BatchLayout: A Batch-Parallel Force-Directed Graph Layout Algorithm in Shared Memory

Md. Khaledur Rahman†
Indiana University Bloomington

Majedul Haque Sujon‡
Indiana University Bloomington

Ariful Azad † §
Indiana University Bloomington

ABSTRACT

Force-directed algorithms are widely used to generate aesthetically-pleasing layouts of graphs or networks arisen in many scientific disciplines. To visualize large-scale graphs, several parallel algorithms have been discussed in the literature. However, existing parallel algorithms do not utilize memory hierarchy efficiently and often offer limited parallelism. This paper addresses these limitations with BatchLayout, an algorithm that groups vertices into minibatches and processes them in parallel. BatchLayout also employs cache blocking techniques to utilize memory hierarchy efficiently. More parallelism and improved memory access coupled with force approximating techniques, better initialization, and optimized learning rate make BatchLayout significantly faster than other state-of-the-art algorithms such as ForceAtlas2 and OpenOrd. The visualization quality of layouts from BatchLayout is comparable or better than similar visualization tools. All of our source code, links to datasets, results and log files are available at https://github.com/khaled-rahman/BatchLayout.

Index Terms: Human-centered computing—Visualization—Visualization techniques—Graph drawings; Human-centered computing—Visualization—Visualization systems and tools—Visualization toolkits

1 INTRODUCTION

Networks or graphs are common representations of scientific, social and business data. In a graph, a set of vertices represents entities (e.g., persons, brain neurons, atoms) and a set of edges indicates relationships among entities (friendship, neuron synapses, chemical bonds). A key aspect of data-driven graph analytics is to visually study large-scale networks, such as biological and social networks, with millions or even billions of vertices and edges. In network visualization, the first step is to generate a layout in a 2D or 3D coordinate system which can be fed into visualization tools such as Gephi [4] and Cytoscape [29]. Therefore, the quality and computational complexity of network visualization are often dominated by the graph layout algorithms.

Force-directed layout algorithms are among the most popularly used techniques to generate the layout of a graph. Following the philosophy of the spring energy model, these algorithms calculate attractive and repulsive forces among vertices in a graph and iteratively minimize an energy function. Classical force-directed algorithms, such as the Fruchterman and Reingold (FR) algorithm [9], require $O(n^2)$ time per iteration where $n$ is the number of vertices in a graph. By approximating the repulsive force between non-adjacent nodes, we can get a faster $O(n \log n)$ algorithm. In this paper, we used the Barnes-Hut approximation [3] based on the quad-tree data structure.

Even though the layout quality (measured by stress, neighborhood preservation, and other well established quality metrics as shown in Table 4) from the Barnes-Hut approximation can be worse than an exact $O(n^2)$ algorithm, the former is often used to visualize large-scale networks. In this paper, we carefully analyzed the quality and runtime of both exact and approximate force-directed algorithms.

To visualize large graphs, several prior work discussed parallel algorithms for multicore processors [23], graphics processing unit (GPU) [5, 17], and distributed clusters [1, 2]. While we summarize state-of-the-art algorithms and software in the Related Work section, our primary focus lies in parallel algorithms for multicore servers. Some state-of-the-art force-directed algorithms such as ForceAtlas2 [17] and OpenOrd [23] have parallel implementations for multicore processors. However, these algorithms do not utilize full advantage of present memory hierarchy efficiently and offer limited parallelism because they calculate forces for one vertex at a time. We aim to improve these two aspects of parallel algorithms. We develop a parallel algorithm called BatchLayout that offers more parallelism by grouping vertices into minibatches and processing all vertices in a minibatch in parallel. This approach of increasing parallelism via minibatches is widely used in training deep neural networks in the form of minibatch Stochastic Gradient Descent (SGD) [11]. We adapt this approach to graph layout algorithms for better parallel performance.

Existing algorithms also access memory irregularly when processing sparse graphs with skewed degree distributions. Irregular memory accesses do not efficiently utilize multi-level caches available in current processors, impeding the performance of algorithms. In BatchLayout, we regularize memory accesses by using linear algebra operations similar to matrix-vector multiplication and employing “cache blocking” techniques to utilize memory hierarchy efficiently. More parallelism and better memory accesses made BatchLayout faster than existing parallel force-directed algorithms. On average, the exact $O(n^2)$ version of BatchLayout is $15 \times$ faster than ForceAtlas2. The Barnes-Hut approximation in BatchLayout is about $4 \times$ and $2 \times$ faster than ForceAtlas2 (with BH approximation) and OpenOrd, respectively (the quality of an OpenOrd layout is usually worse than BatchLayout and ForceAtlas2).

BatchLayout attains high performance without sacrificing the quality of the layout. We used three aesthetic metrics to quantitatively measure the quality of layouts and also visualized the networks in Python for human verification. According to all quality measures, BatchLayout generates similar or better-quality layouts than ForceAtlas2 and OpenOrd.

Overall, BatchLayout covers all important aspects of force-directed layout algorithms. We investigated four energy models, two initialization techniques, different learning rates and convergence criteria, various minibatch sizes, and three aesthetic metrics to study the performance and quality of our algorithms. All of these options are available in an open source software that can be run on any multicore processor. Therefore, similar to ForceAtlas2, BatchLayout provides plenty of options to users to generate a readable layout for an input graph.

In this paper, we present new insights on parallel force-directed graph layout algorithms. We summarize key contributions below:

- **Improved parallelism**: We develop a class of exact and approxi-
mate parallel force-directed graph layout algorithms called BatchLayout. Our algorithms expose more parallel work to keep hundreds of processors busy.

- **Better quality layouts**: BatchLayout generates layouts with better or similar qualities (according to various metrics) compared to other force-directed algorithms.
- **Comprehensive coverage**: BatchLayout provides multiple options for all aspects of force-directed algorithms. Especially, we cover new initialization techniques and four energy models.
- **High-performance implementation**: We provide an implementation of BatchLayout for multicore processors. BatchLayout improves data locality and decreases thread synchronizations for better performance.
- **Faster than ForceAtlas2 and OpenOrd**: On a diverse set of graphs, BatchLayout is significantly faster than ForceAtlas2 and OpenOrd.

## 2 BACKGROUND

### 2.1 Notations

We present an unweighted graph by $G = (V, E)$ on the set of vertices $V$ and set of edges $E$. The number of vertices and edges are denoted by $n$ and $m$, respectively. In this paper, we only considered undirected graphs, but directed graphs can be used by ignoring the directionality of the edges. The degree of vertex $i$ is denoted by $\text{deg}(i)$. We denote the coordinate of vertex $i$ with $c_i$, which represents $(x, y)$ Cartesian coordinate in a $2D$ plane. The distance between two vertices $i$ and $j$ is represented by $\| c_i - c_j \|$. In our algorithm, we used the standard Compressed Sparse Row (CSR) data structure to store the adjacency matrix of a sparse graph. For unweighted graphs, the CSR format consists of two arrays: rowptr and colidx. rowptr, an array of length $n + 1$, stores the starting indices of rows. colidx is an array of length $m$ that stores the column indices in each row. Here, the CSR data structure is used for space and computational efficiency, but any other data structure would also work with our algorithm.

