, $\theta = 1$ and the gravity mode to ‘strong’. During the layout process, we adapt the view after each iteration to ensure that the layout continues to span the image which we render. This is important to, for example, ensure that no parts of the layout get clipped and discarded, which would affect performance.

As our results in Fig. 11 depict, performance scales up to six GPUs without significant saturation for most datasets. Notable are the reduced speedups when using uneven numbers of GPUs. This effect is strongest when using three GPUs, and it diminishes as the number of GPUs increases. We explain this as the result of sub-optimal load balancing between the GPUs. When using an odd number of GPUs, the layout space is partitioned into an odd number of columns. As a result, the center of the layout will be assigned to a single GPU for rendering. Since the layouts that are generated are circular, there might be a tendency for most edges to cross the center of the layout, which would result in an increased amount of work being assigned to the GPU rendering the center of the layout.

Since the majority of the network rendering time consists of drawing the network’s edges, see Table 2, we would expect networks with many edges to benefit especially from a multi-GPU implementation of the network renderer. This is not the pattern revealed by Fig. 11. To investigate the cause of this, we considered the layouts for the ‘com-amazon’ and ‘cnr_200’ networks, since these are the networks that benefit the most, and the least from a multi-GPU implementation,

respectively. Fig. 12 depicts the layouts for both networks. Clearly, the ‘cnr_2000’ network has more larger clusters than the ‘com-amazon’ network, whereas the ‘com-amazon’ network resulted in a layout that is not readable, but is uniform. This might explain the scalability differences for these networks, as the lack of clusters in the ‘com-amazon’ network results in more uniform load-distribution between GPUs.

5.4. Network visualization on a tiled display system

The preceding experiments show reasonable to good performance scaling of our distributed network visualization implementation when evaluated on six GPUs in dedicated multi-GPU machine. In this section, we evaluate the combined effect of our distributed ForceAtlas2 implementation and network renderer on BigEye, using the same set of real-world networks. Subsequently, this is embedded into the tiled visualization approach in order to evaluate the final system.

5.4.1. Distributed network visualization on BigEye

The combined performance improvements provided by distributed visualization approach on BigEye, involving both distributed layout and rendering, are presented in Table 3. For our experiments we used all of the three GPUs in BigEye and the same set of real-world networks used when determining the baseline performance using a single GPU, of which the results are presented in Table 2. Note that we now enable peer-to-peer data transfers between the GPUs in BigEye, thus bypassing CPU memory. This was not possible between all of the GPUs used for the scalability experiments, which is why we only consider this option at this point.

As shown in Table 3, the multi-GPU approach provides speedups for all networks we evaluated. As the network size increases, a speedup of approximately $1.7\times$ is realized. Consequently, this extends the set of networks that can be visualized in real-time with the ‘ca-CondMat’, ‘Newman-Cond_mat’ and ‘ca-HepPh’ networks.

5.4.2. Tiled network visualization

As a final step, we embed the distributed network visualization in our tiled visualization approach and evaluate whether it meets the demands of real-time visualization for a range of real-world networks. With real-time visualization, the displayed visualization reflects user input with the next monitor refresh, making interactive exploration a reality. In the case of BigEye, this corresponds to achieving a frame-rate of 60 frames per second (fps). This bounds the time available to generate a frame to approximately 16.67 ms and as such the combined running time of the ForceAtlas2 layout algorithm and the network renderer must remain below 16.67 ms. Note that this bound is lower

