Eurographics Symposium on Parallel Graphics and Visualization (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Distributed Force-Directed Graph Layout and Visualization

Christopher Mueller, Douglas Gregor, Andrew Lumsdaine

Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{chemuell,dgregor,lums}@osl.iu.edu

Abstract

While there exist many interactive tools for the visualization of small graphs and networks, these tools do not address the fundamental problems associated with the visualization of large graphs. In particular, larger graphs require much larger display areas (e.g., display walls) to reduce visual clutter, allowing users to determine the structure of large graphs. Moreover, the layout algorithms employed by these graph visualization tools do not scale to larger graphs, thereby forcing users into a batch-oriented process of generating layouts offline and later viewing of static graph images. In this paper, we present a parallel graph layout algorithm based on the Fruchterman-Reingold force-directed layout algorithm and demonstrate its implementation in a distributed rendering environment. The algorithm uses available distributed resources for both compute and rendering tasks, animating the graph as the layout evolves. We evaluate the algorithm for scalability and interactivity and discuss variations that minimize communication for specific types of graphs and applications.

1. Introduction

The visualization of large graphs is becoming increasingly important in diverse fields including bioinformatics, social network analysis, and network optimization (e.g., [ADWM04]. However, visualization of large graphs is typically a batch-oriented, sequential process: users must first construct a layout offline, which can easily require tens of minutes to compute. Users can then view the resulting static image on a single display or display wall. This situation stands in stark contrast to the plethora of visualization tools [JM04] for small graphs that produce animated, interactive layouts on single workstations. Early results with visualizing dense graphs on large surfaces suggest that they "qualitatively [change] interactive network visualization" [AKGN99].

Visualization tools for small graphs do not scale well to larger graphs for two reasons. The first reason is that the graph layout algorithms themselves rarely scale well. For instance, many of these tools use *force-directed* layout algorithms, which determine the position of each node in a graph by iteratively computing attractive forces between connected nodes and repulsive forces between all pairs of nodes. While these algorithms are ideal for animation—the visualization system can display the graph after each iteration—the force calculations are expensive and can easily take several seconds per iteration on medium-sized graphs, making them unacceptably slow for interactive use.

The second reason that small-graph visualization tools cannot scale to larger graphs is that few of these tools provide support for distributed rendering to large format display devices such as tiled display walls. Graph visualizations are visually limited by the number of nodes and edges that can be rendered on the display. The node-link graph diagram displays nodes as points or glyphs and edges as lines between the nodes. On most displays, only a few hundred edges can be effectively displayed before the image becomes too cluttered to interpret. While techniques exist for

© The Eurographics Association 2006.

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

improving the layout of planar graphs (e.g., circuit layout), most graphs that arise in real-world applications (e.g., bioinformatics, telephone networks) are not planar. The characteristics of these graphs limit the use of node-link visualizations to small graphs on workstation-class displays.

In this paper we present an approach to solving the two scalability problems associated with interactive graph visualization tools. We first employ distributed rendering techniques to permit the display of larger graphs on tiled display walls, without the visual clutter of a single display. We then describe a distributed-memory parallel formulation of the Fruchterman-Reingold force-directed layout algorithm [FR91], which permits layout of medium-sized graphs on tiled displays at interactive frame rates. Finally, we evaluate the performance and scalability of a distributed graph visualization environment based on Chromium [HHN*02] and the Parallel Boost Graph Library [GL05].

