SyWoWa: a System & Workload Aware Graph Partitioner

Milad Rezaei Hajidehi, Aryan Tajmir Riahi

Distributed Systems, Winter Term 1 of 2021

1 Introduction

Graphs have always been an interesting class of data and drew a lot of attention in the field of data-intensive computing since many real-world interconnected entities and their relationships can correspond to a graph's vertices and edges. Social networks, road networks, web graphs[3], protein structures[26] are few examples where people view their data as a graph and extract information utilizing graph algorithms such as PageRank, shortest path, betweenness centrality, etc. [22]

Although map-reduce[9] is a master key for many big data applications, it has been shown that it is not a proper solution for graph-structured data due to the unique access pattern of graphs and their interdependencies[8]. For this purpose, people build special systems for doing their analytical and transactional jobs, referring to them as graph processing systems or GPS in short form. Google Pregel[16], Apache Giraph, GraphChi[14], PowerGraph[13] are some instances of GPSs.

Each GPS has a unique design and architecture, exploiting different ideas to tackle a particular class of problems. For example, PowerLyra[7] is designed to work with skewed graphs like social networks where the degree of vertices follow a power-law distribution. Some GPSs, such as Ligra[23] and GraphChi are designed to work in a single-machine environment, avoiding distributed system complexities. However, they suffer from the low degree of parallelization since the number of cores in a single machine is limited. Also, there is a limitation for size of graph and storing it in the main memory of a single machine.

With distributed settings, GPSs become more scalable. They can store very large graphs with trillion edges in the main memory of multiple machines. The other advantage is supporting a high degree of parallelization and more threads than a single-machine system. However, distributed graph processing has some challenges and drawbacks: communication and synchronization between machines, distributing the graph, and balancing workloads. In particular, distributing the main graph between machines, known as graph partitioning, plays a vital role in the speed and network overhead of a distributed GPS. Bad partitioning results in either more network overhead or an unbalanced workload. We use the term machine/partition interchangeably. The term node here is confusing since it can be interpreted as a single machine in our distributed system or a vertex in our graphs. Thus we avoid using this term in this proposal.

Network communication in a distributed GPS is needed when there is an edge (u, v) where u,v are not in same partitions(machines), or for synchronization of data between a vertex and its replicas in other partitions. Bad partitioning results in more network overhead and less speed. In addition to network overhead, we are interested to get balanced subgraphs in each partition to eqaualize the load of processing at each machine. The speed of processing is limited to the machine which finishes its jobs latest. Unbalanced partitioning, therefore, may lead to slower processing. We will elaborate and formulate the partitioning problem and its parameters precisely in section 2.

The relation between partitions balance and network overhead is not trivial. They may have inverse relations where optimizing balance leads to more network overhead and vice versa. Also, it is not clear that how these two parameters affect execution time for a graph workload like PageRank. Besides, this relationship between execution time, network overhead, and balance factor can vary per *workload feature*. In this proposal, workload features are the type of workload(PageRank, BFS, Shortest Path), underlying graph features, number of partitions, number of threads, hardware settings, and GPSs design decisions such as async/sync communication. We will explain these workload features in section 2.

Our motivation for making a system and workload aware partitioning is based on existing widely used partitioning algorithms such as HDRF[20], Ginger[7], Fennel[25] try to optimize theoretical values like edge-cuts rather than execution time. Pacaci et al. in a recent SigMod paper[19] shows that this optimization would not always lead to better execution times. Also, these algorithms perform the same process per different systems and workloads, which is a potential for better partitioning and correspondingly faster computation[12]. In sum, in the project's scope, first, we aim to unravel the role of workload features in the quality of a partition respecting execution time. Then, we want to take a deep dive into this problem and design a learned partitioner based on the workload features.

Our deliverables for this project would be, first, a pipeline for partitioning and doing workloads with different features and settings to measure execution time and train a learning model. Then we add a learning model to our pipeline to learn a score function for a streaming graph partitioning algorithm based on execution times. In the end, we evaluate our learned workload and system aware partitioner, SyWoWa, in terms of execution time with state-of-the-art and widely used graph partitioners. We also measure the cost of the learning process in terms of time, resources, and the number of experiments needed.