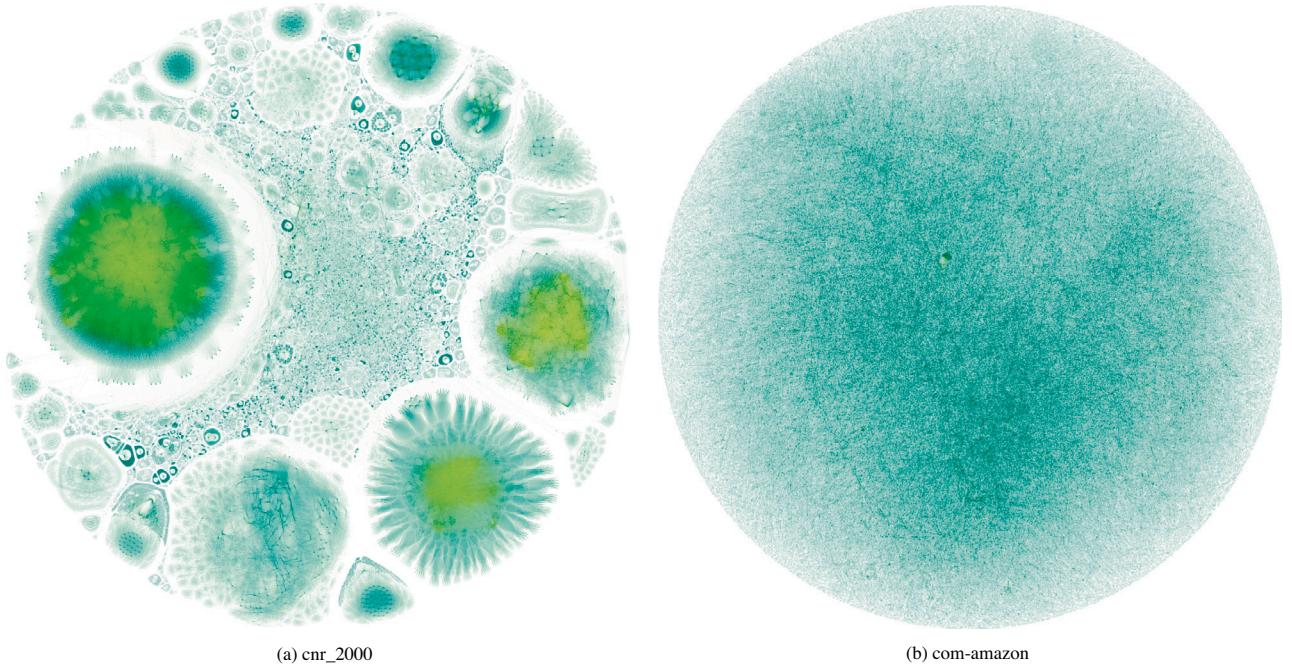


Fig. 12. Layouts for two datasets, where (a) benefited the least from the multi-GPU implementation and (b) the most. Saturation corresponds to node degree.

Table 3

Average layout (t_{fa2}), render (t_{render}) and total iteration times (t_{total}) obtained using our multi-GPU visualization approach with three GPUs. Times are in milliseconds, speedups are between parentheses and in comparison to the initial single GPU implementation. Real-time threshold indicated by dashed line.

Network	Nodes	Edges	t_{fa2} (Speedup)	t_{render} (Speedup)	t_{total} (Speedup)
CA-GrQc	4,158	13,422	2.00 (0.9x)	1.39 (1.5x)	3.39 (1.1x)
petster	1,788	12,475	1.62 (0.8x)	2.75 (1.5x)	4.37 (1.2x)
ppi_dip_swiss	3,766	11,922	2.00 (0.9x)	2.40 (1.6x)	4.39 (1.3x)
PGPgiantcompo	10,680	24,316	2.73 (1.4x)	1.95 (1.6x)	4.68 (1.5x)
ca-HepTh	8,638	24,806	2.63 (1.4x)	2.65 (1.7x)	5.28 (1.6x)
dip	19,928	41,202	4.99 (1.6x)	5.23 (1.6x)	10.21 (1.6x)
ca-CondMat	21,363	91,286	5.39 (1.5x)	9.33 (1.7x)	14.72 (1.7x)
Newman-Cond_mat	22,015	58,578	5.86 (1.6x)	8.87 (1.9x)	14.73 (1.8x)
ca-HepPh	11,204	117,619	3.45 (1.3x)	13.06 (2.0x)	16.51 (1.9x)
wiki-Vote	7,066	100,735	2.62 (1.3x)	18.91 (1.7x)	21.53 (1.7x)
ppi	37,333	135,618	7.85 (1.8x)	10.51 (2.2x)	18.36 (2.0x)
GoogleNw	15,763	148,585	3.96 (1.6x)	26.01 (1.6x)	29.97 (1.6x)
ca-AstroPh	17,903	196,972	4.57 (1.6x)	25.71 (1.7x)	30.28 (1.7x)
email-Enron	33,696	180,811	7.40 (1.7x)	26.88 (1.7x)	34.28 (1.7x)
p2p-Gnutella31	62,561	147,877	13.48 (1.8x)	24.50 (1.8x)	37.99 (1.8x)
Brightkite	56,739	212,945	12.42 (1.8x)	26.43 (1.7x)	38.84 (1.8x)
Cit-HepPh	34,401	420,784	7.69 (1.9x)	28.65 (2.2x)	36.33 (2.1x)
Cit-HepTh	27,400	352,021	6.96 (1.7x)	39.85 (1.9x)	46.80 (1.8x)
soc-Epinions1	75,877	405,738	17.30 (1.8x)	78.88 (1.6x)	96.19 (1.6x)
email-EuAll	224,832	339,924	51.27 (2.0x)	53.07 (1.7x)	104.34 (1.8x)
soc-Slashdot0902	82,168	504,230	20.05 (1.8x)	106.99 (1.7x)	127.03 (1.7x)

in practice, since the resulting drawings also need to be transferred to the framebuffer for display on the system.

We evaluate its performance in comparison to a baseline implementation, that uses a single GPU for the layout and rendering processes, and Xinerama to display resulting visualizations across all monitors of the tiled display system. The baseline system uses the implementations of the layout algorithm and the network renderer that served as the starting point for the multi-GPU variants we discussed in Section 4.4. The tiled system uses our tiled visualization approach and the multi-GPU network layout algorithm and renderer.

We evaluate both systems on BigEye, comparing the average framerate over the first 500 frames. As for previous experiments, we set $f_r = 80$, $f_g = 1$, $\theta = 1$ and the gravity setting to 'strong'. After each

layout step, we update the view such that the entire network drawing remains on the displays. Table 4 presents our results for a number of real-world networks.

As shown in Table 4, the performance improvements obtained using the tiled visualization framework, over a baseline implementation, approach 4.5x as the network size increases. This is in part due to the multi-GPU implementation of the layout algorithm and network renderer, as discussed in Section 4.4 and evaluated in Sections 5.3 and 5.3.4. However, we observe an additional speedup that can be attributed to the performance benefits of using the tiled visualization approach instead of Xinerama. Real-time visualization at 60 fps is now possible for an additional number of the networks, compared to the baseline implementation.

