Eurographics Symposium on Parallel Graphics and Visualization (2008)
J. Favre, K. - L. Ma, and D. Weiskopf (Editors)

# A Scalable Parallel Force-Directed Graph Layout Algorithm

Anna Tikhonova[†] and Kwan-Liu Ma[‡]

Visualization and Interface Design Innovation (VIDi) Lab
University of California at Davis, USA

**Abstract**

*Understanding the structure, dynamics, and evolution of large graphs is becoming increasingly important in a variety of fields. The demand for visual tools to aid in this process is rising accordingly. Yet, many algorithms that create good representations of small and medium-sized graphs do not scale to larger graph sizes. The exploitation of the massive computational power provided by parallel and distributed computing is a natural progression for handling important problems such as large graph layout. In this paper, we present a scalable parallel graph layout algorithm based on the force-directed model. Our algorithm requires minimal pre-processing and achieves scalability by conducting the layout computation in stages - a portion of the graph at a time, and decreasing the amount of inter-processor communication as the layout computation progresses. We provide the implementation details of our algorithm, evaluate its performance and scalability, and compare the visual quality of the resulting drawings against some of the classic and the fastest algorithms for the layout of general graphs.*

Categories and Subject Descriptors (according to ACM CCS): G.2.2 [Discrete Mathematics]: Graph Theory: Graph Algorithms; H.5.0 [Information Systems]: Information Interfaces and Presentation: General

## 1. Introduction

In many diverse fields, including software engineering, bioinformatics, and social network analysis, graphs provide a powerful visual notation for representing relationships among data items. Such graphical representation is useful only if it reveals global and local structures within the data and provides a means of discovering patterns, which are often hidden. It is the goal of a graph layout algorithm to generate such a representation.

In the last few decades, the graph drawing community proposed numerous algorithms for automatically computing graph layouts. Refer to [DBETT99, KW01] for a comprehensive survey of the field. Although graph layout and visualization have often been the subject of research, most of this research focuses on creating more aesthetically pleasing layouts of relatively small graphs or special classes of graphs, such as trees and planar graphs. There are fewer general purpose graph drawing algorithms, and even fewer systems that deal effectively with large graphs. Systems that can handle datasets with thousands of vertices include NicheWorks [Wil99], H3Viewer [Mun98], LGL [LGL], Tulip [Tul], and Royére [Roy] (built using GVF [MHM01]).

One reason for this discrepancy is that the running times of many graph layout methods are too slow to accommodate larger graph sizes. For this study, we looked at various existing approaches for decreasing the running time of graph drawing algorithms. A reduction in running time is only the first step to designing a scalable solution. The large size of a graph can compromise the performance of an algorithm that works reasonably well on small and medium-sized graphs. If a layout algorithm produces nice drawings for graphs of several hundred vertices, this is not a guarantee that it will scale up to several hundred thousand vertices. We argue that scalability should act as a guiding heuristic for the design of graph visualization systems.

Parallel and distributed computing provide massive computational power demanded by the coming generations of scientific applications. It seems natural to exploit this power to provide fast and scalable algorithms for important problems such as large graph layout. It is difficult to compute graph layouts in parallel because we cannot simply use a divide-and-conquer approach and expect the algorithm to

[†] e-mail: tikhonov@cs.ucdavis.edu
[‡] e-mail: ma@cs.udavis.edu

scale. A scalable solution must be able to use increasing numbers of computing resources to process large datasets more efficiently (i.e., it must minimize the communication overhead relative to the amount of local computation).

In this paper, we use parallel processing to develop a scalable parallel algorithm that delivers an approximation to a force-directed model. Our algorithm requires minimal preprocessing and achieves scalability by conducting the layout computation in stages - a portion of the graph at a time, and decreasing the amount of inter-processor communication as the layout computation progresses. The criteria we use for evaluating our graph layout algorithm is performance (running time), scalability, and visual quality of the resulting drawings. The visual assessment task involves comparing the aesthetic qualities of our layouts with the drawings produced by the other graph drawing algorithms.

The rest of the paper is organized as follows. In Section 2, we review existing work on sequential and parallel graph layout algorithms. In Section 3, we provide the details of our algorithm and discuss the compromises we made to design an algorithm that scales to large problem sizes. We provide a visual assessment of results and evaluate the algorithm's performance and scalability in Sections 4 and 5, respectively. Finally, in Section 6, we provide some concluding remarks and discuss future work.

## 2. Related Work

Because of their conceptual simplicity and ability to produce aesthetically pleasing layouts, force-directed algorithms have become the methods of choice for drawing general graphs. Many visualization tools employ classical force-directed algorithms that start with a random placement of graph vertices and apply optimization methods to find the minimum of a chosen energy function. The minimum energy is achieved when the magnitude of net force on every vertex is zero, i.e. an equilibrium state is reached. A layout with minimum energy represents the most aesthetically pleasing drawing of a graph.