### 2.2 Force Calculations

Force-directed graph drawing algorithms compute attractive forces between every pair of adjacent vertices and repulsive forces between nonadjacent vertices. In the spring-electrical model, a pair of connected vertices attract each other like a spring based on Hooke’s law and a pair of nonadjacent vertices repulse each other like electrically charged particles based on Coulomb’s law. Following this spring-electrical model, we define the attractive force $f_a(i, j)$ and the repulsive force $f_r(i, j)$ between vertices $i$ and $j$ as follows:

$$
\begin{align*}
  f_a(i, j) &= \frac{c_i - c_j}{K}, \\
  f_r(i, j) &= \frac{-RK^2}{\| c_i - c_j \|^3}.
\end{align*}
$$

Here, $K$ acts as an optimal spring length and $R$ is considered as the regulator of relative strength of forces. Different values of $a$ and $r$ give different energy models, known as the $(a,r)$-energy model [17, 27]. The combined force $f(i, c)$ of a vertex $i$ with respect to the current coordinates $c$ of all vertices is computed by:

$$
\sum_{(i,j) \in E} f_a(i, j) \times \frac{c_j - c_i}{\| c_i - c_j \|} + \sum_{(i,j) \not\in E} f_r(i, j) \times \frac{c_j - c_i}{\| c_i - c_j \|}.
$$

Here, $c$ is an array of coordinates of all vertices i.e., $c = \{ c_i | i \in V \}$. We multiplied attractive and repulsive forces by a unit vector to get the direction of movement of vertex $i$. Hence, following the spring-electrical model, the energy of the whole graph will be proportional to $\sum_{i \in V} F^2(i, c)$. The goal of a force-directed algorithm is to minimize the energy so that the layout of the graph becomes stable, readable and visually appealing.

Algorithm 1: A Sequential Force-Directed Algorithm

**Input:** G(V, E) and an initial layout $c$

1. $Step = 1.0$

2. $Loop = 0$

3. **while** $Loop < \text{MaxIteration}$ **do**

4. $Energy = 0$

5. **for** $i \leftarrow 0$ to $n - 1$ **do**

6. $f = 0$

7. **for** $j \leftarrow 0$ to $n - 1$ **do**

8. **if** $(i, j) \in E$ **then**

9. $f = f + f_a(i, j) \times \frac{c_i - c_j}{\| c_i - c_j \|}$

10. **else**

11. $f = f + f_r(i, j) \times \frac{c_i - c_j}{\| c_i - c_j \|}$

12. $c_i = c_i + Step \times \frac{f}{\| f \|}$

13. $Energy = Energy + \| f \|^2$

14. $Step = Step \times 0.999$

15. $Loop = Loop + 1$

16. **return** the final layout $c$

### 2.3 A Standard Sequential Algorithm

Algorithm 1 describes a standard sequential force-directed graph layout algorithm. We closely followed the notation of an Adaptive Cooling Scheme by Y. Hu [15]. Algorithm 1 starts with an initial layout and iteratively improves it by minimizing energy. In each iteration, vertices are chosen in a predefined order (line 5), and attractive and repulsive forces are calculated for a selected vertex with respect to all other vertices in the graph. Next, the relative position of the selected vertex is updated (line 12) and this updated position is used in processing the subsequent vertices. Step used in line 12 is a hyper-parameter that influences the convergence of the algorithm. We will discuss possible values of this hyper-parameter in the experimental section. A working example of this algorithm is shown in Fig. S1 of the supplementary file. Each iteration of Algorithm 1 computes $O(m)$ attractive forces and $O(n^2)$ repulsive forces, giving us a sequential $O(n^2)$ algorithm. Using the Barnes-Hut approximation [31] for repulsive forces, Algorithm 1 can easily be made into a sequential $O(n \log n)$ algorithm.
After all vertices in a minibatch finish force calculations, we update vertices in different orders. This considers four energy models and calculates forces for each vertex in the minibatch independently. BatchLayout algorithm selects a subset of vertices called a mini-batch approach. Across a mini-batch, we compute forces for all vertices with respect to all other vertices in the graph. Suppose, in batch 0, we compute forces for vertex 0 and 1 with respect to vertices 0...2 (blue). When forces are calculated for vertex 0, coordinates of vertices 0...2 and other vertices in the graph. In our experiment, we observed that the benefit of concurrent work is significantly more apparent when using a minibatch approach. We noticed that force calculations can be viewed as a matrix-vector multiplication. Fig. 3 shows a simple example where the adjacency matrix of the graph used in Fig. 2 is considered. The vector of current coordinates is shown twice in dense vectors A and B. A' holds the updated coordinates. In our minibatch model, we compute the forces for all vertices in a minibatch with respect to all other vertices in the graph. We highlighted “batch processing” in Step 2a since it is a required option for efficient parallel algorithms discussed in this paper. We explored many options within our parallel algorithm.
are brought in from the main memory and stored in a CPU cache. The cumulative sum of force calculations are stored in a temporary vector, $A'$. When we compute forces for the next vertex, all necessary coordinates are already available in cache (i.e., using data locality), which reduces the memory traffic. Batch 0 gets a similar performance advantage when forces are computed with respect to vertices 3...5 (green). Batch 1, with the cache-blocking scheme is presented in Algorithm 2. Here, $G.rowptr$ and $G.colids$ are two arrays which hold the row pointers and column IDs, respectively, of the graph in a CSR format. $BS$ is the size of minibatch and $Step$ length is initialized to 1.0. In line 6 of Algorithm 2, we iterate over a subset of vertices (minibatch) and in line 10, we iterate over all vertices to calculate $f_{a_0}$ and $f_{r}$. In line 14, when two vertices are connected by an edge we calculate $f_{e}$, otherwise, we calculate $f_r$ (line 18). After calculating forces for a vertex, we store it in a temporary location (see line 19). This temporary storage helps us to calculate forces in parallel. Next, we iterate over all vertices in a minibatch to update relative position (see line 22). We provide a flexible rectangular cache blocking area which is bounded by variables $p$ and $q$. This increases the data locality in the L1 cache of each core, improving the performance of our algorithm. Loop counters $i$ and $j$ are increased by $p$ and $q$, respectively. Variable $k_i$ holds the starting index of neighbors for $p$ vertices (see line 8). After the loop completes, in line 24, we update step length. To keep Algorithm 2 simple, we assume that the number of vertices in a graph is multiple of $(b + 1) \times BS$. In our implementation, we incorporated other cases. The overall running time of this procedure is same as Algorithm 1 i.e., $O(n^2)$. However, the main advantage of this technique is in parallel computation of forces. Note that Algorithm 1 is a special case of Algorithm 2 when the size of the minibatch is 1.

