Collaboration spotting visual analytics tool optimized for very large graphs

Richard Forster
forceuse@inf.elte.hu

ELTE Eötvös Loránd University
Budapest, Hungary

Abstract

As big data is getting collected in every sciences and other applications too, visual analytics is starting to gain traction with graph based applications to help the users understand their data. Community detection has become an important operation in such applications. It reveals the groups that exist within real world networks without imposing prior size or cardinality constraints on the set of communities. The Louvain method is a multi-phase, iterative heuristic for modularity optimization. It was originally developed by Blondel et al. [1], the method has become increasingly popular owing to its ability to detect high modularity community partitions in a fast and memory-efficient manner. To parallelize this solution multiple heuristics are used, that were first introduced in [2]. For graph organization the ForceAtlas algorithm is used that is part of the Gephi toolkit [3]. This method is responsible to assign coordinates in a 2D space for every node in such a way that they will not overlap on each other. In this paper CPU based parallel implementations are provided for these sequential algorithms, and are tested on a single processor with 8 threads, where a 7 fold performance increase was achieved on test data taken from the CERN developed Collaboration Spotting's database, that involves patents and publications to visualize the connections in technologies among its collaborators.

1 Introduction

The Collaboration Spotting [4] graph based tool is a generic version of the CERN and JRC developed TIM (Technology Innovation Monitor), that serves as a visual analytic application. The data about technologies are stored in a graph database and those are traversed to get a specific subgraph based on user queries that will be visualized as an interactive image of the graph. The user can extract different informations from publications and patents, such as authors, keywords, organizations, etc. On organization landscapes it is a requirement to be able to distinguish the different groups working together on the visualized papers, hence community detection plays an important role in the tool's calculations.

Interactive means that the user can always navigate the map that he or she is looking at. Every user starts from their own user defined technology-gram, that is populated by the users as they are exploring the different

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

technologies, requesting the connected papers and patents. From there they can start navigating based on different properties of the papers and patents at hand. They can take a look at the organizations working on a specific technology or finding out what kind of keywords the authors used in their respective publications.

The tool is generic as it is capable to handle all kind of data until it fits a predefined graph database schema. Right now the tool is not only about patents and publications, but within CERN it starts to provide visualization to different experiments and departments based on their data and their own needs.

As this application may handle graphs with tens of thousands of nodes and hundreds of thousands of edges per user, it is clear that the overall system needs to be able to process even millions of nodes and edges at any given time. And because the tool is supposed to give an interactive session for every user it has to be fast and reliable to serve the user requests seamlessly.

We take a look at the community detection problem, what procedures are used to do the computations and what heuristics were applied on the system and we will also describe how to make the interactive visualization useful to the users by applying force-directed algorithms on the graphs. Taking all processes and providing parallelization on them, in overall we were able to achive a speedups up to 7.

2 Problem statement and notation

Let $G(V, E, \omega)$ be an undirected weighted graph, where V represents the set of vertices, E is the set of edges and ω is a weighting function that gives every edge in E a positive weight. If the input graph is unweighted, then we consider the weight of the edges to be 1. The graph is allowed to have loops, so edges like (i, i) are valid, while multiple edges should not be present. Let the adjacency list of vertex i be the following: $\Gamma(i) = j|(i, j) \in E$. Let δ_i denote the weighted degree of vertex i, such as $\delta_i = \sum_{j\in\Gamma(i)} \omega(i, j)$. Let $\#V$ denote the number of vertices in graph G, $\#E$ the number of edges, and W the sum of all edge weights such as $\#E = \frac{1}{2}\sum_{i\in V} \delta_i$. To detect the communities we would like to partition the vertex set V into an arbitrary number of disjoint communities, each with different sizes $(0 < n \leq \#V)$. We will use $C(i)$ to denote the community containing vertex i. Let $E_{i\to C}$ be the set of all edges connecting vertex i to vertices in community C. Also let $e_{i\to C}$ contain the sum of the edge weights in $E_{i\to C}$.

$$e_{i\to C} = \sum_{(i,j)\in E_{i\to C}} \omega(i, j) \tag{1}$$

Let deg_C denote the sum of the degrees of every vertices in community C, which will give us the degree of the whole community.

$$deg_C = \sum_{i\in C} \delta_i \tag{2}$$

Even with the groups generated by community detection helping to understand what is in the data we are looking at, it is still difficult to handle. To be fully useful we need to provide a readable representation that could be easily understood by users. This helps us to make interactions with the graph possible, while maintaining a fast and versatile system.

