Graph Partitioning Algorithm for Social Network Model Transformation Frameworks

Gergely Mezei, László Deák, Krisztián Fekete and Tamás Vajk
Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Magyar tudósok krt. 2. Q, Budapest, 1117, Hungary

Keywords: Graph Partitioning, Model Transformation, Cloud Computing, KL Algorithm, Social Networks.

Abstract: Dealing with extra-large models in software modeling is getting more and more common. In these cases, both memory and computational capacity of a single computer might be insufficient. A solution to overcome this barrier is to use cloud computing. However, existing algorithms have to be extended/modified to support cloud computing and use the advantages of its architecture efficiently. We focus on creating an algorithm to partition graphs representing models. Based on the algorithm, models should be able to be mapped onto several computational instances and processed in a distributed fashion efficiently. Previously, we have presented an algorithm that was based on the heuristic Kernighan-Lin partitioning method with two extensions: no limit on the number of partitions and not building on the knowledge of the whole model at beginning (nodes are received and processed one by one). However, when applying social network-based case studies, we have identified weaknesses of the algorithm. This paper elaborates an enhanced algorithm that produces better results for extra-large models. Detailed measurements are also presented in order to show the improvement.

1 INTRODUCTION

Nowadays, software modeling has become a usual pattern in software development. We often meet extra-large models from different segments of the industry. For example refactoring a huge, industrial source code model, processing bio-chemical systems and examination of DNA related processes requires huge storage and computational resources, or processing complex embedded systems to locate errors. Similarly, working with social network models is also not an easy. These networks tend to contain millions and sometime even more than one billion users (Facebook, 2012). Models of this size may require TBs of space. Supporting these models and offer an efficient solution to process them is a challenging task. We are not always able to use the usual, comfortable technique of loading the whole model into the memory of a computer. One solution can be to partition our model and to use a network of computers to store and process the model parts. It is not easy to achieve this though.

Our modeling approach focuses on domain-specific models (Fowler, 2010), where models are often processed and transformed into another models or artifacts. For example, you can specify a pattern, when suggesting groups based on your interest, friends and previous posts. Finding an appropriate suggestion and updating the model can be applied by model transformation. However, current model transformation approaches does not support or at least not optimized for distributed environments. Most of the approaches use models stored completely in the memory. The reason for this is that processing models directly from hard drives is 3-4 orders of magnitude slower. However, as mentioned before, some of the domains produce extra-large models, which do not fit into the memory of one computer. Partitioning the models and applying transformation on these partitions is a viable solution. We are working to make this mid-term goal possible.

By dividing models into partitions, we are not limited by architectural limits anymore; we can handle models of arbitrary size (supposing that we have as many computers as needed in our network). The emerging world of could computing is a natural selection. We can easily extend our computational capacity whenever needed and we do not need to invest into personal super-computers. Instead, we only pay for the resources we really use. The most

Mezei G., Deák L., Fekete K. and Vajk T..
Graph Partitioning Algorithm for Social Network Model Transformation Frameworks.
DOI: 10.5220/0004475104800487
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-PT-2013), pages 480-487
ISBN: 978-989-8565-68-6

widespread cloud computing platforms include Microsoft Windows Azure (Microsoft, 2012), Amazon AWS (Amazon, 2012) and Google App Engine (Google, 2012). Although the cloud computing architecture is a great aid in realizing our vision on partitioned modeling environment, it does not solve all the difficulties at once: The efficiency of using partitioned models is heavily affected by the communication overhead between the computers storing the partitions. Therefore, our goal is to minimize the communication between the computers.

In our approach, models are represented by graphs. Partitioning the model means partitioning this graph. We would like to search and transform the models, thus we need to find graph patterns and replace them. In order to minimize communication between computer instances, we have to minimize the number of navigation steps between the partitions. Therefore, we need to minimize the weight of edges that connect nodes in different partitions. More precisely, we have a mid-term and a short-term goal. The mid-term goal is to support minimize inter-partition edges (their weight) for all kinds of models, while the short-term goal is to be optimal in case of social network models. First we have created a general algorithm and then started to optimize it to social networks. This paper presents both the original algorithm and its optimization. The algorithm does not depend on a concrete cloud computing environment; it can fit each of them. Note that although the motivation behind the enhancements is based on the field of social networks, they improve efficiency in general as well.