### 3.5 Repulsive Force Approximation

Exact repulsive force computations need $O(n^2)$ time per iteration, which is too expensive for large graphs. Therefore, we have developed a parallel Barnes-Hut algorithm [3] for approximating repulsive forces. Barnes-Hut takes $O(n \log n)$ time per iteration. For 2D visualization, a quad-tree data structure is often used to approximate repulsive forces. An example is shown in Fig. 4. Given the coordinates of vertices in Fig. 4(a), a high-level block is divided into four equal blocks as shown in Fig. 4(b). In this example, we split up to 2 levels, creating $4^2 = 16$ small blocks in the 2D plane. Empty blocks are merged with their adjacent nonempty blocks. We show how the algorithm approximates the repulsive force $f_r$ for the green vertex. First, a centroid (shown in red) is calculated for each block by averaging the coordinates of all vertices in the block. Next, an approximate repulsive force is computed with respect to the centroids, as shown by the dashed gray lines.

![Algorithm 2: Cache Blocking BatchLayout](image)

**Algorithm 2: Cache Blocking BatchLayout**

**Input:** $G(V, E)$

1. $Step = 1.0$ \hspace{1cm} $\triangleright$ initial step length
2. $Loop = 0$
3. **while** $Loop < MaxIteration$ **do**
4. $Energy = 0$
5. for $b \leftarrow 0$ to $\frac{n}{BS} - 1$ **do**
6. for $i \leftarrow b \ast BS$ to $(b + 1) \ast BS - 1$ by $p$ **do** in parallel
7. for $x \leftarrow 0$ to $p - 1$ **do**
8. $k_i = G.rowptr[i + x]$
9. $ct_{i+x} = 0$
10. for $j \leftarrow 0$ to $q - 1$ by $q$ **do**
11. for $x \leftarrow 0$ to $p - 1$ **do**
12. $f = 0$
13. for $y \leftarrow 0$ to $q - 1$ **do**
14. if $j + y = G.colids[k_i]$ then
15. $f + = f_i(i, j) \times \frac{r_{i-c_i}[y-q_i]}{|r_{i-c_i}|}$
16. $k_i + = 1$
17. else
18. $f + = f_i(i, j) \times \frac{r_{i-c_i}[y-q_i]}{|r_{i-c_i}|}$
19. $ct_{i+x} + = f$
20. end parallel for
21. for $i \leftarrow b \ast BS$ to $(b + 1) \ast BS - 1$ **do**
22. $c_{i} + = Step \times \frac{ct_i}{|ct_i|}$
23. $Energy + = ||ct_i||^2$
24. $Step = Step \times 0.999$
25. $Loop + = 1$
26. **return** the final layout $c$

![Figure 4](image)

(a) An example graph. (b) Calculating forces of the green vertex using a quad-tree. (c) Blue solid lines show attractive forces, and dashed gray lines denote approximate repulsive forces with respect to centroids (red).

![Figure 5](image)

(a) A quad-tree structure. (b) Morton order of traversal for 16 vertices in 16 different blocks.

Among several choices of quad-tree implementation [31], we adopted an approach based on Warren and Salmon [30]. Fig. 5(a) shows that we define a drawing plane by bounding coordinates $(x_{min}, y_{max})$ and $(x_{max}, y_{min})$. This bounding plane is the root of the quad-tree. Next, we split the bounding plane (root) into four equal blocks, each of which represents a child node (shown by a unique color) of the root. We sort the positions of all vertices based on Morton code (also called Z-order curve) [25] and obtain an ordered grid of vertices. Each child node is then divided into four equal sub-blocks, and the process is repeated until all vertices are placed in the tree.
Algorithm 3: Greedy Initialization

1: \textbf{Input:} G(V, E) \hspace{1cm} \triangleright \text{initial empty stack}
2: \qquad 1: S = \emptyset
3: \qquad 2: \text{visited} = [False, \ldots, False]
4: \qquad 3: V_{0} = (0,0) \hspace{1cm} V_{0x} = 0, V_{0y} = 0
5: \qquad 4: S.push(V_{0})
6: \qquad 5: visited_{0} = True
7: \qquad 6: \textbf{while} S is not empty \textbf{do}
8: \qquad 7: \quad U = S.pop()
9: \qquad 8: \quad d = \frac{200}{\deg(u)}
10: \qquad 9: \quad D = 0
11: \qquad 10: \quad \textbf{for} u \text{ neighbors of } U \textbf{do}
12: \qquad 11: \quad \text{if visited}_{u} \text{ is False then}
13: \qquad 12: \quad \quad V_{u} = (U_{x} + \cos (\frac{\pi}{\deg(u)}), U_{y} + \sin (\frac{\pi}{\deg(u)}))
14: \qquad 13: \quad \quad \text{visited}_{u} = True
15: \qquad 14: \quad S.push(V_{u})
16: \qquad 15: \quad \quad D = D + d
17: \textbf{return } V

3.6 Initialization

The initial layout plays an important role in the convergence and visualization quality of the final layout. Many existing algorithms start with random layouts where vertex positions are assigned randomly [15]. In addition to random initial layouts, we developed a novel greedy initialization technique as shown in Algorithm 3. We maintain a stack \(S\) to keep track of visited vertices. The algorithm starts with a randomly selected vertex \(V_0\), and \((0,0)\) is used as its coordinate. In every iteration, the algorithm extracts a vertex \(U\) from the stack and places \(U\)’s neighbors on a unit circle centered at \(U\) (line 10–14 in Algorithm 3). Additionally, the distance between every pair of \(U\)’s neighbors is also equal. In line 12, \(PI\) is a constant whose value is 3.1416. In section 3.7, we will discuss that uniform edge length is a good aesthetic metric for assessing the quality of a layout. Hence, we consider this unit-radius approach to be a better initializer. Our experimental results do show that the greedy initialization indeed converge faster than the random initialization. The greedy approach is similar to a DFS with \(O(n + m)\) complexity.

3.7 Performance Metrics

There are several aesthetic metrics in the literature to assess the quality of a layout generation algorithm [8, 21, 28]. Though there does not exist a single metric that can uniquely determine the effectiveness of a layout, it gives a quantitative value that tells us how layout algorithms perform. For the sake of completeness, we have used the following aesthetic metrics along with running time analysis.