2.1 Modularity

Let $S = C_1, C_2, ..., C_k$ be the set of every community in a given partitioning of the vertex set V in $G(V, E, \omega)$, where $1 \leq k \leq \#V$. Modularity Q of the partitioning S is given by the following [5]:

$$Q = \frac{1}{2W}\sum_{i\in V} e_{i\to C(i)} - \sum_{C\in S}(\frac{deg_C}{2W} \cdot \frac{deg_C}{2W}) \tag{3}$$

Modularity is widely used for community detection while it has issues such as resolution limit [6][7], and the definition itself has multiple variants [7][8][9]. However, the definition in Eq. 3 is still the more common version, as it is in the Louvain method [1].

2.2 Community detection

On a given $G(V, E, \omega)$ the problem of community detection is to compute the partitioning S of communities that produces the maximum modularity. This problem has been shown to be NP-Complete [10]. This problem basically is different from the graph partitioning and its variants [11], where the number and size of the clusters and their distribution are known at the beginning. For community detection, both quantities are unknown for the computation.

3 The Louvain algorithm

In 2008 Blondel et al. presented an algorithm for community detection[1]. The Louvain method, is a multi-phase, iterative, greedy algorithm capable of producing the community hierarchy. The main idea is the following: the algorithm has multiple phases, each phase with multiple iterations that are running until a convergence criteria is met. At the beginning of the first phase each vertex is going to be assigned to a separate community. Going on, the algorithm progresses from one iteration to another until the modularity gain becomes lower than a predefined threshold. With every iteration, the algorithm checks all the vertices in an arbitrary but predefined order. For every vertex i, all its neighboring communities (where the neighbors can be found) are explored and the modularity gain that would be the result of the movement of i into each of those neighboring communities from its current one is calculated.

Once this calculation is done, the algorithm selects one of the neighboring communities that would provide the maximum modularity gain, as the new community for i, and it also updates the necessary structures that are maintained to hold the communities and its properties. On the other hand, if all the gains are negative, the vertex stays in its own community. An iteration ends once all vertices are checked. As a result the modularity is a monotonically increasing function, that is spread across the multiple iterations of a phase. When the algorithm converges within a phase, it moves to the next one by reducing all vertices of a community to a single "meta-vertex"[12], placing an edge from such meta-vertex to itself with a weight that is the sum of the weights of all the edges within that community and putting an edge between two meta-vertices with a weight that is equal to the sum of the weights of all the edges between the corresponding two communities.

The result is graph $G'(V', E', \omega')$, which then becomes the input for the consecutive phase. Multiple phases are processed until the modularity converges. At any given iteration, let $\Delta Q_{i \to C(j)}$ represent the modularity gain that resulting from moving vertex i from its current community $C(i)$ to a different community $C(j)$. This is given by:

$$\Delta Q_{i \to C(j)} = \frac{e_{i \to C(j)}}{W} + \frac{2 \cdot \delta_i \cdot mod_{C(i)/i} - 2 \cdot \delta_i \cdot mod_{C(j)}}{(2W)^2} \tag{4}$$

The new community assignment for vertex i will be determined as follows. For $j \in \Gamma(i) \cup i$:

$$C(i) = arg \max_{C}(j) \Delta Q_{i \to C(j)} \tag{5}$$

In [10] thanks to the introduced data structures, the calculation time for $\Delta Q_{i \to C(j)}$ is constant $O(1)$. This way the full iteration's complexity is $O(\#E)$. Effectively we let the iterations and phases run forever, but because the modularity is monotonically increasing, it is guaranteed to terminate at some point. Generally the method needs only tens of iterations and fewer phases to terminate on real world data.

4 Louvain parallel heuristics

In [12] the challenges were explored that arise from parallelizing the Louvain method. To solve those issues multiple heuristics were introduced that can be used to effectively introduce the basically sequential algorithm to parallel systems. From the proposed heuristics in this paper we will see two applied on our system. For them we assume the communities at any given stage of the algorithm be labeled numerically. We will use the notation $l(C)$ to return the label of community C.

4.1 Singlet minimum label heuristic

In the parallel algorithm, if at any given iteration the vertex i which is in a community by itself ($C(i) = i$, singlet community [12]), that vertex will decide to move into another community possibly holding also one single vertex j in hope for modularity gain. This change will only be applied if $l(C(j) < l(C(i)))$.