2 Background

In the following section, we will elaborate on the fundamental concepts necessary to understand the problem mentioned in the previous section and our idea about that.

2.1 Graph partitioning

To process large graphs, graph partitioner algorithms are used to distribute the graph between different machines. More formally, assume we have a graph $G = (V, E)$ and we want to partition it into a set of partitions known as P. Traditional graph partitioning research line focuses on partitioning vertices, in other words, they aim to divide V into disjoint the subsets V_p such that $\bigcup_{p\in P} V_p = V$. In this approach, the number of *edge cuts* is the usual measurement that models network overhead, which is defined as $|\{(v,u) \in E | v \in V_i \wedge u \in V_j \wedge i \neq j\}|$. Also, there are some other approaches in graph partitioning such as *edge partitioning*. Similarly, these algorithms want to divide E into disjoint the subsets E_p such that $\bigcup_{p\in P} E_p = E$. However, in this case the number of *vertex replications* is an alternative measurement of the number of edge cuts, which is defined as $\sum_{v\in V} |\{p | \exists(u,v) \in E : (u,v) \in E_p\}|$. To ensure getting a balanced partitioning we add an extra condition to the vertex partitioning problem which is $\max_{p\in P} V_p < \alpha\frac{|V|}{|P|}$ (respectively

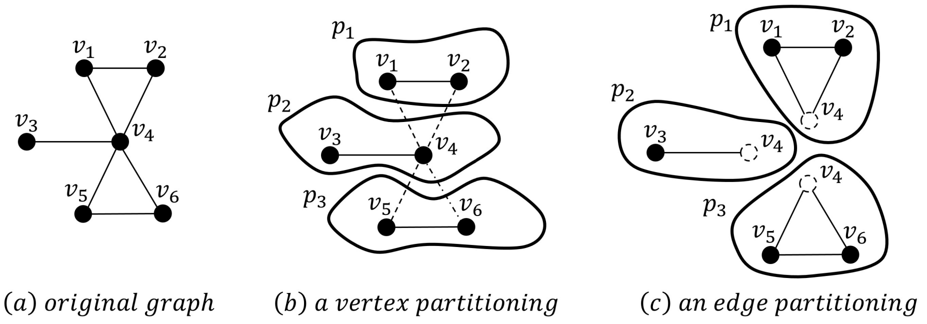


Figure 1: An example of graph partitioning with two different approaches.

$\max_{p\in P} E_p < \alpha\frac{|E|}{|P|}$ for edge partitioning), which $\alpha > 1$ is called the *imbalance factor* here. Figure 1 shows an example of these two different approaches with the edge cut factor of 4 (for vertex partitioning) and the vertex replication factor of 8 (for edge partitioning).

There is a strong intuition of the relation between these aforementioned factors of graph partitioning and the total execution time of a workload. However, as we mentioned in Section 1, the relationship among these factors with each other and the relationship between these factors and execution time is not crystal clear. Also, partitioning algorithms have the same logic regardless of the underlying system and workload features.

Graph partitioning with aim to reduce number of edge-cuts has been proven to be an NP-hard problem[11]. Although it was an old problem in computer science and there are approximation algorithms for this problem, most of those algorithms are not usable since those algorithms consider that graph is in main memory, which is not valid in context of our problem.[4]

Streaming algorithms or random hashing are usable approaches for large graphs. At each stage T, a streaming partitioning algorithm uses a chunk of data as a set of edges or vertex and its neighbors and assigns those edges and vertices to a partition based on a score function. Instead of being aware of the whole graph and partitioned edges and vertices, these algorithms maintain a limited set of information about what they did before. For example, they can only save what vertices they partitioned and the partition index for those vertices. We explained this concept more in Section 3 where we propose our solution for partitioning.

2.2 Think like a vertex model

Most of the GPSs are built based on Think Like a Vertex(TLAV) or vertex programming idea[18]. TLAV is a computation paradigm like map-reduce, where a graph workload must be implemented as a piece of user-defined code. The system then iteratively execute the code per each *active* vertex.