2. Distributed Graph Rendering

Display walls are a fairly recent addition to scientific computing environments. A display wall is composed of a collection of displays connected to one or more computers. When a single machine drives more than two displays, custom video switching and scaling hardware is used to partition the image across all displays. When a cluster drives the display wall, each node in the cluster is connected to one or two displays and a middleware application manages rendering across all the displays. From a systems perspective, it is convenient to define three types of compute nodes used with display walls [RRJ*04]:

1. *Display nodes* drive the displays
2. *Rendering nodes* generate the pixels and geometry
3. *Application nodes* generate the raw data

Aside from custom hardware solutions, there are three primary middleware tools in common use for rendering to display walls. The most general, Distributed Multihead X (DMX), is an X server that distributes X events to clients running on the display nodes. Any application can connect to the DMX server and display windows on the display wall. SAGE [RRJ*04], developed for geo-sciences applications, acts as a virtual framebuffer for application and rendering nodes. SAGE applications send pixels to the display nodes, which manage the final layout of the images for each application. Chromium [HHN*02] is a wrapper around OpenGL that intercepts the OpenGL calls from an application and distributes them across the display wall. Chromium supports multiple strategies for compositing geometry from parallel applications, making it a flexible tool for individual applications. Extensions to DMX allow X applications to use Chromium for parallel rendering.

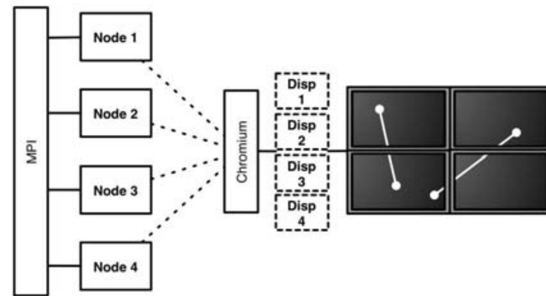


Figure 1: The system architecture for a parallel application in a distributed rendering environment. MPI provides support for running the algorithmic components of the application in parallel while Chromium manages the rendering OpenGL pipeline and display wall. Chromium composites the geometry from the application nodes into a global display space and properly segments lines that span displays.

There are other systems available for distributed rendering (e.g. ParaView [Kit06]) and many animation systems support network rendering. However, these systems are not designed for use with display walls and do not provide robust and comprehenisive support for interactive use.

To render a graph across multiple displays, the rendering system must ensure that edges are drawn on all displays they intersect. In Figure 1, the right-most line spans three displays, with no explicit endpoints on the lower-right display. If each display is represented as a single tile or framebuffer, as in SAGE, the application developer must ensure that lines are properly segmented and rendered to each tile. Geometry based display wall renderers such as DMX and Chromium, on the other hand, handle segmentation automatically.

Most display walls also introduce gaps, or borders, between the displays. On high-end, projector-based walls, these gaps are minimal. However, on LCD-based walls, there is typically a 2-inch gap between display surfaces that must be considered when lines are drawn across displays. Chromium handles such gaps smoothly by allowing the user to position the displays anywhere in the scene space. Experience has shown that users quickly learn to ignore the borders introduced by LCDs. The borders can also be exploited by rendering algorithms as a physical method for partitioning data visually; we use this approach to show the user vertex partitions assigned to individual nodes.

Parallel applications add an extra level of complexity to the rendering pipeline. The middleware must accept geometry streams from multiple sources and properly synchronize and composite them prior to ren-

© The Eurographics Association 2006.

dering each frame. Of the display wall tools, only Chromium has full support for parallel applications and arbitrary composition schemes. While SAGE can support multiple pixel streams from a parallel application, it assigns each stream to a distinct area of the framebuffer, making it difficult to manage applications that use geometric primitives, rather than pixels, for rendering. Chromium supports different strategies for compositing the geometry stream on the display nodes. *Sort-first* rendering divides the display area into tiles and each rendering node is assigned one tile, similar to SAGE's framebuffer approach. *Sort-last* rendering combines the streams from all rendering nodes into a single, global display space, making it possible to combine complex geometry from multiple sources into a single image.

[AKGN99] discusses the utility and challenges for displaying large graphs using tiled display. Their display system uses an SGI Onyx to drive a display wall, allowing them to use algorithms designed for shared-memory parallel systems rather than distributed memory systems.

We originally developed prototype implementations using SAGE, Chromium, and a custom, OpenGL-based rendering system. Sage and our custom system worked fine for simple examples. Unfortunately, SAGE failed to integrate smoothly with MPI and required reimplementing the compositing algorithms already available in Chromium. Our custom solution worked well with MPI but also required reinventing a large amount of Chromium's technology.