4.2 Generalized minimum label heuristic

In the parallel algorithm, if at any given iteration the vertex i has multiple neighboring communities providing the maximum modularity gain, then the community with the minimum label will be selected as its destination community. It is possible that swap situations where two vertices from each others target community will change place delay the convergence, but they can never lead to nontermination as the algorithm uses a minimum required modularity gain threshold.

4.3 Parallel algorithm

The parallel algorithm has the following major parts:

- Phases: Execute phases one at a time. Within each phase, multiple iterations are running and each iteration performs a parallel evaluation of the vertices without any locking mechanism using only the community information from the previous iteration. This is going on until the modularity gain between the iterations is not below the threshold.

- Graph rebuilding: After a successive phase the new community assignment is used to construct a new input graph for the next phase. This is done by introducing the communities in the new graph as vertices and the edges are added based on their connection to these communities.

Algorithm 1 The parallel Louvain algorithm for a single phase, inputs are the graph $G(V, E, \omega)$ and a Status object containing the initial community informations

```
 1: procedure PARALLEL LOUVAIN((G(E, V, ω), Status))
 2:     Q_curr ← 0
 3:     Q_prev ← −∞
 4:     while true do
 5:         for each i ∈ V_k do
 6:             N_i ← Status.nodesToCom[i]
 7:             for each j ∈ Γ(i) do
 8:                 N_i ∪ Status.nodesToCom[j]
 9:             end for
10:             target ← arg max_{t∈N_i} ΔQ_{i→t}
11:             if ΔQ_{i→target} > 0 then
12:                 Status.nodesToCom[i] ← target
13:             end if
14:         end for
15:         Q_curr ← Compute modularity for new assignments
16:         if |(Q_curr − Q_prev)/Q_prev| < θ then, θ being a user specified threshold
17:             break
18:         else
19:             Q_prev ← Q_curr
20:         end if
21:     end while
22: end procedure
```

The cycle in line 4 represents a parallel evaluation of all nodes in the input graph. The possible modularity gain is calculated for all nodes in parallel with the resulting new collaboration assignment.

5 ForceAtlas

The ForceAtlas [3] algorithm generates a force-directed layout on a given set of nodes, acting as an n-body simulation. Nodes repulse each other while edges attract the nodes they connect. These forces are driving the system to converge to a balanced state. This final configuration is expected to give a map of the input nodes, that will help the interpretation of the data they represent. It is generally true, by applying this technique not all graphs will have exactly the same output given multiple runs. The end result highly depends on the forces applied, but it can be influenced by the initial state of the system too. The result is very different from a Cartesian projection, the position of a node cannot be read, it has to be compared to other nodes. Still, this approach has a unique advantage: it makes the graph's visual interpretation much easier. As it works the structurally relevant data is represented in a visually connected map.
ForceAtlas has two main features, that makes it stand out from other force-directed algorithms:

- The final layout highly depends on the given graph's energy model, which represents how the repulsion and attraction is handled.
- In a force-directed algorithm, the layout is applied iteratively to the graph. These steps are simulating the movements that will make the system reach the balance. In every step the forces are recomputed and the nodes are moved from their original position according to the results.

5.1 Energy Model

The force-directed algorithms relies on a certain formula to calculate the repulsion and another one to get the attraction. As described above, force-directed algorithms are simulating a physical system, that involves repulsion and attraction. As in any physical system, such forces are proportional to the distance between the interacting nodes. The nodes that are closer to each other has a higher repulsion force, but lesser attraction and vice versa. The proportionality can be linear, exponential or logarithmic.

5.2 Repulsion by degree

ForceAtlas was designed to work well with different web graphs and social networks. Commonly these networks have many *leaves*. This is because of the power-law distribution of degrees that describes many real-world data. The forest of *leaves* surrounding the highly connected nodes is one of the most significant sources of visual cluttering.

To solve this, the idea is to bring less strongly connected nodes closer to highly connected ones. In this algorithm the repulsion force is calculated (Equation 6) in such way so the poorly connected nodes and strongly connected nodes repulse less. As a result they will get closer to each other in the balanced state. The repulsion force F_r is proportional to the degrees plus one of the two nodes. The coefficient k_r is provided by the user settings.

$$F_r(n_1, n_2) = k_r \frac{(deg(n_1) + 1)(deg(n_2) + 1)}{d(n_1, n_2)} \tag{6}$$

In this formula the +1 ensures that even nodes with a degree of zero still have some repulsion force.

