THANOS: HIGH-PERFORMANCE CPU-GPU BASED BALANCED GRAPH PARTITIONING USING CROSS-DECOMPOSITION

BY

DAE HEE KIM

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

As graphs become larger and more complex, it is becoming nearly impossible to process them without graph partitioning. Graph partitioning creates many subgraphs which can be processed in parallel thus delivering high-speed computation results. However, graph partitioning is a difficult task. In this work, we introduce Thanos, a fast graph partitioning tool which uses the cross-decomposition algorithm that iteratively partitions a graph. It also produces balanced loads of partitions. The algorithm is well suited for parallel GPU programming which leads to fast and high-quality graph partitioning solutions. Experimental results show that we have achieved a 30x speedup and 35% better edge cut reduction compared to the CPU version of METIS on average.

To my family, for their love and support.

ACKNOWLEDGMENTS

I give the most gratitude to my advisor, Professor Deming Chen, from the University of Illinois at Urbana-Champaign, for his guidance, inspiration, kindness and trust. I appreciate him for inviting me to join his research group and for the faith he bestowed on me in all aspects of the research process. I also thank Professor Rakesh Nagi and my colleagues, Zaid Qureshi and Vikram Sharma Mailthody, for their help on this work.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

As data gathered through Internet-of-Things are becoming larger or the number of transistors on a circuit continues to grow higher, the graph that represents the complex data connections for these, namely social network, logic gate netlist, or cell placement graph, is also increasing in complexity and size. Naively processing such enormous graphs on a CPU is practically impossible as it will either take too long to finish or run out of memory due to the shear size of the data. Oftentimes, it is much better to process such a graph with multiple or many sub-graphs. When a graph is partitioned, we want to put as many connections as possible inside of each partition and minimize the connections among partitions, in other words, minimizing edge cuts. In this thesis, we focus on minimizing edge cuts as well as the partitioning time. We also maintain perfect load balancing for each partition. Partitioning a graph into equal sizes while minimizing the edges among different partitions is an important task, finding various useful applications including scalable logic synthesis and physical design. Meanwhile, partitioning also helps graph processing itself through parallel computing. Parallelizing many applications involves the problem of assigning data or processes evenly to processors, while minimizing the communication among the processors [1, 2]. The partitioning problem is known to be NP-complete [3, 4]. Since graphs are getting complex in various way, it is difficult to establish a standard approximation algorithm in general [5] and heuristic algorithms are typically used.

In this work, we introduce Thanos, a fast graph partitioning tool that uses the cross-decomposition algorithm [6]. The algorithm was not designed for the graph partitioning problem, but for a job scheduling problem in the industrial engineering field. However, we realized that the characteristics of the algorithm can help

solve the graph partitioning problem. The algorithm is also well suited for parallel GPU programming which leads to fast and high-quality graph partitioning solutions. Compared to CPU which has only a few cores, GPU generally has thousands of cores that can execute computations in parallel. Many well-known libraries such as Tensorflow [7], are developed targeting GPUs. The CUDA platform [8] that runs on any NVIDIA GPUs has made GPU programming easy to use. In our work, we are able to achieve 30x speed up and 35% better edge cut reduction among partitions compared to the CPU version of the famous graph partitioner METIS [9]. The main contributions of this work are as follows.

- Optimized cross-decomposition algorithm to fit into large-scale graph partitioning problem
- Implemented and optimized the algorithm on GPU
- Provide perfect load balance with high-quality graph partition

We organize the thesis as follows. Chapter 2 discusses all the background knowledge required for this work including the cross-decomposition algorithm. Chapter 3 explains how optimization for the algorithm is accomplished and how it is implemented on a GPU. Chapter 4 shows the result and analysis of this work. Chapter 5 concludes the thesis.

CHAPTER 2

BACKGROUND

2.1 Cross-Decomposition

In this section, we will cover the algorithm of cross-decomposition proposed in [6, 10]. The main idea of cross-decomposition on graph partitioning is to compute each vertex's cost for each different partition and assign it to the partition based on the cost in such a way that meets two conditions [11]:
(1) Partitions contain as many non-zero values (connections) as possible.
(2) One finds as many zero values (no connection) as possible outside the partitions.

