Scalable Force Directed Graph Layout Algorithms Using Fast Multipole Methods

Enas Yunis, Rio Yokota and Aron Ahmadia
King Abdullah University of Science and Technology
4700 KAUST, Thuwal, KSA 23955-6900
{*enas.yunis, rio.yokota, aron.ahmadia*} *@kaust.edu.sa*

***Abstract*—We present an extension to ExaFMM, a Fast Multipole Method library, as a generalized approach for fast and scalable execution of the Force-Directed Graph Layout algorithm. The Force-Directed Graph Layout algorithm is a physics-based approach to graph layout that treats the vertices V as repelling charged particles with the edges E connecting them acting as springs. Traditionally, the amount of work required in applying the Force-Directed Graph Layout algorithm is $\mathcal{O}(|V|^2 + |E|)$ using direct calculations and $\mathcal{O}(|V| \log |V| + |E|)$ using truncation, filtering, and/or multi-level techniques. Correct application of the Fast Multipole Method allows us to maintain a lower complexity of $\mathcal{O}(|V| + |E|)$ while regaining most of the precision lost in other techniques. Solving layout problems for truly large graphs with millions of vertices still requires a scalable algorithm and implementation. We have been able to leverage the scalability and architectural adaptability of the ExaFMM library to create a Force-Directed Graph Layout implementation that runs efficiently on distributed multicore and multi-GPU architectures.**

***Keywords*-Force directed graph layout; Fast multipole methods; Multi-GPUs**

I. Introduction

Force-Directed Graph Layout (FDGL) is a popular algorithm for automatically generating aesthetically pleasing $2D$ and $3D$ visualizations of large sparse graphs [1], [2]. FDGL is a physics-based approach to graph layout that treats the vertices as repelling charged particles and the edges connecting them as springs. Given a graph $G = (V, E)$ consisting of a set of vertices V and a set of undirected edges E, a graph layout algorithm associates a position x^d for each vertex v in V. FDGL works by assigning an initial random position $\boldsymbol{x_0}$ to each vertex and interactive forces between each vertex (usually an electromagnetic-style repulsion between each vertex modeled after Coulomb's Law and attractive linear springs among edges modeled after Hooke's Law), then using either global optimization heuristics to find a configuration of minimal potential energy, where the system's potential energy P is computed by summing the repulsive vertex-vertex potential interaction energies P_v and the edge potential energies P_e over all vertices.

A common heuristic for minimizing potential energy is simulated annealing, [1], [3] where a "warm" system with vertices allowed to freely move is gradually "cooled" allowing the algorithm to occasionally escape local minima in optimizing P. There are several commonly used stopping criterion; [3] bases their approach on total system energy, while [1] relies on a direct temperature cooling technique. Additionally, there are a large variety of force models for interactions, some can be easily connected to existing physical models, while others rely on purely mathematical constructions.

Calculating the repulsive interactions naïvely using direct integration requires $\mathcal{O}(|V|^2)$ computations [1], [2]. Additionally, $\mathcal{O}(|E|)$ computations per iteration are needed for spring calculations. Given that graphs are usually sparsely connected ($|E| \ll |V|$), much effort has been directed at improving the repulsive interaction time complexity from $\mathcal{O}(|V|^2)$ to $\mathcal{O}(|V| \log |V|)$ by using various techniques such as grid modeling [1], Multi-Level Clustering [4], Barnes-Hut Oct-tree [3], Sub-Graph Fixing [5], Fast Multipole Multilevel methods [6], [7], and Edge Filtering [8].

There is also continued interest in pushing the frontier of large graph layout problems; several promising applications such as the display of complex high-dimensional relationships data such as those found in protein-to-protein interactions, disease time studies, network (components and/or traffic) topologies, and global social network realizations remain just over the horizon. An important distinguishing characteristic of these relationships is that they follow a power law function in the distribution of edges to vertices, with some vertices having many connections and most vertices having a few connections. This disparity gives rise to the need to optimize for clarity and readability and also leads to strong non-uniformity in the input data.