The rest of the paper is organized as follows: in Section 2 the background work is introduced. In Section 3 the algorithm is presented in detail. Section 4 evaluates the results of the algorithm and reveals the advantages of the improved algorithm. Finally, some conclusions are drawn together with a brief account of the future directions in Section 5.

2 BACKGROUND

The problem of graph partitioning has been present for over 40 year. Although the problem is NP-complete, several fast and heuristic algorithms exist.

B.W. Kernighan and S.Lin have worked out an efficient heuristic procedure (the KL algorithm) for partitioning graphs. In their paper (Kernighan and Lin, 1970) the KL algorithm is discussed in detail. The basic idea of the work is to count the difference between the external and internal sum of edge weight for each $a \in A$ nodes, where A is a subset of nodes. Their work grounds the base for many graph partitioning algorithms from the automotive industry to medicine.

To improve the efficiency of the KL algorithm C.M. Fidducia and R.M. Mattheyses introduce a new data structure in (Fiduccia, 1982). They used the KL algorithm to improve Computer Network Partitions. Bucket list structure has been used to store and maintain the gain G_a for each node.

These algorithms are not applicable directly in our scenario, since the number of partitions is predefined, and do not map on streaming input model.

Konrad Voigt has provided a brief conclusion for each available algorithm for a really similar problem to the one introduced in this paper. His dissertation (Voigt, 2011) provides an algorithm for partitioning planar graphs. In his work, he states that he could not use the spectral bisection because it does not support explicitly the variable number of clusters.

George Karypis and Vipin Kumar introduced the Metis framework (Karypis, 1998). Their paper divides graph partitioning into three phases: coarsening, partitioning and uncoarsening. The phases are explained later in this paper as well. For each phase, they describe different type of algorithms that can be used. The Random, HEM, LEM and HCM algorithms for coarsening; bisection, KL, GGP, GGGP for partitioning; and KL refinement, Boundary KL refinement for uncoarsening. Their work also concludes the efficiency of these methods.

Burkhard Monien (Monien, 1999) and his team elaborate the latest type of coarsening algorithms and their efficiency on different kind of sample graphs.

Finally, in 2011 Xin Sui, Donald Nguyen, Martin Burtscher, and Keshav Pingali presented parallel graph partitioning methods for shared memory – multicore systems (Sui et al., 2011).

To run model transformation on the model of social networks we have done basic research in Social Network Sites (SNS). Several papers conclude different aspects of Social Networks, from the definition to methods and applications. The paper (Boyd, 2006) of Danah M. Boyd and Nicole B. Ellison conducted a short history, features and future of social networks. The work of Nathan Eagle, Alex Pentland and David Lazer (Nathan Eagle, 2009) introduces the different ways to collect data for social networks. They use the traditional, predominant self-reporting and new automatized mobile phone based data collection. Their article compares the results and does initial steps to merge

these data. Their paper also conducts different social networks.

Graph partitioning in general has a huge mathematical background. However, the area of, the social networks and cloud environment define special constraints, like the communication overhead between computational instances or graph structure characteristics for social networks. In this paper, we present an approach applicable in all these areas.

3 MODEL PARTITIONING

Models can easily be transformed into graphs, each entity in the model is transformed to a vertex in the graph and each relation in the model, is transformed into an edge. These graphs may have vertices and edges reflecting the types and attributes of model items but at the current stage, we ignore this information to be able to handle all domains and all models the same way.

Industrial sized models are difficult to handle without partitioning them. However, partitioning is not easy, since models cannot be decomposed into independent components. The main challenge is to create partitions with minimal number of edges between the partitions. We have created a solution to this issue in the form of a fast, heuristic graph partitioning algorithm.

As mentioned earlier, we have a short-term goal (partitioning social network model graphs) and a mid-term goal (partitioning all kinds of graphs). We have created a general algorithm and fine tuned it later for social network-like models. In this paper firstly we present our original algorithm and then we elaborate the enhancements applied on it.