Given a graph with total number of vertices, N, we first build an adjacency matrix A whose size is $N * N$ and all the elements belong to [0,1], $A = [a_{i,j}]$ with $i = 0, ..., N-1$; $j = 0, ..., N-1$ and $0 \leq a_{i,j} \leq 1$. Next, we need to build two types of partitions, row partition, P_X^v and, column partition, P_Y^v where v denotes the total number of partitions. Each vertex is assigned to one of the partitions between 0 to $v-1$ for both types of row and column partition using uniform random distribution. This initial random assignment is denoted as $P_X^v(0)$. Then from the initial partition, we repeat the following two phases until there is no change from the previous partition [11]:
(1) Build a new partition on $Y : P_Y^v(K)$ using $P_X^v(K-1)$.
(2) Build a new partition on $X : P_X^v(K)$ using $P_Y^v(K)$.

Figure 2.1 visualizes this process. Only the row partition is shown as the column partition uses exactly the same process.

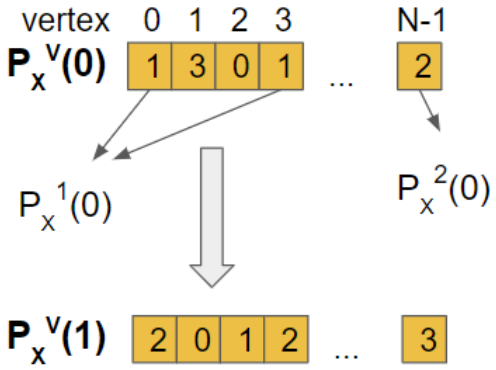


Figure 2.1: Row Partition

For phase 1, we are basically implementing the following equation. For $j = 0, ..., N-1$ and $r = 0, ..., v-1$ do the following equation:

$$y(j,r) = \beta_j \cdot (h \cdot \sum_{i \in X_r(k-1)} \alpha_i \cdot a_{i,j}^{\lambda} + (1-h) \cdot \sum_{i \notin X_r(k-1)} \alpha_i \cdot (1 - a_{i,j})^{1/\lambda}) \quad (2.1)$$

Then search r^* such that

$$y(j,r^*) = \max_{0 \le r \le v-1} (y(j,r))$$

and assign j to the class $Y_{r^*}(k)$. The values for α_i, β_j, h and λ are adjusted by the user. Here, α_i and β_j assign weights to connected vertices and not connected vertices respectively. If h increases, the connected vertices inside the same partition become more important than those outside of partition. On the other hand, if λ decreases, the small values which are outside the partition as well as the large values inside the partitions make a larger contribution in the equation. After phase 1, we perform phase 2. Phase 2 is not shown as it is exactly the same as phase 1 but with a different partition, $P_Y^v(K)$, and α_i, β_j switched.

It was proved in [10] that either $P_X^v(k), P_Y^v(k)$ yields a greater value than $P_X^v(k-1), P_Y^v(k-1)$, or both yield the same value. Since it can lead to a local optima,

we use it with several initial P_X^v and we keep the best obtained solution. This proof verifies that the cross-decomposition algorithm converges [11]. In our work, we repeat this process three times since all the graph data sets we ran empirically show convergence within three iterations. Summarizing the algorithm in simple words, we are assigning each vertex to the best partition based on number of connections it has in different partitions. Therefore, we selected this algorithm for graph partitioning as it can cluster the vertices that are close together into the same partition based on the equation. The algorithm has $O(N^2)$ complexity which will be optimized in Chapter 3.

2.2 Graph Representation

In this work, we used real-world graph data sets that are given from Graph Challenge [12]. All the graphs are given as an edge list where each line of the list shows the source and destination of the edge. If the graph is large and sparse, it would be very memory inefficient or even impossible to store as an adjacency matrix. Therefore, after reading this edge list, we decided to use the Coordinate (COO) format and one more array, row pointer, which is found in the Compressed Row Storage (CSR) format. COO format will require a total number of edge spaces and the row pointer will require a total number of vertex spaces.

CHAPTER 3

IMPLEMENTATION

3.1 Optimization and Basic Implementation

As mentioned in Section 2.2, we store a graph using the COO format and a row pointer. As shown in Figure 3.1, each thread uses its thread ID as the index for the row pointer. Then, based on the content of the row pointer, threads can access the start point of their neighbor list that is stored in the COO format. Each thread can get its number of neighbors by subtracting the content of its row pointer from content of its row pointer plus one since the row pointer contains the prefix sum of neighbors of all the vertices. Next, when a thread is checking its neighbors, it uses its neighbors' number as the index to access partition array.