**Stress (ST):** Stress computes the difference between geometric distance and graph theoretic distance for any pair of vertices in a graph [5, 8]. Since different algorithms may require different drawing space based on which stress may change, the drawing space is scaled before computing the stress for all algorithms. Thus, the reported value is the minimum value achievable after scaling the layout. A lower ST value indicates a better layout, and it is computed using \(\sum_{i \neq j} w_{ij} (|c_{i} - c_{j}| - d_{ij})^{2}\) where, \(c_{i}\) and \(c_{j}\) are coordinates of vertices \(i\) and \(j\), respectively. The value of \(d_{ij}\) represents a graph theoretic distance between vertices \(i\) and \(j\), and \(w_{ij} = \frac{1}{d_{ij}}\).

**Edge Uniformity (EU):** Sometimes the uniformity of edge lengths represents better readability of a layout [16]. We define EU in a similar way as defined in [8, 13]. It finds the normalized standard deviation of edge length which can be computed using \(\sqrt{\frac{\sum_{i \neq j} (|c_{i} - c_{j}| - l_{ij})^{2}}{|E|}}\) where, \(l_{ij}\) is the average length of all edges and \(|E|\) represents the number of edges in the graph. Lower value of EU represents better quality of the layout.

**Neighborhood Preservation (NP):** This measure computes the number of neighbors that are close to a vertex in a graph as well as in the driven layout. It has been used to compare graph layouts in many tools like tsNET [20] and Maxent [10]. This is a normalized similarity measure where 0 means that the adjacency of vertices are not preserved in the layout whereas 1 means that all vertices preserve their respected neighbors.

4 RESULTS

4.1 Experimental Setup

We implemented BatchLayout in C++ with multithreading support from OpenMP. We ran all experiments in a server consisting of Intel Xeon Platinum 8160 processors (2.10GHz) arranged in two NUMA sockets. The system has 256GB memory, 48 cores (24 cores/socket), and 32MB L3 cache/socket. For comparison, we used ForceAtlas2 implemented in Gephi and OpenOrd from its GitHub repository. We also experimented with Gephi’s OpenOrd implementation (denoted by OpenOrdG), but found that the GitHub implementation produced better visualizations. We use Gephi’s toolkit v0.9.2 [4] to run multi-threaded ForceAtlas2 and OpenOrdG. The Barnes-Hut variant of ForceAtlas2 is termed as ForceAtlas2BH. For BatchLayoutBH, we use the same threshold value (1.2) for MAC as in ForceAtlas2BH. Unless otherwise specified, we use the default settings for all of these tools. For OpenOrd, we set the edge-cutting parameter to 0, and we set the default time distribution in different stages of simulated annealing. While the BatchLayout software includes four variants of \((a, r, \rho)\)-energy model considered in the ForceAtlas2 paper, we only show results from BatchLayout using \((2, -1)\)-energy model (equivalent to the Fruchterman and Reingold algorithm). We selected this model because it usually generates better visualizations [17].

4.2 Datasets

Table 1 shows a diverse collection of test graphs including small world networks, scale free networks, mesh, structural problem, optimization problem, linear programming, economic problem, social networks, and road networks. These graphs were collected from the SuiteSparse matrix collection (https://sparse.tamu.edu). Specifically, we use Luxembourg OSM (lxOSM) and Flan_1565 datasets to test the ability of algorithms to visualize large-scale graphs. For scalability experiments, we also generated benchmark networks by Lancichinetti et al. [22]. For this tool, we set mixing parameter, avg. degree, community minsize, community maxsize to 0.034, 80, 128 and 300, respectively, and vary maximum degree from 100 to 300. We generated a total of 5 random graphs, each of which has \(2^{v}\) vertices, where \(v \in \{16, 17, 18, 19, 20\}$.1

---

1https://github.com/SciTechStrategies/OpenOrd
We now select a minibatch size that provides enough parallelism without affecting the convergence rate significantly. Fig. 6(b) shows layout energies for different minibatches for the \textit{3elt_dual} graph. Here, BL256 denotes BatchLayout with minibatch size of 256. For this graph, we observe that small minibatches converge faster (in terms of iterations) than large minibatches. For example, BL1 and BL256 achieve the same energy at 1000 and 1100 iterations, respectively (marked by brown squares in Fig. 6(b)). Consequently, BL256 needs to run 100 more iterations to reach the same energy achieved by BL1. However, the cost of extra iterations is offset by the faster parallel runtime of BL256. Fig. 6(c) shows the runtime for different minibatches when we run BatchLayout with 48 threads for the same graph. The factor of 0.999 (that is $0.999 \times$ faster than BatchLayoutBH) is on average 2.66 times faster than BL256. Scale Free networks play a central role in attaining high performance by our parallel algorithm. However, using larger minibatches beyond 256 does not improve the performance further. For example, BL256 and BL2048 behave almost similarly as can be seen in Figs. 6(b) and 6(c). We also observe similar minibatch profiles for other graphs. Hence, we set 256 as the default minibatch size for the rest of our experiments (a user can change the minibatch size in our software).

### 4.3 Algorithmic Options and Their Impacts

#### 4.3.1 Initialization

At first, we compare the differences between greedy initialization using Algorithm 3 and random initialization using the C++ function \texttt{rand}. Both initialization techniques generate random coordinates within a given range $[-\text{MAXMIN}, \text{MAXMIN}]$. We feed initial coordinates to BatchLayout, and Fig. 6(a) shows the layout energy for the \textit{3elt_dual} graph after various iterations. We observe that in the $i^{th}$ iteration, the greedy-initialized layout has lower energy than the randomly-initialized layout. This result is consistent across other graphs as well. Fig. 6(a) reveals that greedy initialization accelerates the convergence of BatchLayout. Hence, we use the greedy initialization in subsequent experiments in the paper (both initializations are available in our software).

#### 4.3.2 Minibatch size

We now select a minibatch size that provides enough parallelism without affecting the convergence rate significantly. Fig. 6(b) shows layout energies for different minibatches for the \textit{pkustk02} graph. We feed initial coordinates to BatchLayout, and Fig. 6(a) shows the layout energy for the \textit{3elt_dual} graph after various iterations. We observe that in the $i^{th}$ iteration, the greedy-initialized layout has lower energy than the randomly-initialized layout. This result is consistent across other graphs as well. Fig. 6(a) reveals that greedy initialization accelerates the convergence of BatchLayout. Hence, we use the greedy initialization in subsequent experiments in the paper (both initializations are available in our software).

#### 4.3.3 Batch Randomization

We conduct experiments for the randomization of batches which is a common practice in machine learning. We reported standard deviation as well as energy curves for different runs using \textit{3elt_dual} graph (Figure S3 in supplementary file). From our experiments, we observed that randomized batch selection achieves better energy curve than fixed sequential batch selection but results are not deterministic i.e., for each run, output is different (similar but not same) even though all hyper-parameters are same. We can see a greater difference for a smaller number of iteration. Randomization also introduces overhead in total running time for reshuffling and it can not take advantage of optimal cache usage. Moreover, randomized batch selection does not make any significant improvement in layout. So, we keep sequential batch selection as our default option in BatchLayout.