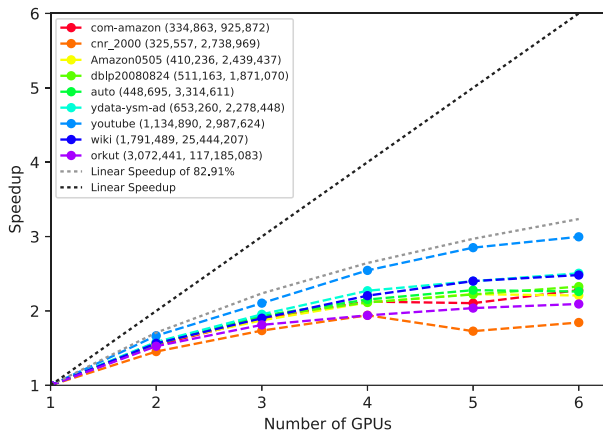


Fig. 13. Performance scaling of our multi-GPU implementation of ForceAtlas2 on a machine equipped with six NVIDIA GeForce GTX980Ti GPUs. Legend denotes network name and $(|V|, |E|)$.

Table 4

Average framerates (in fps), over the first 500 frames, for the tiled visualization approach including use of the multi-GPU network layout algorithm and renderer, in comparison to the baseline (BL) implementation using Xinerama and single-GPU layouting and redrawing.

Network	Nodes	Edges	BL	Speedup
CA-GrQc	4,158	13,422	82.04	229.0 (2.8×)
petster	1,788	12,475	64.93	201.0 (3.1×)
ppi_dip_swiss	3,766	11,922	64.91	182.7 (2.8×)
PGPgiantcompo	10,680	24,316	61.29	168.4 (2.7×)
ca-HepTh	8,638	24,806	51.10	155.9 (3.1×)
dip	19,928	41,202	28.07	88.28 (3.1×)
Newman-Cond_mat	22,015	58,578	15.86	62.34 (3.9×)
ca-CondMat	21,363	91,286	16.67	62.26 (3.7×)
ppi	37,333	135,618	11.30	49.85 (4.4×)
ca-HepPh	11,204	117,619	11.69	46.38 (4.0×)
wiki-Vote	7,066	100,735	9.38	45.81 (4.9×)
GoogleNw	15,763	148,585	7.05	33.10 (4.7×)
ca-AstroPh	17,903	196,972	6.86	31.85 (4.6×)
email-Enron	33,696	180,811	6.73	31.01 (4.6×)
p2p-Gnutella31	62,561	147,877	6.20	25.68 (4.1×)
Brightkite	56,739	212,945	5.90	24.37 (4.1×)
Cit-HepPh	34,401	420,784	4.17	23.96 (5.8×)
Cit-HepTh	27,400	352,021	4.03	21.14 (5.2×)
soc-Epinions1	75,877	405,738	2.39	10.74 (4.5×)
email-EuAll	224,832	339,924	2.58	9.64 (3.7×)
soc-Slashdot0902	82,168	504,230	1.68	7.13 (4.2×)

5.5. Interaction

The tiled visualization approach enabled performance levels suitable to real-time interaction for a number of the networks we evaluated. The interactive capabilities of the resulting system are assessed by implementing a number of interactions using a Nintendo Wii remote. Since an evaluation of the human computer interaction (HCI) aspects related to these interactions is beyond the scope of this study, which focuses rather on the technical challenges related to facilitating interactivity, we do not consider any usability experiments. Still, for the reader's information, we will report on our personal observations, and provide a number of videos demonstrating our results on the supporting web page <https://liacs.leidenuniv.nl/~rietveldkfd/tiledvis>.

In our experience the WiiMote was effective at allowing users to navigate through networks. By means of simple point-and-drag gestures most users of the system were able to navigate around large networks, to reveal and inspect its substructures. We perceived that the wireless connectivity provided by the controller, enabling one to freely position oneself in front of the displays, was both effective and natural whilst exploring networks. Positioning oneself at distance of the displays

improved the contrast between different parts of the network, causing the overall structure of the network to be better perceivable, most likely due to the blending of fine structures. The ability to interactively adjust the opacity of nodes and edges also proved to be valuable in revealing the structure of the network. This is illustrated in Fig. 14, which depicts how the structure of dense parts layouts can be revealed by reducing the opacity of edges.

Regarding the two interaction modes, we found that the local repulsion mode indeed allowed users to force parts of the layout to converge to a different configuration. Based on our specific experience it is not possible to conclude whether this enables better layouts in general. However, we did observe an interesting, and unintended, use for the local repulsion mode. As illustrated in Fig. 15, the local repulsion mode can be used to arrange nodes based on their degree. When positioned at the origin of the layout, the local repulsion mode imposes a 'sorting' on the layout, forcing low-degree nodes to the periphery of the layout. This likely results from the combined effects of gravitational forces and the local repulsive force. Local repulsion forces nodes away from the origin of the layout, whereas the gravitational force moves nodes towards the origin, but with a magnitude proportional to the node's degree. The magnitude by which nodes are forced to the periphery of the layout is thus based on their degree. If a network consists of different components, the components with a similar topology, are likely to be positioned in proximity of each other.

The local heat mode proved to be less intuitive to use. Correctly configuring the temperature offset to be used near the mouse was important to prevent chaotic displacements of all nodes. Once a correct setting was found, nodes near the pointer were indeed displaced by a larger distance than other nodes in the network. However, in our experience this had no clear effect on the layout quality in this part of the network.

5.6. Discussion and future research

We consider our results in relation to the previous work on interactive network visualization using tiled display systems that was discussed in Section 3. We specifically consider the work by Mueller et al. [8], which also describes a general-purpose interactive network visualization system.