Equation (2.1), consists of two parts. The first part is where h is multiplied and the second part is where $(1 - h)$ is multiplied. In words, the first part basically checks all the neighbors of j which are in partition r. The second part checks all the non-neighbors of j which are NOT in partition r.

The first part can be calculated quickly as we can just traverse the neighbor list which can be N in the worst case, but in practice, it is relatively small. However, the second part cannot be done quickly as we have to traverse all the non-neighbors and check whether they are in partition r or not.

Analyzing the problem, we realized that the cross-decomposition algorithm which is designed for any general decomposing problems, is used specifically for graphs, and more specifically for unweighted graphs. An unweighted graph has a value of 1 for every connection in an adjacency matrix. Taking advantage of this fact, we found a

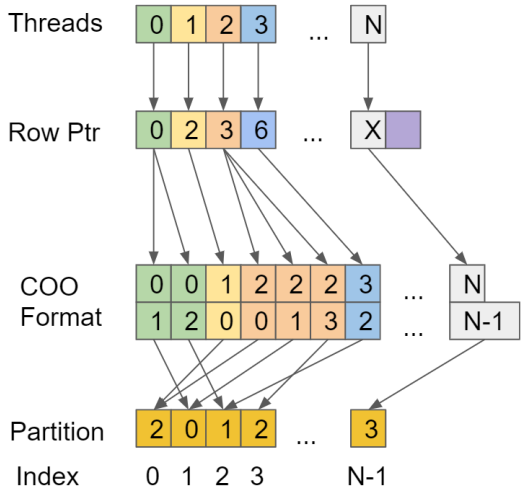


Figure 3.1: Work of Each Kernel

novel way to mathematically calculate the second part of the equation, rather than traversing the entire adjacency matrix to see if there is connection or not. If it was a weighted graph, we need to check the weight value for every connection that exists outside of current partition which will require traversing entire non-neighbors.

To explain the calculation, we use Figure 3.2 which shows a graph with 29 vertices. Each circle is a vertex. Now assume we are working on vertex 0 (white circle) and red circles indicate vertex 0's neighbor and also in the current partition that we are checking, P0. Then the green circles indicate the neighbors that are outside of current partition. If vertex 0 starts to check for partition 1, then the red circles in partition 0 will be changed to green and the two green circles in partition 1 will be changed to red. First, we count how many circles are in each partition. That is the cardinality array in Algorithm 2. Second, we count the number of red circles in the current partition which is the first part of the equation. That is the 'connected_and_in_curpart' variable in Algorithm 2. Next, we can now simply calculate the second part of the equation by performing the total number of vertices, N, minus the current partition's cardinality minus 'connected_and_in_cur_part' plus the number of neighbors of the current vertex, 'Degree'. This calculation will immediately give the non-neighbors of j which are NOT in partition r.

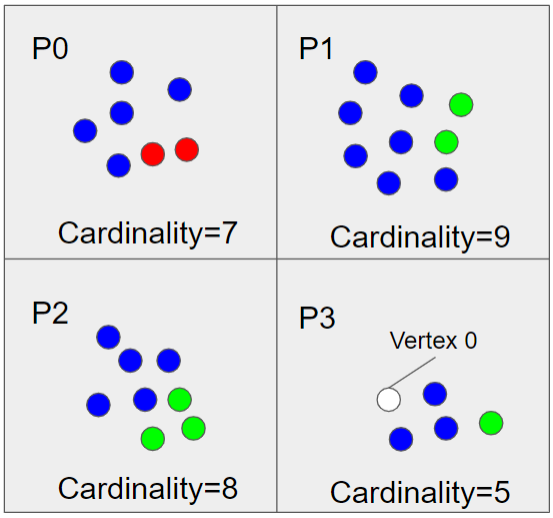


Figure 3.2: Partition Example

This approach now has the runtime complexity of $O(ND)$ where D is the degree, number of neighbors, for each vertex. The degree can be N in the worst case leading to $O(N^2)$, but it will less likely happen in a large real graph. Let us have a graph with N number of vertices and K many partitions.