#### 4.3.4 Number of threads

Using more threads not necessarily translates to faster runtime, especially for small graphs. This is because of the overhead of thread fork-join model used in OpenOrd. For larger graphs with more than 100,000 vertices, we recommend employing all available threads. For smaller graphs, users can consider using fewer threads. Hence, the number of threads is provided as an option in BatchLayout.

#### 4.3.5 Step length or learning rate

Line 14 of Algorithm 1 uses a multiplicative factor for Step, which dictates both convergence and the quality of the final layout. Intuitively, we want to adapt the step length so that the algorithm takes bigger steps in the beginning but increasingly smaller steps as we move closer to the minimum energy. Fig. 7(a) shows layout energies at convergence with various multiplicative factors for the \textit{3elt_dual} graph. We observe that the factor of 0.999 (that is 0.1% reduction of step length in every iteration) provides the best balance as the converged energy is minimum for this factor. Hence, we use this multiplicative factor in all experiments with BatchLayout. We note that adaptive steps are also used in other force-directed layouts such as in the algorithm developed by Yifan Hu [15]. In this paper, we empirically identify an effective adaption policy so that BatchLayout converges faster.

#### 4.4 Running Time and Scalability

We run all algorithms for 500 iterations using 47 threads (one thread per core) and report their runtime in Table 2. Since OpenOrdG leaves out a core for kernel-specific tasks, we also leave a core unutilized for fairness in comparison. In Table 2, we include time for all steps except file input/output operations. If we consider exact force calculations, BatchLayout is on average 15 $\times$ faster than ForceAtlas2 (min:7.6 $\times$, max:21.8 $\times$). For approximate force calculation, BatchLayoutBH is on average 2 $\times$ faster than OpenOrd (min:1.4 $\times$, max:9.5 $\times$). Both BatchLayoutBH and OpenOrd are faster than ForceAtlas2BH. Notice that OpenOrdG (that is Gephi’s OpenOrd) also runs fast, but its layout quality is very low as seen in the next section. Overall, BatchLayoutBH is faster than all of them. BatchLayoutBH and OpenOrd are faster than ForceAtlas2BH. We observe that tsNET can generate good layouts for small graphs, but failed to generate any layout for medium or bigger graphs in our experiments. For small graphs like \textit{3elt_dual}, tsNET took around 1 hour whereas other multi-threaded tools generated layout within few seconds. Hence, we did not include tsNET runtimes in Table 2.

**Thread and vertex scalability**. We now discuss the thread and vertex scalability of the three fastest algorithms: BatchLayoutBH (FLBH), ForceAtlas2BH (FA2BH), and OpenOrd (OO). As discussed in section 4.2, we use a benchmark tool [22] to generate graphs for the scalability experiment. Fig. 7(b) shows the thread runtimes for different graph sizes.
Figure 6: (a) Energy curves for greedy and random initialization (3elt_dual graph). (b) Energy curves (log scale) for different batches (pkustk02 graph). (c) Running time (log scale) for different batches of BatchLayout (pkustk02 graph).

Figure 7: (a) Energy values (log scale) for different multiplicative factors of Step using 3elt_dual graph. (b) Strong scaling: Running time (log scale) for different number of threads using a synthetic graph of $2^{18}$ vertices generated by [22]. (c) Weak scaling: Running time (seconds) for different number of threads and graph sizes.

We now visualize the layouts generated by all algorithms considered were run for the same number of iterations that BatchLayout took without consuming significant memory. We observe that both BatchLayoutBH and OpenOrd scale linearly with threads (cores), but BatchLayoutBH is $\sim 4x$ faster than OpenOrd on all thread counts. ForceAtlas2BH does not scale linearly, and it is generally not faster than OpenOrd. Fig. 7(c) shows the weak scalability where we use increasingly larger graphs varying number of threads. Fig. 7(c) starts with a graph of $2^{18}$ vertices running on 2 threads. Then, we increasingly double the problem size as well as computing resource, run experiments and report the runtime. Among all three algorithms, we observe that BatchLayoutBH’s work distribution remains balanced (i.e., curve remains horizontal). Hence, BatchLayoutBH shows superior weak scaling performance compared to other two algorithms.

**Memory usage.** We measured the memory consumption of all algorithms using the memory-profiler python package. For the lxOSM graph, BatchLayout, BatchLayoutBH, ForceAtlas2, ForceAtlas2BH, OpenOrdG, and OpenOrd consumed maximum memory of 16.36MB, 30.34MB, 676.38MB, 1348.38MB, 1265.32MB and 14.12MB, respectively. Gephi’s ForceAtlas2, ForceAtlas2BH and OpenOrdG consumed higher memory than others possibly because of the high memory requirements of Gephi itself. In fact, ForceAtlas2BH failed to generate layouts for two large graphs due to its high memory consumption. In our algorithm, Barnes-Hut force approximation requires additional space to store the quad-tree data structure, and hence it consumed more memory than the exact force computation. Overall, BatchLayoutBH can compute layouts faster without consuming significant memory.

### 4.5 Visualization

We now visualize the layouts generated by all algorithms considered in the paper. At first, we run BatchLayout until convergence (when the energy difference between successive iterations becomes less than $10^{-6}$). Then, all other algorithms including BatchLayoutBH were run for the same number of iterations that BatchLayout took to converge. The visualization from these layouts are shown in Table 3. The runtime and converged energy are provided in supplementary Table S2, which shows that BatchLayoutBH runs much faster than other tools. Table 3 demonstrates that BatchLayout and BatchLayoutBH produce readable layouts that are comparable or better than ForceAtlas2 and ForceAtlas2BH, respectively. Generally, BatchLayoutBH and ForceAtlas2BH layouts are much better than OpenOrd. Gephi’s OpenOrd generates inferior layouts; hence, we did not show them in Table 3. Note that BatchLayout uses Fruchterman and Reingold as its default energy model, which is different from the default model in ForceAtlas2. Hence, BatchLayout’s superior layouts for some graphs (e.g., finan256) are not surprising because the FR model usually generates better layouts as was also reported in the ForceAtlas2 paper [17]. ForceAtlas2 does not use FR as its default model possibly because of its higher computational cost (also reported in [17]). In this paper, we show that the FR algorithm runs much faster in our BatchLayout framework and can generate better layouts for some graphs.