Mueller et al. consider an eight node cluster-based tiled display system composed of eight monitors, whereas we consider a single-node system with three GPUs connected to twelve monitors for the present study. Although Mueller et al. do not report the resolution of their system, the photos in the paper do not suggest it exceeds the 25 megapixel resolution of the system considered for the present study. Mueller et al. report framerates of up to 5 frames per second (fps) for randomly generated graphs with 8000 nodes and (approximately) 80,000 edges. For a real-world graph of similar size we achieve a framerate of approximately 60 fps. This corresponds to a performance improvement of 12×, at the same (but most likely higher) resolution, using only a single node with three graphics cards. We should note this improvement is in part due to the more recent hardware used in the present study, which came to market almost a decade after the study.

The tiled visualization approach enabled performance levels suitable to real-time interaction for a number of the networks that we evaluated. We successfully exploited this possibility in our assessment using the WiiMote. We consider our results a proof-of-concept, which is not directly applicable for data-exploration and -analysis. This is mainly due to the limited number of interactions we implemented. In the context of the standard 'overview first, zoom and filter, then details on demand' information seeking mantra [4], we now provide overview for larger networks, but still lack filter and details on demand. However, we expect the implementation of these to be significantly less complicated than the performance-related challenges focused on in this paper, and applicable in the framework we presented. In general we expect our findings to translate to larger networks on more recent

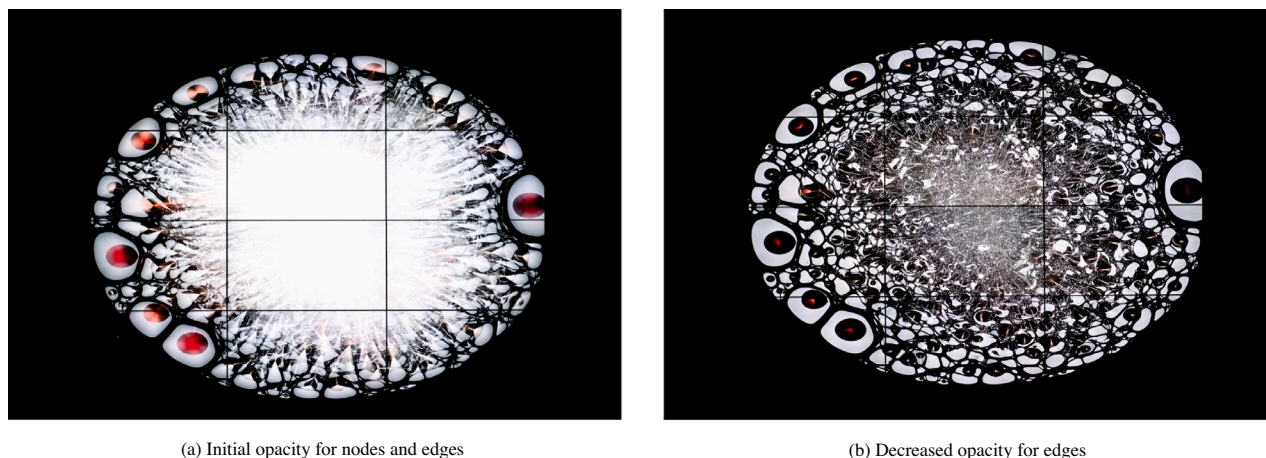


Fig. 14. Interactively adjusting the opacity of nodes and edges reveals modular network structure.

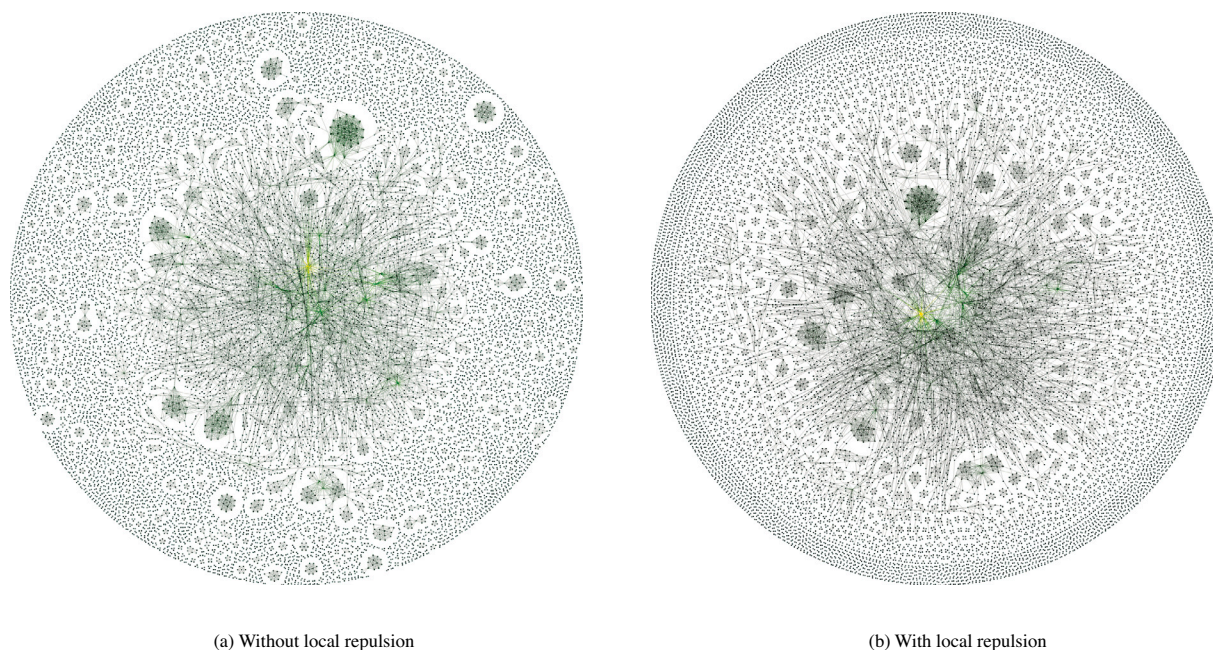


Fig. 15. The same network visualized with and without a local repulsive force at the origin of the layout. Saturation of node color corresponds to degree.

graphics cards and GPUs, which provide factors of additional compute units and memory bandwidth. This may further advance the real-time threshold as well. To conclude this section, we indicate a number of areas where we expect further improvements can be made.