Figure 2 shows an example of the PageRank algorithm implemented based on TLAV where all of the vertices are active at the beginning. In this example, each vertex v has a buffer v_buffer that is used by its neighbor to notify v about their new ranks. This code will be executed for every vertex at each iteration. Each vertex v first updates its rank with the values that in-neighbors of v has put in the v_buffer in the previous iteration. Then, v calculate it is new rank based on sum of its neighbors new rank and sends this value to the buffer of all out-neighbors. If the out-neighbor was not co-located in the same partition with v, this update needs to be done over the network. Iterations will continue till all of the vertices converge. This code is for a vertex-partitioned system where there is no replica for vertices. For edge partitioned systems, a vertex should synchronize its

```
Compute(v, v_buffer)
    sum = 0.0
    for message in v_buffer:
        sum += message.value

    v.rank = 0.15 + 0.85 * sum

    if (converged(v)):
        deactivate(v)
        return

    for neigh in v.out_neigh():
        notify(neigh, v.rank)
```

Figure 2: Psuedo code of a user-defined function for PageRank

rank with other replicas, but the communication for edges is unnecessary because every neighbor would be local.

2.3 Workloads and execution of vertex programs

The execution model and activity/inactivity of vertices in a TLAV workload depends on the type of algorithm. For example, in PageRank, all vertices are active at the beginning and remain active till they reach convergence. On the other hand, execution pattern of shortest past algorithm is quite different(Figure 3). In the shortest path algorithm, at the beginning, only the source vertex is active. The set of active vertices changes at each iteration. Each vertex deactivates itself and activates its neighbors if they were not active from the beginning. Thus, the volume of communication in a workload is a function of workload type. Correspondingly impact of edge-cuts or replication factor on execution time depends on workload type.

2.4 Async/sync model of computation

There are two ways to schedule execution of vertex programs in a TLAV system. The first one is *synchronous* computation, similar to the bulk-synchronous-parallel model (bsp) [6]. In this method, we divide computation into *supersteps*. Each superstep consists of two phases. In phase one, all of the machines start to run user-defined programs per each active vertex. When we finish running vertex programs for all vertices in a machine we proceed to the second phase. In the second phase, each machine starts to communicate with other machines, and it blocked by a *synchronization barrier* and have to wait for other machines to finish their superstep, in order to proceed next superstep.

In the *asynchronous model*, on the other hand, there is not any synchronization barrier. Vertex programs are executed independently without having to wait or coordinate with other vertices. These vertex programs progress independently and make network calls whenever the resource is available.

We mentioned these two models of computation because it completely changes computation and correspondingly definition of a good partition. In the synchronous model, we are able to batch messages for all of the vertices in a single machine which notably reduces network overhead. Therefore, we can relax cuts optimization in partitioning for the synchronous systems to gain more

