Graph drawing in parallel with distributed memory

PETER LY

A parallel algorithm for graph drawing is presented. The algorithm is based on the Barnes-Hut approximation for the n-body problem.

CCS Concepts: • **Theory of computation → Parallel algorithms**; **Computational geometry**.

Additional Key Words and Phrases: Graph drawing, Parallel algorithms

ACM Reference Format:
Peter Ly. 2023. Graph drawing in parallel with distributed memory. 1, 1 (December 2023), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Given a set of data points and a set of values which indicate the strength of relationships between the points, one can define a *graph*, a mathematical object which combines these sets. One fundamental requirement of a graph from a viewer's perspective is a convenient representation of the data contained within the graph. While it is possible to list out the relationship between each pair of points for every pair of points, this representation is rarely intuitive and often unwieldy. Often, what is preferred for any data set of respectable size is a *graph drawing*, which is a 2D or 3D embedding of the data points along with line segments connecting the data points (one could also ask for 1D embeddings, but these become are generally unsatisfactory for all but the most simple data set, and embeddings into spaces of dimension greater than 3 are hard to visualize, naturally). The goal of this project is to propose an algorithm which draws a graph using many processors.

2 BACKGROUND

2.1 Graphs and graph drawings

An **graph** is a set of vertices V, and a set of edges $E \subseteq V \times V$. The relationships indicated by E may be symmetric or asymmetric – the graph is said to be *undirected* in the former case and *directed* in the latter case. The graph may additionally be equipped with a function which maps edges to numbers (often referred to as weights); in this case, the graph is said to be *weighted*.

An **embedding** of a graph is a way to place its vertices and edges in some space. The vertices are modelled as points whereas the edges are modelled curves between points. Desirable embeddings typically place vertices which have edges between them close to each other, and vertices which do not have edges between them farther away from each other. When the edges have weights, depending on the weights, it may be desirable to place the edges farther or closer away from each other. For example, if the edges indicate the distances between two points, then one might want smaller edge

Author's address: Peter Ly, pmly@umich.edu.

Manuscript submitted to ACM

weights to correspond to closer points in the embedding (and therefore a shorter curves), whereas if the edges indicate a number of matching features, then one might want larger edge weights to correspond to closer points.

We will refer to embeddings of graphs as **graph drawings**, as we will be mainly concerned with drawings which place the vertices and edges in 2D or 3D Euclidean space and model edges as curves. We will only address unweighted graphs for simplicity.

2.2 The Barnes-Hut approximation

The Barnes-Hut approximation [Barnes and Hut 1986] is an algorithm used in the simulation of n-body problems to reduce the number of force computations from $\Theta(n^2)$ to $\Theta(n \log n)$ (for example, in galaxy simulation where gravitational forces exist between all objects in the galaxy). At a high-level, the main idea of the Barnes-Hut approximation is that the force between close points should be computed directly while the force between distant points can be estimated forces between clusters of points which can be approximated as appropriately defined "average" points. More specifically, the approximation works in four steps:

(1) Represent the simulation space as a hierarchical tree structure, where each node in the tree represents a region of space and contains information about the total mass and center of mass of all points in the region. For example, the region can be divided into quadrants. Each region corresponds to a node in the tree.
(2) For each region, if it has many points, the region is recursively divided (this gives the lower levels of the tree structure), and if it does not, the quadrant is not divided (corresponding to a leaf in the tree structure).
(3) For each non-leaf in the tree, the center of the points in the tree is computed.
(4) The tree is traversed to compute the forces on each node. For each point, the approximation decides whether to compute pairwise forces (for example, if the points exist in the same region) or to compute approximate forces if the points are in completely different regions. The approximate forces are computed using the centers computed in step (3) in the latter case.

2.3 Force-based Graph Drawing

Force-based graph drawing is a technique which places the vertices in Euclidean space based on forces which are placed between vertices. The force-based method of Eadse [Eades 1984] and later Fruchterman and Reingold [Fruchterman and Reingold 1991] models these forces as springs which obey Hooke's law[1] between vertices. Every vertex pair has repulsive spring forces between the pair and each vertex pair which has an edge in the graph will have attractive spring forces. (Other possible models for forces between vertices are possible and we make no effort to characterize these different force models.) Thus, vertices which have an edge between them will be drawn to each other more than vertices which do not. One can also define forces based on the shortest path metric in a graph as specified in [Kamada and Kawai 1989].