Much recent work has been concentrated on creating better heuristics for faster convergence (less iterations) to an optimal layout, such as design of a temperature decrease/decay function [3], clustered initial layout strategy [6], edge filtering [8], and multiscale eigenvector computations for energy minimization [9]. It has become clear that this increased exploration of the design space requires a commensurate increase in computational throughput.

There have been several efforts to improve the performance of layout computations and to take advantage of GPU acceleration. Notably, early CPU parallelization attempts by Tikhonova [5] handled systems with almost 300 thousand vertices, and an implementation featuring single GPU parallelization by Godiyal [6] has computed layouts for graphs with nearly 500 thousand vertices and almost 1

million edges. Jia's edge filtering technique was able to lay out graphs featuring 1 million vertices and 7 million edges [8], and multiscale eigenvector computations allow rough solutions of systems with 7 million vertices and 15 million edges [9].

In this work, we describe an open, efficient, easily extensible, and error-bounded implementation of the FDGL algorithm, enabling experimentation and design of new techniques by domain experts. Our code leverages the ExaFMM library, a scalable Fast Multipole Method (FMM) library built specifically for solving n-body problems that allows us to compute each iteration in $\mathcal{O}(|V| + |E|)$ time without loss of precision. The implementation's open framework allows easy modification of the core kernels, allowing it to be extended as a comparative testbed for many different algorithms. Additionally, accurate bounds allow designers greater confidence in reproducing their experiments. We demonstrate the capability of our implementation on graph problems containing as many as 10 million vertices with a max per-iteration runtime of 10^{-5} sec/vertex on a single CPU and 3×10^{-6} sec/vertex on a single GPU. Our approach enables visualizing truly massive datasets such as the entire Facebook social graph and protein-protein interactions using traditional or GPU-accelerated supercomputing facilities.

II. Fast Multipole Method

The repulsion between vertices is modeled by a Coulomb potential, which is a long-range potential that considers the effect of all-pairs of vertices. The direct summation of an all-pairs interaction requires $\mathcal{O}(|V|^2)$ for $|V|$ vertices, and becomes prohibitive for large graphs. Fast Multipole Methods (FMM) approximate the far field and near field using multipole expansions and local expansions, respectively. This brings the complexity down to $\mathcal{O}(|V|)$, while the approximation error remains bounded as a function of the order of expansion p. This reduction in the complexity of the algorithm allows us to handle graphs of unprecedented sizes.

?

A schematic of the flow of calculation of the FMM is shown in Figure 1. In order to illustrate the difference between treecodes used by previous FDGL calculations[3], [6] and our current FMM, we show the stages for both methods. There are 6 different stages in the $\mathcal{O}(|V|)$ FMM and 4 different stages in the $\mathcal{O}(|V| \log |V|)$ treecode. First, at the *P2M* stage, the charge of each particle in the leaf cell is transformed to a multipole expansion at the center of the cell. Then, the multipole expansions are shifted to the center of larger cells in the tree structure at the *M2M* stage. Up to this point treecodes and FMMs are identical. Once all multipole expansions are calculated treecodes evaluate the effect of the multipole expansion the particles directly in the *M2P* stage. On the other hand, FMMs first transform the multipole expansions into local expansions at well-separated cells (*M2L*). Then, the local expansions are shifted to the center of smaller cells in the tree (*L2L*). Finally, the local expansions at the leaf cells are used to evaluate the effect on each particle (*L2P*). Note that the multipole expansion does not converge in the vicinity of the target particles so these neighboring particles must be calculated directly (*P2P*).

FDGL is a challenging application for FMMs because the distribution of vertices are often clustered, which results in a highly adaptive tree structure in the FMM. Furthermore, vertices can move much faster than other applications in physics, since the temporal resolution does not need to be very high in order to achieve an optimal graph layout. This means that the tree structure changes rapidly at every iteration, and incremental load-balancing techniques that repartition based on the previous load imbalance become useless.