For future work on the distributed ForceAtlas2 implementation, we would initially consider improving the performance of the force approximation component. As Table 2 reveals, the force approximation component of ForceAtlas2 continues to comprise the majority of the iteration duration, even after parallelization using multiple GPUs. Further performance improvements might also be realized by distributing additional ForceAtlas2 components, besides the force approximation computation, across multiple GPUs. As the scalability results in Fig. 13 shows, this is crucial to ensure performance scales when using increasing numbers of GPUs. Distributing the computation of attractive forces between different GPUs might be considered first to account for dense networks. Finally, alternative force approximation approaches could be considered to improve performance. For example, the Bonsai code [50],

which is designed to utilize multiple GPUs. However, such codes might need to be adapted and optimized for operation on a 2-d plane instead of in 3-d space.

We suggest future work on the network rendering implementation to focus on two problems. First, the factors limiting the performance of the edge drawing implementation could be assessed. We hypothesize that performance might be improved by using OpenGL triangle primitives instead of line primitives to draw the edges, given the prevalence of this procedure in most computer graphics applications. Second, the multi-GPU implementation could be improved by assuring a more uniform work distribution between the different GPUs. Given our results, we would consider partitioning the render work based on graph topology, e.g. by assigning individual edges and nodes to GPUs, rather than through the spatial partitioning we employed. The effect this would have on inter-GPU communication would need to be considered.

The employed tiled visualization approach has to synchronize updates to the different monitors to ensure a coherent image spanning all monitors. To this end we synchronize the initiation of the OpenGL buffer swap procedure for different GPUs, on the CPU. As discussed this is a sub-optimal approach, since it synchronizes the initiation of the buffer swap process rather than the actual buffer swap. Given the importance of image coherency we deem improvements in this area valuable.

With the purpose of evaluating the interactive capabilities of our system, we implemented a number of interactions, enabling users of the system to navigate the network and to locally destabilize it through the ‘local repulse’ and ‘local heat’ modes. There are various other interactions that we did not consider here, which would yield a significant improvement in the system’s data-exploration capabilities, based on our experience with the system and users’ comments. A user may wish to obtain properties of selected nodes and edges in the networks, for example through pop-up menu’s or node labels providing information on selected nodes, or by highlighting the nodes and edges connected to a selected node. The possibility to filter for nodes or edges with certain properties, or to color and size them based on these properties, would also be valuable for the purposes of data-exploration. To improve users’ sense of context, i.e., what part of the drawing they are viewing, a map of the network drawing indicating the part that is viewed could be superimposed in one of the display’s corners.

6. Conclusion

In this work we presented an approach to using the graphics processing units (GPUs) in a tiled display system for interactive network visualization at high resolutions. We showed how using the GPU as a platform for both network layout and network drawing allows for high performance levels. Our method extends an existing GPU implementation of force-directed graph layout to be capable of distributing the work over multiple GPUs as well as proposed a distributed rendering approach in which each GPU in the system draws the part of the network to be displayed on the monitors attached to it. An evaluation of our approach demonstrated real-time performance at 60 frames per second for networks with tens of thousands of nodes and edges on a 25 megapixel tiled display system with twelve monitors and three NVIDIA GeForce GTX660 graphics cards. This constitutes a performance improvement of approximately 4× over the standard single GPU implementation that served as the starting point for our multi-GPU approach. We were furthermore able to successfully implement real-time navigation and a number of interactions with the layouts using a Nintendo Wii remote as input device.

Although our results are promising, two main challenges remain. First, the performance of our multi-GPU network visualization approach could be improved by optimizing the time spent on repulsive force approximation, by reducing the number of non-distributed components in the layout algorithm, and by optimizing the approach to render the network’s edges. Second, synchronous updates between the different monitors in the tiled display system would resolve the remaining discontinuities that currently appear, especially if frame rates decrease as result of increased network sizes. We believe these problems can be resolved without a major revision of the framework presented in this study, and as such we expect future work to be effective at scaling the approach to even larger real-world networks.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Network properties

See [Table A.5](#).

Table A.5

Properties of the network datasets used. Original data, obtained from well-known network repositories KONECT [5] and SNAP [45], was filtered for the largest connected component, self-loops were removed and the resulting graph was considered to be undirected. “Avg. deg.” denotes “average degree”.