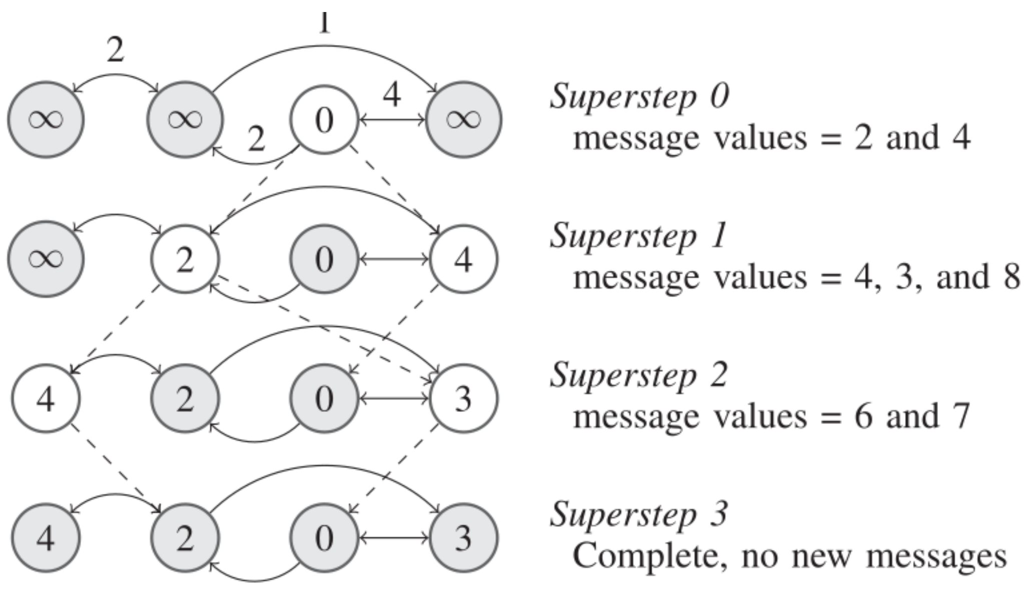


Figure 3: Execution of single source shortest path in TLAV paradigm. The process takes four iteration(supersteps). White vertices are active vertices in each iteration and dotted lines are communications.

balance partitions. On the other hand, in the asynchronous model, each vertex makes network calls independently which amplifies the role of cuts. However, this architecture would not suffer from lagging behind a synchronization barrier. This model has an advantage in skewed graphs and all-vertices-active workloads like PageRank where there are some vertices with a high degree and other vertices should wait for them.

2.5 Gaussian process

We use Gaussian process regression model for analyzing the execution time of a GPS workload. A Gaussian process (GP) can be viewed as a distribution over function space. More formally assume that we are analyzing functions of the form $f : X \to \mathbb{R}$ where X is a bounded subset of $\mathbb{R}^d$. GP assigns a random variable to each point of X with the property that any finite subset of these random variables comes from a multivariate Gaussian distribution. The properties of the induced distribution are completely determined by the mean function $m : X \to \mathbb{R}$ and the covariance function (also known as the kernel) $K : X \times X \to \mathbb{R}$ of the process [21]. These functions are defined as

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})],$$

$$K(\mathbf{x_1}, \mathbf{x_2}) = \mathbb{E}[(f(\mathbf{x_1}) - m(\mathbf{x_1}))(f(\mathbf{x_2}) - m(\mathbf{x_2}))].$$

Covariance functions are typically chosen in a way that close points in X have more correlation [10]. One common choice of these functions is the *squared exponential kernel*. In this covariance function, we have

$$K(\mathbf{x_1}, \mathbf{x_2}) = \alpha_0 e^{-\frac{1}{2} r^2(\mathbf{x_1}, \mathbf{x_2})},$$

where r is a normalization of euclidean distance in $\mathbb{R}^d$ defined by $r^2(\mathbf{x_1}, \mathbf{x_2}) = \sum_{i=1}^{d} \alpha_i (x_{1,i} - x_{2,i})^2$ and $\alpha_0, \ldots, \alpha_d$ are the hyperparameters of the function [24]. These hyperparameters determine the speed of change of functions (see Figure 4).

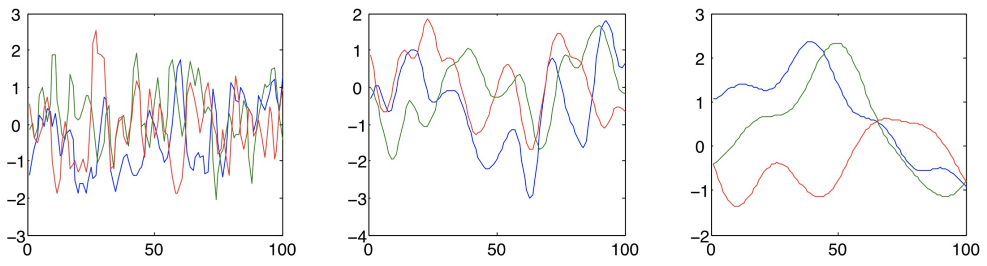


Figure 4: Each plot illustrates three random functions drawn from a GP prior with the same hyperparameters except for α_1. The value of α_1 is decreasing from the left figure to the right one.

2.6 Bayesian optimization

In Bayesian optimization[24][10] or BO in short form, we aim to find the minimum of a function $f : X \to \mathbb{R}$ with a probabilistic prior without using any other information from f such as its gradient. For the prior over functions Gaussian process prior (see Section 2.5) is the default choice due to its generality and flexibility. Generally the BO algorithm, in each iteration, updates the posterior probability distribution and chooses the next point in a way that maximizes a function called *acquisition function* $a : X \to \mathbb{R}$. Two typical choices for this function are the *Probability of Improvement* and *Expected Improvement.* In the former function, we choose the next point to maximize the probability of finding a point with a lower value $f(\mathbf{x})$. However, in the latter case, we choose points to maximize the expected improvement over the current best. More formally, an overview of this algorithm is shown in Algorithm 1. An example of this algorithm's process is shown in Figure 5.