5.3 A classical attraction force

The attraction force (Equation 7) F_a between two connected nodes n_1 and n_2 depends linearly on the distance $d(n_1, n_2)$.

$$F_a(n_1, n_2) = d(n_1, n_2) \tag{7}$$

6 Parallel ForceAtlas

As our tool needs to process very big graphs with even tens of thousands of nodes and hundreds of thousand of edges it is important to increase the performance of the ForceAtlas computation too. To optimize the algorithm in the current implementation we worked on the repulsion (Subsection 5.2) and attraction (Subsection 5.3) part of process. The original algorithm was designed with user interaction in mind, which means the user could see some animation as the iterations were progressing. This is a nice feature considering the calculations were running on the client side. Because we would like to focus on performance and thus would push the calculations to the server side, we left this feature out.

6.1 Repulsion phase

For this part first we use the Barnes-Hut algorithm [13] to recursively divide our nodes into groups by storing them in a quad-tree. The nodes of this tree represents a region of the two-dimensional space. The root node stands for the whole space, and its four children represent the four quartets of the space. The full space is recursively subdivided into quartets until each subdivision contains 0 or 1 nodes. We distinguish two types of nodes in the tree: internal and external nodes. The external node has no children and is either empty or points to single node from the original space. Each internal node represents the group of nodes belonging to it, and stores the center of mass and the total mass of all its children nodes.

After this we use the regions as nodes and repulse them from each other. In those cases where the region contains multiple nodes, they will repulse together with the region. Naturally those regions where only one node is present will repulse the node itself.

Using the created regions, a Θ user provided threshold and *coefficient* predefined value, **algorithm 2** shows how the repulsion is calculated.

Algorithm 2 Parallel ForceAtlas repulsion phase with inputs of the arrays of regions and nodes

1: **procedure** PARALLEL REPULSION((regions, nodes))
2:   **for each** $n \in nodes$ **do**
3:     $r \leftarrow root\ regions$
4:     **while** true **do**
5:       **if** r has sub-regions **then**
6:         *distance*←distance between n and r
7:         **if** $2 * regions[r].size/distance < \Theta$ **then**
8:           $xDist \leftarrow n.x - r.massCenterX$
9:           $yDist \leftarrow n.y - r.massCenterY$
10:           $factor \leftarrow coefficient * n.mass * r.mass/distance^2$
11:           $n.dx \leftarrow xDist * factor$
12:           $n.dy \leftarrow yDist * factor$
13:           $r \leftarrow next\ sibling$
14:         **else**
15:           $r \leftarrow first\ child$
16:         **end if**
17:       **else**
18:         There is only 1 node in the region, repulse from that
19:         **if** There are no more siblings **then**
20:           **break**
21:         **end if**
22:       **end if**
23:     **end while**
24:   **end for**
25: **end procedure**

From line 2 repulsion is calculated for all nodes in parallel.

6.2 Attraction phase

Here we would like to calculate how the links between the nodes will affect the repulsion of them. The nodes connected by edges will be attracted to each other, keeping them closer to other connected nodes. All the edges are processed in parallel and because multiple edges can connect to the same node to change the distance of the effected elements, we use atomic operations to accumulate all the modifications. Because the distance values are stored as floating point numbers and the generic C++ floating point types do not have atomic addition operations defined, we use the *compare-exchange* functions to define the summation.

The user can control through the ForceAtlas settings how much the edge weights should influence the attraction. Different graphs possibly will react differently to a specific set of settings, resulting in a different layout in the end. Users are encouraged to experiment with the different values to see how will they yield the

best layout that suits best their use cases.

Algorithm 3, using the *coefficient* predefined value shows how the attraction is done.