Dataset	Nodes	Edges	Density	Avg. deg.
Amazon0505	410,236	2,439,437	0.0000	11.9
Brightkite	56,739	212,945	0.0001	7.5
CA-GrQc	4,158	13,422	0.0016	6.5
Cit-HepPh	34,401	420,784	0.0007	24.5
Cit-HepTh	27,400	352,021	0.0009	25.7
GoogleNw	15,763	148,585	0.0012	18.9
Newman-Cond_mat	22,015	58,578	0.0002	5.3
PGPgiantcompo	10,680	24,316	0.0004	4.6
auto	448,695	3,314,611	0.0000	14.8
ca-AstroPh	17,903	196,972	0.0012	22.0
ca-CondMat	21,363	91,286	0.0004	8.5
ca-HepPh	11,204	117,619	0.0019	21.0
ca-HepTh	8,638	24,806	0.0007	5.7
cnr_2000	325,557	2,738,969	0.0001	16.8
com-amazon	334,863	925,872	0.0000	5.5
dblp20080824	511,163	1,871,070	0.0000	7.3
dip	19,928	41,202	0.0002	4.1
email-Enron	33,696	180,811	0.0003	10.7
email-EuAll	224,832	339,924	0.0000	3.0
orkut	3,072,441	117,185,083	0.0000	76.3
p2p-Gnutella31	62,561	147,877	0.0001	4.7
petster	1,788	12,475	0.0078	14.0
p2p_dip_swiss	3,766	11,922	0.0017	6.3
ppi	37,333	135,618	0.0002	7.3
soc-Epinions1	75,877	405,738	0.0001	10.7
soc-Slashdot0902	82,168	504,230	0.0001	12.3
wiki-Vote	7,066	100,735	0.0040	28.5
wiki	1,791,489	25,444,207	0.0000	28.4
ydata-ysm-ad	653,260	2,278,448	0.0000	7.0
youtube	1,134,890	2,987,624	0.0000	5.3

Appendix B. Additional results

See [Tables B.6](#) and [B.7](#).

Table B.6

Average amount of time (in ms) spent on the three most time consuming components of the ForceAtlas2 algorithm, during a single iteration. Fractions, relative to all components, are given between parentheses. Results are averaged over the first 500 iterations, using a **single GPU**.

Dataset	Force approx.	Tree summarize	Tree building
petster	0.91 (77.41%)	0.14 (11.88%)	0.04 (3.36%)
ppi_dip_swiss	1.24 (76.08%)	0.24 (14.83%)	0.05 (3.00%)
CA-GrQc	1.26 (75.95%)	0.25 (15.21%)	0.05 (2.90%)
wiki-Vote	2.38 (74.94%)	0.49 (15.27%)	0.07 (2.19%)
ca-HepTh	2.83 (80.06%)	0.48 (13.55%)	0.09 (2.58%)
PGPgiantcompo	2.80 (77.33%)	0.58 (15.93%)	0.10 (2.67%)
ca-HepPh	3.45 (76.93%)	0.68 (15.11%)	0.10 (2.28%)
GoogleNw	4.88 (78.86%)	0.91 (14.63%)	0.13 (2.06%)
ca-AstroPh	5.50 (78.34%)	0.99 (14.15%)	0.14 (1.94%)
dip	6.20 (81.98%)	1.00 (13.19%)	0.14 (1.84%)
ca-CondMat	6.55 (80.99%)	1.12 (13.84%)	0.15 (1.80%)
Newman-Cond_mat	7.96 (84.59%)	1.04 (11.10%)	0.14 (1.54%)
Cit-HepTh	9.06 (79.70%)	1.47 (12.90%)	0.18 (1.56%)
email-Enron	9.62 (78.72%)	1.92 (15.74%)	0.21 (1.71%)
ppi	11.20 (80.44%)	2.05 (14.74%)	0.23 (1.62%)
Cit-HepPh	11.40 (81.44%)	1.62 (11.55%)	0.20 (1.45%)
Brightkite	18.20 (81.41%)	3.13 (13.98%)	0.34 (1.50%)
p2p-Gnutella31	19.39 (81.47%)	3.29 (13.84%)	0.36 (1.51%)
soc-Epinions1	24.37 (80.00%)	4.39 (14.43%)	0.46 (1.51%)
soc-Slashdot0902	27.94 (79.62%)	4.90 (13.96%)	0.50 (1.44%)
email-EuAll	86.19 (83.79%)	12.70 (12.35%)	1.63 (1.59%)
Average	79.53%	13.91%	2.00%

Table B.7

Average amount of time (in ms) spent on the three most time consuming components of the ForceAtlas2 algorithm, during a single iteration. Fractions, relative to all components, are given between braces. Results are averaged over the first 500 iterations, using **two GPUs**.

Dataset	Force approx.	Tree summarize	Tree building
petster	0.87 (76.25%)	0.14 (12.12%)	0.04 (3.42%)
ppi_dip_swiss	1.17 (74.32%)	0.24 (15.35%)	0.05 (3.12%)
CA-GrQc	1.19 (74.18%)	0.25 (15.74%)	0.05 (3.03%)
wiki-Vote	1.36 (62.35%)	0.49 (22.21%)	0.07 (3.17%)
ca-HepTh	1.49 (66.68%)	0.49 (21.85%)	0.09 (4.12%)
PGPgiantcompo	1.48 (63.29%)	0.58 (24.78%)	0.10 (4.17%)
ca-HepPh	1.87 (63.71%)	0.68 (22.96%)	0.10 (3.48%)
GoogleNw	2.94 (68.63%)	0.90 (20.90%)	0.13 (2.96%)
dip	3.38 (70.36%)	1.00 (20.76%)	0.14 (2.91%)
ca-AstroPh	3.28 (67.56%)	0.99 (20.48%)	0.14 (2.81%)
ca-CondMat	3.34 (67.70%)	1.11 (22.51%)	0.15 (2.94%)
Newman-Cond_mat	4.06 (72.95%)	1.04 (18.62%)	0.14 (2.59%)
Cit-HepTh	5.13 (68.17%)	1.47 (19.54%)	0.18 (2.36%)
email-Enron	4.98 (64.77%)	1.93 (25.09%)	0.21 (2.73%)
Cit-HepPh	5.86 (68.47%)	1.61 (18.87%)	0.20 (2.39%)
ppi	5.92 (67.62%)	2.05 (23.45%)	0.22 (2.56%)
Brightkite	9.73 (69.18%)	3.13 (22.27%)	0.34 (2.40%)
p2p-Gnutella31	9.80 (68.05%)	3.30 (22.90%)	0.36 (2.50%)
soc-Epinions1	12.34 (66.15%)	4.39 (23.54%)	0.46 (2.48%)
soc-Slashdot0902	14.58 (66.31%)	4.89 (22.24%)	0.51 (2.30%)
email-EuAll	42.85 (71.49%)	12.48 (20.82%)	1.60 (2.66%)
Average	68.49%	20.81%	2.91%