The first practical algorithms for graph drawing include the force-directed placement algorithm of Quinn and Breur [QB79] and the spring embedder of Eades [Ead84]. Some other popular methods include the algorithms developed by Kamada and Kawai [KK89], Fruchterman and Reingold [FR91], Davidson and Harel [DH96], and Noack [Noa06].

The problem with force-directed layout algorithms is that they do not scale to graphs of large size. Even some of the best variants such as the algorithm developed by Frick et al. [FLM94] are estimated to have complexity of $O(|V|^3)$, where $|V|$ is the number of vertices in a graph. Given a large number of nodes, this running time is unacceptably slow.

To generate graph layouts in less time, one natural approach is to compute approximate solutions that employ heuristics rather than use expensive optimal solutions. Some algorithms employ partitioning of the spatial domain in order to approximate or even ignore interactions between graph vertices located far away from each other. For example, the grid-variant algorithm (GVA) of Fruchterman and Reingold [FR91] subdivides the rectangular drawing area into a regular grid. The grid is used to determine whether two vertices are located close enough to each other in order to significantly influence each other's positions in space. The forces between the vertices located far enough apart are ignored. An example of a force-directed graph layout algorithm that uses a more complex hierarchical partitioning of the spatial domain is the FADE [QE01] algorithm by Quigley and Eades, which is based on the well-known $N$-body method by Barnes and Hut [BH86]. The recent survey paper by Hachul and Jünger [HJ05b] cites more highly-efficient force-directed techniques. Their paper also cites other very fast methods for drawing large graphs that are based on techniques of linear algebra instead of physical analogies, but strive for the similar esthetic drawing criteria.

Multi-level approaches such as the ones by Harel and Koren [HK00] and Walshaw [Wal00] provide a heuristic method that utilizes a coarsening technique to cluster a graph. The coarsened graph is laid out in space and the other vertices are reintroduced in un-coarsening steps until a final drawing is produced. The recent work by Frishman and Tal [FT07] proposes a multi-level force-directed graph layout on the GPU. The authors employ geometrical spectral partitioning to break the graphs into smaller, similarly-sized pieces, based on graph connectivity. The partitioning strategy is used both for the multi-level approach and for computing the layouts in parallel using graphics hardware. Graph partitioning has been an active field of research in the HPC community. Refer to [KL70,FM82,DFF*03] for information on parallel partition refinement algorithms.

There are several other parallel graph layout algorithms available. Monien et al. [MRS95] parallelized the simulated annealing algorithm by Davidson and Harel [DH96]. Both of these algorithms were developed in the mid nineties. Though Coleman and Parker improved the sequential method, as described in [CP96], the parallelized algorithm has not been altered. Mueller et al. [MGL06] address the problem of large graph layout and visualization using larger display areas (e.g., display walls). In their force-directed algorithm, each processor is assigned its own visual sub-domain and the vertices are divided evenly among the processors. Similarly to the $N$-body method by Barnes and Hut, their algorithm computes repulsion forces only between neighboring vertices. For that purpose, each local sub-domain is divided into a grid of cells, and the repulsion forces are computed only between neighboring cells. In their paper, Mueller et al. demonstrate scalability for problems containing up to eight thousand vertices, run on up to eight processors.

Our research is aimed at developing a scheme that re-

quires minimal pre-processing and computes layouts for graphs of up to a several hundred thousand vertices, using a significantly larger number of processors.

## 3. Naive Approach

In this section, we describe a naive approach to designing a parallel graph layout algorithm that approximates a force-directed model. The following subsections delve into greater detail about specific steps of the algorithm and discuss the design challenges we faced to develop an algorithm that scales to large problem sizes.

Before the computation of a graph layout can take place, the graph vertices have to be distributed among processors. In our method, the number of vertices assigned to each processor does not depend solely on the total number of vertices in a graph, but also on vertex degrees, i.e. the number of vertex neighbors. We provide a more detailed explanation of our vertex distribution strategy in Section 3.1.

As in the classic force-directed methods, the new position of a vertex in a graph is determined by iteratively computing local and inter-processor (global) attractive and repulsive forces during each iteration of the algorithm. Vertices that are connected via an edge (neighbors) attract each other and all vertices repel. The basic idea is that neighboring vertices need to be placed close (but not too close) to each other, and clusters must be separated from other clusters to avoid clutter. The equilibrium state is reached when the attractive and repulsive forces for each vertex cancel each other out and no further movement of vertices is possible.