Algorithm 3 Parallel ForceAtlas attraction phase with inputs of the arrays of nodes and edges

```
1: procedure PARALLEL ATTRACTION((regions, nodes))
2:     for each e ∈ edges do
3:         n1 ← e.source
4:         n2 ← e.target
5:         w ← e.weight
6:         Testing edge influence user setting
7:         if influence = 0 then
8:             ewc ← 1
9:         end if
10:        if influence = 1 then
11:            ewc ← w
12:        else
13:            ewc ← w^influence
14:        end if
15:        xDist ← n1.x − n2.x
16:        yDist ← n1.y − n2.y
17:        distance ← √(xDist² + yDist²)
18:        factor ← −coefficient ∗ ewc
19:        if distance > 0 then
20:            AtomicAdd(n1.dx,xDist ∗ factor)
21:            AtomicAdd(n1.dy,yDist ∗ factor)
22:            AtomicAdd(n2.dx,−(xDist ∗ factor))
23:            AtomicAdd(n2.dx,−(yDist ∗ factor))
24:        end if
25:    end for
26: end procedure
```

The cycle in line 2 represents the parallel evaluation of the attraction applying on each node of the input graph.

7 Evaluation

For testing purposes we took 4 graphs from the Collaboration Spotting's database, that is based on data, coming from the Web of Knowledge. For the test set the searches were run to gather the papers currently in the database related to *3D, CT, Database* and *Silicon*. The produced graphs are containing all the organizations collaborating on that specific technology. The reason why we choose these graphs is that they can effectively demonstrate the use of community detection and layout generation. We will take a look at the details of the graphs as we evaluate the different runtimes. The evaluation is strictly done on the described algorithms, the database processing and retrieval time is not under discussion in this paper.

The optimized version is implemented in C++, so for a fair comparison between the two, we prepared the implementation of the standard versions of the Louvain and ForceAtlas algorithms also in the same environment. The tests of both versions were running on the following (Table 1) system:

Table 1: The test system

CPU	Intel Core i7 4710HQ
RAM	24 GB DDR3
OS	Windows 10 x64
Threads	8

The optimized version was running on 8 threads in both algorithms.

7.1 Runtimes

We are interested in examining the performance of the whole compuation, so we measure the runtime on the Louvain community detection algorithm (Section 3) in its entirety, running its original and optimized version on all test graphs and then we do same with the ForceAtlas (Section 5) calculation. These runtimes are compared to each other, showing the performance gain of our modified algorithms, while also showing the ratio how these computations are dividing the whole process among them.

7.1.1 3D graph

For the testing first we use the graph that contains the collaborators on all the papers that are related to 3D technology. This graph has 35763 nodes and 192336 edges. The runtime of the algorithms on this input can be seen on Figure 1.

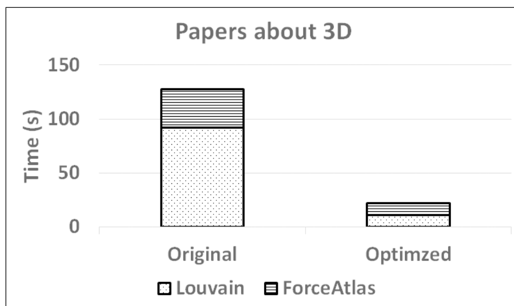

Figure 1: Runtime on a graph containing papers about 3D

These results are showing us that the overall computation time of the original implementation is $t_{original} = 127,4s$ and the optimized version is just $t_{optimized} = 21,8s$. For the "3D" graph the original Louvain algorithm took $t_{louvain_{original}} = 92,2s$, the optimized code took $t_{louvain_{optimized}} = 10,8s$ to finish the whole community detection. The original ForceAtlas took $t_{forceatlas_{original}} = 35,2s$ to finish and the parallel version took only $t_{forceatlas_{optimized}} = 11s$ to generate the final balanced layout.

Overall, the full process was able to finish with everything $\Delta_t = 5,8$ times faster on the optimized algorithms.

7.1.2 CT graph

The next graph contains the collaborators working on papers referring to the CT technology. This graph has 53039 nodes and 325490 edges. The runtime of the processing can be seen on Figure 2.

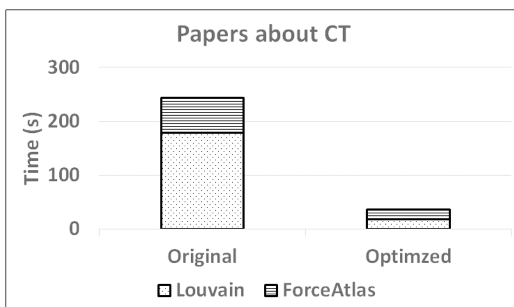

Figure 2: Runtime on a graph containing papers about CT

The results are showing that the overall computation time of the original implementation is $t_{original} = 243,82s$ and the optimized version is handling the same processing in just $t_{optimized} = 35,707s$. For the "CT" graph the original Louvain algorithm took $t_{louvain_{original}} = 179,1s$, the optimized took $t_{louvain_{optimized}} = 18,1s$ to fully

generate the communities. The original ForceAtlas took $t_{forceatlas_{original}} = 64,72s$ and the optimized took only $t_{forceatlas_{optimized}} = 17,6s$ to return with the finished layout.