References

- [1] J. Scott, *Social Network Analysis*, Sage, 2012.
- [2] A.-L. Barabási, *Network Science*, Cambridge University Press, Cambridge, 2016.
- [3] Y. Hu, L. Shi, Visualizing large graphs, *Wiley Interdiscip. Rev. Comput. Stat.* 7 (2) (2015) 115–136.
- [4] B. Shneiderman, The eyes have it: A task by data type taxonomy for information visualizations, in: *Proceedings of the IEEE Symposium on Visual Languages*, 1996, pp. 336–343.
- [5] J. Kunegis, KONECT – the koblenz network collection, in: *Proceedings of the 22nd International World Wide Web Conference, WWW*, 2013, pp. 1343–1350.
- [6] F.W. Takes, E.M. Heemsker, Centrality in the global network of corporate control, *Soc. Netw. Anal. Min.* 6 (1) (2016) 97:1–97:18.
- [7] S. Chae, HD-GraphViz: Highly Distributed Graph Visualization on Tiled Displays (Ph.D. thesis), University of California, Irvine, 2013.
- [8] C. Mueller, D.P. Gregor, A. Lumsdaine, Distributed force-directed graph layout and visualization, in: *Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV, 2006, pp. 83–90.
- [9] J. Davis, X. Chen, Lumipoint: Multi-user laser-based interaction on large tiled displays, *Displays* 23 (5) (2002) 205–211.
- [10] J.-y. Huang, K.M. Wang, K.-W. Hsu, The frame synchronization mechanism for the multi-rendering surrounding display environment, *Displays* 25 (2–3) (2004) 89–98.
- [11] W.-J. Li, C.-C. Chang, K.-Y. Hsu, M.-D. Kuo, D.-L. Way, A PC-based distributed multiple display virtual reality system, *Displays* 22 (5) (2001) 177–181.
- [12] G.G. Brinkmann, K.F.D. Rietveld, F.W. Takes, Exploiting GPUs for fast force-directed visualization of large-scale networks, in: *Proceedings of the 46th International Conference on Parallel Processing, ICPP*, 2017, pp. 382–391.
- [13] A. Godiyal, J. Hoberock, M. Garland, J.C. Hart, Rapid multipole graph drawing on the GPU, in: *Proceedings of the 16th International Symposium on Graph Drawing*, 2008, pp. 90–101.
- [14] P. Mi, M. Sun, M. Masiane, Y. Cao, C. North, Interactive graph layout of a million nodes, *Informatics* 3 (4) (2016) 23.
- [15] M. Jacomy, T. Venturini, S. Heymann, M. Bastian, ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software, *PLoS One* 9 (6) (2014) 1–12.
- [16] H. Gibson, J. Faith, P. Vickers, A survey of two-dimensional graph layout techniques for information visualisation, *Inf. Vis.* 12 (3–4) (2013) 324–357.
- [17] M. Garland, D.B. Kirk, Understanding throughput-oriented architectures, *Commun. ACM* 53 (11) (2010) 58–66.
- [18] D. Blythe, Rise of the graphics processor, in: *Proceedings of the IEEE*, 2008, pp. 761–778.
- [19] M. Segal, K. Akeley, The OpenGL Graphics System: A Specification, The Khronos Group Inc. 2017, <https://www.khronos.org/registry/OpenGL/specs/gl/glspec41.core.pdf>. (Accessed 30 July 2018).
- [20] NVIDIA Corporation, CUDA toolkit documentation, 2018, <https://docs.nvidia.com/cuda/>. (Accessed 30 July 2018).
- [21] K.O.W. Group, in: A. Bourd (Ed.), *OpenCL Specification*, 2017, <https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf>. (Accessed 30 July 2018).
- [22] A. Godiyal, J. Hoberock, M. Garland, J.C. Hart, Rapid multipole graph drawing on the GPU, in: *Graph Drawing, 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21–24, 2008. Revised Papers*, 2008, pp. 90–101, URL http://dx.doi.org/10.1007/978-3-642-00219-9_10.
- [23] Y. Frishman, A. Tal, Multi-level graph layout on the GPU, *IEEE Trans. Vis. Comput. Graph.* 13 (6) (2007) 1310–1319.
- [24] NVIDIA Corporation, Programming guide :: CUDA toolkit documentation, 2018, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. (Accessed 30 July 2018).
- [25] M. Bastian, S. Heymann, M. Jacomy, Gephi: An open source software for exploring and manipulating networks, in: *Proceedings of the 3rd International Conference on Weblogs and Social Media, ICWSM*, 2009, p. 2.
- [26] J. Barnes, P. Hut, A hierarchical O(N log N) force-calculation algorithm, *Nature* 324 (1986) 446–449.
- [27] M. Bertscher, K. Pingali, An efficient CUDA implementation of the tree-based barnes hut n-body algorithm, in: W. mei W. Hwu (Ed.), *GPU Computing Gems Emerald Edition*, Morgan Kaufmann, 2011, pp. 75–92.