Because the extraordinary size of the models, our algorithms needs to fulfill special constraints: Existing graph partitioning approaches (e.g. the KL algorithm) usually suppose that the whole graph is known, when partitions are created. In our case, this presupposition is not fulfilled, since the model may not fit into the memory. We have decided to create an algorithm that does not build on knowledge of the whole model, but can handle receiving model parts (nodes) one by one in a stream. This way, the size of manageable models is not limited.

Our partitioning method consists of three phases:

1. Growing partitions
2. Separating partition into sub-partitions
3. Refining partitions

The skeleton of the algorithm is the following:

```
 1.NewNode([Node])
 2.  [partition]= Select partition with
     the highest sum of edges weight
     from [Node]
 3.  add [Node] to [partition]
 4.  if([partition] has too many Nodes)
 5.    Separate([partition])
 6.  if([Node] is the last)
 7.    Coarsen([partitions])
 8.      for each partition pair
 9.        KL(partition pair)
10.    UnCoarsen([partitions])
```

Figure 1: Skeleton of the algorithm.

3.1 Growing Partitions

In the first phase (*growing partitions)*, the nodes are received from the stream and inserted into a partition (Figure 1– Line 2 and 3).

Selection of the partition to insert the node into is not random. To determine the chosen partition for a give node a, we compute the sum of weight for edges from a to each partition P:

$$w_P(a) = \sum_{\forall b \epsilon P} w(a,b) \quad (1)$$

where $w(a,b)$ is the weight between nodes a and b. The chosen partition is the one with the maximum sum of edge weights $\max\{w_P\}$.

After the insertion, if the number of nodes in partition P is more than a predefined constant value, we divide the partition into two sub-partitions. As the result, the partitions cannot grow beyond the memory of one instance. The details of this separation are discussed in the second phase.

3.2 Separation

The second phase is called *Separation* (Figure 2). Firstly, two nodes are chosen from the partition. These two nodes considered to be the centers of the new partitions from now on. The nodes can be incident nodes; however the Graph Growing Partitioning (GGP) is sensitive to the choice of the vertexes from which we start to grow the partitions. The basic idea here is to select two nodes with the highest weight of edges. Starting from these points, we use a modified GGP algorithm (Karypis, 1998). We begin to grow regions around the two chosen points in a breath-first fashion until all vertices have been included (Figure 3).

```
1.Separate([partition])
2. Choose [node1] and [node2] with the
   highest Edge weight
3. Grow regions around [node1] and
```

```
   [node2] in breath-first fashion to
   create initial [partition1] and
   [partition2]
4. KL([partition1],[partition2]);
5. Merge([partition1]);
6. Merge([partition2]);
```

Figure 2: Separating the nodes into sub-partitions.

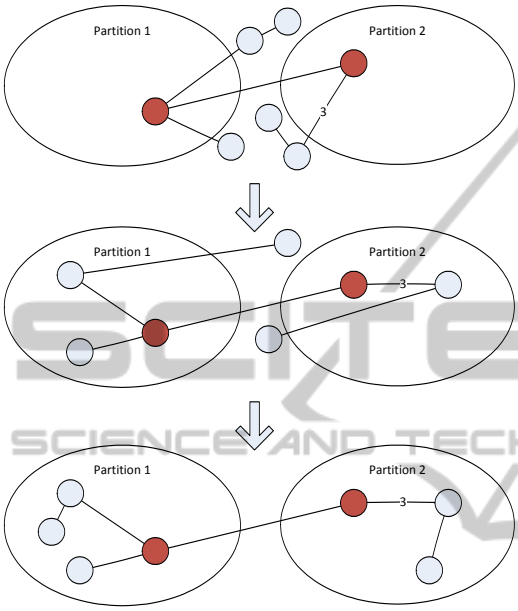

Figure 3: The GGP algorithm.

When the regions are full sized (all nodes are included in one of the partitions), the partitions are further refined by a modified KL algorithm.

In the KL part of the second phase (Figure 4), the basic idea is to compute the distance

$$D_a = \forall b,d \sum_{\substack{a\in P_1 \\ b\in P_2}} w_{ab} - \sum_{\substack{a\in P_1 \\ d\in P_1}} w_{ad} \qquad (2)$$

where a, b, d are nodes, w_{ab} is the cost of edge between nodes (a, b) and P_1, P_2 are the partitions of the nodes.