In the current work, we use a highly scalable adaptive FMM library for heterogeneous architectures `ExaFMM` [10]. The `ExaFMM` library has a suite of optimized kernels for both Cartesian and Spherical expansions on both CPUs and GPUs. `ExaFMM` can also select between an $\mathcal{O}(|V| \log |V|)$ treecode, an $\mathcal{O}(|V|)$ FMM, or a hybridization of the two with a single compile option. It also has an auto-tuning mechanism which selects between P2P, M2P, and M2L kernels based on the runtime of each kernel on the given architecture. Further discussion of `ExaFMM` can be found in [11], [12].

Our approach has been tested for up to 10 million vertices and 20 million edges on a modern multi-core multi-GPU workstation `Intel(R) Xeon(R) CPU X5650 @ 2.67GHz and nVidia Corporation GF100 [Quadro 5000]` and on the TSUBAME 2.0 GPU Supercomputer at Tokyo Institute of Technology. Given the stability and scalability of FMM to 8 billion nodes [13], we aim to eventually scale the implementation further to even larger graphs.

`ExaFMM` runs serially or in parallel on workstations and distributed-memory supercomputers. Additionally, `ExaFMM` can access on-node acceleration of CPU+GPU architectures that support the CUDA programming interface. `ExaFMM` is implemented as a library that services one time step in adaptive FMM, leaving the remaining implementation details to the application user. A `unit_test` example `fdgl.cxx` has been provided to showcase one of the many ways to take advantage of the library and to support FDGL requirements following the basic techniques specified by Fruchterman and Reingold [1], with several improvements noted below.

The authors in [1] calculated the maximum velocity for vertices at each time step using a limiter approach: $min(t, |v.\boldsymbol{disp}|)$. In physical annealing, the temperature acts as a dampening factor on molecular vibrations (displacement) while here it is acting as a simple limiter, acting as a blocking force only to vertices that have a higher

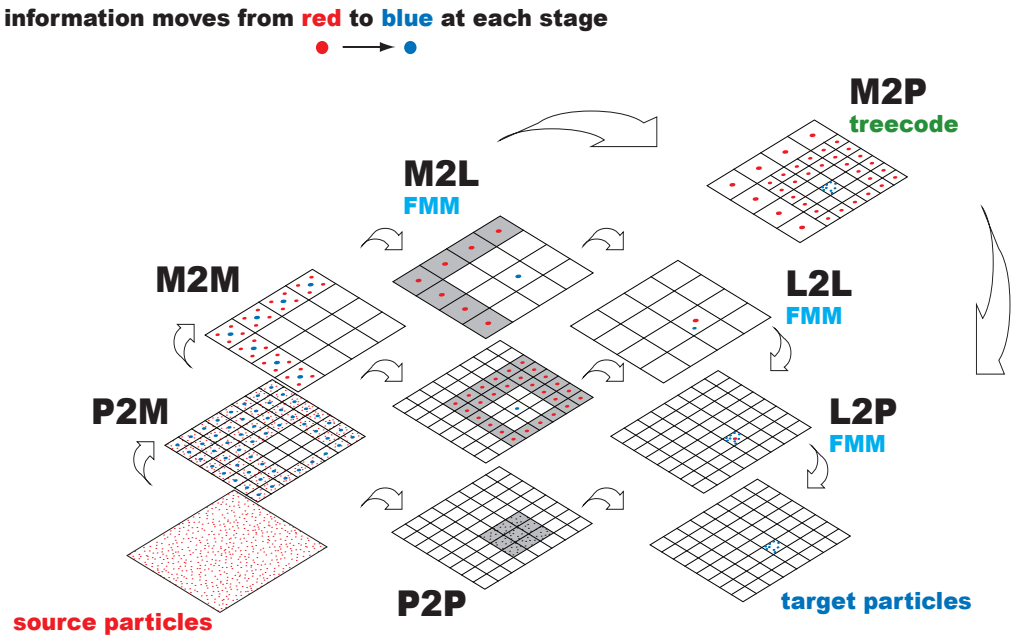


Figure 1. FMM and Treecode Algorithm Operations

displacement than the current temperature. We have changed our version of the implementation to use temperature in the sense of dampening the movement across all vertices rather than halting it at a specific distance for some vertices. Additionally, the original work limited the new vertex positions within walls with a user specified frame length and a tuning constant. Given the progress in visualization software since 1991, we were able to avoid restricting ourselves to predefined walls, and consequently found several aesthetically improved layouts that had previously suffered from this restriction. Finally, we also separated the tuning parameter for spring calculation from repulsive calculations, giving us the freedom to tune each independently.