In the scalability and runtime experiments, we showed results with a fixed 500 iterations. Supplementary Table S1 shows the visualization from all algorithms after 500 iterations. In many cases, layouts after 500 iterations are close to the layouts at convergence.

| Graph      | BL | BLBH | FA2 | FA2BH | OOG | OO |
|------------|----|------|-----|-------|-----|----|
| Powergrid  | 0.95 | 0.98 | 20.73 | 3.04 | 1.67 | 1.43 |
| add32      | 1.03 | 1.02 | 20.75 | 3.15 | 1.71 | 1.45 |
| ba_network | 1.42 | 1.22 | 25.47 | 3.61 | 2.18 | 1.78 |
| 3elt_dual  | 2.92 | 1.54 | 54.6  | 4.87 | 3.17 | 2.67 |
| PGP        | 4   | 1.83 | 68.67 | 6.26 | 4.18 | 3.26 |
| pkustk01   | 4.23 | 1.98 | 76.69 | 10.72 | 10.32 | 3.83 |
| fe_4elt2   | 4.33 | 1.86 | 75.4  | 6.34 | 4.21 | 3.35 |
| bodyy6     | 12.3 | 2.95 | 143.08 | 9.69 | 6.75 | 5.85 |
| pkustk04   | 16.23 | 3.43 | 242.72 | 18.49 | 15.33 | 7.25 |
| OPP_6000   | 29.75 | 4.69 | 266.99 | 18.31 | 12.45 | 8.95 |
| finance256 | 45.49 | 5.4  | 412.64 | 20.37 | 14.89 | 11.18 |
| finan512   | 185.8 | 10.53 | 1408.46 | 46.98 | 31.74 | 22.57 |
| lxOSM      | - | 21.59 | - | 72.30 | - | 33.62 |
| comYout    | - | 174.5 | - | 1504.7 | - | 1670.52 |
| Flan_1565  | - | 224.3 | - | - | - | 609.56 |

Table 2: Runtime for different methods. Lower value represents better result. BL, BLBH, FA2, FA2BH, OOG and OO represent BatchLayout, BatchLayoutBH, ForceAtlas2, ForceAtlas2BH, OpenOrdG and OpenOrd, respectively. Better results are shown in bold font.
Table 3: Layouts of different graphs generated by different algorithms. BatchLayout convergence criteria was set to 1E-6. Number of iterations and running time are shown in supplementary file, Table S2.

| Graph     | BatchLayout | BatchLayoutBH | ForceAtlas2 | ForceAtlas2BH | OpenOrd |
|-----------|-------------|---------------|-------------|---------------|---------|
| Powergrid | ![Image]    | ![Image]      | ![Image]    | ![Image]      | ![Image] |
| ba_network| ![Image]    | ![Image]      | ![Image]    | ![Image]      | ![Image] |
| 3elt_dual | ![Image]    | ![Image]      | ![Image]    | ![Image]      | ![Image] |
| pkustk02  | ![Image]    | ![Image]      | ![Image]    | ![Image]      | ![Image] |
| fe_4elt2  | ![Image]    | ![Image]      | ![Image]    | ![Image]      | ![Image] |
| bodyy6    | ![Image]    | ![Image]      | ![Image]    | ![Image]      | ![Image] |
| pkustk01  | ![Image]    | ![Image]      | ![Image]    | ![Image]      | ![Image] |
| finan256  | ![Image]    | ![Image]      | ![Image]    | ![Image]      | ![Image] |

Hence, many algorithms including BatchLayout provide “number of iterations” as an option to users.

Both BatchLayout and ForceAtlas2 provide many options beside default options used in our experiments. It is possible that tuning these parameters will generate better-quality layouts for some graphs. However, we stick to default parameters in this paper for simplicity and fairness. OpenOrd may generate different layouts for the same graph when different numbers of threads are used. By contrast, BatchLayout and BatchLayoutBH generate the same layout for a given graph irrespective of number of threads.

Finally, Fig. 8 shows the layout of Flan.I565, the largest graph in our dataset. We show layouts from BatchLayoutBH and OpenOrd after running them for 5000 iterations (ForceAtlas2BH went out of memory for this graph). For this graph, OpenOrd’s layout is not good enough even though we allow both algorithms to run for 5000 thousands of iterations. Furthermore, OpenOrd took 1 hour and 39 minutes whereas BatchLayoutBH took 49 minutes to finish 5000 iterations. This clearly demonstrates the effectiveness of BatchLayoutBH to visualize graph with hundreds of millions of edges. However, generating layouts in half an hour may not be good enough for certain applications. We plan to address this by implementing BatchLayoutBH for distributed systems.
Table 4: Comparison of Stress (ST) and Neighborhood preservation (NP) measures among all tools. For ST, a lower value means a better result and for NP, a higher value represents a better result. Better results are shown in bold font.

| Graph     | BL   | BLBH | FA2 | FA2BH | OO  | OOGH | BL   | BLBH | FA2 | FA2BH | OO  | OOGH |
|-----------|------|------|-----|-------|-----|------|------|------|-----|-------|-----|------|
| Powergrid | 1.3E+6 | 1.6E+6 | 2.4E+6 | 2.2E+6 | 2.5E+6 | 2.4E+6 | 0.39 | 0.266 | 0.403 | 0.387 | 0.324 | 0.336 |
| add32     | 2.2E+6 | 1.9E+6 | 5.3E+6 | 5.4E+6 | 6.2E+6 | 2.3E+6 | 0.453 | 0.347 | 0.538 | 0.523 | 0.362 | 0.409 |
| ba_network | 2.8E+6 | 2.3E+6 | 3.4E+6 | 3.4E+6 | 3.9E+6 | 2.9E+6 | 0.357 | 0.295 | 0.457 | 0.452 | 0.384 | 0.465 |
| 3elt_dual | 5.6E+6 | 6.6E+6 | 9.2E+6 | 8.0E+6 | 1.4E+7 | 1.1E+7 | 0.38 | 0.271 | 0.253 | 0.334 | 0.171 | 0.174 |
| PGP       | 8.2E+6 | 1.2E+7 | 9.9E+6 | 9.8E+6 | 1.1E+7 | 8.8E+6 | 0.18 | 0.119 | 0.24 | 0.243 | 0.266 | 0.231 |
| pkustk02  | 3.7E+6 | 4.4E+6 | 1.0E+7 | 1.1E+7 | 1.9E+7 | 8.1E+6 | 0.649 | 0.583 | 0.549 | 0.544 | 0.38 | 0.513 |
| fe_4elt2  | 1.0E+7 | 1.7E+7 | 1.2E+7 | 1.5E+7 | 2.1E+7 | 1.8E+7 | 0.36 | 0.25 | 0.319 | 0.289 | 0.222 | 0.207 |
| body06    | 3.1E+7 | 4.1E+7 | 3.4E+7 | 4.3E+7 | 6.1E+7 | 5.2E+7 | 0.213 | 0.205 | 0.278 | 0.251 | 0.217 | 0.124 |
| pkustk01  | 1.7E+7 | 1.9E+7 | 1.8E+7 | 2.0E+7 | 4.1E+7 | 3.1E+7 | 0.513 | 0.398 | 0.528 | 0.523 | 0.373 | 0.396 |