Algorithms 1 and 2 are given to show implementation steps. We first initialize all the data structures we need as shown in Algorithm 1 where 'rP,cP' stands for row partition and column partition respectively. Since phase 2 is basically the same operation with different partitions and α, β values, only phase 1 will be shown. Then we launch a number of vertices many of threads to perform a computation for the

Algorithm 1: Pseudo Code for Initialization

```
Generate COO representation of graph(RowInd, ColInd, RowPtr);
Build Partition arrays, rP and cP, with size of N. rP[N] & cP[N];
Initialize rP & cP with uniformly distributed random number < partition
  size(K);
Generate cardinality arrays with size K, rC[K], cC[K];
Count the number of vertices in k partition from partition arrays and assign
  the count to r, cC[K];
```

cross-decomposition. Therefore, each thread is responsible for one vertex. Figure 3.1 shows how each thread is taking its portion, the neighbors, from the COO representation of the graph. We can see that since we have a number of vertices many workers to compute in parallel, each thread's ID is matched with each vertex's ID. This enables us to use thread ID as the index to access COO arrays. Algorithm 2 shows the pseudo code for each kernel which performs the calculation explained above.

3.2 Load Balancing

Note that compared to how original cross-decomposition works, it no longer simply updates the partition based on maximum cost. Instead, we set a capacity for each partition, and if one partition is already full, it goes to other partition based on cost. By setting the capacity for each partition, we can prevent one partition from getting very large causing load imbalance among partitions. Since the index of cost array is used as the partition number, we need the sorted index too. To achieve this, we create an index array and sort it along with the cost array. This operation is basically the same as the 'arg sort' operation in Python. If the first partition is full, it checks the next partition based on the order of index array. Note that this process is done atomically. Unless it does not perform an atomic operation, we will still see load imbalance among partitions due to race condition. A race condition happens when all threads are trying to read and write to the same memory. To force threads to read/write in order, we use an atomic operation. However, even if we perform atomic operations, we can see the output will be non-deterministic. If each partition has a capacity of 100 and 150 threads trying to get into the partition, it depends on which reads the memory first and updates. Since our work is not finding exactly the right partition that each vertex belongs to but rather finding the relatively best partition based on the number of partitions, we decided to use this first-come first-served strategy to achieve speed up.

Algorithm 2: GPU Kernel Code

cur_vertex = thread ID;
create array with size K to count the number of vertices that is connected to current vertex and in current partition, connected_and_in_curpart[K];
degree = RowPtr[cur_vertex+1] - RowPtr[cur_vertex];
for *j in neighbor of current vertex* **do**
    current_neighbor = current neighbor that is connected to current vertex
    for *cur_part in all the partition* **do**
        if *rP[cur_neighbor]==cur_part* **then**
            increment connected_and_in_curpart[cur_part]
        end
    end
end
Create a cost array with size K, cost[K];
for *i in all the partition* **do**
    cost[i] = $\beta * (h * (\alpha * (connected\_and\_in\_curpart[i])^{\lambda})) + \alpha((1-h) * (N - rC[i] + Degree - \beta * (connectd\_and\_in\_curpart[i])^{1/\lambda})))$
end
Create index array, idx = {0,1,2,...,K};
Sort the cost array based on maximum value along with idx array(arg_sort);
Assign new partition atomically until one partition has size of N/K;
If the partition with the max cost is already full, check the next one in idx array;

CHAPTER 4

RESULTS

The purpose of our work is to partition a large graph into balanced sub-graphs quickly. To see the performance of Thanos, we ran real-world graphs provided in the graph challenge competition [12].

4.1 Runtime

First, we measured both the CPU and GPU runtimes and compared with the CPU runtime of METIS [9] since it is consistently updated, maintained and used for graph partitioning as a state-of-the-art graph partitioning tool. For this measurement, we used P100 from NVIDIA for GPU, and Intel Quad Core for CPU. In Table 4.1, 'N' and 'M' denote the number of total vertices and edges in the graph respectively. The times are measured in seconds. Partition size of 4 is used for this measurement. We can see for some of the small graphs, Thanos is actually slower than METIS. However, for larger graphs, Thanos is much faster. Thanos achieves 163x faster runtime than METIS at the best for a large graph, 'roadNet-CA'. This is due to two factors. First, the graph has a huge number of vertices that can be processed in parallel utilizing the power of GPU. Second, as discussed earlier, each kernel has to run a loop that has bound of number of neighbors of the vertex, $O(Degree)$. For Thanos, if few vertices have a huge number of edges compared to the rest, the other threads will be idling when the few threads are processing the vertices that have a large number of neighbors. To reduce the runtime, we assigned the vertex that has a maximum outgoing degree to the CPU. However, for 'roadNet-CA', all vertices have