We choose to simplify the parallelization and communication process by fixing a shadow copy of the vertices and mapping them into the available cores using static parallelization. We also keep a static copy of the edges with their associated source vertices. At each completed FMM time step, the shadow vertices are updated with their electrostatic forces before the rest of the FDGL algorithm is run against the shadow vertices. Keeping the simple vertex division static across the cores allows for simple communication patterns for the positions of the destination vertices. Although, this process can likely be improved using more advanced techniques, we have found that both communication and spring calculation take a negligible amount of time when compared to electrostatic calculations. Given this is only an example extension, we have chosen to maintain the simple model for this paper.

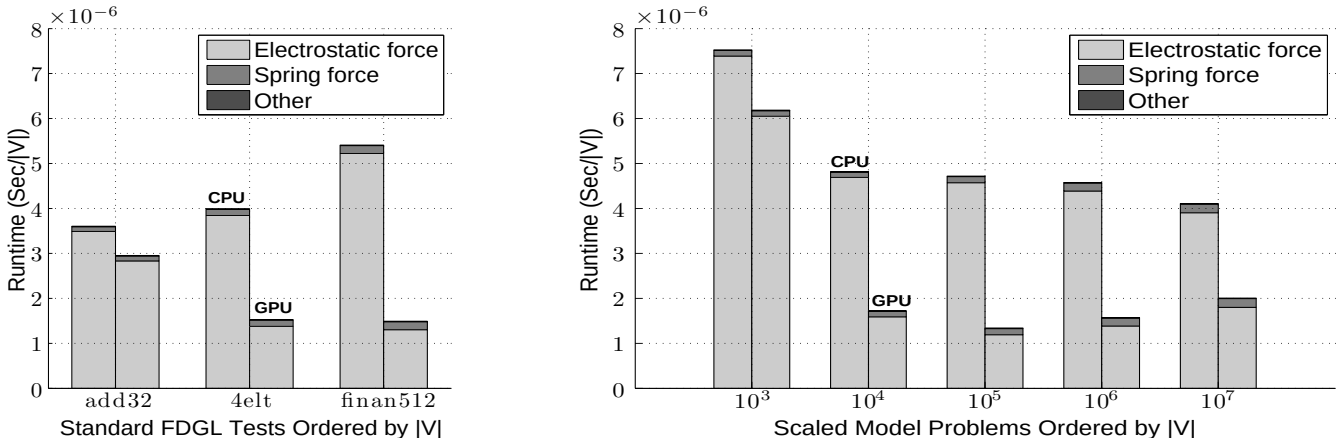
Spring attraction force computation capability is not present in `ExaFMM`, so the attraction forces are directly added at the end of each FMM iteration (this code runs on the CPU only). This addition to the framework results in a total time complexity of $\mathcal{O}(|V| + |E|)$. Because the output of the single repulsion iteration is always returned to the CPU, the FMM tree gets rebuilt on each subsequent call. Unfortunately, this does not take full advantage of GPU caching, and we notice a performance degradation in comparison with [6].

III. Performance Analysis

We have conducted a performance analysis on several well-known community graphs (Fig. 2a) as well as artificial graphs (Fig. 2b) generated by a modified version of `pywebgraph-2.72` [14]. Although the `pywebgraph` library can generate random graphs featuring power law connectivity relationships reflective of networks, protein

Table I
Source Data

Graph	Vertices	Edges	Source
10^3	10^3	2×10^3	pywebgraph [14]
10^4	10^4	2×10^4	pywebgraph
10^5	10^5	2×10^5	pywebgraph
10^6	10^6	2×10^6	pywebgraph
10^7	10^7	2×10^7	pywebgraph
add32	4,960	9,462	Walshaw [4]
4elt	15,606	45,878	Walshaw
finan512	74,752	261,120	Walshaw