Table 5: Comparison of Edge uniformity (EU) measures among all tools. For this measure, lower value means better result. Better result of each graph is shown in bold font.

| Graph     | BL   | BLBH | FA2 | FA2BH | OO  | OOGH | BL   | BLBH | FA2 | FA2BH | OO  | OOGH |
|-----------|------|------|-----|-------|-----|------|------|------|-----|-------|-----|------|
| Powergrid | 0.83 | 0.5 | 1.46 | 1.33 | 1.93 | 1.18 | 0.83 | 0.5 | 1.46 | 1.33 | 1.93 | 1.18 |
| add32     | 1.38 | 1.04 | 1.30 | 1.26 | 1.86 | 1.76 | 1.07 | 0.58 | 3.25 | 3.30 | 3.01 | 2.93 |
| ba_network | 0.39 | 0.39 | 0.59 | 0.63 | 1.37 | 0.53 | 0.39 | 0.39 | 0.59 | 0.63 | 1.37 | 0.53 |
| PGP       | 0.81 | 0.65 | 1.43 | 1.42 | 2.44 | 1.43 | 0.73 | 0.67 | 1.09 | 1.12 | 1.93 | 0.88 |
| pkustk02  | 0.39 | 0.53 | 0.53 | 0.53 | 1.55 | 0.48 | 0.39 | 0.53 | 0.53 | 0.53 | 1.55 | 0.48 |
| fe_4elt2  | 0.76 | 0.80 | 0.49 | 0.51 | 1.64 | 0.79 | 0.83 | 0.70 | 1.01 | 1.02 | 1.62 | 0.93 |

4.6 Comparison of Aesthetic Metrics

We quantitatively measure the quality of graph layouts using standard aesthetic metrics discussed in Section 3.7. Similar to the runtime analysis, we measure aesthetic quality of layouts after running each algorithm for 500 iterations and report the numbers in Tables 4 and 5. There is no clear winner according to all metrics, which is expected since different algorithms optimize different energy functions. Table 4 shows ST (lower is better) and NP measures (higher is better), where BatchLayout and BatchLayoutBH are better than other tools for ST measure and competitive with their peers for NP measure. Table 5 shows EU (lower is better), where we observe that BatchLayoutBH is winner for most of the graph instances. Overall, BatchLayout and BatchLayoutBH perform better according to ST and EU measures and competitive according to NP measure.

5 Related Work

Graph drawing is a well studied problem in the literature. In addition to the spring-electrical approach, there are other models such as the spectral method [19], and high-dimensional embedding [14]. Recently, Krueger et al. [20] introduced tsNET based on stochastic neighbor embedding technique. Due to space limitation, we restrict our discussion to force-directed approaches.

Kamada and Kawai [18] proposed a spring model where optimal drawing is obtained by minimizing stress. In their model, stress is represented by the difference between geometric distance and graph-theoretic shortest path distance. The computational complexity of this model is high, and quad-tree based force approximation can not be applied to it [15]. Other graph drawing algorithms that optimize a stress function also produce readable layouts [10, 24], but they are generally prohibitively expensive for large graphs.

Fruchterman and Reingold (FR) proposed a spring-electrical model where an energy function is defined in terms of attractive and repulsive forces [9]. This model is computationally faster than the spring model, and quad-tree based force approximation can also be applied. Yifan Hu introduced a multi-level version of force-directed algorithm [15]. In his approach, repulsive force can be approximated by a quad-tree data structure, resulting in one of the fastest sequential algorithms. Other efficient multi-level algorithm includes FaP [12] which was later integrated in OGDF library [7]. Interestingly, Andreas Noack introduced the LinLog energy model, which is very effective in capturing clusters in the graph layout [26]. OpenOrd [23] by Martin et al. is a parallel multi-level algorithm following the force-directed model. OpenOrd is reasonably fast and generate layouts of large-scale graphs. Finally, ForceAtlas2 by Jacomy et al. introduced a more general framework for graph layout combining various features (repulsive force approximation by Barnes-Hut approach, LinLog mode, etc.) [17]. ForceAtlas2 is known to generate continuous layouts as a user can stop the program anytime with valid (possibly unoptimized) layouts. Our work is influenced by ForceAtlas2 and covers many features available in ForceAtlas2. The ForceAtlas2 paper [17] demonstrated that FR model can generate readable layouts but it can be slower than other energy models. In this paper, we addressed this challenge with a multi-threaded cache-efficient algorithm that is both faster and generates readable layouts.

Recently, Zheng et al. used a sequential SGD approach for graph drawing, which optimizes stress to generate the layout of a graph [32]. However, their software is very slow as it considers graph-theoretic distances in the optimization function. Both ForceAtlas2 and OpenOrd can run in parallel and are available in Gephi’s toolkit [4]. Force-directed algorithm has also been implemented in distributed platforms [1, 2] as well as for GPUs [6]. These tools cannot be run on any server because they rely on special hardware. We did not compare with these implementations because our objective in this paper is to develop a general-purpose and fast algorithm for multicore servers.

6 Discussions and Conclusions

In this paper, we present a parallel force-directed algorithm BatchLayout and its Barnes-Hut approximation BatchLayoutBH for generating 2D layouts of graphs. The presented algorithms are highly scalable, robust with respect to diverse classes of graphs, can be run on any multicore computer, and runs faster than other state-of-the-art algorithms. Aside from carefully chosen default hyper-parameters (e.g., initialization, minibatch size, energy model, learning rate, convergence conditions), BatchLayout provides flexibility to let users choose hyper-parameters that are suitable for the graph. In terms of flexibility, BatchLayout is comparable to the flexibility of ForceAtlas2 and can be integrated with any visualization tool.

The high performance of BatchLayout comes from two important optimizations that we made. First, BatchLayout exposes more parallel work to keep many processors busy. More parallel work comes from our minibatch scheme, which is also a widely-used technique in training Deep Neural Networks. Second, we incorporate cache-blocking technique commonly used to optimize linear algebra operations. Hence, this paper unifies two proven techniques from...
machine learning and linear algebra and delivers a high-performance graph layout algorithm. As a result, BatchLayout is highly scalable and runs significantly faster than other state-of-the-art tools.

BatchLayout generates graph layouts without sacrificing their aesthetic qualities. We visually and analytically verified the quality of layouts for different classes of graphs covering grid networks, small world networks, scale-free networks, etc. In all cases, BatchLayout generates good layouts that are similar or better than its peers.