We evaluated the performance of this approach using several test graphs of varying size and complexity. The sizes and descriptions of the test graphs are provided in Table 1. The performance results, provided in Table 2, do not demonstrate the desired scalability. Subsection 3.3 provides an overview of our optimized scalable parallel algorithm.

### 3.1. Vertex Distribution Scheme

Many parallel methods employ a simple strategy for distributing data elements among processors: given two parameters $n$ and $p$, where $n$ is the number of data elements and $p$ is the number of processors, each processor is assigned $\lfloor \frac{n}{p} \rfloor$ elements. This approach works well in cases where all the elements are equivalent/independent.

Albert and Barabási [RB02] have shown that the degree distribution of vertices in many graph models of real-world systems has high variance. Examples of such models include social and neural networks, citations of scientific articles, hyperlink structures, etc. For instance, in one of the real-world graphs we used in our experiments the minimum vertex degree is 1 and the maximum vertex degree is 105. Using a strategy where every processor is assigned an approximately equal number of graph vertices, but which does not account for the number of vertex neighbors, may lead to poor load balancing. Imagine a graph that contains only a few vertices of high degree that share edges with many low degree vertices that have been distributed to other processors. If all the higher degree vertices are assigned to a single processor, that processor might have to conduct more inter-processor communication than other processors.

Load balancing is especially important for a scheme that does not employ vertex migration. In such a scheme, the vertices and their neighbors are always located on the same processors they were assigned to during the initial distribution. Our arguments against vertex migration are provided in Section 3.2. In our algorithm, we employ a distribution strategy that takes into account the number of edges each vertex shares with its neighbors.

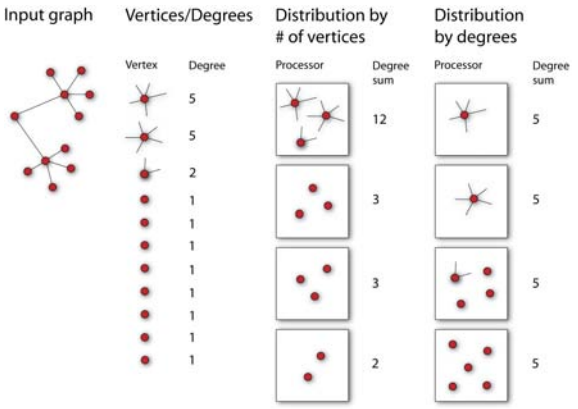



**Figure 1:** *An illustrative example of two different vertex distribution strategies. The vertices ($n = 11$) are represented as points, edges - as line segments, and processors ($p = 4$) - as boxes. The first strategy shown is distribution that depends solely on the number of vertices. Each processor is assigned $\lfloor \frac{n}{p} \rfloor$ vertices. The use of this strategy might result in one processor being assigned a large number of high degree vertices, thus having to conduct more inter-processor communication than other processors. The second strategy shown tries to alleviate this situation by taking into account vertex degrees. This strategy guarantees that the sums of degrees of the vertices assigned to each processor are approximately equal, thus providing a more balanced distribution.*

Figure 1 illustrates the two distribution strategies. The vertices are represented as points ($n = 11$), edges - as line segments, and processors - as boxes ($p = 4$). The first strategy shown is the simple distribution scheme that depends only on the number of vertices. In this example each processor is assigned $\lfloor \frac{11}{4} \rfloor$ vertices. In a case where the neighbors of vertices assigned to processor 0 have been distributed mostly to other processors, processor 0 would have to conduct more inter-processor communication than other processors. The second vertex distribution scheme alleviates this possible

imbalance by distributing the vertices in a way that guarantees that the sums of degrees of the vertices assigned to each processor are approximately equal.

### 3.2. Data Locality

As we mentioned previously, most graph layout algorithms assign random initial positions to the vertices (i.e., positions that do not depend on graph properties such as connectivity) before the computation takes place. Then, the vertices are allowed to move around until an equilibrium state is reached. One challenge is to decide whether to implement vertex migration (move vertices from one processor to another when they leave their processor's spatial sub-domain or coordinate range into another processor's sub-domain) or to allow the processor to keep its vertices throughout the entire computation even though their positions in space are changing. This decision depends on the possible communication overhead associated with each of the two schemes.

A naive vertex migration strategy, in which the vertices are not regularly re-distributed, may lead to situations in which the clustered vertices are gathered on just a few processors. These processors might not only have to perform more local computation, but also conduct more inter-processor communication than other processors. Our goal for the design of this algorithm is to minimize the amount of pre-processing and inter-processor communication. Therefore, we decided against vertex migration. In our algorithm, each processor keeps the original number of vertices throughout the entire execution.