Table 4.1: Runtime Comparison with METIS

Graph Name [12]	N	M	Thanos(CPU)	Thanos(GPU)	METIS(CPU)	Speed Up(CPU)	Speed Up(GPU)
as20000102	6,474	12,572	0.009	0.035	0.02	2.22x	0.57x
ca-CondMat	23,133	93,439	0.04	0.0067	0.038	0.95x	5.67x
oregon1 010331	10,670	22,002	0.012	0.029	0.02	1.66x	0.68x
p2p-Gnutella04	10,876	39,994	0.019	0.0031	0.04	2.1x	12.9x
as-caida20071105	26,475	53,381	0.031	0.05	0.05	1.6x	1x
facebook combined	4,039	88,234	0.026	0.0021	0.004	0.15x	1.9x
email-Enron	36,692	183,831	0.071	0.031	0.077	1.08x	2.48x
loc-brightkite edges	58,228	214,078	0.09	0.021	0.12	1.3x	5.71x
cit-HepPh	34,546	420,877	0.14	0.02	0.067	0.47x	3.35x
cit-Patent	3,774,768	16,518,947	8.2	0.2	25.6	3.12x	128x
soc-Epinions1	75,879	405,740	0.15	0.057	0.07	0.46x	1.22x
soc-Slashdot0811	77,360	469,180	0.17	0.051	0.36	2.11x	7.05x
soc-Slashdot0902	82,168	504,230	0.19	0.049	0.4	2.1x	8.16x
amazon0302	262,111	899,792	0.4	0.016	0.36	0.9x	22.5x
amazon0312	400,727	2,349,869	0.9	0.071	0.67	0.7x	9.43x
amazon0505	410,236	2,439,437	0.93	0.076	0.81	0.87x	10.65x
amazon0601	403,394	2,443,408	0.95	0.071	0.67	0.7x	9.43x
roadNet-PA	1,088,092	1,541,898	1.1	0.015	2.4	2.1x	160x
roadNet-TX	1,379,917	1,921,660	1.12	0.022	3.11	2.77x	141.36x
roadNet-CA	1,965,206	2,766,607	1.6	0.03	4.9	3x	163.33x
flickerEdges	105,938	2,316,948	0.42	0.1	0.5	1.19x	5x
graph500-scale18-ef16	174,147	7,600,696	0.7	0.49	3.72	5.3x	7.59x
graph500-scale19-ef16	335,318	15,459,350	1.4	0.76	7.9	5.6x	10.39x
graph500-scale20-ef16	645,820	31,361,722	2.6	1.2	18.5	7.1x	15.41x
graph500-scale21-ef16	1,243,072	63,463,300	5.2	2	48.4	9.3x	24.2x
Speed Up in Average						14x	30x

almost the same number of neighbors, between 1 to 12 enabling all threads to finish their jobs very quickly and in the same time. On average, Thanos using GPU is 30x faster than METIS.