Our final system architecture for processing and rendering graphs in distributed environments (Figure 1) uses MPI for parallel processing and Chromium for distributed rendering to the display wall. For a given run, the application first partitions the graph across the application nodes and sets up display nodes on each computer connected to a display. The application proceeds by executing one iteration of the layout algorithm followed by a rendering pass. In a rendering pass, each application node sends the geometry for its vertices and edges along with a synchronization signal to Chromium. Chromium composites the stream using sort-last compositing and sends the final geometry data to the individual display nodes. As with traditional OpenGL-based systems, once the rendering commands have been issued, the application is free to continue, allowing it to enter the next layout phase before the rendering is complete.

3. Distributed Force-Directed Layout

While there exist many vertex/edge layout algorithms for graph drawing, the distributed graph rendering architecture places some severe limitations on the underlying layout algorithm. First, the algorithm must be efficient even with a large numbers of vertices and edges, so that interactive layout can be achieved. Second, to limit data aggregation bottlenecks, the algorithm must be able to perform layout of local vertices without requiring extensive global information. Finally, the algorithm must be efficiently parallelizable in a distributed memory system, minimizing the communication overhead relative to the local computation required even as the number of compute nodes and displays increases. To the best of our knowledge, there are no prior results for interactive graph rendering algorithms on distributed memory systems.

We have adapted the Fruchterman-Reingold force-directed layout algorithm [FR91]. Force-directed algorithms place all of the vertices within a space, then iteratively compute the forces acting on each vertex and reposition the vertices accordingly until a minimal energy state is reached. Given a graph with vertices V and edges E, Vertices connected by an edge $(u, v) \in E$ are "pulled" together by an attractive force $f_a(d)$, where d is the distance between u and v. Intuitively, this force should be stronger when the vertices are farther apart, so that adjacent vertices will be positioned close together. Each vertex also exhibits a repulsive force $f_r(d)$ on every other vertex, whose strength is inversely related to the distance. The functions $f_a(d)$ and $f_r(d)$ are selected to ensure that the forces cancel each other when two adjacent vertices reach some optimal distance apart.

The computation of attractive forces is efficient, requiring $\mathcal{O}(|E|)$ time. However, the computation of repulsive forces requires every pair of vertices to be compared in $\mathcal{O}(|V|^2)$ time. To improve the performance of the latter operation, the repulsive forces computed for a vertex u are limited to those vertices within a circle of radius $k = \sqrt{\text{width} \cdot \text{height}}$. Fruchterman and Reingold suggest dividing the drawing space into a grid with cells of size $k \times k$. Vertices are sorted into cells based on their position, and for each vertex repulsive forces need only be computed within its own cell and neighboring cells, drastically reducing the average number of distance comparisons. Pseudo-code for the complete distributed Fruchterman-Reingold layout algorithm is provided in Figure 4. The following subsections describe the evolution and evaluation of this algorithm.

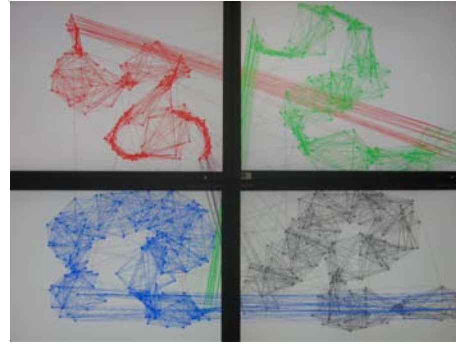
3.1. Naïve Distribution and Parallelization

The Fruchterman-Reingold layout algorithm can be trivially distributed by considering the layout on each compute node independently of the other nodes. Each compute node is given its own display in which it can render its local portion of the distributed graph. Interprocessor edges in the graph are filtered out, so that attractive forces are computed only between vertices

© The Eurographics Association 2006.

on the same compute node. Since each display is considered independent of the others, repulsive forces are only calculated between vertices that reside on the same compute node.