When the local force computation takes place, no communication is necessary, because all the data required for the computation is local. On the other hand, when the inter-processor forces need to be computed, each processor needs to know with which processors it has to communicate to conduct its own force calculations. Note that this process has to be performed only once, since we do not implement vertex migration. All the vertices with non-local neighbors, assigned to a particular processor, are grouped together, and an aggregated list of their coordinates is sent to that processor. The receiving processor computes the forces between the received vertices and its own vertices. Then, it sends back the list of computed forces.

### 3.3. Optimized Algorithm

Our optimized algorithm uses a preprocessing step that employs an *importance metric* to assign a level of *importance* to each graph vertex. Then, the vertices are divided into groups according to their importance. In our experiments, we observed that clusters generally form around vertices of high degree. Therefore, we use vertex degree as a measure of importance and initially sort all the vertices by degree.

Then, we take a certain percentage of the vertices of highest importance and compute a good layout using our parallel force-directed method. We call the resulting layout the *core* of the graph. Then, in each stage of the algorithm, we add another group of vertices in order of their importance, until all the graph vertices are included.

The most important aspect of this technique is that once a layout is computed for the newly introduced group of vertices, these vertices are made *static*, i.e. the positions of these vertices do not change in the future iterations of the algorithm. Note that no further force computations are performed for the stationary vertices as the algorithm progresses. When the vertices belonging to the next group in the queue are added to the graph, the algorithm computes the positions of the newly added vertices taking into account the edge connections to static vertices. In the later stages of the algorithm, the number of these connections decreases (vertices introduced later have lower degrees). Once the new layout is computed, all the vertices in the layout are also made static.

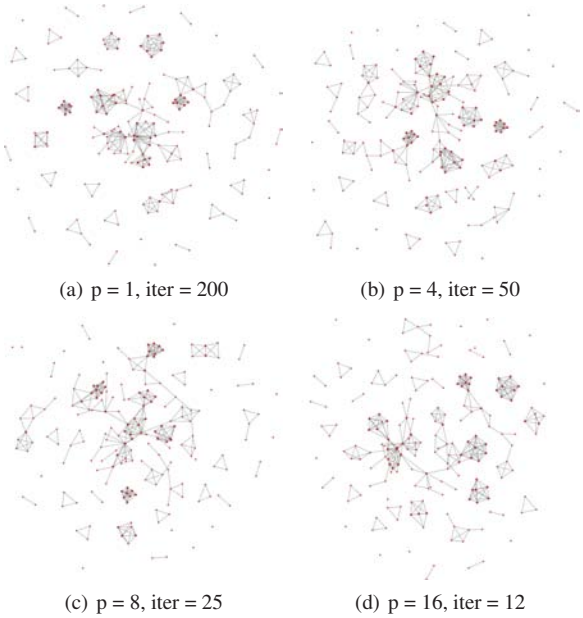
The number of stages in the process of layout construction can vary depending on the graph structure. The percentage of vertices to be inserted at each stage may also vary. This percentage should depend on the distribution of graph degrees. For example, if the goal is to see the clustering of high degree vertices and most of the nodes in a graph have low degrees, a smaller number of stages may yield a result of sufficient quality. In our experiments, we observed that the randomly placed vertices of low degree do not obscure the clusters formed by vertices of high degree. On the other hand, given a very balanced distribution of vertex degrees, a larger number of stages is necessary to produce a meaningful representation of the graph structure.

To improve the performance of the algorithm, the inter-processor attractive and repulsive forces are computed between all pairs of vertices only during the first stage of the algorithm, when the core layout is computed. This is done because the vertices are more volatile in the first few iterations of the force-directed algorithm and are expected to move longer distances. Afterwards, all the local attractive and repulsive forces are still computed between the local vertices, but the global forces are computed only between the neighboring processors. The remaining vertices are considered too far away to significantly influence the minimum energy computation and are not used in the force calculations. This modification greatly reduces the amount of necessary intra-processor communication in the successive stages of the algorithm. The size of the processor neighborhood depends on the total number of processors used in the layout computation. In the last stage of the algorithm, the size of the neighborhood is eight processors (the closest neighbors), unless the total number of processors is less than eight.

In Section 5, we discuss the performance of our optimized algorithm and demonstrate that it is a scalable solution for computing layouts of large graphs.

| *graph* | *\|V\|* | *\|E\|* | *Description* |
|---|---|---|---|
| coauthor | 200 | 664 | Bibliography dataset |
| kazaa288 | 1550 | 8028 | P2P network |
| add32 | 4960 | 9462 | 32-bit adder |
| 4elt | 15606 | 45878 | 2D FE mesh |
| gcc-fail | 71991 | 81764 | Memory graph |
| 0109 | 209581 | 260385 | Internet map |

**Table 1:** *Test graph sizes and descriptions.*



(a) p = 1, iter = 200 (b) p = 4, iter = 50

(c) p = 8, iter = 25 (d) p = 16, iter = 12

**Figure 2:** *Drawings of the coauthor graph produced using different numbers of processors (p). Also shown are the numbers of iterations (iter) required to generate each particular drawing. Naturally, the larger the number of processors, the fewer iterations of the algorithm are required to generate a similar quality of layout.*

## 4. Visual Assessment

A formal visual assessment of a graph drawing might include the calculation of the total energy in an underlying force model or the measurement of a desired esthetic criteria, for example, the crossing number. In practice, the quality evaluation of a graph drawing is a highly subjective process. A graph representation may be useful to an individual user only if he or she is satisfied with it.