(a) Walshaw's datasets

(b) *pywebgraph* generated datasets

Figure 2. Runtime Breakdown using a single core and GPU on the local Workstation

interactions, and social graphs, we modified the library to generate graphs with both a specific requested number of vertices and a minimum number of edges while maintaining a pseudorandom power law connectivity relationship to the graph. Previously, the library could only generate graphs with a specified maximum number of edges. We have noticed that most of the available datasets online reflect a low edge density while this library permitted us to test at higher and controlled vertex count and edge density. All generated datasets have been used for most of our time analysis. We also verified our approach against several standard test graph datasets. This comparison is summarized in Table I.

`ExaFMM` has several tuning parameters that allow it to take advantage of the different architectures [11]. In Fig. 2 we compare a single iteration run on a single CPU vs. that for a single core/GPU on the local workstation for different data sizes. The runtime is normalized by the number of vertices $|V|$, and the datasets are sorted according to $|V|$ as well. For each dataset there are two bars; one for the CPU runtime, and the other for the GPU runtime. Different colors in the bar represent different stages of the FDGL calculation. It can be seen that the repulsion dominates the runtime for all cases, however, the runtime per vertex varies. For smaller datasets the runtime per vertex is larger due to the overhead of tree construction and data copies in the FMM. This trend is more prominent on the GPUs since there is a large overhead for the data copies between the host and device.

In Figure 2a the data sets are in ascending order as shown in Table I. The timings are averaged over 100 steps. Each data set has a different connectivity pattern, where *add32* is a computer component design for a 32-bit adder, *4elt* is a finite element mesh, and *finan512* is a linear programming matrix. The different connectivity results in different vertex distribution at later time steps. On the other hand, all graphs in Figure 2b are power law graphs generated by `pywebgraph`, and have similar connectivity. This is why we see the runtime per vertex increase in Figure 2a, whereas the runtime per vertex decreases in Figure 2b.

For a more thorough investigation of the dependence of the problem size on the runtime of the FMM, we tested for a broader range of data sizes in smaller increments. The runtime of the FMM on the CPU and GPU are shown in Fig. 3, where $|V|$ is the number of vertices. On the CPU the strong scaling of the FMM is quite poor until $|V| > 10^5$. On the GPU, good strong scalability is achieved only when $|V| > 10^6$.

The breakdown of the CPU and GPU runtime for $|V| = 10^7$ is shown in Fig. 4. The runtime is multiplied by the number of cores/GPUs for better visibility of the breakdown. The dashed line represents perfect strong scaling. Both the "Electrostatic force" and "Spring force" include the communication time. The "Spring force" is always calculated on the CPU, and takes a significant portion of the runtime when using many GPUs.

Fig. 5 shows the overall speedup of the FDGL code on up to 32 CPU cores/GPUs for $|V| = 10^7$. Super-linear scaling is observed in the GPU runs, which is consistent with previous reports on FMMs on GPUs. [11]

IV. Comparative Analysis

Godiyal et. al. [6] implements FM^3, a CPU Fast Multipole Multilevel Method [7], on a single GPU. To reduce the running time of the system the authors decided to skip the local expansion, making the hierarchical computation equivalent to a Barnes & Hut's treecode [15]. Their work starts by finding an optimal placement of vertices in close Euclidean distance to their connected vertices. Then, the authors solve the problem at a coarse mesh level and use the result to solve downwards to the finer level until the