Naïve distribution and parallelization of Fruchterman-Reingold is both efficient and scalable. Since interprocessor edges are not considered, the distributed layout algorithm itself requires no communication. Visually, this leads to a layout on each compute node that only illustrates local structure. Vertices may be placed far from their neighboring vertices that reside on other processors. Figure 2(a) illustrates these long interprocessor edges, which traverse nearly the entire width of the display wall. Naïve distribution is only as effective as the initial partitioning. When interesting substructures are assigned to a single compute node, the layout will expose these substructures.



(a) Naïve layout

(b) Interprocessor attraction

Figure 2: Comparing the naïve distributed Fruchterman-Reingold layout to the version that permit interprocessor attraction between vertices on a small world graph.

3.2. Attraction Across Processors

Graph substructures that span multiple compute nodes can be pulled closer together by considering attractive forces for interprocessor edges. These attractive forces can be computed by replicating the position information of neighboring vertices across processes. Thus, prior to the computation of attractive forces, processors exchange the current location of each vertex with its remote neighbors. Attractive forces are then computed for each vertex using both intraprocessor edges and interprocessor edges.

Attraction across processor boundaries introduces communication into the distributed layout algorithm. In each iteration, the position information of each vertex must be sent to the compute nodes that store a neighboring vertex, resulting in $\mathcal{O}(|V| \cdot p)$ additional communication per step, where p is the number of processors. A good partitioning can improve the performance of the algorithm by minimizing the amount of positional data that must be transferred.

Visually, interprocessor attraction forces bring together graph substructures that span multiple compute nodes. Even though vertices themselves are bound to the display connected to the compute node on which they reside, they may move toward the display on which their neighbors reside, providing the illusion of global layout. Figure 2(b) illustrates how the addition of interprocessor attractive forces avoids long interprocessor edges in the layout. We found that it the algorithm produced more pleasing layouts when the interprocessor attractive forces were somewhat weaker than intraprocessor attractive forces. We therefore multiply the magnitude of the attractive force for an interprocessor edge by some constant $C < 1$. When C is nearly 1, graphs with many interprocessor edges tend to clump near other displays (i.e., near the center of the tiled display wall), as interprocessor attractive forces outweigh intraprocessor repulsive forces. Extending the Fruchterman-Reingold algorithm slightly, the sides of the display exert a repulsive force on nearby vertices, pushing the graph layout toward the center of the display to generate more pleasing layouts.

3.3. Repulsion Across Processors

The distributed Fruchterman-Reingold layout with interprocessor attraction pulls vertices near their neighbors, regardless of the compute node on which the neighbors reside. However, this interprocessor attractive force is not balanced by an interprocessor repulsive force, causing clumping of vertices near the shared borders of each display. The grid used in distributed Fruchterman-Reingold layout can be extended to multiple compute nodes, as shown in Figure 3. Prior to the computation of repulsive forces, each processor gathers the local vertices that reside in grid cells adjacent to another display and exchanges those with the corresponding compute node. The process is identical to that of exchanging boundary information in distributed finite element analyses.

© The Eurographics Association 2006.

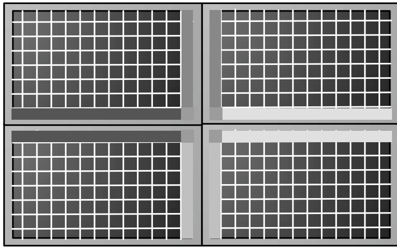


Figure 3: Each display is divided by a grid of cells used to compute repulsive forces. Vertices that fall into cells adjacent to nearby displays will repulse vertices on those nearby displays.

```
do for n iterations:
  // Interprocessor repulsion
  for each local vertex v:
    // place v in grid cell (i, j) based on its position
  // Repulsive forces
  for each grid cell (i, j):
    for each vertex u in grid cell (i, j):
      for each vertex v in neighboring grid cells:
        // compute repulsive force between u and v
  // Interprocessor attraction
  for each edge (u, v):
    if v is remote:
      // send position of u to the owner of v
  // Attractive forces
  for each edge (u, v):
    // compute attractive force between u and v
  // Position update
  for each local vertex v:
    // move v based on the forces acting on it
```

Figure 4: Pseudo-code for the distributed Fruchterman-Reingold implementation.