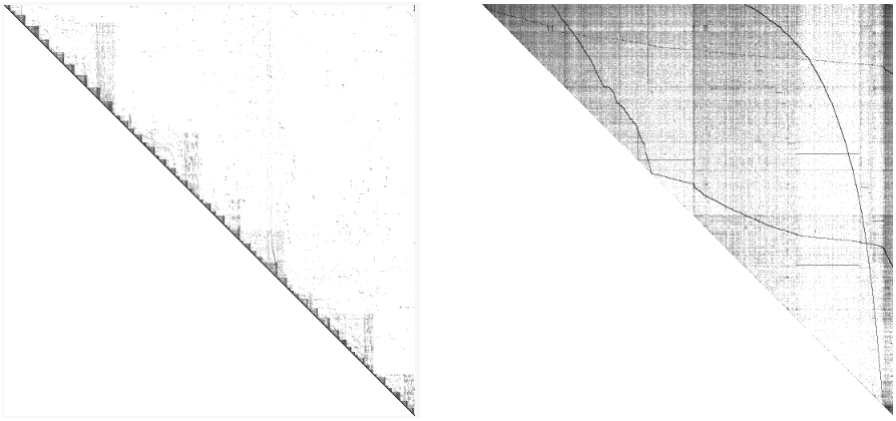
4.2 Partition Quality

Since those real graphs are very large, we cannot visualize easily with graph visual tools to see the partition quality. Table 4.2 shows the result of comparing the edge cut reduction results among partitions with Thanos and METIS [9]. To compare, we used random partitions as the baseline since partitioning a graph randomly is the fastest method although quality might be poor. In the table, 'P0' denotes the total number of edges inside partition 0. 'P0↔P1' denotes the total number of edges that are connecting partition 0 and 1. '#External Edges' denotes the total number of edges that exist among partitions. Finally 'Reduction %' shows the edge cut reduction percentage compared to the solution done by random partitioning. The ideal result should be maximized internal edges for each partition and minimized outgoing edges among partitions. From data set 'roadNet-CA', for Thanos, we can see the number of edges that are leaving one partition to another are dramatically reduced and the number of internal edges for each partition are dramatically increased compared to the random partition. For Thanos, 99% of edges that were originally connecting partitions are now put inside of partition making each partition more dense while METIS is achieving only 44% reduction from the random partition. We achieved the best reduction result for this data set. Unfortunately, Thanos is not effective on some data sets. On data set 'soc-Slashdot0902', both Thanos and METIS were not able to achieve any benefit from just partitioning a graph randomly resulting in edge reduction of 0%.

To see how original graphs look, we sampled with every 500 vertices and visualized the adjacency matrix since visualizing the entire graph is not possible. Figure 4.1 shows the upper triangular of adjacency matrix for the 'roadNet-CA' graph and 'soc-Slashdot0902'. From this visualization, we can see that vertices are not densely

Table 4.2: Edge Cut Reduction Among Partitions with Partition Size of 4

Graphs		P0	P1	P2	P3	P0↔P1	P0↔P2	P0↔P3	P1↔P2	P1↔P3	P2↔P3	#External Edges	Reduction %
roadNet-CA	Random	172,390	172,867	172,430	172,523	346,209	346,157	346,815	345,936	345,621	345,659	2,076,397	N/A
	Thanos	680,502	699,690	684,636	672,945	6,758	1,281	2,028	1,569	5,517	11,681	28,834	99
	METIS	399,954	392,910	408,183	401,105	210,990	175,644	177,903	208,743	216,038	175,137	1,164,455	44
FlickerEdges	Random	144,189	144,475	138,344	152,537	288,925	282,395	296,628	282,981	296,896	289,578	1,737,403	N/A
	Thanos	166,462	57,546	64,264	1,623,507	58,574	59,214	181,482	57,003	25,126	25,126	405,169	77
	METIS	132,563	135,752	134,291	210,271	269,319	267,067	296,367	270,731	300,823	299,764	1,704,071	2
cit-Patent	Random	1,032,127	1,034,990	1,036,009	1,028,355	2,063,059	2,069,316	2,061,663	2,067,036	2,061,341	2,065,051	12,387,466	N/A
	Thanos	764,078	2,021,047	4,610,788	3,062,972	401,710	530,587	662,338	1,018,738	1,375,605	2,071,084	6,060,062	51
	METIS	1,375,139	958,054	1,226,628	663,725	2,248,090	2,574,299	1,887,625	2,167,479	1,599,996	1,817,912	12,295,401	1
graph500_ scale19-ef16	Random	468,606	485,219	481,329	498,033	953,225	950,909	964,874	966,016	982,068	979,396	5,796,488	N/A
	Thanos	6,234	1,186,379	365,016	943,312	143,649	59,822	90,287	1,439,322	2,264,775	1,230,879	5,228,734	10
	METIS	530,753	528,151	437,942	438,250	1,059,959	966,271	966,352	963,401	963,184	875,412	5,794,579	0
soc-Slashdot0902	Random	29,704	32,641	32,039	31,329	62,434	62,215	60,808	64,855	64,228	63,977	378,517	N/A
	Thanos	8,603	24,422	34,365	57,373	10,946	14,350	44,717	47,037	108,516	153,901	379,467	0
	METIS	32,246	29,979	36,535	27,483	61,787	68,703	59,620	66,853	57,919	63,105	377,987	0