BatchLayout is implemented as an open-source software with detailed documentation. This software depends on standard C++ and OpenMP libraries and is portable to most computers. We use simple Python scripts to visualize layouts generated by our software. Hence, we believe that BatchLayout will benefit many users in visually analyzing complex networks from diverse scientific domains.

While this paper only focuses on parallel force-directed algorithms, other classes of algorithm might generate better layouts than the algorithms considered in this paper. For example, tSNE can generate better quality layouts when it converges. However, such tools are often prohibitively expensive for large-scale graphs. For example, tSNE took 16 hours to generate a layout for OPF_6000, whereas BatchLayout generates a comparable layout in just few seconds. In fact, users can set a very low threshold in BatchLayout, decrease the batch size and increase number of iterations to get superior layouts from our software (e.g., see supplementary Table S4). We also compared our results with FM as available in OGDF library and found that BatchLayoutBH is always faster than FM for non-engineered settings in OGDF library (see supplementary Table S9 for runtime details). For two large graphs of Table 1, BatchLayoutBH is approximately 5x faster than FM.

This paper only considered shared-memory parallel algorithms since multicore computers and servers are prevalent in scientific community. However, BatchLayout can be easily implemented for GPUs and distributed-memory systems. A distributed algorithm will require larger minibatches if only “data parallelism” is used. However, if we also use graph partitioning (which is equivalent to model parallelism in deep learning), we can have enough parallelism for large-scale distributed systems. Expanding BatchLayout for other high-performance architectures and improving the aesthetic quality for massive graphs remain our future work.

Acknowledgements
We would like to thank Stephen Kobourov, Katy Borner, Iqbal Hossain, Felice De Luca and Bruce Herr for helpful discussions and comments on measures. Funding for this work was provided by the Indiana University Grand Challenge Precision Health Initiative.

References
[1] A. Arleo, W. Didimo, G. Liotta, and F. Montecchiani. Large graph visualizations using a distributed computing platform. Information Sciences, 381:124–141, 2017.
[2] A. Arleo, W. Didimo, G. Liotta, and F. Montecchiani. A distributed multilevel force-directed algorithm. IEEE Transactions on Parallel and Distributed Systems, 30(4):754–765, 2018.
[3] J. Barnes and P. Hut. A hierarchical o(nlogn) force–calculation algorithm. nature, 324(6096):446, 1986.
[4] M. Bastian, S. Heymann, and M. Jacomy. Gephi: an open source software for exploring and manipulating networks. In Third international AAAI conference on weblogs and social media, 2009.
[5] U. Brandes and C. Pich. An experimental study on distance-based graph drawing. In International Symposium on Graph Drawing, pp. 218–229. Springer, 2008.
[6] G. G. Brinkmann, K. F. Rietveld, and F. W. Takes. Exploiting gpus for fast force-directed visualization of large-scale networks. In 2017 45th International Conference on Parallel Processing (ICPP), pp. 382–391. IEEE, 2017.
[7] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel. The open graph drawing framework (ogdf). Handbook of Graph Drawing and Visualization, 2013:543–569, 2013.
[8] F. De Luca, I. Hossain, S. Kobourov, and K. Börner. Multi-level tree based approach for interactive graph visualization with semantic zoom. arXiv preprint arXiv:1906.05996, 2019.
[9] T. M. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. Software: Practice and experience, 21(11):1129–1164, 1991.
[10] E. R. Gansner, Y. Hu, and S. North. A maxent-stress model for graph layout. IEEE transactions on visualization and computer graphics, 19(6):927–940, 2012.
[11] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677, 2017.
[12] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In International Symposium on Graph Drawing, pp. 285–295. Springer, 2004.
[13] S. Hachul and M. Jünger. Large-graph layout algorithms at work: An experimental study. J. Graph Algorithms Appl., 11(2):345–369, 2007.
[14] D. Harel and Y. Koren. Graph drawing by high-dimensional embedding. In Int. symp. on Graph Drawing, pp. 207–219. Springer, 2002.
[15] Y. Hu. Efficient, high-quality force-directed graph drawing. Mathematical Journal, 10(1):37–71, 2005.
[16] W. Huang, S.-H. Hong, and P. Eades. Effects of sociogram drawing conventions and edge crossings in social network visualization. Journal of graph algorithms and applications, 11(2):397–429, 2007.
[17] M. Jacomy, T. Venturini, S. Heymann, and M. Bastian. ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. PloS one, 9(6):e98679, 2014.
[18] T. Kamada, S. Kawai, et al. An algorithm for drawing general undirected graphs. Information processing letters, 31(1):7–15, 1989.
[19] Y. Koren, L. Carmel, and D. Harel. Drawing huge graphs by algebraic multigrid optimization. Multiscale Modeling & Simulation, 1(4):645–673, 2003.
[20] J. F. Krüger, P. E. Rauber, R. M. Martins, A. Kerren, S. Kobourov, and A. C. Telea. Graph layouts by t-SNE. In Computer Graphics Forum, vol. 36, pp. 283–294. Wiley Online Library, 2017.
[21] O.-H. Kwon, T. Cnrowsanin, and K.-L. Ma. What would a graph look like in this layout? a machine learning approach to large graph visualization. IEEE transactions on visualization and computer graphics, 24(1):478–488, 2017.
[22] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. Physical Review E, 78(4):046110, 2008.
[23] S. Martin, W. M. Brown, R. Klavans, and K. W. Boyack. OpenOrd: an open-source toolbox for large graph layout. In Visualization and Data Analysis 2011, vol. 7868, p. 786806. International Society for Optics and Photonics, 2011.
[24] H. Meyerhenke, M. Nöllenburg, and C. Schulz. Drawing large graphs by multilevel maxent-stress optimization. IEEE transactions on visualization and computer graphics, 24(5):1814–1827, 2017.
[25] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.
[26] A. Noack. An energy model for visual graph clustering. In International symposium on graph drawing, pp. 425–436. Springer, 2003.
[27] A. Noack. Modularity clustering is force-directed layout. Physical Review E, 79(2):026102, 2009.
[28] H. C. Purchase. Metrics for graph drawing aesthetics. Journal of Visual Languages & Computing, 13(5):501–516, 2002.
[29] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome research, 13(11):2498–2504, 2003.
[30] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In Supercomputing ’93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing, pp. 12–21. IEEE, 1993.
[31] J. Zhang, B. Behzad, and M. Snir. Design of a multithreaded barnes-hut algorithm for multicore clusters. IEEE Transactions on parallel and distributed systems, 26(7):1861–1873, 2014.
[32] J. X. Zheng, S. Pawar, and D. F. M. Goodman. Graph drawing by stochastic gradient descent. IEEE Transactions on Visualization and Computer Graphics, 2019. doi: 10.1109/TVCG.2018.285997