The intent of interprocessor repulsion is to counteract the interprocessor attractive forces that tend to clump vertices around the borders of the display. However, in our experience the addition of the repulsive forces did not produce layouts that were noticeably better than those using the simpler (and more efficient) repulsion from the display borders. Rather, by decreasing the constant C that governs the strength of interprocessor attraction, clumping can be avoided while still exposing graph structure across processors.

3.4. Vertex Migration

Our distribution and parallelization of the Fruchterman-Reingold layout algorithm has restricted vertices to the local display, although edges may cross several displays. Relaxing this condition, vertices may be permitted to migrate to other displays when the interprocessor attractive forces are strong enough to overcome the repulsive forces exerted by the edges of the (local) display. There exist two kinds of vertex migration: visual and data migration.

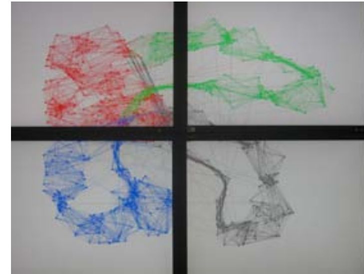


Figure 5: Visual migration across processors permits vertices to be placed closest to their neighbors, even when those neighbors are stored on a different compute node.

Visual migration of vertices involves drawing a vertex (and its edges) on a display other than the local display. The data associated with the vertex is not actually moved to the compute node associated with the remote display; rather, the rendering system simply displays the vertex and edge on the other display. The computation of attractive forces (both inter- and intra-processor) is unchanged by visual migration. However, visual migration of a vertex implies that repulsive forces can no longer be efficiently computed: once a vertex is displayed outside the confines of its owner's display area, it does not fall within any local grid cells and therefore no repulsive forces act on it on either its owning processor or the processor displaying it. For this reason, vertices that have visually migrated may be placed very near other vertices on the remote processor, causing visual clutter. In our experiments, the resulting visual clutter was mitigated by the use of processor-specific colors for each vertex. Thus, a vertex that had migrated would retain the color of its owning processor, and would therefore be visually distinct from the vertices stored on the remote processor. Figure 5 illustrates the result of permitting vertex migration for a distributed small-world graph. A long chain of vertices owned by the upper-right and lower-left displays has been pulled together on the upper-left display, and the vast majority of edges within the display are very short. Visual migration has an interesting side effect: when a vertex migrates to another processor, it is because the vertex is pulled much more strongly to that processor by its neighbors, potentially indicating that layout would be better—and computations more efficient—if the vertex were stored on that remote processor. In a sense, visual migration of vertices can indicate a better partitioning for a graph.

Data migration of vertices expands visual migration to involve physically moving the data associated with a vertex to the processor displaying the vertex.

© The Eurographics Association 2006.

Data migration eliminates the communication associated with drawing vertices on remote displays and restores the proper behavior of the local repulsive forces. However, data migration has severe performance implications. Migrating a vertex involves passing a vertex and all of its edges (including any additional data attached to the vertex and edges) to a different processor, notifying all processors with vertices adjacent to that vertex of the move, and assimilating the vertex into its new owner's data structure. Data migration within the Parallel Boost Graph Library is particularly expensive, requiring (in the worst case) reallocation of the graph data structure on each compute node. Moreover, data migration can introduce load-balancing problems when a large number of vertices migrate to only a few compute nodes. For instance, the compute node displayed in the upper-left corner of Figure 5 contains a disproportionate number of vertices due to migration.