To assess the visual quality of the drawings produced by our algorithms, we compare them to the layouts produced by the classic and the fastest force-directed layout methods. For our experiments, we chose some of the most popular and widely accessible graphs commonly used to evaluate graph layout algorithms. We tested both connected graphs and graphs that have multiple unconnected components. We tested only unweighted graphs. The graph sizes and descriptions are provided in Table 1.

The first graph we used for our visual assessment task is the *coauthor* graph, which is a bibliography dataset with multiple unconnected components. Figure 2 shows drawings of this graph produced using different numbers of processors. The *coauthor* graph is small yet interesting enough to demonstrate the visual quality of our algorithm as the number of processors increases. This graph was used as one of the experimental graphs in the paper by Shen et al. [SOTM06], where they present BiblioViz - a system for visualizing bibliography information. The layout produced by our algorithm shows the overall shape of the graph with the largest connected component in the middle, encircled by smaller unconnected components. The results presented in Figure 2 demonstrate that our algorithm handles disconnected components well and that the layout quality remains acceptable when the number of processors increases. Figure 4 also shows that the larger the number of processors used in the computation, the fewer iterations of the algorithm are required to generate a similar quality of layout.

The second graph we used for our experimental evaluation is add32, which is a representation of an electronic circuit, a 32-bit adder. It has a tree-like structure with many outlying branches. The progression and the final result of our optimized algorithm can be seen in Figure 3. This graph was used as one of the experimental graphs in the paper by Walshaw [Wal00]. It was also used in the survey paper by Hachul and Jünger [HJ05b]. In the survey paper, the drawing produced by our algorithm is similar in quality to the one produced by the $FM^3$ algorithm by Hachul and Jünger [HJ05a]. Our optimized algorithm is able to show the overall structure of the graph much better than the classic algorithm and the more efficient force-directed techniques.

The third graph we used for our evaluation is the *gcc-fail* graph, a memory graph depicting the state of GCC 2.95.2. It represents a faulty program state - one single pointer points to the wrong element, causing GCC to crash. Figure 5 shows a drawing of this graph produced by our optimized algorithm using 32 processors. The important feature of the graph (incorrect pointer) is clearly revealed by our algorithm.

## 5. Performance and Scalability Evaluation

Our primary goal was to design a graph layout algorithm that scales to large graph sizes. We evaluated the performance and scalability of our algorithm on a variety of real-world graphs of sizes up to approximately 200,000 vertices.

Table 3 provides the performance results achieved by our optimized algorithm for eight graphs of varying size and complexity. Figure 6 contains plots showing the performance results of our algorithm on three different graphs. These experiments were performed on the PSC's BigBen

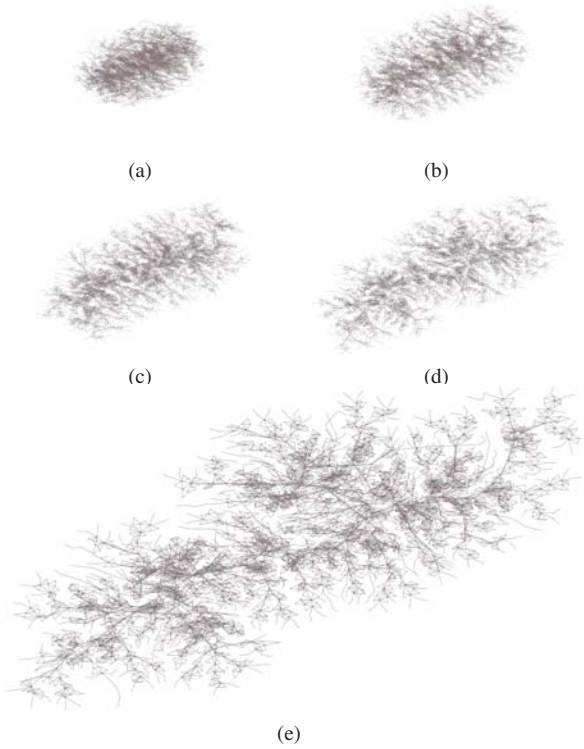



**Figure 3:** *Layout progression of add32, using 4 processors.*

| *p* | *kazaa288* | *add32* | *4elt* | *gcc-fail* |
|---|---|---|---|---|
| 4 | 21.78 | 154.37 | 1777.95 | N/A |
| 8 | 12.38 | 87.06 | 1004.62 | 18989.61 |
| 16 | 6.95 | 49.07 | 549.37 | 9817.32 |
| 32 | 5.08 | 28.89 | 291.530 | 4940.71 |
| 64 | 3.56 | 16.23 | 157.08 | 2850.63 |