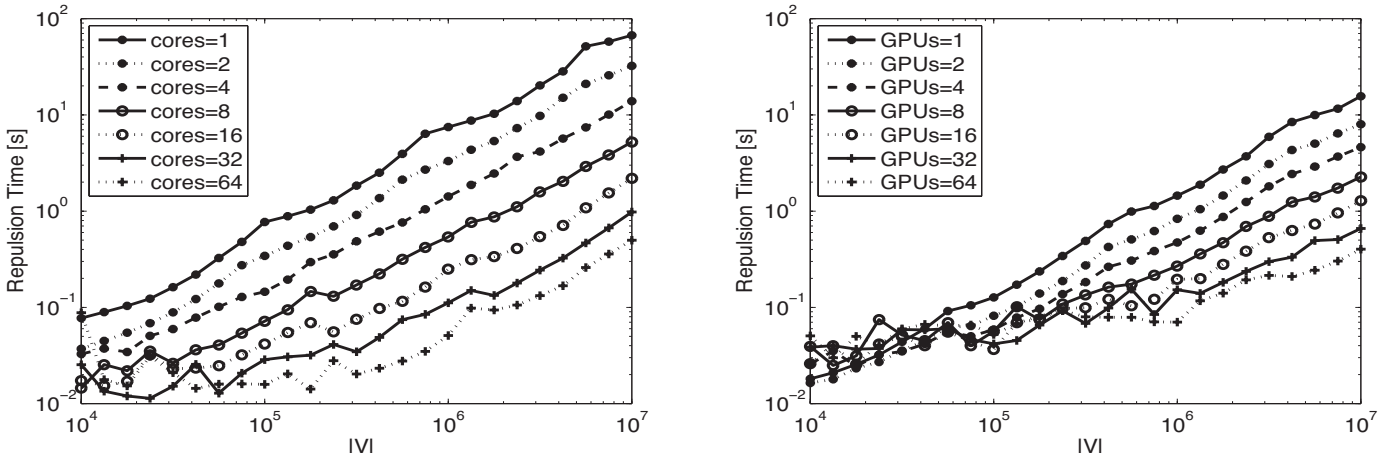


Figure 3. One Iteration Repulsion Time on TSUBAME

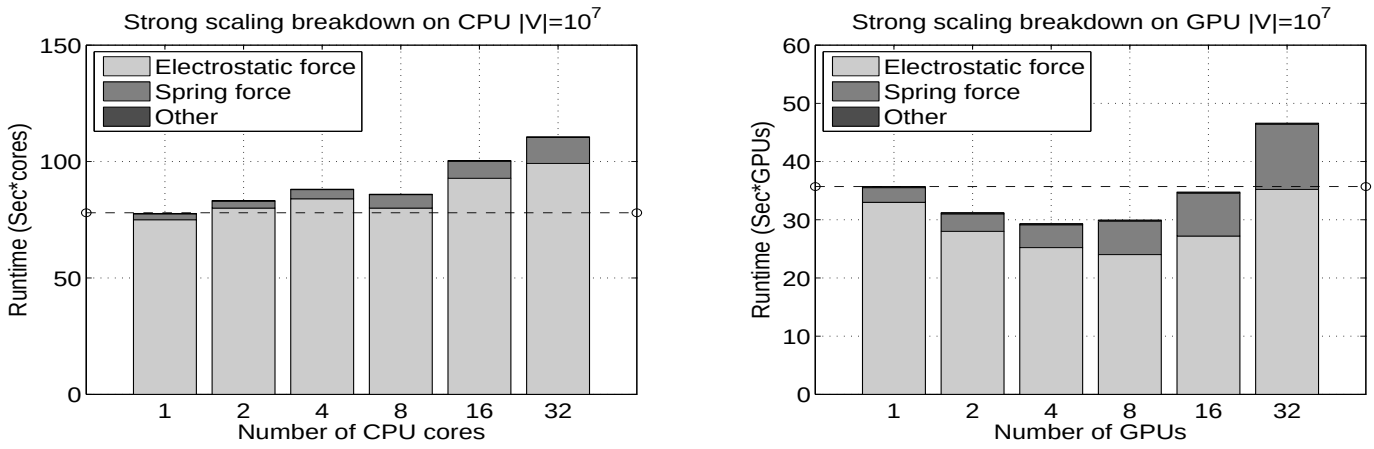


Figure 4. Strong Scaling Breakdown on TSUBAME for $|V| = 10^7$

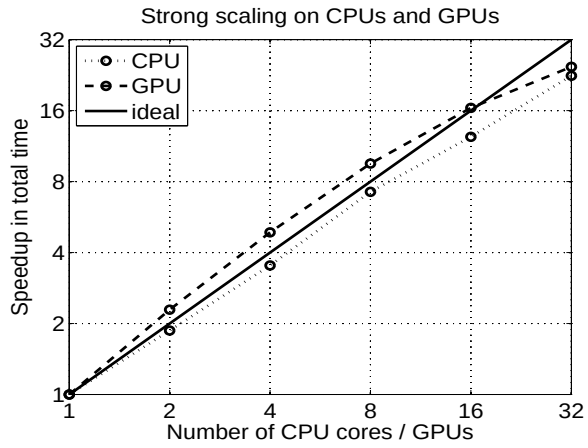


Figure 5. Speedup on TSUBAME for $|V| = 10^7$

complete graph is displayed.

The displayed graphs by Godiyal look appealing and the running times are compelling (though the number of iterations is not available, making direct comparison impossible). The authors hand modify the tuning parameters until they get the optimal graph and convergence speed. We can not be certain that the idea of pre-placing the vertices is helpful given that the tuning parameters are still being hand modified. Per our experience with several of the community graphs, a small modification in even a single tuning parameter is enough to cause the layout algorithm to vary from quick convergence (~ 100 iterations) to slow convergence (> 1000 iterations) or even a failure to converge. Equally concerning is the sensitivity in graph quality to tunable parameters; slightly adjusting a parameter can result in twisted or otherwise poor layouts.

ExaFMM has a scalable multi-GPU adaptive FMM im-

plementation that does not skip over the local expansion calculations, thus allowing for further scaling up the size of the graphs (though with a time penalty on smaller graphs).

Tikhonova et. al. [5] implements an algorithm that runs on multi-processors and balances the load by pinning edges with their vertex on the same processor. Tikhonova then balances allocation on the different processors using the sum of the number of vertices and edges. Given the cost of repulsion being orders of magnitude larger than the calculation of springs, balancing only the number of vertices should give a better overall load balance. The imbalance in the edge calculations will not have a significant impact on the overall runtime, since it consumes only a very small fraction. A better technique would be to also keep as much of the connected vertices on the same machine as possible. `ParMETIS`, a parallel graph partitioner by Karypis et. al. [16], handles this problem in an efficient and robust manner.