4. Methods

We implemented the distributed graph layout algorithm using the Parallel Boost Graph Library [GL05] over MPI and visualized the results on a tiled display using Chromium. Rendering nodes were connected to the display nodes using Chromium's `tilesort` SPU, enabling distributed sort-last rendering.

We tested this visualization environment using randomly generated graphs from three different models: Erdös-Renyi, Power Law, and Small World graphs. Erdös-Renyi graphs, also known as random graphs, randomly join pairs of vertices. Power-Law graphs exhibit a power-law distribution in the degree of vertices, as seen in graphs such as the Web graph. Small-world graphs exhibit a regular local structure with many small cliques and several long-range connections that bring together otherwise unrelated groups.

For each graph type, we constructed graphs with between $n = 500$ and $n = 8000$ vertices and distributed these graphs across 1, 2, 4, and 8 processors. The resulting distributed graphs were rendered on an 8-node display wall. The number of edges in the graphs depend on the type of graph: Erdös-Renyi graphs contained approximately $10n$ edges, Power Law graphs contained from 2 to 12 thousand edges, and Small World graphs contained $10n$ edges. The graphs were partitioned across the nodes in the visualization cluster. Small-world graphs had optimal partitions, minimizing cross-processor edges; Erdös-Renyi and Power Law graphs, on the other hand, exhibited poor, random partitioning.

Our visualization cluster consisted of 8 nodes with two AMD Opteron 246 processors and 8GB RAM per node running Fedora Core 2 with the kernel upgraded

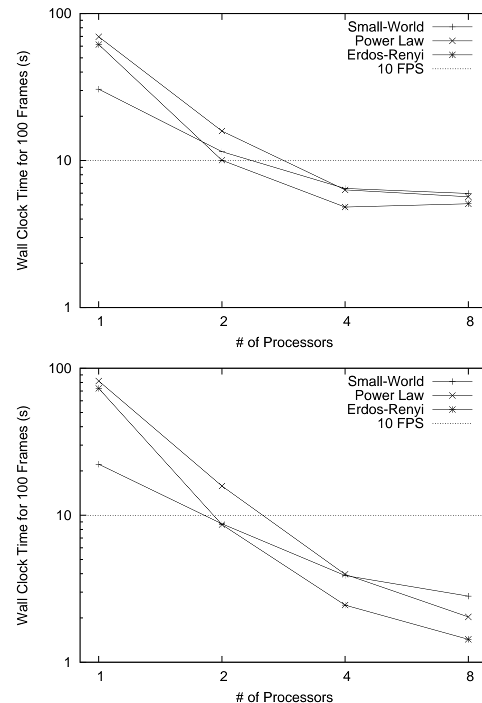


Figure 6: Scalability with the display enabled (top) and disabled (bottom) for 2000 vertex graphs.

to 2.6.10. The eight nodes contained GeForce 5950Fx graphics cards with 128MB video memory, running X and connected by Gigabit Ethernet. We used LAM-MPI 7.1.1, Chromium 1.8, and the current development version of the Parallel BGL. All times were measured using wall clock time covering the execution of the algorithm and the setup code, excluding command line parsing. The algorithm was executed for 100 iterations, which was sufficient to stabilize the layout. With the display enabled, the graph was rendered once per iteration.

4.1. Results

We evaluated the performance and scalability of our distributed graph visualization environment on a variety of randomly-generated graphs, measuring the costs of various algorithm features and of the distributed rendering infrastructure itself. We were particularly interested in whether the distributed layout algorithm itself is scalable and if it can be harnessed to provide animated visualizations of graph layout on a tiled display wall at interactive frame rates.

Scalability Figure 6 illustrates the wall-clock times when executing 100 iterations of our distributed graph layout algorithm on graphs with 2000 vertices. With the display disabled (bottom), the algorithm provides excellent scalability with near-linear scaling up to 8

© The Eurographics Association 2006.

processors. Scalability was similar with the display enabled, with less steep curves due to the time taken by the rendering infrastructure. With the display enabled, we required four processors to achieve interactive frame rates of 10 FPS or better for all graph types with 2000 vertices. At 8000 vertices, scalability remained reasonable and we were able to achieve acceptable frame rates of 2–5 FPS.