**Table 2:** *Performance results (in seconds) for the naive algorithm on a variety of graphs. Graph sizes and descriptions are provided in Table 1.*

Cray XT3 cluster. All the measurements were conducted for 100 iterations of the algorithm.

As Table 3 and Figure 6 show, our optimized algorithm scales well. The kazaa288 graph demonstrates poorer scalability because this problem is too small to benefit significantly from parallel processing. For 512 processors, each processor is conducting the layout computation on just a few (3 - 4) vertices. In this case, the amount of communication overhead outweighs the almost insignificant amount of local computation. The larger two graphs show very good scalability achieved through the combination of two factors: conducting layout computation on only a portion of the graph vertices in each stage of the algorithm and incrementally reducing the amount of inter-processor communication as the layout computation progresses.

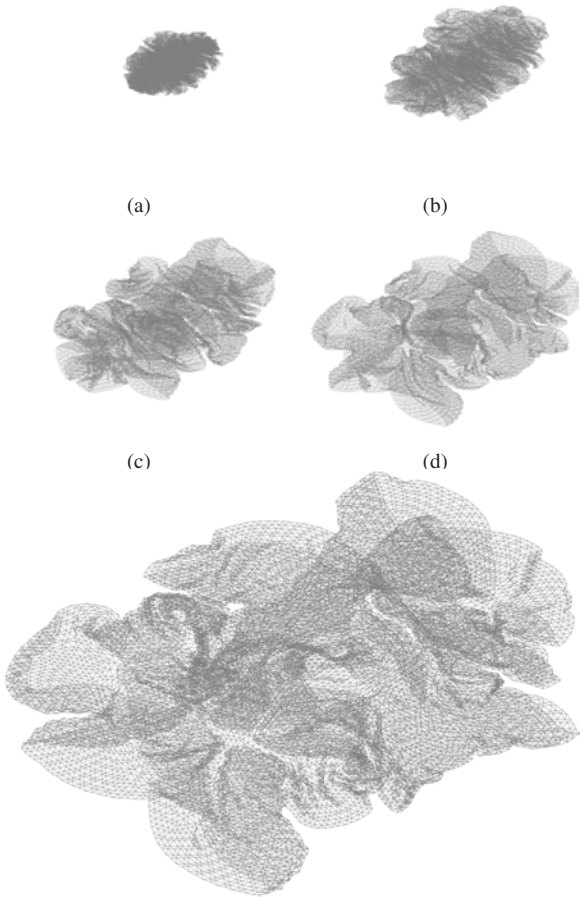



**Figure 4:** *Layout progression of 4elt, using 32 processors.*

| *p* | *kazaa288* | *add32* | *4elt* | *gcc-fail* | *0109* |
|---|---|---|---|---|---|
| 4 | 14.53 | 74.76 | 440.57 | 9309.18 | N/A |
| 8 | 6.10 | 19.91 | 169.80 | 3518.41 | 26533.60 |
| 16 | 2.28 | 5.06 | 69.87 | 1099.88 | 6845.16 |
| 32 | 0.95 | 1.33 | 31.14 | 638.34 | 2567.89 |
| 64 | 0.47 | 0.39 | 7.89 | 299.40 | 694.24 |
| 128 | 0.18 | 0.14 | 2.66 | 84.12 | 228.89 |
| 256 | 0.12 | 0.08 | 0.81 | 27.45 | 77.07 |
| 512 | 0.09 | 0.06 | 0.38 | 9.91 | 29.09 |

**Table 3:** *Performance results (in seconds) for the optimized algorithm on a variety of graphs. Graph sizes and descriptions are provided in Table 1.*

## 6. Discussion and Future Work

Layout and navigation of extremely large graphs can be computed using our algorithm in conjunction with a fisheye view [Fur86, SB94] or the multi-level display algorithms of Eades and Feng [EF96]. Utilizing these approaches would allow us to accommodate graphs with more vertices than the number of pixels of our display device.