Note that the nodes are not ordered by their D value in their partitions. However, we select the two unprocessed nodes with the highest D value to exchange in both partitions. Ordering the partitions may seem to be more optimal, but finding the exchangeable nodes are faster than reordering the neighbors in each step.

After exchanging, the D value of the unprocessed incident nodes has to be updated. Notice that the size of partitions does not change, but the number of unprocessed nodes continuously decrease. We define the gain in each iteration: $G = D_a + D_b -$ w_{ab} the sum of the D values for nodes a and b minus the weight of edge(s) between. We repeat this step until the value of gain is a positive number.

```
1.KL([partition1], [partition2])
2.  foreach [node] in the partitions
3.    compute [D] for [node]
4.  do
5.    choose unprocessed [node1] and
      [node2] from partitions where
      [gain]=[node1].[D]+[node2].[D]-
      [node1_node2_edge] is highest
6.    exchange [node1] and [node2]
7.    update [D] values of the
      incident nodes of [node1] and
      [node2]
8.  while([gain]> 0)
```

Figure 4: Modified KL algorithm.

3.3 Refining Partitions

In the *refining partitions* phase (Figure 1 – Line 7 – 10), the whole model is already processed, and the nodes are partitioned. The purpose of the third phase is to further refine the existing partitions. The phase consists of three steps: coarsening, refinement, uncoarsening. In the coarsening step, a series of simpler graphs with fewer and fewer vertices are created. Graph coarsening can be done with several algorithms. Suppose that the graph to be coarsened is G_0, in each step of the series we produce a new G_{i+1} from G_i where $|V_i| > |V_{i+1}|$.

```
1.Coarsen( [partitions] )
2.  for each [partition]
3.    do
4.      for random order of each [node]
        in [partition]
5.        select a connected unmatched
          [node1] in [partition] with
          the highest edge weight
6.        merge [node] and [node1]
7.    while the sum of nodes in
      [partition] is below [X]
```

Figure 5: HEM coarsening.

In each iteration, nodes are merged into a *multinode*. Multinodes are nodes, which weight is the sum of the original nodes' weight and the edges are the union of the original edges except the edges connecting the nodes in the multinode. In the case when both nodes have edges to a vertex *v*, the weight of the edge from the multinode to vertex *v* is the sum of the original weights of these edges.

An edge-cut of the partition in a coarser graph will be equal to the edge-cut of the same partition in the finer graph. A *matching* of a graph is a set of

edges which does not have any endpoint in common. The coarser graph is constructed by finding a matching and merging the nodes into multinodes, while unmatched vertices remain in the coarsened graph as well. A matching is *maximal* if any edge in the graph that is not in the matching has at least one of its endpoints matched. A matching can be *maximum* when it is a maximal matching and it has the maximum number of edges. For computational reasons, in practice, the maximal matching is preferred instead of the maximum matching.

To find a maximal matching we have used the *Heavy edge matching* (HEM) algorithm, which is a greedy algorithm for the number of coarsening iterations (Figure 5). It can be shown that the total edge weight of the graph is reduced by the total weight of the matching. Although, HEM is based on a randomized algorithm, thus it does not guarantee to find the maximum matching, it gives a good approximation. The vertices are visited in a random order, and for each vertex u an unmatched vertex v is chosen, so that edge (u,v) has the highest weight from the all possible v vertices. The computational complexity of HEM is $O(|E|)$ (Burkhard Monien, 1999).

In the coarsened graph, the KL algorithm is used to improve the efficiency. Finally, in the uncoarsening phase, the coarsened graph is projected back to the original graph. For each vertex, the coarsened graph contains a subset of vertices. These vertices are assigned back to the partition of the original vertices during the projection. During the projection it is still possible to improve the partitions by running KL refinement or Boundary KL refinement algorithms.

Although this heuristic algorithm does not find a global optimum, it finds a local optimum for graph partitioning in a really effective way.

Concluding, our algorithm consists of three major phases. In the first phase, we grow partitions, in case of large partitions we separated these partitions into sub-partitions in the second phase. The second phase is based on the Kernighan-Lin method. Finally, in the last phase we refine these partitions using coarsening and uncoarsening techniques. Now, we present our enhancements on the algorithm.