Regardless of the way edge weights are defined, force-directed methods then rank graph drawings by a number which is sometimes called the *energy* of the layout. Graph drawings in which adjacent vertices are some specified distance away from each other and non-adjacent vertices are far away have low energy. Force-directed graph drawings seek to find graph drawings which minimize energy. Often, the output graph drawing is a locally minimum[2] energy drawing rather than a globally minimum energy drawing. Of course, a force-based graph drawing is not computed

[1] Hooke's law states that the amount of force required to stretch (or compress) a spring is proportional to the distance by which the spring is stretched (or compressed). Moreover, the coefficient of proportionality is a characteristic of the spring.
[2] Of course, the meaning of locally minimum depends on the exact algorithm and what local changes can be made to the drawing by the algorithm

in "one-shot". The methods of [Eades 1984] and [Fruchterman and Reingold 1991] work over several iterations and incrementally update the positions of vertices within each iteration. The algorithm of [Eades 1984] updates the positions of vertices by a small factor within each iteration, while the algorithm of [Fruchterman and Reingold 1991] updates the positions of vertices by larger amounts in earlier iterations and smaller amounts in later iterations (using the general framework of simulated annealing).

However, a force-directed graph drawing is difficult to compute directly for large graphs, as the number of forces is quadratic in the number of vertices in the graph. For moderately large graphs, this number of forces is impractical to compute outright and it can be difficult to guarantee that a drawing is not a locally minimum energy drawing rather than a globally minimum energy drawing.

One main technique has been of use in resolving the quadratic growth problem; rather than compute all forces between all pairs of vertices, one can instead compute the approximations of the forces. One technique to do so is to form a family of laminar sets of vertices[3] which indicates a hierarchy of sets of vertices. Then each set in the family can be placed in the graph drawing by modelling each set as a "supervertex" (so at higher levels of the laminar family, supervertices will be "collapsed" together to form higher-level supervertices). This is the approach used by several algorithms including but not limited to the algorithms of [Hadany and Harel 2001; Walshaw 2003] and many others.

3 WHY PARALLELIZATION IS DIFFICULT

The main difficulty one encounters when parallelizing a force-directed graph drawing algorithm is the computation of forces between the vertices. This is the bulk of computation as the movements of the vertices must be simulated by small time steps to compute the forces acting between vertices accurately, while still allowing for the graph to converge.

As mentioned above, one of the major approaches in literature thus far in parallelizing force-directed graph drawing algorithms is to first is to apply a Barnes-Hut tree or other tree-like structure to reduce the number of pairwise force computations from $\Theta(n^2)$ to $\Theta(n \log n)$ where n is the number of vertices in the graph and then compute forces approximately. While the gain in performance in this case is quite sizeable, there are still some difficulties to this approach. One is that the vertices in a Barnes-Hut tree will move around throughout the execution of the algorithm, which necessitates recomputation of the tree periodically.

Additionally, graphs may have strong local dependencies which cause there to be hard to separate sections of the graph. These sections may need to be treated as a single block to be processed by a single process, so this can make it difficult to develop parallel tasks from a snapshot of the force-simulations. These large blocks can then make the load-balancing difficult for the processors, leading to decreased efficiency.

One additional difficulty is the communication required between processes. If all processes working on drawing a graph try to draw independently from each other, one can find that nodes start overlapping when they should not[4], necessitating a non-trivial amount of communication between the processes to ensure that nodes do not overlap when they should not.

An orthogonal difficulty to all these is the desire for the drawing to satisfy some constraints. For example, one might want to minimize the edge lengths of a graph to make is as compact as possible. Generally, minimizing edge lengths for a local subgraph does not ensure that the edges lengths will be approximately minimized overall in the global graph. However, these are rather domain dependent, so general parallelization techniques will not by universally effective on these.

[3] A laminar set family is a family of sets where each pair of sets is either disjoint or related by containment.
[4] Much like how if several people enter a room blind-folded and choose seats independently, it is possible that there is a seat conflict.

4 METHODOLOGY

The code implementing the algorithm is available upon request. It currently is not in any publicly available source. The implementation uses:

- MPI for communication among processes in a distributed-memory parallel setting,
- METIS for computation of the graph partitions,
- OGDF for computing the drawings of subgraphs.

The development was quite fortunate, in that my preliminary analysis matched the actual outcome, in that separating the drawing problem into smaller subgraphs would provide reasonable speedup by decreasing the total amount of forces required to be approximated for each node.

4.1 Overview of the Algorithm

The algorithm has three main steps (and additionally reads a graph as input and outputs a graph drawing). Given an unweighted graph as input:

(1) Partition the graph into several subgraphs.
(2) Draw each subgraph.
(3) Assemble the local drawings to form a drawing of the original graph.

We proceed to give more details on each step.