Algorithm 1 Overview of Bayesian optimization

```
    Input a GP prior on $f$, maximum iteration number $N \in \mathbb{N}$, an acquisition function $a$
1: $i = 0$
2: Select $\mathbf{x_0}$ randomly from $X$
3: while $i \leq N$ do
4:     Observe $y_i = f(\mathbf{x_i})$
5:     Update the posterior probability distribution on $f$ with respect to new data
6:     Set $\mathbf{x_{i+1}} = \text{argmax}_{\mathbf{x} \in X}\, a(\mathbf{x})$
7:     Update $i = i + 1$
8: end while
9: return $x_i$ with the largest $y_i$
```

3 Proposed Solution

In the following subsections, we first formalize the problem, solution, and the variables we look for. Then we propose our learning methodology. At last, we explain SyWoWa pipeline to show how we run different graph workloads to report the execution times, which is the training data for our learning model.

Please note that we may change some of these proposed solutions in the future. For example, we may change our Machine learning model or the GPS we use for running experiments, but we

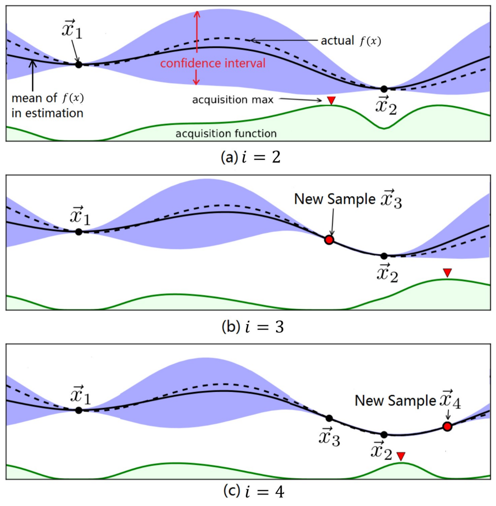


Figure 5: An example of three iterations of the BO algorithm. In each iteration, the bold line and the shaded purple area are representing the posterior of mean and variance of f respectively while the dashed line is showing its actual value. The green line represents the acquisition function in iteration.

state the rationale behind changes in the final report.

3.1 Score Function & Formalization

As we discussed in the previous section, a streaming graph partitioning algorithm, at each step, takes a vertex or an edge and tries to assign it to the partition that maximizes the score function. Our proposed solution is learning a linear score function with its coefficients(weights) per different workload features. More precisely, for assigning vertex x to a partition P_i, our streaming partitioning algorithms have a function F, which calculates the score of assigning x to P_i:

$$F(x, P_i) = A_1X_1 + A_2X_2 + A_3X_3 + ... + A_kX_k$$

Where each X_i is a parameter that gives us some information and heuristic about vertex x, P_i, and vertices and edges in P_i at the moment of partitioning x. For example, the number of x's low/high degree neighbors in partition i. Another example for X_i is the size of partition i at the moment of partitioning vertex x, in terms of the number of edges or vertices in P_i. We will normalize A_is in a way that:

$$\sum_{i=1}^{k} A_i = 1$$

Each A_i is a real number that indicates the impact of its corresponding parameter X_i in the score function. Our hypothesis is these impacts are different per workload features. In fact, in different systems and scenarios, factors have various impacts on the execution time. Recall our argument about how asynchronous and synchronous GPSs have distinct drawbacks and bottlenecks in Section 2.4, or how the activity of vertices is different per type of workloads in Section 2.3.

Existing successfull and state-of-the-art streaming partitioners use limited heuristic parameters (X_is). Balance factor and number of common neighbors are two conventional parameters used in Fennel, LDG, Ginger. Finding more X_is rather than mentioned parameters is one of our tasks during this project. For instance, we are suspicious to differentiating between low/high degree neighbors because we think high degree neighbors anyway forced to communicate most of the partitions, which is not valid for low degree vertices(Meyer et al.[17], used this fact for a hybrid in-memory/streaming partitioner, however, their streaming part is HDRF). We want to find more heuristic parameters and prove and explain their effect.