3.4 Order of Nodes in GGP

During separation, our goal is to create two new partitions having fairly the same amount of nodes. Two regions are grown in a breath-first fashion around the two center nodes. A list of nodes is used to store nodes for processing in *waiting_for_process*. When a new node is processed, it is popped from the list, and the node's unprocessed incidents are added to this list if they are not already there. However, this method has clearly a huge drawback. Let's take the following graph on Figure 6. The two center nodes are a and b. Now if we first process the incident nodes of node a, none of the nodes will be added to the partition of b node, because they are already added to the *waiting_for_process* list with a's partition. The separation is not balanced. Our solution for this problem is to add the nodes to the waiting list in an alternating manner, especially for the incident nodes' of two center nodes. This means, that when a and b's incident nodes are processed, they are either added to the list with a node's partition or b node's partition alternatingly. This will result two partitions, one with 3 nodes and one with 2 nodes. The result is fairly balanced.

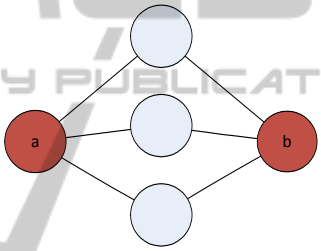

Figure 6: Problem of separation.

3.5 Merging

Merging is a new phase inserted into the original algorithm. The phase solves the defragmentation problem occurred during partitioning. Merging reduces the number of partitions, which results less edges between partitions. Merging partitions can only result less edges between partitions, as new nodes or edges are not created, and if nodes were originally in separate partitions which are merged, edges will no longer running between partitions. Edges to other partitions remain, but their number will not increase. Also note that reducing the number of partitions is not only beneficial because the number of edges is reduced. The secondary advantage is caused by the fact that we need fewer instances in cloud environment, which also results lower bills. (Supposing that partitions are ran on different instances.)

The merging phase can take place either after the whole algorithm, which influences only the final result, or during the algorithm, after each separation phase. The second solution influences not only the final result, but the next step of the algorithm as well.

Therefore, we have chosen the second option as it provides a better solution. This way some partitions can be already corrected at the separation time which could not be corrected at the end of the algorithm. Merging is applied for the two new separated partitions, one after the other. Merging (Figure 7) itself collects the incident partitions and chooses a partition where the number of nodes is still less than the maximum node per partition after the merging, but would create the largest partition. After this, it moves the nodes from one partition to the other partition and deletes the empty partition.

```
1.Merge( [partition1] )
2. Collect all incident partitions
   into [incident]
3. Choose [partition2] where nodes in
   [partition1] + [partition2] < max
   nodes in partitions, and
   [partition2] is the largest
4. Move nodes in [partition2] to
   [partition1]
5. Delete [partition2]
```

Figure 7: Merging phase.

4 EVALUATION

In this section of the paper, the performance results of our implementation are presented. The algorithm has been tested on general graphs and on social network like graphs as well.

A customizable version of the presented algorithm has been implemented. This version emulates the behavior of several computer instances, but runs in local development environment, which emulates the cloud architecture. Although, the algorithm can be run in a highly parallel manner, we have not implemented it in all possible cases yet, as the algorithm is still being developed and optimized. We have tried to keep its performance closer to the real multiple instance scenario, where communicational overhead is a huge constraint.

To implement the algorithm, we used the C# language and .NET Framework. There are two deviations in the implementation from the pseudo codes shown earlier. One is in the third *refining partitions* phase: for all partitions pairs a KL is run (Figure 1 – Line 9). In our case study implementation, a few random pairs are chosen for the KL algorithm, but we do not apply it on all possible combinations. The other deviation is in the implementation of KL algorithm: we have an external loop, thus the algorithm can be repeated several times sequentially. These repetitions can further refine the partitions.

The source model and the algorithm are driven by a set of parameters: (i) Number of nodes; (ii) Maximum nodes in one partition; (iii) Average degree of nodes; (iv) Average weight of edges; (v) The number of KL repetitions.

In the first scenario, we run the algorithm on a general graph. The model consists of 10000 nodes and 6 edges for each node in average. The weight for each edge changes from 1 to 10 in a random manner. The maximum number of nodes in a partition is varied from 100 to 1000. The KL repetitions highly influence the running time of the algorithm. We used values between 10 and 50.