- [28] C.T.M. Forum, MPI: a message passing interface, in: *Proceedings of the International Conference on Supercomputing*, 1993, pp. 878–883.
- [29] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P.D. Kirchner, J.T. Klosowski, Chromium: A stream-processing framework for interactive rendering on clusters, in: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, 2002, pp. 693–702.
- [30] T.M.J. Fruchterman, E.M. Reingold, Graph drawing by force-directed placement, *Softw. Pract. Exp.* 21 (11) (1991) 1129–1164.
- [31] S. Chae, A. Majumder, M. Gopi, HD-GraphViz: Highly distributed graph visualization on tiled displays, in: *The 8th Indian Conference on Vision, Graphics and Image Processing, ICVGIP*, 2012, p. 43.
- [32] R. Jingai, Y. Kido, S. Date, S. Shimojo, Research note: A high resolution graph viewer for multi-monitor visualization environment, *Rev. Socionetw. Strateg.* 9 (1) (2015) 15–27.
- [33] Y. Gu, C. Wang, J. Ma, R.J. Nemiroff, D.L. Kao, Igraph: a graph-based technique for visual analytics of image and text collections, in: *Visualization and Data Analysis 2015*, San Francisco, CA, USA, 2015, February 9–11, 2015 939708.
- [34] G. Mukundan, Drawing anti-aliased circular points using OpenGL/WebGL - A circular reference, 2018, <https://www.desutryquest.com/blog/drawing-anti-aliased-circular-points-using-opengl-slash-webgl>. (Accessed 30 July 2018).
- [35] J. Leigh, A.E. Johnson, L. Renambot, T. Peterka, B. Jeong, D.J. Sandin, J. Talandis, R. Jagodic, S. Nam, H. Hur, Y. Sun, Scalable resolution display walls, *Proc. IEEE* 101 (1) (2013) 115–129.
- [36] R.W. Scheifler, X Window System Protocol X Consortium Standard X Version 11, Release 6.7, X Consortium, Inc. 2004, <https://www.x.org/docs/XProtocol/proto.pdf>. (Accessed 30 July 2018).
- [37] J. Gettys, K. Packard, The X resize, rotate and reflect extension, 2015, X.org, <https://cgit.freedesktop.org/xorg/proto/randrproto/tree/randrproto.txt>. (Accessed 16 March 2018).
- [38] NVIDIA Corporation, Offloading graphics display with RandR 1.4, 2018, https://download.nvidia.com/XFree86/Linux-x86_64/390.42/README/randr14.html. (Accessed 30 July 2018).
- [39] J. Gettys, R.W. Scheifler, C. Adams, V. Joloboff, H. Hiura, B. McMahon, R. Newman, A. Tabayoyon, G. Widener, S. Yamada, Xlib - C language X interface, 2002, <https://www.x.org/releases/X11R7.7/doc/libX11/libX11/libX11.pdf>. (Accessed 30 July 2018).
- [40] xcb.freedesktop.org, The X protocol C-language binding, 2018, (Accessed 30 July 2018).
- [41] NVIDIA Corporation, Specifying OpenGL environment variable settings, 2018, https://download.nvidia.com/XFree86/Linux-x86_64/390.42/README/openglvariables.html. (Accessed 30 July 2018).
- [42] NVIDIA Corporation, EXT_swap_control specification, 2011, https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_swap_control.txt. (Accessed 18 June 2018).
- [43] NVIDIA Corporation, GLX_NV_swap_group specification, 2008, https://www.khronos.org/registry/OpenGL/extensions/NV/GLX_NV_swap_group.txt. (Accessed 18 March 2018).
- [44] D. Herrmann, XWiimote - open-source nintendo wii / wii U device driver, 2018, <https://dvdhrm.github.io/xwiimote/>. (Accessed 30 July 2018).
- [45] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, 2014, <https://snap.stanford.edu/data>. (Accessed 30 July 2018).
- [46] A.A. Hagberg, D.A. Schult, P.J. Swart, Exploring network structure, dynamics, and function using networkx, in: *Proceedings of the 7th Python in Science Conference, SciPy*, 2008, pp. 11–15.

- [47] A. Douady, Julia sets and the mandelbrot set, in: *The Beauty of Fractals: Images of Complex Dynamical Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1986, pp. 161–174.
- [48] NVIDIA Corporation, GitHub - NVIDIA/nvidia-settings: NVIDIA driver control panel, 2018, <https://github.com/NVIDIA/nvidia-settings>. (Accessed 30 July 2018).
- [49] NVIDIA Corporation, Profiler user's guide, 2018, https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf. (Accessed 30 July 2018).
- [50] J. Bédorf, E. Gaburov, M.S. Fujii, K. Nitadori, T. Ishiyama, S.P. Zwart, 24.77 PFlops on a gravitational tree-code to simulate the milky way galaxy with 18600 GPUs, in: *Proceedings of the 26th International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 54–65.