For the overall computation of both the community detection and layout generation, the runtime was $\Delta_t = 6,8$ times faster on the optimized algorithms.

7.1.3 Database graph

The next graph contains the network of collaborators connected to Database technologies. The whole graph consists of 17832 organizations and 113622 corresponding edges among them. The full processes runtime values can be seen on Figure 3.

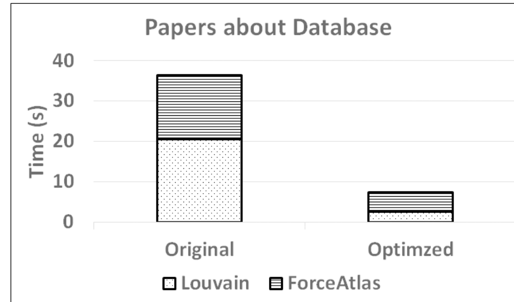

Figure 3: Runtime on a graph containing papers about database

This shows us that the overall computation time of the original implementation is $t_{original} = 36,45s$ and the optimized version is just $t_{optimized} = 7,44s$. For the "Database" graph the original Louvain algorithm took $t_{louvain_{original}} = 20,5s$ to finish, while the optimized took only $t_{louvain_{optimized}} = 2,6s$ to return the communities. The original ForceAtlas took $t_{forceatlas_{original}} = 15,95s$ to generate the layout, while the optimized could finish in $t_{forceatlas_{optimized}} = 4,84s$.

For the overall computation the runtime was $\Delta_t = 4,89$ times faster on the optimized algorithms.

7.1.4 Silicon graph

The last graph contains the network connected to technologies working with silicon. This graph has 24923 nodes and 185594 edges. The final runtime can be seen on Figure 4.

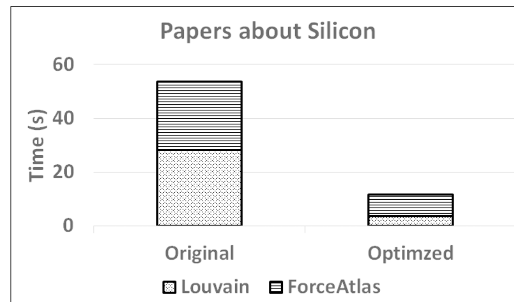

Figure 4: Runtime on a graph containing papers about silicon

The overall computation time of the original implementation is $t_{original} = 53,6s$ and the optimized version is only $t_{optimized} = 11,67s$. For the "Silicon" graph the original Louvain algorithm took $t_{louvain_{original}} = 28,3s$ to finish its computations and the optimized took $t_{louvain_{optimized}} = 3.51s$ on the same dataset. The original ForceAtlas took $t_{forceatlas_{original}} = 25,3s$ to process the graph and the optimized was able to conclude in only $t_{forceatlas_{optimized}} = 8,16s$.

For the whole computation on the tested algorithms the overall runtime was $\Delta_t = 4,6$ times faster on the optimized algorithms.

7.2 Analysis

From the detailed runtimes we can see, that the Louvain computation takes more time to finish from the two algorithms, also it is where the biggest speedup can be achieved. The optimized Louvain algorithm is in average $\Delta_{t(louvain)} = 9,1$ times faster than the sequential implementation, while the ForceAtlas gains in average a $\Delta_{t(forceatlas)} = 3,4$ speedup. This happens because of two things: first the ForceAtlas runs on the quad-tree generated by the Barnes-Hut algorithm to calculate the repulsions (Subsection 6.1) leading to smaller number of nodes than what we have in the original input graphs, requiring less operations to finish the process. On the other hand the Louvain has to check every possible neighbor for the nodes to decide which community will yield the best possible modularity gain (Subsection 3), thus leading to more computation.

The speed up of the Louvain algorithm is in line with the increase of the computing resources, using 8 threads instead of just 1. The higher performance increase can be explained with the usage of the parallel heuristics (Section 4). It reduces the number of operations in the swapping scenarios, where no modularity gain check is required, as the nodes are always simply moved towards the community with a bigger label. Overall the clustering scales well with the available resources.