Display Costs Figure 7 illustrates the costs of rendering the distributed graphs to the tiled display wall. The display cost is the time difference between runs with the display system enabled and disabled (all other variables are fixed). The left graph shows the average display cost per frame, which was generally under .1 second. The *relative* display cost is the ratio of the display cost to the overall execution time with the display enabled. For identical graphs, the absolute display costs were stable against of the number of processing nodes and increased relative to the size of the graph, roughly following a power curve. The relative display cost increased with the number of processors, suggesting that that the amount of time spent in the algorithm decreased at a faster rate than the performance of the display system. However, the relative display cost decreased as the number of vertices increased (right graph), minimizing the overall impact of the display system for larger graphs. For the sizes tested, the display costs generally fell between 20% and 80% of the overall run time. The absolute cost per frame was almost always below 0.1 second of a second and typically below 0.05 second. For 8000 node Erdös-Renyi graphs the display cost per frame peaked at .29 seconds. Thus, for almost all graphs tested, the display system was able to maintain interactive rates.

4.2. Discussion

The distributed algorithm produced similar layouts to the single-processor version. Power law and Erdös-Renyi graphs, with high connectivity, were always difficult to interpret and generally settled to groups of vertices in the center of the display area. However, observing the evolution of the layout provided insight into the overall structure of both graph types. Small world graphs, on the other hand, provide interpretable layouts in all cases. On a single processor, small world graphs tend to form rings and looped curves. As the number of vertices grows and the display space is filled, multiple 'knots' form, obscuring the overall structure. With multiple processors, the portion of the graph assigned to each processor formed its own local loop that was connected to the adjoining loops on remote processors by tight clusters of edges, making it easier to explore local structure on individual displays while discerning the overall structure across all displays, as illustrated in Figure 2(b).

For multiple processors, we colored the vertices and edges by the process number that owned the vertex. In the case of edges that overlapped, an edge is colored by the vertex that last drew it. While useful for debugging, coloring partitions of the graph also made it easier to interpret large, dense graphs. With a single color, overlapping edges and close vertices add a large amount of visual noise to the image. Arbitrarily coloring groups of edges using a small number of colors helped disambiguate the structure in dense regions. With vertex migration enabled, the coloring provided a visual anchor for collections of vertices as they moved from screen to screen.

5. Future Work

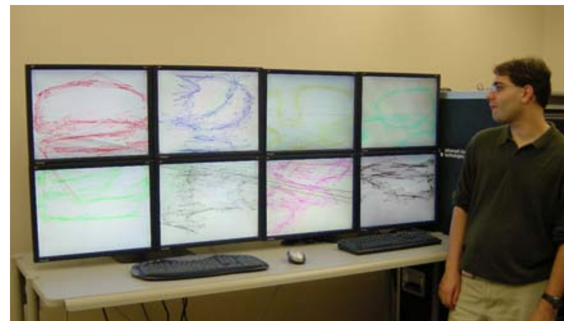


Figure 8: Distributed graph layout of a 1,000 node small world graph on an 8-node rendering cluster.

There are a few immediate opportunities for continuing study of distributed graph layout and rendering. The most important is scaling up to larger display clusters. While large computation clusters are relatively easy to configure, larger display clusters present additional challenges. Display walls require a large amount of physical space, and all of the rendering nodes must contain graphics hardware that is not typically found in compute clusters. Finally, the software used to drive the display wall, while technically sound, requires extensive configuration and expertise to use.

The initial partitioning of the vertices has a direct effect on the final layout. Because the algorithm does not allow vertex migration, vertices assigned to the same processor will always have a stronger effect on each other than distant nodes. Thus, a good partitioning should assign vertices that are most related to the same processor.

While we have done a basic study of repulsion and vertex migration, there are many additional strategies we could employ that are aware of the physical layout of the display. By taking into account the "display physics", we believe that we can improve the display of large, distributed graphs.