4.1.1 Partitioning the graph. The graph is partitioned into several pieces. In the current implementation, the graph is separated into `numProcesses` pieces. This partition is computed using METIS, and the partition is constructed to (approximately) minimize the number of edges crossing between different partitions. The computation of the parts is performed on the root process. After the partition is computed by the root process, but before we begin using all other processes in an interesting way, the root process sends to each other process the vertices and edges which are completely contained in the partition that the process is responsible for drawing.

4.1.2 Drawing each subgraph. Each process, after receiving its subgraph, then can draw its subgraph independently of all other processes, in any way it would like. In this case, we instantiate this step by having each process draw its subgraph using a force-based graph drawing approximates pairwise forces by computing forces in a hierarchical way as mentioned in section 3.1. This is implemented using OGDF.

4.1.3 Assembling the local drawings. The last step is the most challenging in terms of parallelization, this is because each process now has many positions in space that it would like to place nodes at. We cannot place nodes necessarily allow nodes from two different processes to be placed close to each other, as they may have few edges between them (and should therefore be separated).

The key idea here is that we approximate each subgraph by a "supernode", and draw the graph induced by these supernodes (again, there is freedom in how this drawing is computed; the current implementation uses a force-based algorithm which applies the Barnes-Hut approximation). This gives an approximate layout for each subgraph drawing. Because the partitions were computed so that the number of crossing edges is approximately minimized, the overall

quality of the drawing should not be greatly affected by this layout[5]. This approximation is performed solely on the root process (which knows the full graph, and can compute this drawing).

After this drawing is computed, the root node broadcasts the positions of every supernode in the graph drawing to all processes. Then, each process can simultaneously compute (using a deterministic strategy) a rectangle which is close to the position of the supernode corresponding to its subgraph, and move its subgraph drawing to be contained entirely in the rectangle. These rectangles are computed using a greedy strategy. After all this is done by each process, the root process gathers all of the vertex positions from each process, and composes the final drawing.

4.2 Implementation details

We did not find a significant difference between using blocking and non-blocking communication; this is within expectations, as much of the communication is between a process and the root process. The only possible time in which we would likely have found any difference between blocking and non-blocking communication would likely be the all-to-all communication where each process needs to share its rectangle with all other processes, but the amount of data being transferred here is so small that the effect of using non-blocking communication would likely be negligible.

5 EXPERIMENTAL RESULTS

5.1 The input data/graphs

We examine the performance of the algorithm on social network data and road networks. Road networks are interesting because they are planar (and therefore sparse, in graph theoretic terms). Social network data is interesting because messaging graphs typically exhibit highly connected nodes with several isolated nodes. Furthermore, these graphs are relatively sparse, which is required to have a reasonable graph drawing. The data sets have been obtained from the SNAP database and have been suitably processed to be used in our algorithm.

5.1.1 Social network data. In GitHub, users can "star" repositories in which they are interested. Mutual "starring" relationship data among GitHub users who starred at least 10 repositories was collected from GitHub in 2019 [Rozemberczki et al. 2019]. Vertices represent users who starred at least 10 repositories. Edges represent pairs of users who have starred at least one of each others repositories. Feather, a slightly smaller network is provided by [Rozemberczki and Sarkar 2020].

5.1.2 Road networks. Road network data was collected for California, Pennsylvania, and Texas [Leskovec et al. 2008]. Vertices represent intersections. Edges represent roads connecting the intersections.

Table 1. Summary of graph parameters

	GitHub	PA Roads
Number of vertices	37,700	1088092
Number of edges	289,003	1541898

[5]The main concern is that even though vertices which are separated by parts may be drawn closely together in the supernode graph drawing, they may be quite separated still if one local drawing places a node on the far left and the other places a node on the far right.

5.2 Hardware

The algorithm was run using the University of Michigan Great Lakes computing clusters. Results were collected using various numbers of processors.

5.3 Timing results

Timing results for $p = 1$ processors were computed using the OGDF implementation of the drawing algorithm without constructing subgraphs. Timing results for $p = 2, 4, 8, 16, 32, 64$ processors were collected using our algorithm. Timing in both cases start immediately after the input graph is read, and end immediately before the graph drawing is output.

We also collected timings for the three phases of the algorithms. For brevity, we only show the results on the Pennsylvania road network, as that is the largest graph we tested the algorithm on. The efficiency and speedup results are omitted for the Pennsylvania road network, as the serial speed was too long running to keep on the GreatLakes cluster.