In the scope of this project, workload features are the number of machines(partitions), degree of parallelization(thread count per machine), workload type(PageRank, BFS, Shortest Path, ...), underlying graph, and model of communication. So we can say the input of our pipeline is the vector of workload features, and the output is the vector of A_is. Below is an example of our pipeline input and output:

$$G(4, 1000, PageRank, "twitter\_follow\_graph", async) = A = [0.3,\ 0.2,\ ...\quad, -0.1]$$

3.2 Learning Model

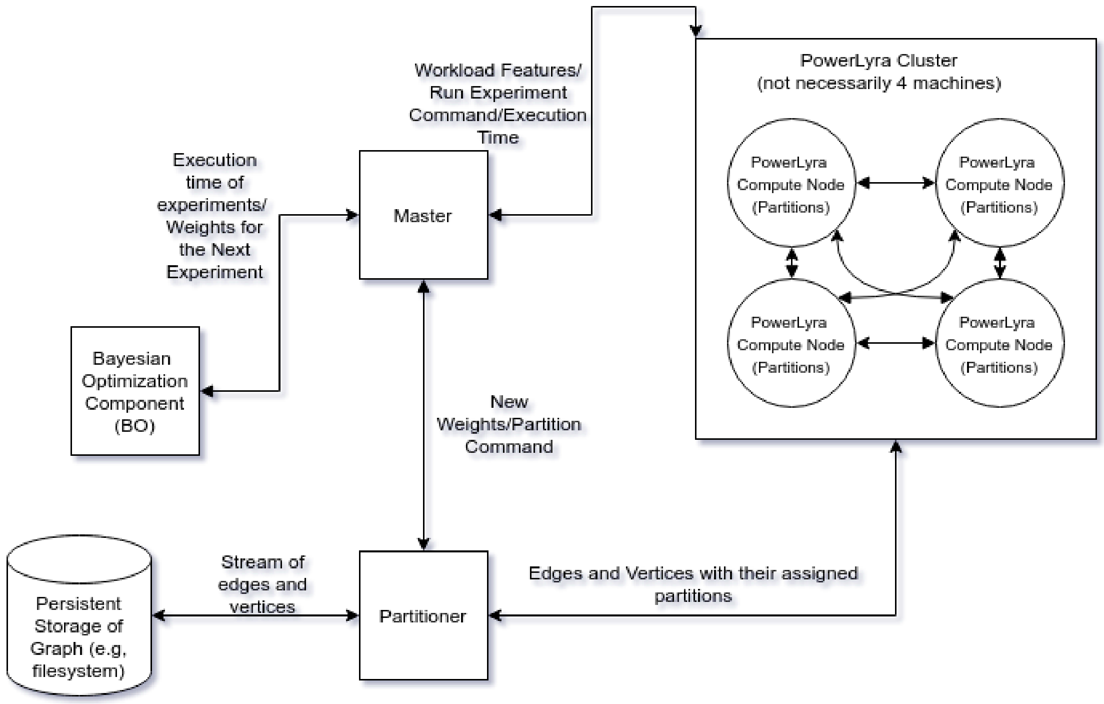
From the previous part, the output of function G is a set of coefficient A per a specific workload feature. From this point, we suppose workload features are set. For finding a vector A that minimize execution time, we define the function H per each workload feature:

$$H_{workload\_feature}(A_1,\ A_2,\ ...,\ A_k) = execution\ time$$

Minimizing function H and finding A vector is hard since we have no definition of the function H. Thus, we cannot use many of the optimization techniques. Also, we are not sure that whether H is convex or not. So, our optimization technique must treat H as a black box and only know the output of H for a given input.

Besides, knowing the output of H for a given input is an expensive process. It needs partitioning a large graph from the beginning, assigning subgraphs to machines of a distributed graph processing cluster, and finally running a workload that may take more than a minute. Therefore, our black box learning approach must find a near-optimal output by asking questions as few as possible. Here, by the question, we mean knowing the output of a function for a given input. In other words, the execution time of a workload per different partitioning coefficients.