© The Eurographics Association 2006.

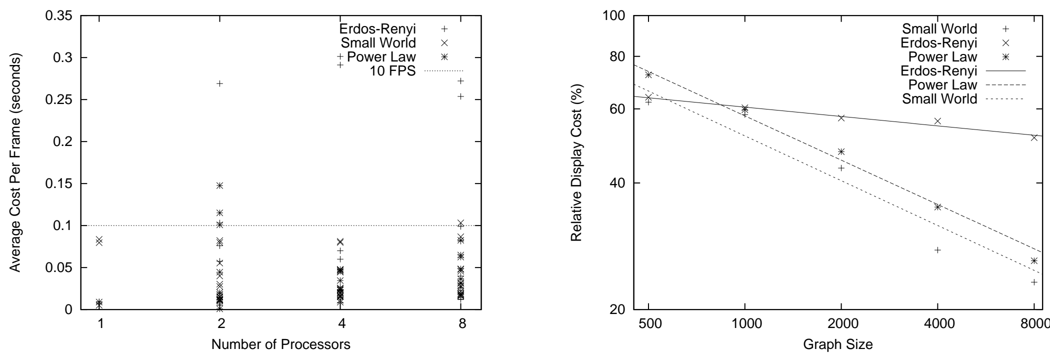


Figure 7: The average display cost per frame (left) was almost always under .1 second, allowing for interactive frame rates from the graphics system. The outliers are from 8000 node Erdös-Renyi graphs, the most densely connected graphs tested. The relative display costs (right) decreased with the size of the graph. The results here are for 8000 node graphs using the 8 processor configuration.

6. Conclusion

The availability of affordable parallel computing systems and tiled display walls, combined with the growing demand for interactive applications that support complex data sets, opens up the opportunity to develop a new class of applications that take advantage of cluster resources for large-data applications. Occupying the space between traditional workstation tools and off-line parallel applications, these applications will provide high-end analytical tools to users for real-time data analysis.

In this paper, we described an adaptation of a force-directed graph layout algorithm for interactive use on a computing cluster. Using MPI and the Parallel BGL for graph layout and Chromium for distributed rendering, we attained interactive frame rates on problems that are not interactive on a single workstation. We were also able to effectively display graphs using a tiled display wall that would be uninterpretable on a smaller display. Using the combined approach of parallel processing and distributed rendering demonstrates the feasability of developing visualization applications that scale to environments that span multiple computers and contain complex data.

Acknowledgments

This work was supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment. We are grateful to John Huffman and Frederick Myers of the Advanced Visualization Laboratory at Indiana University, who built and maintained the visualization clusters we used for this work.

References

[ADWM04] Adai A., Date S., Wieland S., Marcotte E.: LGL: Creating a map of protein function with an algorithm for visualizing very large biological networks. *Journal of Molecular Biology 340*, 1 (June 2004), 179–190.

[AKGN99] Abello J., Koutsofios E., Gansner E. R., North S. C.: Large networks present visualization challenges. *SIGGRAPH Computer Graphics Newsletter 33*, 3 (August 1999).

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

[GL05] Gregor D., Lumsdaine A.: The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)* (July 2005).

[HHN*02] Humphreys G., Houston M., Ng R., Frank R., Ahern S., Kirchner P. D., Klosowski J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 693–702.

[JM04] Jünger M., Mutzel P. (Eds.): *Graph Drawing Software*. Springer, 2004.

[Kit06] Kitware: ParaView. `http://www.paraview.org/`, 2006.

[RRJ*04] Renambot L., Rao A., Jeong R. S. B., Krishnaprasad N., Vishwanath V., Chandrasekhar V., Schwarz N., Spale A., Zhang C., Goldman G., Leigh J., Johnson A.: SAGE: the Scalable Adaptive Graphics Environment. In *Proceedings of WACE* (2004).

© The Eurographics Association 2006.