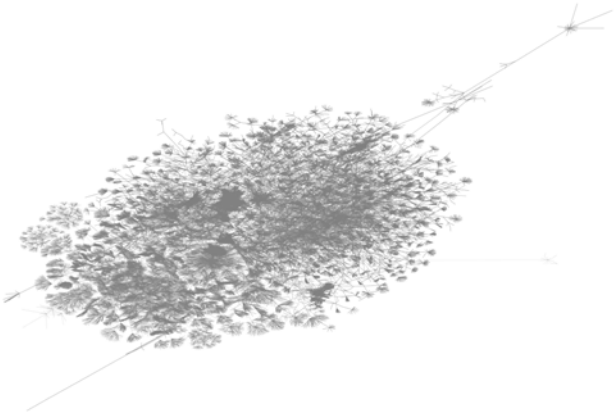


**Figure 5:** *Drawing of the gcc-fail graph produced by our optimized algorithm using 32 processors.*

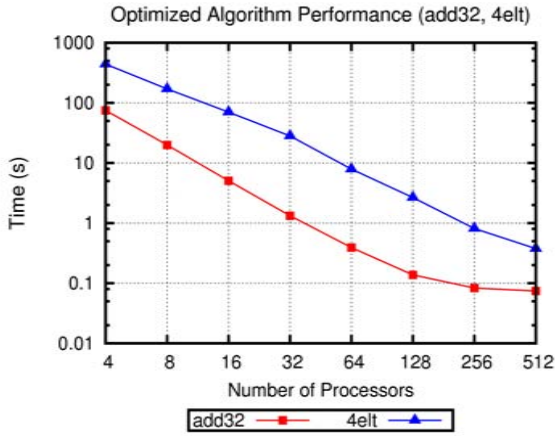
Additional structural information about a graph is sometimes known in advance. For example, it may be known that the graph has a tree-like structure or that it is bipartite. With this knowledge, an algorithm designed to perform well on a specific type of graph can be used to produce more aesthetically pleasing layouts and result in better running times.

A better vertex distribution approach would also improve the quality of our layouts. One option is to ensure that every processor receives a collection of vertices with similar degree distribution. We could also ensure that the number of other processors each processor has to communicate with is minimal and approximately equal for each processor. A parallel graph partitioning scheme may be used to achieve this result. Karypis and Kumar [KK96] demonstrate good scalability for their parallel multilevel graph partitioning method. This method can be used as a pre-processing step for a parallel graph layout algorithm.

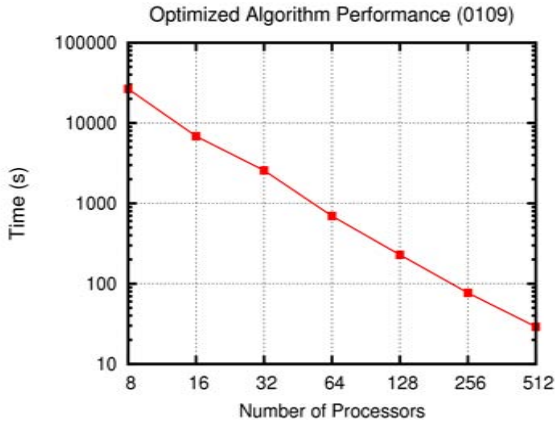
We use an importance metric to assign importance to graph vertices. Vertex degree is our measure of importance, because in our experiments, we observed that clusters generally form around vertices of high degree. Other possible choices for an importance metric include vertex weights, vertex connectedness, or a combination of the above.

## 7. Conclusion

We presented a scalable parallel graph layout algorithm that delivers an approximation to a force-directed model. We compared the performance and visual quality of our method against the classic and the fastest force-directed techniques for layout of general graphs. We have shown that the performance of our optimized algorithm and the quality of the produced graph drawings are comparable to the classic and the more efficient force-directed techniques. We have shown that our algorithm is scalable to moderate numbers of vertices and processors.




(a)




(b)

**Figure 6:** *Graphs of performance results (in seconds) for the optimized algorithm on the graphs shown in Table 3.*

## 8. Acknowledgments

This research was supported in part by the U.S. National Science Foundation through grants CNS-0551727, OCI-0325934, OCI-0749227, and OCI-0749217, and the U.S. Department of Energy through the SciDAC program with Agreement No. DE-FC02-06ER25777.

Datasets provided by the University of Greenwich Graph Partitioning Archive, the Internet Mapping Project, and Adriana Iamnitchi. We would also like to thank Fekete, J.-D. et al. for making available the IEEE InfoVis 2004 Contest dataset.

## References

[BH86] BARNES J., HUT P.: A hierarchical O(N log N) force-calculation algorithm. *Nature 324*, 6096 (1986), 446–449.

[CP96] COLEMAN M. K., PARKER D. S.: Aesthetics-

based graph layout for human consumption. *Software: Practice and Experience 26*, 12 (1996), 1415–1438.