Table 2. Timing results for the GitHub Network

Number of processors	Time	Phase 1 Time	Phase 2 Time	Phase 3 Time	Speedup	Efficiency
1	12.0021	–	–	–	–	–
2	14.2963	0.183941	10.0246	4.08773	0.8393780209	0.4196890104
4	5.74849	0.204528	3.94965	1.59432	2.087504719	0.5218761797
8	2.79488	0.282741	2.46844	0.0436995	4.293565377	0.5366956721
16	1.6478	0.39784	0.730897	0.519067	7.282437189	0.4551523243
32	1.62831	0.520159	0.53946	0.568703	7.369604068	0.2303001271
64	1.51459	0.822847	0.263881	0.427857	7.92293624	0.1237958788

Table 3. Timing results for the Pennsylvania road network

Number of processors	Time	Phase 1 Time	Phase 2 Time	Phase 3 Time
64	209.579	1.49979	96.1309	111.949
128	110.407	1.81997	54.1825	54.4047
256	56.4253	3.20057	26.0663	27.1585
512	36.9475	9.32519	12.8906	14.7317

5.4 Output drawings

We present two different drawings of the same network to demonstrate some qualitative properties of the current drawing strategy. Note that due to the greedy strategy, the nodes in our algorithm tend to the top left, while the nodes in the standard force-directed algorithm are pulled to the center.

6 ANALYSIS

We proceed to provide analysis of our results. Verification is rather difficult, but the speedup and efficiency are plain to see.

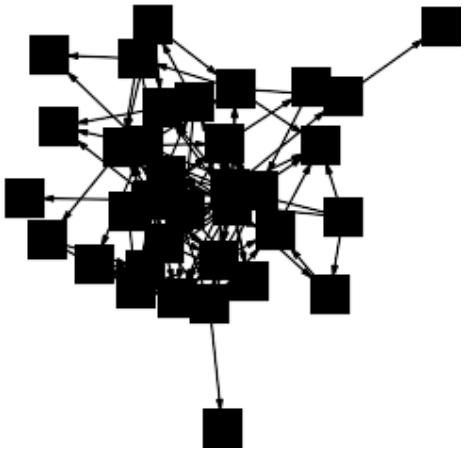

Fig. 1. A drawing of a simple network using a standard force-directed algorithm.

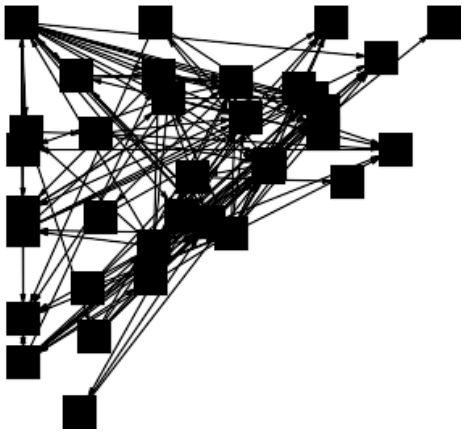

Fig. 2. A drawing of a simple network with 4 processors.

6.1 Verification of results

To determine if the algorithm is functioning correctly, we compare the output the algorithm to the OGDF implementation of the force-directed graph drawing algorithm which uses the Barnes-Hut approximation. Note that while the algorithms will in general output different graph drawings, because both of the algorithms operate on the same core principle (constructing a drawing which minimizes "potential energy" in the vertex configuration), the graphs should share some structural features such as dense subgraphs[6]

6.2 Speedup and Efficiency

As expected, the speedup and do not continue being linear. The timings indicate that the time spent partitioning the graph tends to decrease the local subgraph drawing time significantly (phase 2), as there are fewer forces to worry

[6]This one reason why it is important that we can compute the partitions so that the number of crossing edges is approximately minimized.

about about. However, the time taken to partition (phase 1) increases as the number of processors increases - this is quite natural as it takes more computation to ensure than smaller pieces are evenly balanced in METIS. The time taken to reassemble the drawings generally decreases as the number of processors increases. This is because the adjustment of each processors partition when moving their partition to enter in a rectangle in the overall graph requires less adjustment; there are simply fewer points to move for each processor when the number of processors increases.

Of course, we also see that there are diminishing returns as the number of processors increase (c.f. Table 3). This seems to be because the third phase has a fundamental lower bound of requiring that every node position be transmitted to the root process to output the final drawing.

7 CONCLUSION

It seems like there is some promise to drawing graphs by partitioning them first, thereby reducing the number of approximated forces, and then reassembling the drawing. Based on the data from the Pennsylvania road network, this approach seems promising in that it can be scaled to a relatively large number of processors while maintaining that reasonable speedup. However, it is unclear if this technique can be applied GPUs or more fine-grained models, as the amount of broadcast and gather communications is sizeable – it would likely be prohibitively expensive to apply these operations. A different strategy seems necessary for those kinds of settings.