(a) roadNet-CA (b) soc-Slashdot0902

Figure 4.1: Upper Triangular Adjacency Matrix [13]

connected for the road net. Each vertex is connected to few vertices only. Also, most importantly, the connections are close to each other. This is why we see little triangles in black color and big triangles in shady color in the figure. The cross-decomposition algorithm seems to perform the best with graphs that are already nicely clustered.

In contrast to the road net, 'soc-Slashdot0902' is a very dense graph. Each vertex is connected to many other vertices especially toward the right edge. Also, there is a big triangle formed which is hard to partition. As these cases show, the performance of the algorithm depends on the characteristics of the graph.

For the rest of the graphs, we used a chart, Figure 4.2, to show the percentage comparison for better readability. On average, Thanos was able to achieve 43% edge cut reduction while METIS was achieving 8%. Thanos is also doing well with a very large network graph. For data set, 'friendster' [12], which has 120 million vertices, we were able to achieve 30% edge cut reduction in only 80 seconds. Figure 4.3 shows the average edge cut reduction of all the data set based on a different number of partitions with both Thanos and METIS. We have tested with partition sizes of 4, 8, 12, 16. Thanos always produces around 40% edge cut reduction while METIS always produces around 8%. Based on the result, Thanos performs well on balanced graph partitioning in general.