[DBETT99] DI BATTISTA G., EADES P., TAMASSIA R., TOLLIS I. G.: *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[DFF*03] DONGARRA J., FOSTER I., FOX G., GROPP W., KENNEDY K., TORCZON L., WHITE A.: *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers Inc., 2003.

[DH96] DAVIDSON R., HAREL D.: Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics 15*, 4 (1996), 301–331.

[Ead84] EADES P.: A heuristic for graph drawing. *Congressus Numerantium*, 42 (1984), 149–160.

[EF96] EADES P., FENG Q.-W.: Multilevel visualization of clustered graphs. In *Graph Drawing* (1996), pp. 101–112.

[FLM94] FRICK A., LUDWIG A., MEHLDAU H.: A fast adaptive layout algorithm for undirected graphs. In *Graph Drawing* (1994), pp. 388–403.

[FM82] FIDUCCIA C. M., MATTHEYSES R. M.: A linear-time heuristic for improving network partitions. In *Design Automation* (1982), pp. 175–181.

[FR91] FRUCHTERMAN T. M. J., REINGOLD E. M.: Graph drawing by force-directed placement. *Software: Practice and Experience 21*, 11 (1991), 1129–1164.

[FT07] FRISHMAN Y., TAL A.: Multi-level graph layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1310–1319.

[Fur86] FURNAS G. W.: Generalized fisheye views. In *SIGCHI Bulletin* (1986), vol. 17, pp. 16–23.

[HJ05a] HACHUL S., JÜNGER M.: Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph Drawing* (2005), pp. 285–295.

[HJ05b] HACHUL S., JÜNGER M.: An experimental comparison of fast algorithms for drawing general large graphs. In *Graph Drawing* (2005), pp. 235–250.

[HK00] HAREL D., KOREN Y.: A fast multi-scale method for drawing large graphs. In *Graph Drawing* (2000), pp. 183–196.

[KK89] KAMADA T., KAWAI S.: An algorithm for drawing general undirected graphs. *Information Processing Letters 31*, 1 (1989), 7–15.

[KK96] KARYPIS G., KUMAR V.: Parallel multilevel graph partitioning. In *International Parallel Processing Symposium* (1996), pp. 314–319.

[KL70] KERNIGHAN B. W., LIN S.: An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal 49*, 2 (1970), 291–307.

[KW01] KAUFMANN M., WAGNER D.: *Drawing Graphs: Methods and Models*. Springer-Verlag, 2001.

[LGL] LGL, available at http://bioinformatics.icmb.utexas.edu/lgl.

[MGL06] MUELLER C., GREGOR D., LUMSDAINE A.: Distributed force-directed graph layout and visualization. In *Eurographics Symposium on Parallel Graphics and Visualization* (2006), pp. 83–90.

[MHM01] MARSHALL M. S., HERMAN I., MELANÇON G.: An object-oriented design for graph visualization. *Software: Practice and Experience 31*, 8 (2001), 739–756.

[MRS95] MONIEN B., RAMME F., SALMEN H.: A parallel simulated annealing algorithm for generating 3D layouts of undirected graphs. In *Graph Drawing* (1995), pp. 396–408.

[Mun98] MUNZNER T.: Drawing large graphs with H3Viewer and Site Manager. In *Graph Drawing* (1998), pp. 384–393.

[Noa06] NOACK A.: Energy-based clustering of graphs with nonuniform degrees. In *Graph Drawing* (2006), pp. 309–320.

[QB79] QUINN N., BREUR M.: A force directed component placement procedure for printed circuit boards. In *IEEE Transactions on Circuits and Systems* (1979), vol. 26, pp. 377–388.

[QE01] QUIGLEY A., EADES P.: FADE: Graph drawing, clustering, and visual abstraction. In *Graph Drawing* (2001), pp. 197–210.

[RB02] RÉKA A., BARABÁSI A.-L.: Statistical mechanisms of complex networks. *Reviews of Modern Physics 74*, 1 (2002), 47–97.

[Roy] Royére, available at http://sourceforge.net/projects/gvf.

[SB94] SARKAR M., BROWN M. H.: Graphical fisheye views. *Communications of the ACM 37*, 12 (1994), 73–84.

[SOTM06] SHEN Z., OGAWA M., TEOH S. T., MA K.-L.: BiblioViz: A system for visualizing bibliography information. In *Asia-Pacific Symposium on Information Visualization* (2006), pp. 93–102.

[Tul] Tulip, available at http://www.labri.fr/perso/auber/projects/tulip.

[Wal00] WALSHAW C.: A multilevel algorithm for force-directed graph drawing. In *Graph Drawing* (2000), pp. 171–182.

[Wil99] WILLS G. J.: NicheWorks - interactive visualization of very large graphs. *Journal of Computational and Graphical Statistics 8*, 2 (1999), 190–212.