Our learning model is based on BO. As we mentioned in Section 2.6, BO is a method for optimizing black-box functions that converge a near-optimal output by asking a few questions. BO asks question i based on answers it got in previous iterations. The same approach is used by Alipourfard et al.[2] to find the best cloud hardware configuration for a general big-data workload. As you can see in 1, BO tries to minimize the number of asked questions at cost of high computational complexity for choosing the next point. This computation especially for high dimensional spaces can be a trouble. However, in our case, we are dealing with some chosen factors and the number of these factors barely exceeds 7, so we would not suffer from high dimensionality. Also, after asking each question we should try partitioning and run a workload on it which will be extremely expensive. As a result, this learning algorithm is quite suitable for our work.



Workload Features/ Run Experiment Command/Execution Time
PowerLyra Cluster (not necessarily 4 machines)
PowerLyra Compute Node (Partitions)
PowerLyra Compute Node (Partitions)
PowerLyra Compute Node (Partitions)
PowerLyra Compute Node (Partitions)
Execution time of experiments/ Weights for the Next Experiment
Master
Bayesian Optimization Component (BO)
New Weights/Partition Command
Stream of edges and vertices
Persistent Storage of Graph (e.g, filesystem)
Partitioner
Edges and Vertices with their assigned partitions

Figure 6: Initial design of SyWoWa experiment and learning pipeline.

3.3 Experiments Pipeline

The general overview of our experimenting and model training pipeline is shown in Figure 6. Master, partitioner, BO, persistent storage, and a Powerlyra cluster are components of this pipeline.

We use PowerLyra[7] as a GPS for running our experiments. We selected Powerlyra because it supports both asynchronous and synchronous models of computation. Suppose we use two non-identical GPSs for the asynchronous and the synchronous model. In that case, one may argue that other differences between the two systems may cause the changes in execution times and correspondingly find different weights. Thus, we have to ensure that the only difference between the two experiments is the async and sync computation model.

The graph is a large object in a graph processing pipeline and is usually located in a persistent storage. A *partitioner* reads data in chunks and assign edges or vertices to a partition P_i and then notify ith machine which is a representative for P_i.

We run the training pipeline iteratively. The master first sets a workload feature vector at the first iteration and bootstraps the PowerLyra respecting those features(e.g., number of machines and threads, workload type). Then at each iteration, the master asks the BO component for a weights vector A. Master gives these weights to the partitioner and asks it to assign subgraphs to PowerLyra machines regarding the score function and the weights. After loading the graph, master run workload on PowerLyra and report the execution time to BO components. BO component uses this result to select weights for the next iteration(s). The iterations end when BO reports a convergence. At that point, we have the best weights per workload feature. Master reports the total amount of time spent at each stage in each iteration in addition to final weights. We must hard-code heuristic parameters, X_is, in partitioner code. Also, note that in Figure 6 arrows are not always network calls, and separation of components does not mean they are located in different machines.

In addition to finding the best weights, this pipeline helps us find new heuristics and test our hypothesis about them for graph partitioning. So it can be a boilerplate for finding new partitioners.

4 Evaluation

For evaluating our idea, we first select a workload feature and find the best weights for that. Then we run the experiment with that weights for its partitioner ten times and compute mean time. Then, we do this experiment but this time using different partitioners. We want to compare Sywowa with Ginger- PowerLyra original partitioner-, Fennel- a widely used and succesfull partitoner- and random hasing as a trivial solution. Unlike, typical partitioner comparison[1][19], we only care about the execution time of workload not the number of edge cuts, replication factor, and balance factor because we did not aim to optimize those theoretical values.

We evaluate our method for different models of computation (sync/async), diverse families of graphs (power-law, road-network, and citation graph), various numbers of machines in the system with varying degrees of parallelization, and two different workloads, BFS and PageRank. We have planned to use the SNAP[15] dataset for real-world graphs and the RMAT graph generator[5] for generating large synthetic graphs since there are just a few graphs with more than a billion edges available for the public and research use. (However, graphs with this scale are not rare in real-world and companies are maintaining these large graphs)

We also measure the amount of time it takes to find near-optimal weights, and the answer for this question that how bounding iterations of BO affect the quality of SyWoWa. Also, finding new heuristics for partitioning is a priceless contribution that we hope to find and report.