For reference, a random solution is also implemented, where the nodes are randomly put into one of the partitions. If a partition is larger than the limit, it is separated in a random manner: we create two new partitions, and we move each node into one of the new partitions by a random choice. The random solution is initialized with a single partition. Note, that comparing our results to a random solution may seem meaningless, however there is no other algorithm which would support streaming input model and variable number of partitions. Applying the KL algorithm on our models would be possible if the required extra information is manually set, but it would lead to false results, since KL would build on an extra advantage due to the extra information.

To compare the results, we computed the sum of the edge weight between the partitions. We have observed that random solution provides a good result, in the case of high connectivity and low number of partitions. Note that in these cases, there is no "near-optimal" solution.

Our research has shown that the base algorithm works much better than the random solution under normal conditions. We have tried several different random graphs and the advantage of our algorithm remained approximately the same. Results have clearly shown, that the algorithm is 50% better than the random solution in average, based on the sum of edge weights between partitions.

We have also observed that if the number of edges is higher, the efficiency of our algorithm is reduced, while larger edge weight improves it.

However, currently we intend to run model transformations on the models of social networks primarily. Modeling the connections between the people and other social entities realistically is difficult. It is necessary for us to find a good sample graph that models social networks. This graph must be simple, easy to reproduce, but it also should hold

one or more characteristics of social networks.

We have prepared several smaller graphs to simulate the structure of social networks. These graphs represent different aspects of a social network in an abstract way. We have identified a few primitive patterns and built up complex structures from them. In this paper, we use the model of a star, and the model of loosely connected stars for sake of simplicity. We believe the graph of a connected starts is a good initial approximation for these topic. Figure 8 represents a linked star graph.

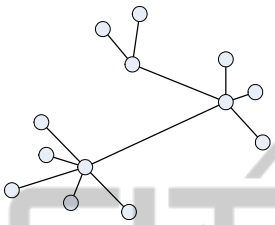

Figure 8: Sample linked star graph.

Our measures have shown that the structure of partitions and number of partitions are dependent on the order of the nodes received in the first (growing partitions) phase. Different ordering results in different number of partitions and inter-partition edges. This was the main reason why we added the merging phase to our algorithm.

The initial problem is the following: let us take a graph with two stars where the center nodes are connected. The size of both stars shall not be greater, but equal to the size of a single partition.

In the first test (Figure 9, solid lines), the nodes of the first star are received and the second star only after that (starting with the center). The first partition is filled by the algorithm and then the center of the second star is added. Since the partition size is exceeded, the partition is separated. The centers of the GGP algorithm will be the center of the first star as it has the highest degree and the center of the second star, because its degree is equal to the other nodes' degree but it is the last received node. Note that at this point we have no information about the final degree of the center of the second star. The result of the separation phase is two partitions, one with the center of the first star and the other nodes in the first star, and one with the center of the second star. As the algorithm continues, the rest of the second star is added to the new partition with the center of the second star. In this case, our algorithm works well, the result concurs to the expected.

In the second test (Figure 9, dotted lines), node *e* of the first star is received only after the second center was added. The beginning of the algorithm results the same as in the previous case. However, as we exceed the limit of a single partition, we choose different nodes for the centers of the GGP algorithm. The first center of GGP will be the center of the first star, because it has the highest degree. The second center of GGP will be node *e* as it has the same degree as the other nodes but it is the last node. The results of the separation phase are two new partitions: all nodes except *e*, and node *e*. As the algorithm follows, and we get a new node from the second star, another separation phase takes place. This is necessary, since the first partition is oversized, again. As the result, there will be three partitions, one with the first star, except node *e*, one with the second star and one with node *e* (Figure 9). Unfortunately, this is not the result we expect, it contains three partitions, one with only one node. The number of edges between the partitions is also higher than in the previous case. The difference between the two cases is the order of the nodes.

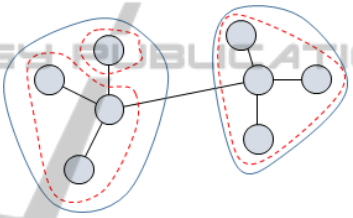

Figure 9: Different separation results.