These can help the Louvain to perform better, than ForceAtlas. Furthermore While the quadratic tree decreases the required amount of computation, it introduces an additional bottleneck in the form of irregular memory accesses. In this case the algorithm cannot effectively utilize the CPU's caching system, thus reducing the scalability in the current implementation.

While the Louvain algorithm is inherently sequential with the heuristics we can drastically increase the performance of the computation, $t_{louvain_{optimized}} << t_{louvain_{original}}$. It may be that the ForceAtlas algorithm did not receive such boost in performance, but the gained runtimes from the optimization are still significantly better, $t_{forceatlas_{optimized}} < t_{forceatlas_{original}}$.

8 Future work

As we stand now our work is tested on a single machine, with a single CPU using 8 threads (Section 7). Naturally there is a limit at how much can be processed on such systems and as we are starting a service based on these graph calculations it is given that the system will run on multiple machines utilizing multiple CPUs and much more resources than what was given for the analysis. We have to further explore the possibilities about how much this system can scale on a bigger environment and how much performance boost it can provide using the described techniques.

9 Conclusion

By applying the heuristics described in Section 4 for the Louvain algorithm and by using an optimized version of the ForceAtlas (Section 5) we were able on our biggest test graph about "CT" technology (Subsection 7.1.2) to achieve a $\Delta_t = 6,8$ times overall speedup (Subsection 7.1) compared to the original CPU based implementation. Based on what we see in the detailed analysis (Section 7.2) we can expect the system to scale well on distributed systems when the graph that it's used on does not only have a high volume of edges, but also a larger amount of nodes. In all test cases the community detection enjoyed a significantly higher speedup than the ForceAtlas algorithm, thanks to the introduced heuristics (Section 4): $t_{louvain_{optimized}} << t_{forceatlas_{optimized}}$. Even though the ForceAtlas computation was also more efficient using the parallel solutions (Section 6): $t_{forceatlas_{optimized}} < t_{forceatlas_{original}}$. This leads us to the conclusion, the parallel Louvain modularity (Table 1) can greatly improve the overall performance of the computations thanks to the introduced heuristics and the parallel ForceAtlas (Tables 0, 0) also increases the overall experience of the tools usage. Also, this makes the Collaboration Spotting tool capable of handling very large graphs without compromising the users work.

References

[1] V. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, J. Stat. Mech. Theory Exp. (2008) P10008

[2] Hao Lu, Mahantesh Halappanavar, Ananth Kalyanaraman, Parallel heuristics for scalable community detection, Parallel Computing 47 (2015) 1937

[3] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, Mathieu Bastian, ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software, http://dx.doi.org/10.1371/journal.pone.0098679 (2014)

[4] Collaboration Spotting Visual Analytics Tool, CERN, http://collspotting.web.cern.ch

[5] M.E.J. Newman, M. Girvan, Finding and evaluating community structure in networks, Phys. Rev. E 69 (2) (2004) 026113.

[6] S. Fortunato, Community detection in graphs, Phys. Rep. 486 (35) (2010) 75174, http://dx.doi.org/10.1016/j.physrep.2009.11.002.

[7] V.A. Traag, P. Van Dooren, Y. Nesterov, Narrow scope for resolution-limit-free community detection, Phys. Rev. E 84 (1) (2011) 016114.

[8] D. Bader, J. McCloskey, Modularity and graph algorithms, SIAM AN10 Minisymposium on Analyzing Massive Real-World Graphs (2009) 1216.

[9] J.W. Berry, B. Hendrickson, R.A. LaViolette, C.A. Phillips, Tolerating the community detection resolution limit with edge weighting, Phys. Rev. E 83 (5) (2011) 056119.

[10] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, D. Wagner, On modularity clustering, IEEE Trans. Knowl. Data Eng. 20 (2) (2008) 172188.

[11] B. Hendrickson, T.G. Kolda, Graph partitioning models for parallel computing, Parallel Comput. 26 (12) (2000) 15191534.

[12] Hao Lu, Mahantesh Halappanavar, Ananth Kalyanaraman, Parallel heuristics for scalable community detection, Parallel Computing 47 (2015) 1937

[13] J. Barnes and P. Hut (December 1986). "A hierarchical O(N log N) force-calculation algorithm". Nature 324 (4): 446449.