While not much can be said for aesthetics of a graph, one thing which can be stated is that a sophisticated algorithm (at least a more sophisticated algorithm than a greedy algorithm for repositioning rectangles) is likely necessary to ensure that the layout of the nodes when reassembling local subgraphs into a global subgraph is satisfactory.

8 FUTURE WORK

It would be good to explore other graph layout algorithms as a base algorithm, such as spectral algorithms[Koren 2005] and projective drawings[Harel and Koren 2002]. It would also be good to examine the performance of the algorithm on many other datasets to evaluate the performance of the algorithm and to determine if any parameters of the algorithm can be optimized. One parameter which certainly requires tuning is the batch size, which is dependent on the data set and desired resolution of the graph drawing. One type of graph which would be interesting to examine are dynamic graphs, which change over time and ask for incremental updates of the graph drawing.

For force-based graph drawing, it would be interesting to compare the performance of an MPI implementation of the algorithm with a CUDA implementation of the algorithm. A CUDA implementation may be promising based on the improvement in performance for simple arithmetic operations which would need to be performed over each iteration. However, the performance of a GPU may not be ideal in the dynamic graph setting because updates to a graph may not be large enough to utilize the full capabilities of a GPU – small updates to the graph should only require small changes to the graph drawing, so it is unclear whether the time saved performing incremental updates on the graph would offset any time take from loading data to and from the GPU.

Outside of force-based graph drawing, it may be worthwhile to consider parallelization of the spectral graph drawing and projected graph drawing methods. Such methods have the benefit of having a body of linear algebra to support the methods analytically and one may be able to show stronger convergence guarantees based on convergence guarantees from eigenvector approximating algorithms and principle component analysis. One challenge to overcome is implementing the relevant numerical linear algebra tools in parallel. Depending on the context, it may be difficult to ensure that the load-balancing is appropriate among processors, especially as the structure of the graph cause some processors to handle primarily isolated vertices.

Additionally, it would be interesting to see if a GPU-based implementation of the spectral and projected graph drawing methods, as there seem to be parallel implementations of the relevant eigenvalue/eigenvector computing algorithms. As with the force-based graph drawings, one parameter which may make this implementation inefficient is the ratio of updates which must be made to the amount of data which must be loaded/unloaded from the GPU (for example, from inserting and deleting vertices).

REFERENCES

Josh Barnes and Piet Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature* 324, 6096 (01 Dec 1986), 446–449. https://doi.org/10.1038/324446a0

Peter Eades. 1984. A Heuristic for Graph Drawing. https://api.semanticscholar.org/CorpusID:63936426

Thomas M. J. Fruchterman and Edward M. Reingold. 1991. Graph Drawing by Force-Directed Placement. *Softw. Pract. Exper.* 21, 11 (nov 1991), 1129–1164. https://doi.org/10.1002/spe.4380211102

Ronny Hadany and David Harel. 2001. A multi-scale algorithm for drawing graphs nicely. *Discrete Applied Mathematics* 113, 1 (2001), 3–21. https://doi.org/10.1016/S0166-218X(00)00389-9 Selected Papers: 12th Workshop on Graph-Theoretic Concepts in Com puter Science.

David Harel and Yehuda Koren. 2002. Graph Drawing by High-Dimensional Embedding. In *Graph Drawing*, Michael T. Goodrich and Stephen G. Kobourov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–219.

Tomihisa Kamada and Satoru Kawai. 1989. An algorithm for drawing general undirected graphs. *Inform. Process. Lett.* 31, 1 (1989), 7–15. https://doi.org/10.1016/0020-0190(89)90102-6

Y. Koren. 2005. Drawing graphs by eigenvectors: theory and practice. *Computers & Mathematics with Applications* 49, 11 (2005), 1867–1888. https://doi.org/10.1016/j.camwa.2004.08.015

Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR* abs/0810.1355 (2008). arXiv:0810.1355 http://arxiv.org/abs/0810.1355

Benedek Rozemberczki, Carl Allen, and Rik Sarkar. 2019. Multi-scale Attributed Node Embedding. arXiv:1909.13021 [cs.LG]

Benedek Rozemberczki and Rik Sarkar. 2020. Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*. ACM, 1325–1334.

Chris Walshaw. 2003. A multilevel algorithm for force-directed graph-drawing. *Journal of Graph Algorithms and Applications* 7, 3 (2003), 253–285. http://eudml.org/doc/51536