For infrastructure setup, we may use ComputeCanada but we should consider its limitation and gather information about that. Running in Docker containers and faking distributed systems would not work for us because we care about actual network delay.

5 Timeline

5.1 Weekly Tasks

Here is our desired timeline for doing project. It is a draft and we want to finalize it and make it more precise for final proposal(as it is said that timeline is not necessary for draft.)

[11 Oct, 18 Oct): Select and request hardware resouce, Research and learn about Bayesian Optimization

[18 Oct, 25 Oct): Select and learn about datasets for our experiments, Implement BO component

[25 Oct, 1 Nov): Gain insight about PowerLyra configuration and codebase, Finish implementation of BO

[1 Nov, 8 Nov): Read original FENNEL, LDG paper, Gain insights about implementation of mentioned algorithms

[8 Nov, 15 Nov): Make decision about heuristic parameters, Implement a partitioner based on defined heuristic parameters, Run experiments on PowerLyra

[15 Nov, 22 Nov): Implement the master component, Make the connection between different component

[22 Nov, 29 Nov): Run pipeline to find optimal weights for workloads with different graphs, and communication pattern

[29 Nov, 6 Dec): Prepare for presentation

[6 Dec, 13 Dec):Learn optimal weights per different workload algorithms, and possibly different number of machines

[13 Dec, 20 Dec): Compare results of SyWoWa with existing partitioners

5.2 Milestones Deliverables

In this section we define what we going to show you in each milestone session.

Milestone 1 - 29 October

- A document containing information, characteristics, and structure of our selected datasets for experiments.
- A document that summarizes our meeting with people in Lab who have experience in this line of work. (Thomas, Mayank, and Sadaf)
- We finish the BO component (Either our implementation or a library implementation based on their speed). We show you an optimization process of an example function. We should get an insight into the details of its implementation and time complexity.
- Select and prepare hardware environment for running PowerLyra.
- Select the proper version of PowerLyra.(There are many forks and versions with different features)
- Gain insight about PowerLyra codebase. Understand the components that we have to change for our experiments. (e.g., ingress). Understand the configuration.

Milestone 2 - 15 November

- A document containing the results of the experiments in different workload features highlights the role of each feature.
- A document containing a summary of our reading about existing partitioners and more importantly details of their implementation.
- Identifying online available and unavailable partitioning codes for reuse.
- Identifying the hyperparameters of BO that can affect our work. Gain deeper insight into BO.
- We show you the running of workloads on Powerlyra, which indicates we successfully deployed it on an environment, and we are confident with using that in experiments.

Milestone 3 - 29 November

- A document containing our selected heuristic parameters and the reason behind choosing each of them.
- We show you the code of master part and how we make connections between different components.
- We will run a optimization task for you to show you our pipeline is successful to run find weights iteratively.

References

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment*, 11(11):1590–1603, 2018.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[4] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *Algorithm engineering*, pages 117–158, 2016.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[6] Thomas Cheatham, Amr Fahmy, Dan Stefanescu, and Leslie Valiant. Bulk synchronous parallel computing—a paradigm for transportable software. In *Tools and Environments for Parallel and Distributed Systems*, pages 61–76. Springer, 1996.

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[8] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29–41, 2009.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[10] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[11] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[12] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A study of partitioning policies for graph analytics on large-scale distributed platforms. *Proceedings of the VLDB Endowment*, 12(4):321–334, 2018.

[13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[14] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a {PC}. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 31–46, 2012.

[15] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford stanford network analysis project. http://snap.stanford.edu/.

[16] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[17] Ruben Mayer and Hans-Arno Jacobsen. Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 1289–1302, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[19] Anil Pacaci and M Tamer Özsu. Experimental analysis of streaming algorithms for graph partitioning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1375–1392, 2019.

[20] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 243–252, 2015.

[21] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

[22] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[23] Julian Shun and Guy Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 135–146. ACM, 2013.

[24] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[25] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014.

[26] Yan Yan, Shenggui Zhang, and Fang-Xiang Wu. Applications of graph theory in protein structure identification. *Proteome science*, 9(1):1–10, 2011.