Remark, that in the second test, partition of node *e* could be merged with the partition of the first star. Note that the problem is not really specific to linked stars, or social networks, it is a general issue. The original algorithm was able to create partitions, but it was unable to *merge* partitions. Since we may create extra partitions (we do not know all the information (the whole model) during partitioning), we should be able to merge the partitions later, as our knowledge evolves. As we see more-and-more from the graph we can always apply merging again.

We have run tests on the original algorithm and with merging enabled. Our algorithm has resulted approximately one order of magnitude less edge between partitions than the random solution if each star has the same amount of nodes. The difference is even more remarkable (45 vs. 2919), if the size of the star changes. The number of partitions has also decreased by 25% compared to the merge-less algorithm. The generation of nodes for the stars does not influence the final results, all methods has the same outcome. The number of edges between partitions scales with the number of nodes overall. Results are concluded in (Gergely Mezei, 2013).

5 CONCLUSIONS AND FUTURE WORK

In model-based software engineering, new methods have to be applied because the size of the models can grow beyond the capacity of a common, single computer. For extra-large models such as social networks, several instances of computers can be used and transformations can be applied efficiently in a distributed fashion. Cloud services provide a whole new perspective for the multi-instance infrastructure. Their payment model also allows us to pay only after the resources we really use. In order to process models in the cloud, they have to be partitioned and existing modeling approaches should be modified. The goal of partitioning is to have the least amount of edges between different computers, thus network communication can be reduced.

We have provided an improved heuristic approach to partition a social network like graphs into subsets of nodes. The algorithm is based on the Kernighan-Lin method and graph coarsening. With this algorithm, a social network like graph can be partitioned into unconstrained number of partitions, where the maximum size of partitions is constrained.

In the near future, we aim to further refine the algorithm by considering further characteristics of social networks. We need more case studies for this, thus we plan to use a real world, Facebook-based case study with real data as the input model with a small world assumption.

Since structure of social networks tend to change rapidly, we have to find a way to adopt to these changes. A promising solution would be to transforming both models and transformation logic via model transformations.

We plan to evaluate time performance measurements for each phase of the algorithm. As well as, we plan to implement a several magnitudes bigger scenario, which runs on real network connected computer instances. In the further future, we will improve and extend the existing distributed model transformation methods.

ACKNOWLEDGEMENTS

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

This work has also been supported by the project "Talent care and cultivation in the scientific workshops of BME" financed by the grant TÁMOP - 4.2.2.B-10/1–2010-0009.

REFERENCES

Amazon, 2012. *Amazon AWS.* [Online] Available at: http://aws.amazon.com/ (Accessed 10 10 2012).

Burkhard Monien, R. P. R. D., 1999. *Quality Matching and Local Improvement for Multilevel Graph Partitioning.* s.l., s.n.

Danah M. Boyd, N. B. E., 2006. *Social Network Sites: Definition, History, and Scholarship.* s.l., s.n.

Facebook, 2012. *Facebook. (*Online) Available at: http://www.facebook.com (Accessed 17 10 2012).

Fiduccia, C. M. a. M. R. M., 1982. *A linear-time heuristic for improving network partitions.* Piscataway, NJ, USA, s.n.

Fowler, M., 2010. *Domain Specific Languages.* s.l.:Prentice Hall.

Gergely Mezei, L. D. K. F. T. V., 2013. *Results of the case study.* (Online) Available at: http://avalon.aut.bme.hu/~mesztam/vmts/icsoft_results.docx (Accessed 22 04 2013).

Google, 2012. *Google App Engine.* [Online] Available at: https://cloud.google.com/index (Accessed 10 10 2012).

Karypis, G. a. K. V., 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.,* pp. 359-392.

Kernighan, B. W. & Lin, S., 1970. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal,* pp. 291-307.

Microsoft, 2012. *Windows Azure.* (Online) Available at: https://www.windowsazure.com/en-us/ (Accessed 10 10 2012).

Nathan Eagle, A. (. P. a. D. L., 2009. *Inferring friendship network structure by using mobile phone data.* s.l., s.n.

Sui, X., Nguyen, D., Burtscher, M. & Pingali, K., 2011. *Parallel graph partitioning on multicore architectures.* Houston, TX, Springer-Verlag, pp. 246-260.

Voigt, K., 2011. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* Dresden, Germany: Technische Universität Dresden.