Tikhonova also uses the idea of partial graph layouts. Their approach is different from Godiyal's in that they are introducing a subset of vertices at each stage and making all the previously added vertices static to cut down the cost of computing the repulsion forces. No further optimizations have been made, and their work focused primarily on the parallelization of the algorithm, thus the count of iterations is given but a convergence test is not done.

Tikhonova's work on the `BigBen - Cray XT3, 2.4 GHz` system required 64 CPUs 7.89 seconds to compute 100 iterations for a test problem. On the same problem, we were able to compute the same number of iterations on a single CPU in 6.32 seconds, GPU acceleration brought this number further down to 2.43 seconds. The highly tuned exaFMM library grants us a clear performance advantage even without its added capability of distributed-memory parallelization.

V. Limitations and Future Work

FDGL has many tunable parameters and heuristics that play a direct role in the efficiency and quality of the layout algorithm. This current state makes research in the area cumbersome and repetitive. Amalgamating the different solutions into a centralized repository will allow for more concentrated work on improving the heuristic for not only the current set of test graphs, but graphs beyond the capabilities of today's approaches.

The amount of movement that the vertices experience on each iteration is high, so it is impractical to assume the distribution of the vertices across processes to be the same from one time step to the next. Instead, the building of the FMM tree is repeated on each time step. To keep the costs of running FMM reasonable, vertex migration from one processor to the next should be done during the tree build phase of the algorithm.

One promising strategy is to use ACE's multi-scale eigenvector computation [9] for a fast and coarse preplacement, then distribute the vertices and their edges using `ParMETIS` across processors by minimum edge cuts. We hope that this will correspond to a better initial heuristic in the distribution of vertices and their later movements, lowering both per iteration runtime and the total number of iterations.