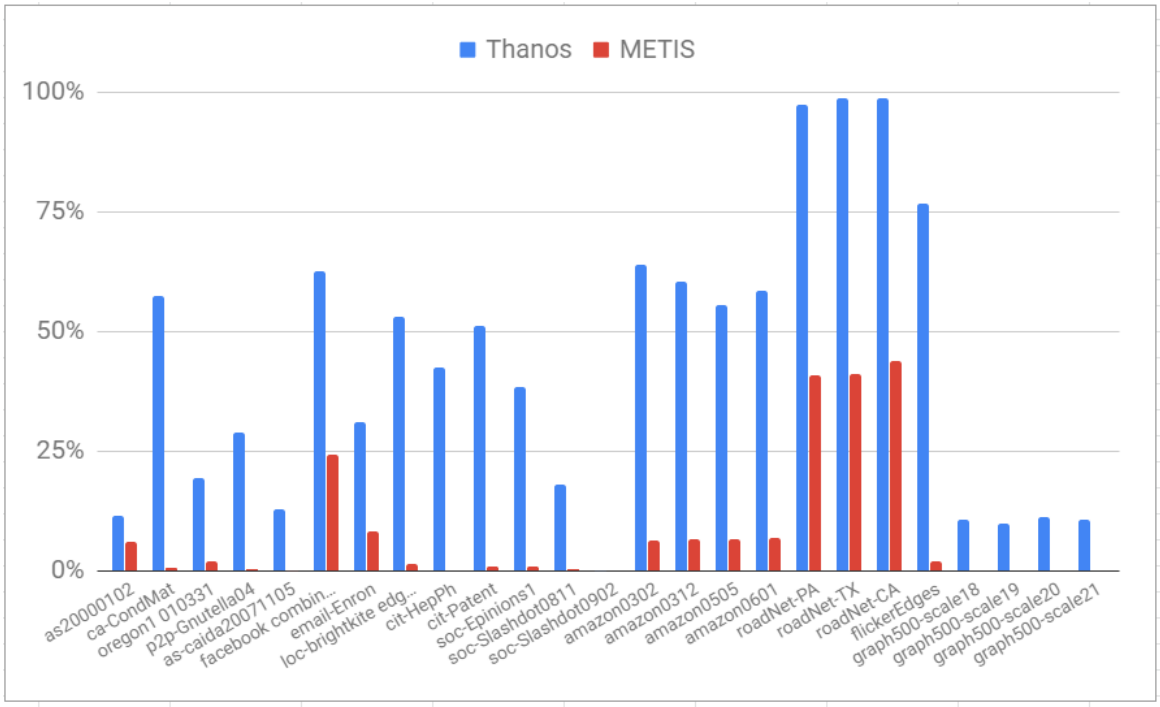


Figure 4.2: Edge Cut Reduction with Different Data Set with Partition Size 4

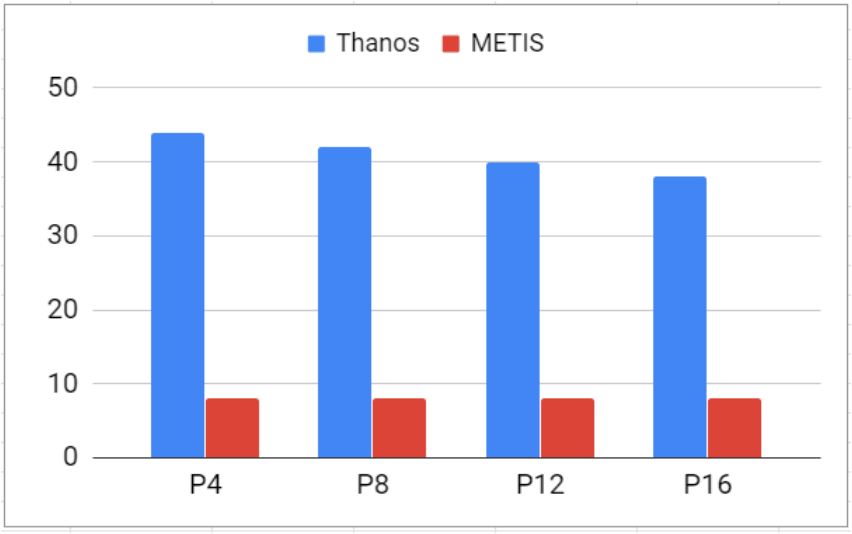


Figure 4.3: Average Edge Cut Reduction Based on Different Partition, 4, 8, 12, 16

CHAPTER 5

CONCLUSION

In this work, we introduced a fast graph partitioing tool Thanos that uses the cross-decomposition algorithm. We have demonstrated that the cross-decomposition algorithm fits well with the large-scale graph partitioning problem. Not only is the partition fast but the quality of the partition is high. In the best case, Thanos achieved 99% edge cut reduction compared to the random partition. Also, the sizes of all partitions are balanced. Partitioning a graph into equal sizes while minimizing the edges among different partition is very important in parallel computing. With the result of our thesis, we can work on multiple sub-graphs in parallel knowing that each partition is a dense cluster. Our work will be open sourced [14].

REFERENCES

[1] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, Nov 2006. [Online]. Available: https://doi.org/10.1007/s00224-006-1350-7

[2] R. Krauthgamer, J. S. Naor, and R. Schwartz, *Partitioning Graphs into Balanced Components*, pp. 942–949. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611973068.102

[3] L. Hyafil and R. Rivest, "Graph partitioning and constructing optimal decision trees are polynomial complete problems," *IRIA-Laboratorie de Recherche en Informatique et Automatique, Tech. Rep. 33*, 1973.

[4] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified np-complete problems," in *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '74. New York, NY, USA: ACM, 1974. [Online]. Available: http://doi.acm.org/10.1145/800119.803884 pp. 47–63.

[5] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is np-hard," *Inf. Process. Lett.*, vol. 42, no. 3, pp. 153–159, May 1992. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(92)90140-Q

[6] H. Garcia and J. M. Proth, "Group technology in production management: The short horizon planning level," *Applied Stochastic Models and Data Analysis*, vol. 1, no. 1, pp. 25–34, 1985. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/asm.3150010105

[7] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[8] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[9] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997

[10] H. Hillion and J.-M. Proth, "A top-down hierarchical classification method," *Applied Stochastic Models and Data Analysis*, vol. 3, no. 4, pp. 247–255, 1987. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/asm.3150030406

[11] M.-C. Portmann and J.-M. Proth, *A Cross-Decomposition Method for Layout Systems and Scheduling Problem*, 01 1989, pp. 323–327.

[12] S. Samsi et al., "Static graph challenge: Subgraph isomorphism," in *IEEE HPEC*, 2017.

[13] "Graph challenge data set stats," https://graphchallenge-datasets.netlify.com//, 2019.

[14] D. H. Kim, *Thanos*, 2019, https://github.com/dannyk0104/Thanos.