VI. Conclusion

In the present work, a scalable adaptive FMM on heterogeneous architectures is used to accelerate the repulsion calculation in the FDGL. With the current FMM implementation we are able to reduce the complexity from the current state-of-the-art at $\mathcal{O}(|V|\log|V| + |E|)$ to $\mathcal{O}(|V| + |E|)$ per iteration. The particular heuristics to accelerate the convergence of FDGL is beyond the scope of this study. The aim of this work is rather to provide a fast repulsion calculation technique, which accelerates all flavors of FDGL regardless of the convergence rate.

We have tested on graphs with up to 10 million vertices and 20 million edges, and found that the runtime per vertex remains somewhat constant for large enough graphs. On a single node, we observed that the repulsion dominates the runtime, and the spring force calculation is insignificant. On the other hand, on multiple MPI processes the communication for the spring calculation takes a significant amount of time. On multi-GPUs the spring communication time consumes a larger portion of the total runtime since the repulsion is accelerated on the GPU. Strong scalability tests for 10 million vertices show 70 % parallel efficiency for CPUs, and 76.6 % parallel efficiency for GPUs. The runtime on a single CPU is 10^{-5} sec/vertex and 3×10^{-6} sec/vertex on a single GPU. Our work shows significant speed up of a single iteration of FDGL, allowing domain experts to take advantage of the highly scalable FMM library in furthering their work on visualizing their ever increasing datasets.

References

[1] T. M. J. Fruchterman and E. M. Reingold, "Graph Drawing by Force-Directed Placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, Nov. 1991.

[2] T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989.

[3] Y. Hu, "Efficient, High-Quality Force-Directed Graph Drawing," *The Mathematica Journal*, vol. 10, no. 1, pp. 37–71, Jan. 2006.

[4] C. Walshaw, "A Multilevel Algorithm for Force-Directed Graph-Drawing," *Journal of Graph Algorithms and Applications*, vol. 7, no. 3, pp. 253–285, 2003.

[5] A. Tikhonova and K.-l. Ma, "A Scalable Parallel Force-Directed Graph Layout Algorithm," in *Eurographics Symposium on Parrallel Graphics and Visualization*, 2008, pp. 25–32.

[6] A. Godiyal, J. Hoberock, M. Garland, and J. Hart, "Rapid Multipole Graph Drawing on the GPU," in *Graph Drawing*. Springer, 2009, pp. 90–101.

[7] S. Hachul and M. J, "Drawing Large Graphs with a Potential-Field-Based Multilevel Algorithm (Extended Abstract)," in *The International Symposium on Graph Drawinghe International Symposium on Graph Drawing*. Springer-Verlag, 2004, pp. 285–295.

[8] Y. Jia, J. Hoberock, M. Garland, and J. C. Hart, "On the Visualization of Social and Other Scale-Free Networks." *IEEE transactions on visualization and computer graphics*, vol. 14, no. 6, pp. 1285–92, 2008.

[9] Y. Koren, L. Carmel, and D. Harel, "ACE : A Fast Multiscale Eigenvectors Computation for Drawing Huge (full version)," in *Proceedings of the IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, 2002, pp. 137–144.

[10] R. Yokota, "Fast Multipole Method for Exascale Computing." [Online]. Available: https://bitbucket.org/exafmm/exafmm

[11] R. Yokota and L. Barba, "Hierarchical N-body Simulations with Auto-Tuning for Heterogeneous Systems," *Computing in Science Engineering*, 2012.

[12] R. Yokota and L. A. Barba, "A Tuned and Scalable Fast Multipole Method as a Preeminent Algorithm for Exascale Systems." *The International Journal of High Performance Computing Applications*, 2012.

[13] R. Yokota, L. A. Barba, T. Narumi, and K. Yasuoka, "Petascale Turbulence Simulation Using a Highly Parallel Fast Multipole Method," *arXiv:1106.5273v1*, 2011.

[14] S. Sathe, "A Python Open Source Random Power Law Generation Library." [Online]. Available: http://pywebgraph.sourceforge.net/

[15] J. Barnes and P. Hut, "A Hierarchical O(N log N) Force-Calculation Algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, 1986.

[16] G. Karypis and V. Kumar, "ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering," 2003. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview