PARQ: A MEMORY-EFFICIENT APPROACH FOR QUERY-LEVEL PARALLELISM

A Thesis Presented

by

QIANQIAN GAO

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

July 2016

Department of Electrical and Computer Engineering

PARQ: A MEMORY-EFFICIENT APPROACH FOR QUERY-LEVEL PARALLELISM

A Thesis Presented

by

QIANQIAN GAO

Approved as to style and content by:

_____
Lixin Gao, Chair

_____
Michael Zink, Member

_____
David Irwin, Member

_____
C. V. Hollot, Department Chair
Department of Electrical and Computer Engineering

ACKNOWLEDGEMENTS

First, I am honored to express my deepest gratitude to Professor Lixin Gao. She offered me with this intriguing research project and every possible resource. I would also like to extend my appreciation to Professor Michael Zink and Professor David Irwin. Thank them for being the member of my thesis committee. Thank them for their valuable time, comments and encouragement.

Secondly, I give my sincere thanks to all the friends in the laboratory for their help in the completion of this thesis. They gave me many useful suggestions and lots of generous help during my academic period.

In the end, I will never forget the love from my parents. I am indebted to them for their continuous support and encouragement.

ABSTRACT

PARQ: A MEMORY-EFFICIENT APPROACH FOR QUERY-LEVEL PARALLELISM

July 2016

QIANQIAN GAO

B.S., BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lixin Gao

In the era of big data, people not only enjoy what massive information brings, but also experience the problem of information overload. As the volume of both data and users increasing sharply, more and more studies focus on how to answer a query for interesting information from massive data. However, most query systems are designed and implemented to optimize the performance in processing a single query and do not support data sharing among query processing jobs. When they are extended to process multiple concurrent queries, they will suffer the problems of the inefficient use of memory and waste of time.

This thesis aims to design and implement a memory-efficient system, ParQ, which could be adopted by query systems to realize query-level parallelism. The main idea includes constructing a common memory block for maintaining sharable data. By sharing data, ParQ is able to process multiple queries concurrently while reducing memory usage and running time. We applied ParQ to several existing query systems. The experiment results show that ParQ improves the performance in both job completion time and memory usage when executing multiple concurrent query jobs.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Trends and challenges

The future of big data is promising. Data has become a class of asset. In the areas of Internet, Telecommunications, Finance, and etc., many companies have achieved commercial success by using big data.

On the other hand, the development of big data also faces many challenges, such as, how to reduce data fragment. Nowadays, data is organized into different databases that may belong to different organizations. Even in the same company, data may be maintained by different departments. It is necessary to integrate all the data to better exploit the potential of information.

How to perform data preprocessing is also a big challenge. At the step of data preprocessing, data is cleaned, denoised, and transformed into an easy-to-use structure. Nonstandard preprocessing often leads to inaccuracies, poor quality, and low availability. In big data area, not only data collecting is important but also preprocessing.

Another challenge is how to find requested information from massive data, especially as the volumes of both data and users are increasing sharply. According to recent statistics, by August 2014, Google Knowledge Graph had contained 1.6 billion labeled entities [1] [2]. As of January 2014, Freebase had approximately 44 million topics and 2.4 billion facts [3] [4]. To make use of such databases, many researchers focus on algorithms of data processing. Topics include, for example, how to improve accuracy, how to increase efficiency, and how to reduce computation cost.

1.2 Motivation

A query system refers to an implementation that aims to answer queries on a database. Such applications have been widely applied in the areas of search engine, social network, e-commerce,

and etc. In practical applications, query systems may be queried by more than one user at the same time, therefore they are supposed to process multiple queries concurrently. The job execution logs from a large Chinese social network shows that at peak time, there are more than 20 jobs submitted to the platform at the same time [5]. However, some developing query systems are inherently designed to evaluate the performance in answering a single query and are difficult to be expended to process multiple queries. Therefore, although such query systems are efficient when processing a single query, they require huge memory usage and long processing time when they process a number of queries concurrently. This is because they cannot support multiple query processing jobs to share in-memory data and each job has to keep an exclusive copy of the same graph data, which results in a serious waste of time and memory.

This thesis aims to seek an approach to enable such systems to answer multiple queries concurrently by sharing sharable data among different queries. This approach should ensure 1) a query system uses only one copy of sharable data when it processes multiple queries, 2) the processing time of a query will not be affected by other queries, and 3) this approach should be easily used by different query systems.

1.3 Contribution

This thesis introduces a memory-efficient approach, ParQ system, to enable query systems to answer multiple queries concurrently by sharing data among different query jobs. In this thesis, the sharable data is the dataset from which a query system finds the answers of a query.

The ParQ system offers a standard interface. It provides the ability to automatically process data in different formats as long as the formats conform to the system's requirements, which allows ParQ to be easily used by many query systems with a little manual work.

Besides the applications on the query systems, ParQ could also apply to other similar situations where different processes require the same data.

As case studies, we applied the ParQ system to several query systems. A series of experiments were performed on a real computer cluster and ParQ showed good performance. It achieved approximately n times less memory usage to answer n queries concurrently than existing query systems. When there was limited memory, ParQ also enabled more concurrent jobs and used less processing time.

1.4 Thesis outline

In Chapter 2, we give an explicit problem definition. Chapter 3 descripts the overall design of the proposed approach, gives a detailed explanation of each method provided, and briefly introduces how to use the ParQ system. In Chapter 4, we apply ParQ to one real query systems. In Chapter 5, we design and perform a series experiments to evaluate the performance when using ParQ. Chapter 6 gives the introduction to related work, and Chapter 7 concludes the thesis.

CHAPTER 2

PROBLEM DEFINITION

With the great increase of data volume in information area, more and more applications focus on mining interesting information from massive data. Some classes of massive data are normally stored into structured datasets to be further used. For example, Freebase is an online collection of structured data. It is a global resource，which allows people to access information. A query is a request for information on a target dataset. In the example of Freebase, a query on it can be "When is the date of Kennedy's birthday?" A query system is an implementation that adopts a specific data processing algorithm and aims to answer queries on a target dataset. For an implementation, a job refers to a process that is responsible for answering a single query. In the process of answering queries on the same dataset, the data used can be divided into two parts: static data and dynamic data. Static data is used but not changed during computations. In this thesis, static data refers to the target dataset, because almost all jobs are executed on the same target dataset. For example, Facebook performs most computations on the same friendship graph [5]. Dynamic data refers to the data generated during computations. It is query-specific and constantly updated. For example, the rank score of a candidate answer is dynamic data. Therefore, a query answering process could be viewed as a computation using a set of static data and a set of dynamic data, as shown in Figure 1.

Traditionally, large data is stored in disk, such as database. However, in order to read data faster, a number of query systems load static data into memory. Since many query system frameworks are inherently designed for evaluating the performance of answering a single query, they usually tightly combine static data and dynamic data together. Therefore, they do not allow multiple jobs to share the static data in memory and each individual job has to maintain an exclusive copy of the static data in memory even if the static data is the same. Such query systems are much inefficient when

they try to process multiple queries. The inefficiencies mainly fall in two aspects: longer job completion time and large memory usage.

To be more specific, there are two approaches to answer *n* concurrent queries. First, a query system could keep one copy of the static data in memory and process the *n* queries in serial. In this situation, if the running time for completing one query is *t*, the total running time is *t*n*. Second, the system could also keep *n* copies of the static data in memory and start *n* jobs to answer n queries in parallel, as shown in Figure 2. In this case, if the memory usage for completing one query is *m*, the total memory usage for completing n queries is *m*n*, which could be a huge number. When there is limited memory, the number of running jobs is also limited and the large memory usage also results in longer running time.

The objective in this thesis is to seek a memory-efficient approach to enable a query system to answer multiple queries concurrently. One possible way is to share static data among different jobs. This approach should also ensure that 1) a query system keeps only one copy of static data among different jobs, 2) in the ideal situation, the processing time of one job will not be affected by other jobs, and that 3) it can be adopted by different query systems where datasets, queries, and implementations are all different.

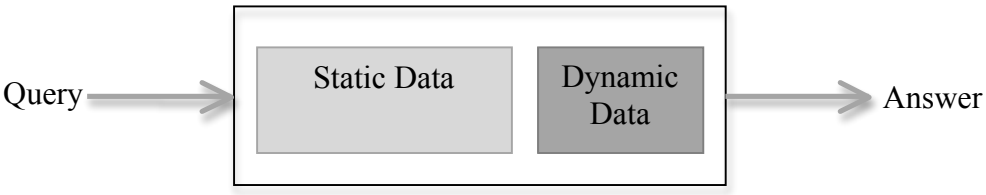


Figure 1: A job of a query system

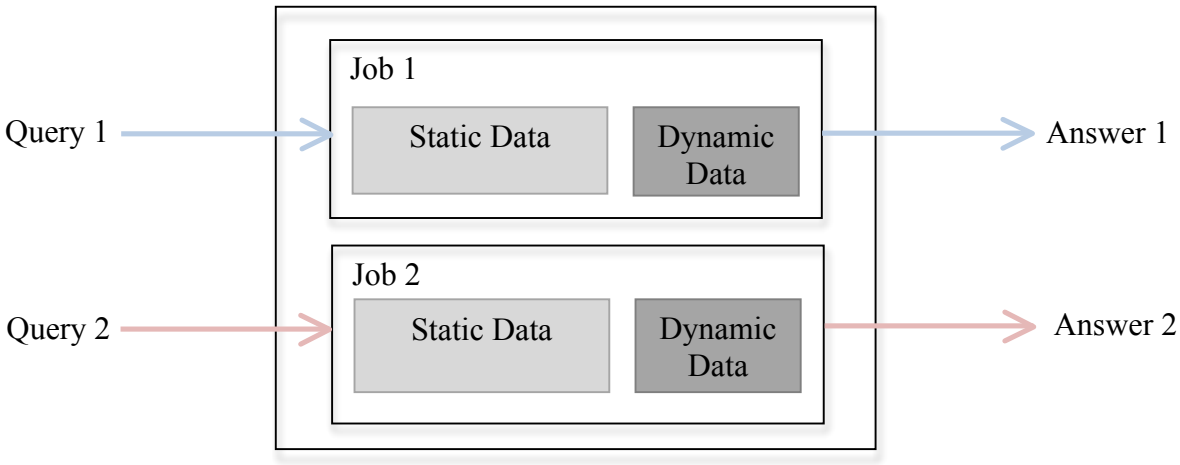


Figure 2: Answering concurrent queries with existing systems

CHAPTER 3

DESIGN AND IMPLEMENTATION OF PARQ

3.1 Overall design

The objective in this thesis is to seek a memory-efficient approach to enable a query system to concurrently answer multiple queries. Since almost all concurrent jobs use the same static data, the main idea of the proposed approach, the ParQ system, is to create a shared memory to realize static data sharing among concurrent query processing jobs.

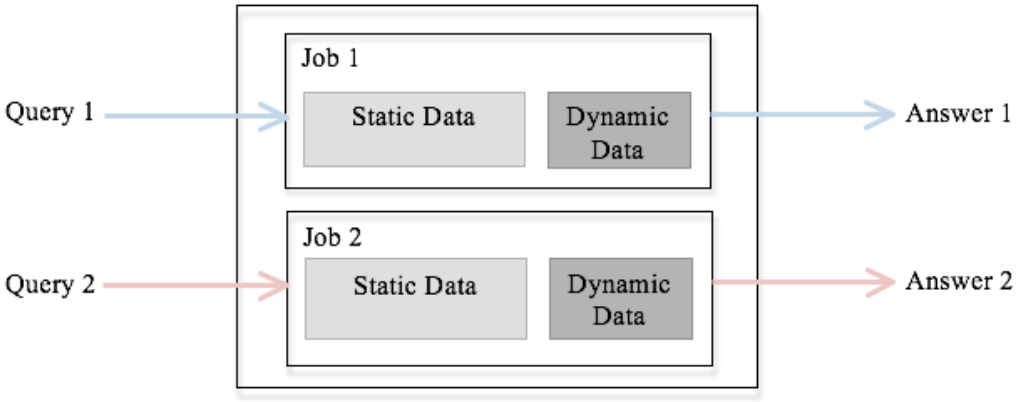
As explained in Chapter 2, a query system can start *n* jobs to answer *n* queries in parallel with *n* copies of the static data in memory. As shown in the first box in Figure 3, when a query system processes two queries on the same static dataset at the same time, two jobs are started and each job is responsible for one query. Each job accepts one query, loads a copy of static data, performs computations, returns answers, and terminates. As a result, the query system has to keep two copies of the static data in memory.

In order to share static data, the design of ParQ splits the task in the original job into two parts: data loader and query processor. A job of data loader creates a shared memory and stores the static data into the memory. A job of query processor accesses the shared memory and fetches requested data from it. ParQ starts *n* jobs of query processor to process *n* queries. Consider as Figure 3 (b) as the example. When processing two queries on the same static dataset, ParQ first starts a job of data loader, Job 0, to create a shared memory and store the static data in the shared memory. Then, it starts two jobs of query processor, Job 1 and Job 2, to handle the two queries. Job 1 and Job 2 can connect to the shared memory to get needed data. In this way, ParQ only needs to keep one copy of the static data in memory.

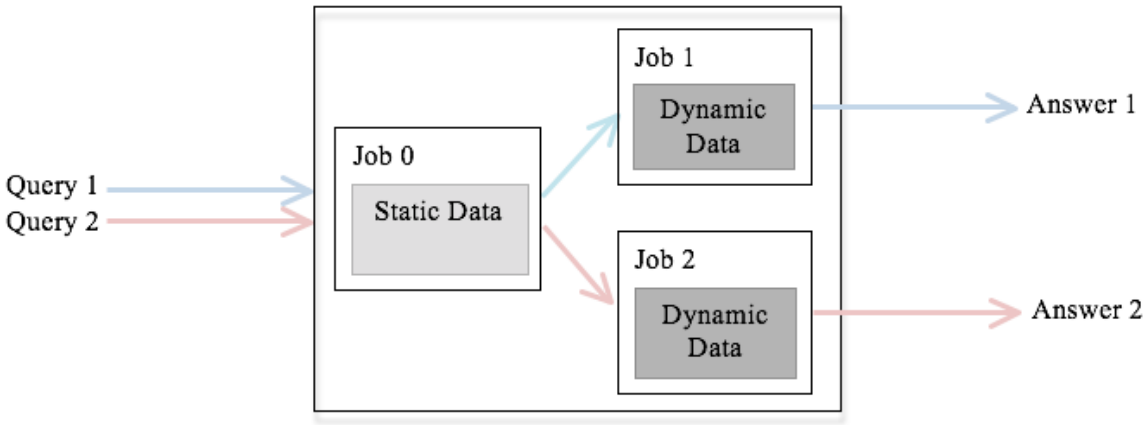
There are two main challenges in implementing ParQ. First of all, it should be easy to apply ParQ to different query systems where datasets, queries, and implementations are all different.

Different datasets have different data structures. ParQ should support diverse static data structures in order to be applied to different query systems with little manual work.

Secondly, ParQ should be able to work in a distributed environment. The shared memory storing the static data may be maintained by more than one worker. Therefore, the segments of the shared memory should be indexed properly to avoid potential errors.



(a) Query system keeps n copies of the static data



(b) ParQ keeps one copy of the static data

Figure 3: Multiple queries processing

The ParQ system contains two classes, one for data loaders to process the static data and the other one for query processors to fetch requested data. It can be applied to different query systems

and is able to work in a distributed environment. The following sections will introduce the API provided by ParQ, including the implementation and usage.

3.2 System implementation

As described in the last section, a data loader of the ParQ system enables static data sharing among different jobs. Each query processor of ParQ is responsible for one job and can connect to the shared memory to get needed data.

This section introduces the implementation of data loader and query processor and explains how to use ParQ.

3.2.1 API for data loading

Data loader creates a shared memory and stores the target static data into the memory. Data loader stores every number in a specific data type and assigns one label to each number. It can support different data formats as long as the formats conform to several requirements.

The ParQ system provides a class, DataLoader, as shown in

Figure 4. Given the input file path, the function loadData(string path) processes the data in the input file.

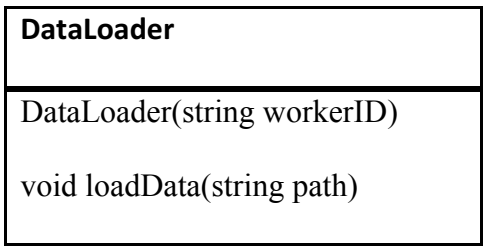


Figure 4: DataLoader class

In order to atomically process different files, ParQ requires the input files to be in a specific format. First, as Line c and Line d in Figure 5, a line represents a set of data and each tap-separated column contains a data item. The data in different lines should be organized in the same sequence, namely, the data in the same column has the same meaning and data type. The first column must be the index. An index is an integer and can specify one line. The following columns can be any data whose data type is supported. At present, the ParQ system supports “int”, “float”, “string”, “vector<int>”, “vector<float>”, and “vector<string>”. Among them, the numbers in vectors are separated by spaces.

Secondly, users should complete configuration file as shown in Figure 5. In the configuration file, users will assign labels and specify data types to corresponding columns. while the first column defaults to integer indices.

Index (Default)	Column Name (Optional)	Column Name (Optional)
int (Default)	Data Type	Data Type

Figure 5: Configuration file

For an input file that meets all the requirements, the data loader of ParQ will store all data into the shared memory in the specific data type and name the data by its index and label. For example, for an input file demonstrated in Figure 6, ParQ will process “-32959029” as an integer, store it into the shared memory, and name it as “45-NodeType”.

Index	Node Type	Neighbor
Int	int	vector<int>

45	-32959029	114661706 131784151
476	-32959029	532524246

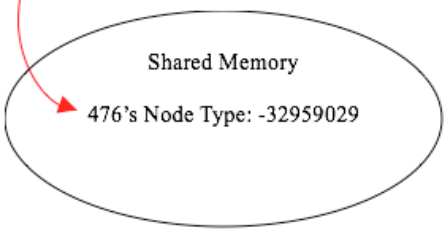


Figure 6: Store an data item into shared memory

3.2.2 API for query processing

After loading the static data, users can start query processors to process query. The ParQ system provides the QueryProcessors class, as shown in Figure 7, for query processors to access the shared memory and fetch needed data.

QueryProcessor
QueryProcessor(string workerID)
int getInt(string index, string label)
float getFloat(string index, string label)
string getString(string index, string label)
int getFromVectorInt(string index, string label, string vectorIndex)
float getFromVectorFloat(string index, string label, string vectorIndex)
string getFromVectorString(string index, string label, string vectorIndex)

Figure 7: QueryProcessor class

To fetch the target data, the users need to provide the data type, the index, and the label. Consider the input file shown in Figure 8 as the example. If a user tries to fetch the node type of node 45 from the shared memory, he/she should provide the data type and the label of the target. Since the data type is integer, the label is "NodeType", and the index is 45, the user could call the function getInt("45", "NodeType"). This function will connect to the shared memory, search for the data item, and return '-32959029'. Similarly, the user can get a float, or a string from the shared memory by using corresponding function.

If the target data is in a vector, users should also provides the vector index. For example, a user wants to get the first neighbor node of node 45. He/she should call function getFromVectorInt(string index, string label, string vectorIndex), namely getFromVectorInt("45",

"Neighbor", "1"). The user could get an item for a vector of floats, a vector of doubles or a vector of strings in the same way.

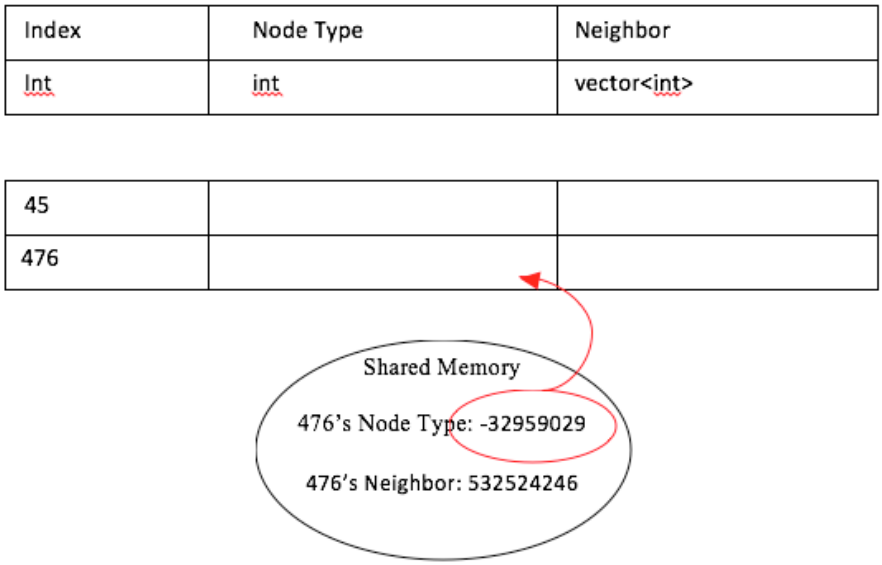


Figure 8: Fetch a data item from shared memory

3.2.3 Ability to adapt to distributed environment

The ParQ system starts a data loading job and *n* query processing jobs to process *n* queries. In distributed model, a job initializes a master and a certain number of works. The master is responsible for work scheduling and the workers share the tasks of the job.

Consider Figure 9 as the example. There are also two workers for each job. Job 0 is a data loading job and aims to load static data. Each worker of Job 0 maintains a part of the static data. Job 1 and Job 2 are query processing jobs and aim to process queries. Each worker of them shares the computations and keeps a portion of dynamic data. The dynamic data is generated during computations.

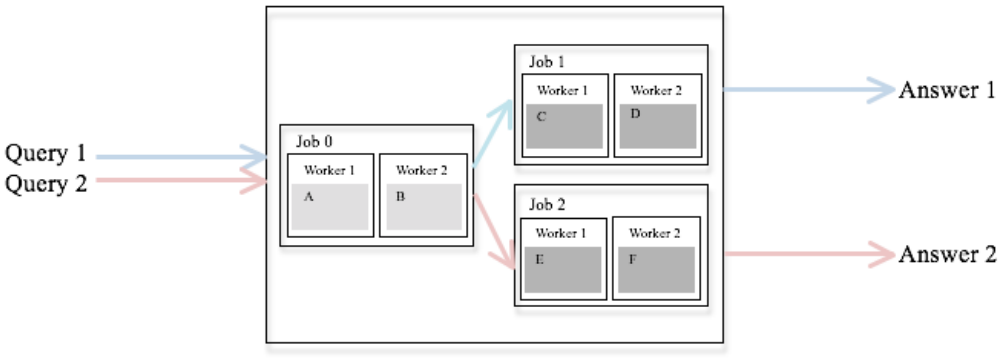


Figure 9: Two jobs in distributed model when ParQ applied

In distributed model, a large static dataset is split into several partitions based on partitioning strategies. Work *i*s are responsible for Part *i*. To be more specifically, Worker *i* of data loader creates a shared memory on a physical machine and loads Part *i* into the memory. Worker *i*s of query processors must be on the same physical machine to access Part *i*. As shown in Figure 9, A, C, and E must be on the same physical machine, so that C and E can access the shared memory creates by A. ParQ ensures Worker *i*s are assigned to the same machine by creating a map between worker ID and machine ID.

It is possible that workers with different ID are assigned to the same physical machine. In this case, by giving identifiers to data loader and query processors based on worker ID, ParQ ensures that Worker *i*s of query processors connects to the shared memory created by Worker *i* of data loader. Consider Figure 9 as the example, A, B, C, D, E, and F are assigned to the same physical machine. The memory created by A is named by "1" and the memory created by B is named by"2".

The query processor tasks on C and E are given an identifier “1”, so that C and E knows they should connects to the shared memory “1”.

3.3 Instructions for application

The ParQ system can be easily used by many query systems with a little manual work. The input static data of an implementation should be presented in required format. The task in an original job will be split into two main tasks: data loading and query processing. A job uses an *if* statement to determine whether it is a data loader or a query processor. If it is a data loader, it will create a shared memory and call corresponding functions to store the static data into the shared memory. If it is a query processor, it will access the shared memory, call corresponding functions to fetch requested data, and perform computations. The following example applications will show how ParQ can be applied to real query systems.

CHAPTER 4

APPLICATION ON EXISTING SYSTEMS

4.1 Application on the top-*k* personalized PageRank system

The ParQ system is first applied to the top-*k* PageRank system [6]. In the following sections, after a brief introduction of the top-*k* PPR system, we will show how to apply ParQ to it.

4.1.1 System specification

PageRank [7] was first proposed by Google. It is an algorithm used to rank websites for the results of their search engine. PageRank works by counting the number and quality of links to a page to get a rough estimate of how important the website page is. PageRank assumes that a more important website page is likely to receive more links from other websites [8].

Personalized PageRank is an algorithm used to obtain the items highly-relevant to a given set of facts. It has been widely used in various applications. For example, some social networking sites adopts personalized PageRank to give recommendation and relationship prediction.

The top-*k* PPR system makes personalized recommendation by implementing the Personalized PageRank computation. It aims to find *k* items that are the most relevant to a query set from an item graph. The query item set is formed based on known personal preferences. Personalized PageRank is used to measure the relevance of candidates by computing the stationary distribution of a random walk. In each computing step, the random walk reaches a random out-neighbor with probability *d* or jumps to a query node with probability *1 – d*.

The top-*k* PPR system is an extension of the Maiter framework [9]. In this system, relevance score is computed by accumulative computation in an asynchronous manner. A fraction of nodes are selected to update their scores in each pass. The chosen nodes will update its value (*v*) and

propagate the change of value (Δv) to their neighbors. The top-k PPR system uses score bounds to speed up the computation.

To process a query, the top-k PPR system will start a process，or job, that loads the graph into memory and performs an accumulative computation on the in-memory graph to measure the relevance of candidates. The computation is performed until the top-k answers are obtained.

4.1.2 Implementation process

As descripted in the last section, the top-k PPR system starts a process, or job, to process a query. The job loads the graph into memory and performs an accumulative computation on the in-memory graph to measure the relevance of candidates. The computation is performed until the top-k answers are obtained.

When answering multiple queries on the same graph, the item graph used is static and can be shared among different jobs. Therefore, we apply the ParQ system to enable in-memory data sharing among query processing jobs.

First, Figure 10 shows how an item graph should be represented in a text file to use ParQ. Each line describes one node. The first column is an integer representing the node name or the node key. The next two are the nodes attributes used in bound computation. The following column lists the node's neighbors. As explained in Section 3.2.1, the data types of the columns except the first should be specified in the first two lines.

```
        globalScore     incomingDegree  Neighbors
        float   float   vector<int>
0       30.349953       138.460022      11342 824020 867923 891835
1       0.38410074      7.46896818      53051 203402 223236 276233 552600 569212 635575 748615 862566 893884
2       3.7941647       15.35852867     30957 357310 423174 430119 462435 472889 565424 581609 597621 644135 858904
3       0.8809524       2.466473115     87562 131116 262285 290424 298887 449136 456686 718023 789669 812470
```

Figure 10: A part of an item graph for the top-k PPR system

Second, we write the code of data loader. In the original system, method *read_graph* in *struct PPRGlobalIterateKernel* is responsible for data loading. To apply ParQ, we create an instance of *DataLoader* and pass in the current worker ID as shown in Figure 11. Then, we call the method *loadData* to enable in-memory static data sharing.

```
void read_graph(string& patition_file, TypedGlobalTable<int, double, double, PPRGlobal_data > *table) {
    //..
    DataLoader loader(workerID.str());
    loader.loadData(path);
    //..
}
```

Figure 11: Code segment 1

Third, we write the code for query processing. As shown in Figure 12, we create an instance of *QueryProcessor* in *struct PPRGlobalIterateKernel*. As shown in Figure 12, in *struct PPRGlobalIterateKernel*,we define a variable with type *QueryProcessor*, create an instance of *QueryProcessor*, and pass in the current worker ID. The function *g_func* and *dynamic_incbound* are responsible for processing query. For these two functions, we call corresponding methods to access the shared memory and fetch the requested data.

```
struct PPRGlobalIterateKernel : public IterateKernel<int, double, PPRGlobal_data > {
    //..
    QueryProcessor* processor;
    //..
    PPRGlobalIterateKernel() {
        //..
        processor = new QueryProcessor(workerID.str());
        //..
    }

}


void g_func(const int& key, double& value, double& delta, PPRGlobal_data & data, vector<pair<int, double> >* output) {
    //..
    int neibor_size = processor->get_vecsize(key_str.str().c_str(), "3");
    //..
}


double dynamic_incbound(const int& key, double dv, double maxdv, double sumdv, double hopbound, PPRGlobal_data & data) {
    //..
    float globalScore = processor->getFloat(key_str.str().c_str(), "1");
    float incomingDegree = processor->getFloat(key_str.str().c_str(), "2");
    //..
}
```

Figure 12: Code segment 2

Lastly, we add an *if* statement in method *read_graph* to determine whether the present job is a data loading job or a query processing job. As shown in Figure 13, if the current job is responsible for data loading, it will perform the tasks of data loader. If the current job is for query processing, it will perform the tasks of query processor.

```
void read_graph(string& patition_file, TypedGlobalTable<int, double, double, PPRGlobal_data > *table) {
    //..
    if(FLAGS_is_dataLoader) {
        DataLoader loader(workerID.str());
        loader.loadData(path);
    } else {
        //Process the query.
    }
    //..
}
```

Figure 13: Code segment 3

With the application of ParQ, we can start a data loader job to enable static data sharing and *n* query processors to process *n* queries concurrently.

4.2 Application on the top-*k* star query system

The ParQ system is also applied to the top-*k* star query system [10]. In the following sections, we will first introduce the top-*k* star query system and then present how to apply ParQ to it.

4.2.1 System specification

Massive information networks, such as Freebase, a real-life knowledge graph, contain billions of labeled entities, such as a person. Star queries is an approach used to identify an unknown entity, based on its relationship with other know entities. For example, in Freebase, star query can be used to search for an actor in the movie Cloud Atlas who has worked with the director Steven Spielberg.

An implementation of star query, the top-*k* star query system, answers pattern match queries in massive information networks [11]. A massive information network describes labeled entities and the relationships among the entities. An entity can be a person or a thing with one attribute, type. On a

massive information network, a star query aims to identify an unknown entity based on known facts. When representing the network as a graph, the star query problem can be modeled as a pattern matching problem.

In this problem, a query describes a known node, the type of an unknown node, and their relationship. The top-*k* star query system aims to find the best *k* answers for a query from a graph. The candidates are ranked by their relevance to the known nodes. The relevance score between two nodes is computed by

$$\varphi(u,v) = \begin{cases} 1 & u = v \\ \min\{N, n_{u,v}\} \cdot \alpha^{l_{u,v}} & otherwise \end{cases},$$

where α is a damping factor having the value between 0 and 1. $l_{u,v}$ is the length of the shortest path between node *u* and *v*. $n_{u,v}$ is the number of the shortest paths. *N* is a constant used to bound the value of $n_{u,v}$. That is, the relevance between two nodes is quantified by the length of the shortest path and the number of the shortest paths between them. The top-k star query system uses breadth-first search to traverse a target graph to compute $n_{u,v}$ and $l_{u,v}$. It also adopts a bounding technique to speed up the computation. The bounding technique can detect the top-*k* answers without having to compute converged scores.

To answer a query, the top-*k* star query system will start a process, or job, that loads the graph into memory and performs multiple BFSs on the target graph to measure relevance scores of candidates until the top-*k* answers are obtained.

4.2.2 Implementation process

As descripted in the last section, the top-*k* star query system starts a job to answer a query. The job loads the target graph into memory and performs multiple BFSs on the in-memory target graph to measure relevance scores of candidates until the top-*k* answers are obtained.

When answering different queries on the same target graph, the graph is the same, or static, and can be shared among different jobs. Therefore, we apply the ParQ system to the top-k star query system to enable in-memory static data sharing.

First, we represent the target graph in a text file to apply the ParQ system, as shown in Figure 14. Each node is represented in one line. The first column is an integer representing the node name or the node key. The second column is the node type. The third column lists the node's neighbors. As explained in Section 3.2, the first column is always an integer and used as index. The labels and the data types of the following columns are specified in the first two lines.

```
        type    neighbors
        int     vector<int>
3177    -100    3098 2407 3052 5437 2073 3310 4382
3173    -100    3172
3169    -100    2441 3973 4194 6805 3702 3874 3900 5962 1746 4454 1445 6860 3168 2212
3097    -100    3096 3205 3009 4669 6430
```

Figure 14: A part of an item graph for top-k star query system

Second, we modify the code for data loader. Method *initDataGraph* in class *BoundWorker* is responsible for graph initializing. As shown in Figure 15, in this function we create an instance of *DataLoader* and pass in the current worker ID. Then, we call method *loadData* to create a shared memory and load the static data into the memory.

```
void BoundWorker::initDataGraph(){
    //..
    DataLoader loader(workerID.str());
    loader.loadData(path.str());
    //..
}
```

Figure 15: Code segment 1

Third, we write the code of query processor. Method *runIter* in class *BoundWorker* is responsible for computations. As shown in Figure 16, we define a variable with type *QueryProcessor* in the header file DBRWorker.h. We create an instance of *QueryProcessor*, pass in the current worker ID in *setMaiter*, and call functions in *runIter* to connect to the shared memory and get requested data.

```
class BoundWorker : public DSMKernel, public NetworkController{
    //..
    QueryProcessor* processor;
    //..
};

void BoundWorker::setMaiter(MaiterKernel<uint, BoundList, TargetNode*>* inmaiter) {
    //..
    processor = new QueryProcessor(workerID.str());
    //..
}

void BoundWorker::runIter(int tabId, const uint& k) {
    //..
    QueryProcessor* p = this->processor;
    int node_type = p->get_int(key.str().c_str(), "1");
    int neibor_size = p->get_vecsize(key.str().c_str(), "2");
    //..
}
```

Figure 16: Code segment 2

Lastly, as shown in Figure 17, we add an if statement in the function *Run* to determine whether the current job is a data loader or a query processor. If the current job is a data loader, it will load the static data and terminate. If the current job is a query processor, it will skip data loading and start to process query.

```
void DBRMaster::Run(Master* master){
    //..
    if(FLAGS_is_dataLoader){
        //Load the static data.
    }else{
        //Process the query.
    }
    //..
}
```

Figure 17: Code segment 3

With the application of ParQ, we can start a data loading job and *n* query processing job to process *n* queries concurrently.

4.3 Application on the belief propagation system

In order to further study the performance of the ParQ system, we also applied it to the belief propagation system [12]. The following sections introduce the introduction of that system and present how to apply ParQ to it.

4.3.1 System specification

Belief propagation was first used in artificial intelligence and information theory and has demonstrated great use in many areas. For example, belief propagation can be used to predict the probability of one event based on the probabilities of given events, given the probabilistic interactions between them.

The belief propagation system performs approximate inference on probabilistic graphical models. Probabilistic graphical models use a graph-based representation to capture uncertainty in real-world applications. In this graphical representation, the nodes correspond to the variables in the real world. The edges represent direct probabilistic interactions between them. The query describes a set of variables

In this example, we consider a typical probabilistic graphical models, factor graphs, since any other graphical models can be converted to factor graphs. There are two types of nodes in factor graphs: variable nodes and factor nodes. Each variable node represents a random variable and each factor node describes a specific function that maps variable nodes to non-negative real-valued number.

The belief propagation system here adopts the sum-product algorithm, which answers query by computing marginal probabilities of factor graphs. In order to reduce scheduling cost, the belief

propagation system calculates the priority of nodes and selects a set of messages to update at a time via the priority. In each steps, from the chosen node, the computation reaches the out-neighbors. It keeps propagating messages in both directions along edges until a stable situation is reached.

The belief propagation system extends the Maiter framework [9]. In this system, the message is computed by accumulative computation in an asynchronous manner. A fraction of nodes are selected to update their scores in each pass. The chosen nodes will update its value (v) and propagate the change of value (Δv) to their neighbors.

To process a query, the belief propagation system will start a process, or job, that loads the factor graph into memory and performs an accumulative computation on the in-memory graph to measure the value of variables. The computation is performed until it reaches a stable situation.

4.3.2 Implementation process

As descripted in the last section, the belief propagation system starts a job to perform approximate inference on probabilistic graphical models. The job loads the target graph into memory and performs an accumulative computation on the in-memory graph to measure the value of variables until the values reach a stable situation.

When answering different queries on the same target graph, the graph is the same, or static, and can be shared among different jobs. Therefore, we apply the ParQ system to the top-k star query system to enable in-memory static data sharing.

First, we represent the target graph in a text file to apply the ParQ system, as shown in Figure 18. Each node is represented in one line. The first column is an integer representing the node name or the node key. The second column lists the node's neighbors. As explained in Section 3.2, the first column is always an integer and used as index. The labels and the data types of the following columns are specified in the first two lines.

Second, we modify the code for data loader. Method *read_data* is responsible for graph initializing. As shown in Figure 19, in this function we create an instance of *DataLoader* and pass in the current worker ID. Then, we call method *loadData* to create a shared memory and load the static data into the memory.

```
void read_data() {
        //..
        DataLoader loader(workerID.str());
        loader.loadData(path.str());
        //..
}
```

Figure 18: Code segment 1

Third, we write the code of query processor. Method *c_fun* and *u_fun* are responsible for computations. As shown in Figure 16, we call functions in *runIter* to connect to the shared memory and get requested data.

```
QueryProcessor* p = this-> processor;
int node_type = p->get_int(key.str().c_str(), "1");
int localstatenum = p->get_int(key.str().c_str(), "2");
int outstatenum = p->get_int(key.str().c_str(), "3");
int neibor_size = p->get_vecsize(key.str().c_str(), "4");
```

Figure 19: Code segment 2

Lastly, we add an if statement in the function *Run* to determine whether the current job is a data loader or a query processor. If the current job is a data loader, it will load the static data and terminate. If the current job is a query processor, it will skip data loading and start to process query.

With the application of ParQ, we can start a data loading job and *n* query processing job to process *n* queries concurrently.

CHAPTER 5

EVALUATION

5.1 Overview

The ParQ system has been applied to the top-*k* personalized PageRank system and the top-*k* star query system. In order to examine the performance, we designed and performed a series of experiments, which includes running time and memory usage. For one query system that uses ParQ, multiple queries can be processed in parallel with one target graph in memory. We compare ParQ with the original system running in two ways. First, the original system can process multiple queries in serial with one target graph in memory (or "ONE CPY" for short). Alternatively, it can process queries in parallel but with multiple graphs in memory (or "DUPL CPY" for short).

5.2 Performance of the top-*k* personalized PageRank system using ParQ

We first evaluate the performance of the top-k PPR system adopting ParQ. In the following sections, after a brief introduction of the top-k PPR system, an experiment overview and experimental results are presented.

5.2.1 Experiment settings

We performed the experiments on a local computer cluster. The local cluster consisted of four machines connected by a 1Gb Ethernet switch. Each machine had 16GB RAM and one 1.86GHz CPU.

We used a set of queries on a 916K-line graph. The graph is a web graph that describes the directed links between webpages. The queries are personal preferences.

5.2.2 Running time

Figure 20 shows the comparison of the running time of the approaches for completing queries. First, it can be seen that for ParQ and Parallel/multi-graph, as the number of queries increases, the running time increases slightly. This is because that there is enough free memory as shown in Figure 21, since the top-k PPR system requires less memory to process one query.

Second, the results show that running time of ParQ is smaller than ONE CPY. The first reason is that ParQ processes the queries concurrently while ONE CPY processes queries in serial. The second reason is ONE CPY uses a long time to load the static data before processing queries, while ParQ can directly use the static data in the shared memory.

Third, the results also show that the running time of ParQ is smaller that DUPL CPY. This is because of limited free memory. In order to processes *n* queries concurrently, DUPL CPY keeps *n* copies of the static data in memory. When there is limited free memory, the running time of DUPL CPY increases due to increased workload. The second reason is that DUPL CPY also uses a long time to load the static data for each query.

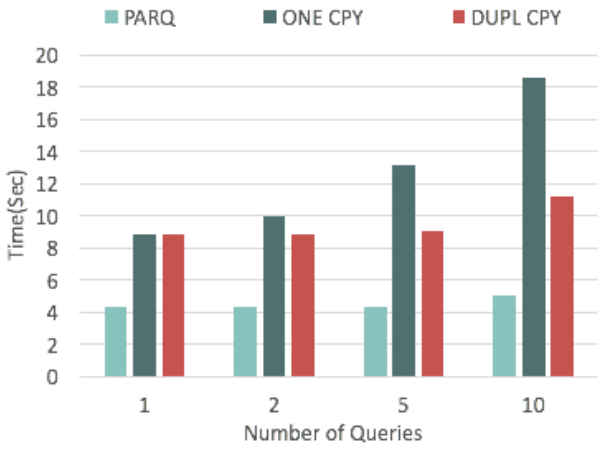


Figure 20: Running time for completing all queries

5.2.3 Memory usage

Figure 21 shows the comparison of the memory usage of the three approaches. First, the results show that ParQ requires less memory than DUPL CPY because ParQ keeps only one copy of the static data in memory while DUPL CPY needs *n* copies to process *n* queries.

Second, it can be seen that ParQ requires more memory than ONE CPY. Since each query job needs to maintain a set of dynamic data generated during computations, ParQ keeps *n* sets of dynamic data in memory to process n queries concurrently, while ONE CPY keeps only one set of dynamic data at a time because it processes *n* queries in serial.

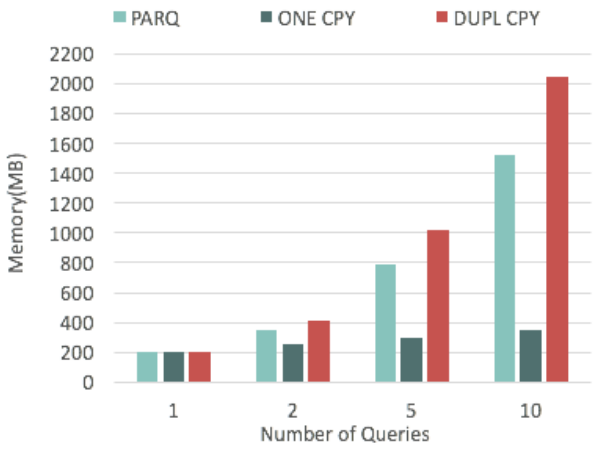


Figure 21: Memory usage

5.3 Performance of the top-*k* star query system using ParQ

The proposed approach is also applied and evaluated on the top-*k* star query system. What shall be introduced in later sections are a brief system introduction, an experiment overview, and experiment results.

5.3.1 Experiment settings

We performed the experiments on the same local computer cluster. The local cluster consisted of four machines connected by a 1Gb Ethernet switch. Each machine had 16GB RAM and one 1.86GHz CPU.

We used a set of queries on a 548K-line graph. The graph describes the collaboration among authors. The queries consisted of a known node set, an unknown node, and their relationship.

5.3.2 Running time

Figure 22 shows the performance comparison of the three approaches, focusing on the running time for completing various numbers of queries. First, the results show that for ParQ and Parallel/multi-graph, as the number of queries increases, the running time increases as well due to increased workload.

Second, it can be seen that the running time of ParQ is smaller than ONE CPY. This is because ParQ processes the queries concurrently while ONE CPY processes queries one by one.

Third, the results show the running time of ParQ is also smaller that DUPL CPY. When there are 10 queries, the running time of DUPL CPY is large. The reason is that although DUPL CPY processes 10 queries concurrently, it keeps 10 copies of the static data in memory. When there is limited free memory, the running time of DUPL CPY increases sharply.

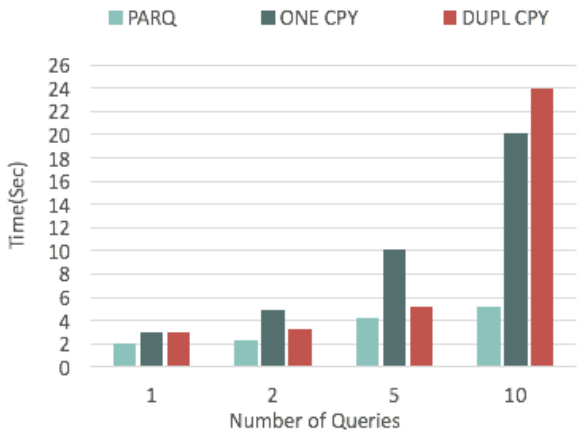


Figure 22: Running time for completing all queries

5.3.3 Memory usage

Figure 23 shows the performance comparison of the approaches, focusing on the memory usage for completing different numbers of queries. First, the results show that ParQ uses less memory than DUPL CPY, because ParQ keeps only one copy of the static data in memory while DUPL CPY requires *n* copies to process *n* queries.

On the other hand, the results show that ParQ requires more memory than ONE CPY. Each query job needs to maintain a set of dynamic data generated during computations. Since ParQ processes *n* queries concurrently, it keeps *n* sets of dynamic data in memory, while ONE CPY keeps only one set of dynamic data at a time because it processes n queries in serial.

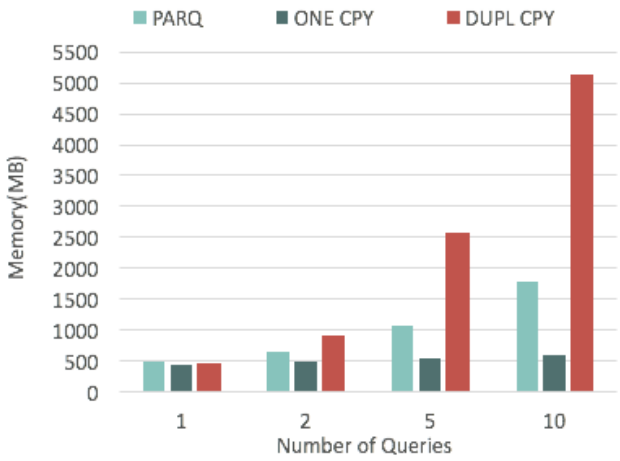


Figure 23: Memory usage

5.4 Performance of the belief propagation system

At last, the proposed approach is applied and evaluated on the belief propagation system. What shall be introduced in later sections are a brief system introduction, an experiment overview, and experiment results.

5.4.1 Experiment settings

We performed the experiments on the same local computer cluster. The local cluster consisted of four machines connected by a 1Gb Ethernet switch. Each machine had 16GB RAM and one 1.86GHz CPU.

We used a set of queries on a 908K-line graph. The graph describes direct probabilistic interactions between given real facts.

5.4.2 Running time

Figure 24 shows the job completion time comparison of the three approaches. First, the results show that for ParQ and DUPL CPY approach, as the number of queries increases, the running time increases as well due to increased workload.

Second, it can be seen that the running time of ParQ is less than which of ONE CPY. The reason is that ParQ processes the queries concurrently while ONE CPY processes queries one by one.

Third, the results also show that the running time of ParQ is less that DUPL CPY as well. Especially when there are 10 queries, the running time of DUPL CPY is large. The reason is that although DUPL CPY processes 10 queries concurrently, it keeps 10 copies of the static data in memory. When there is limited free memory, the running time of DUPL CPY increases sharply.

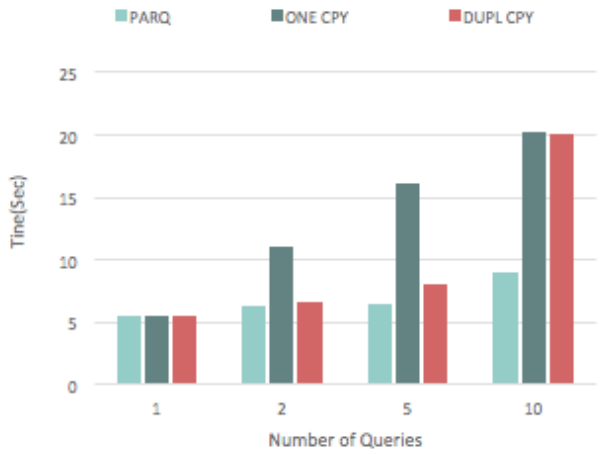


Figure 24: Running time for completing all queries

5.4.3 Memory usage

Figure 25 shows the performance comparison of the approaches, focusing on the memory usage for completing different numbers of queries. Similarly, first, the results show that ParQ uses less

memory than DUPL CPY, because ParQ keeps only one copy of the static data in memory while DUPL CPY requires *n* copies to process *n* queries.

On the other hand, the results show that ParQ requires more memory than ONE CPY. Each query job needs to maintain a set of dynamic data generated during computations. Since ParQ processes *n* queries concurrently, it keeps *n* sets of dynamic data in memory, while ONE CPY keeps only one set of dynamic data at a time because it processes n queries in serial.

However, compared with the performance of the PageRank system and the star query system, ParQ here requires relative much more memory than ONE CPY and is slightly better than DUPL CPY. Recall that the belief propagation system calculates the priority of nodes and selects a set of messages to update at a time via the priority. In each steps, from the chosen node, the computation reaches the out-neighbors. It will visit all graph nodes before the computation reaches a stable situation is reached while the previous two systems will visit a small fraction of nodes. The dynamic data of each query processing job will be relative much more.

As a conclusion, ParQ is more suitable in the situations when the size of static data is much larger than the size of dynamic data, although it is still useful in the opposite situations.

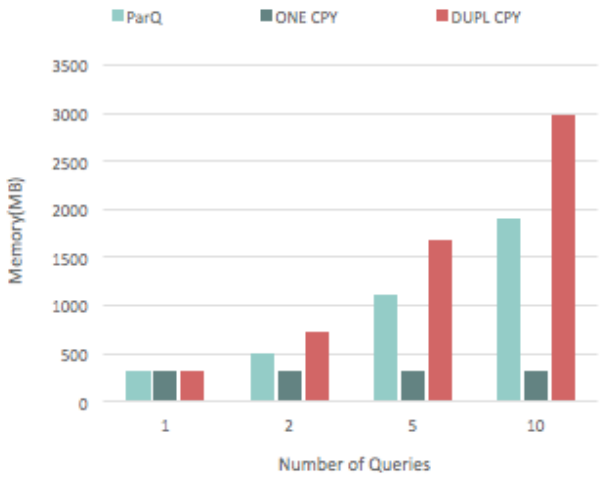


Figure 25: Memory usage

CHAPTER 6

RELATED WORK

As the volume of data increases sharply, more and more researches focus on how to process and analyze large-scale data. Many distributed computing works, such as Dryad [13] [14] and MapReduce [15], have been proposed to process large data in a cluster of machines or in a cloud environment. A lot of frameworks have also been designed and implemented for complementing existing functionality and for accelerating large-scale iterative computations.

Dryad is a computing framework for general-purpose, distributed, and data-parallel applications. An application of Dryad generates a dataflow graph by combining computational vertices with communication channels. It handles the vertices on a cluster of computers which communicates with each other. The Dryad application can make efficient use of the available resources by discovering the size and placement of data at run time and modifying the graph as the computation progresses.

MapReduce is a programming model for generating and processing big data sets by adopting a parallel and distributed algorithm in a cluster of machines or in a cloud environment. A MapReduce program is composed of mapper procedure and reducer procedure [16]. Mapper procedure performs filtering and sorting. For example, in order to count the words in a large file, mapper will sort words by counting into queues, one queue for each word. Reducer performs a summary operation, such as counting the total number of words in each queue and yielding word frequencies. MapReduce has been widely used as the big data processing model and its libraries have been written in many programming languages, such as Java and C++. MapReduce also provides different levels of optimization, including scalability and fault-tolerance for a variety of applications.

The competition advantage of MapReduce reflects in reducing running time with multi-threaded implementations. A single-threaded MapReduce program is usually not faster than a non-

MapReduce program. To design a distributed model, one challenge is how to improve fault-tolerance and to reduce network communication cost among the machines in the cluster. The open-source implementation, Apache Hadoop [17], is a good complementary to MapReduce that supports optimized distributed operation. Another challenge is how to improve the data processing speed. To address this challenge, a series of frameworks, have been proposed for accelerating computations by adopting large-scale iterative algorithms. What's more, many practical problems concern large graphs. For example, the scale of large graphs, like web graphs and social networks, poses challenges to efficient processing. The significant increasing need to process and analyze large volume of graph-structured data leads to lots of recent researches on parallel frameworks, such as Pregel [18], Giraph [19], GraphLab [20]. Those frameworks are the most advantageous complement to MapReduce in order to process graph-structured data.

The Pregel framework is similar to MapReduce, but with a natural graph API and more efficient support for iterative computations over the graph. Programs of Pregel are performed as a sequence of iterations, within which a vertex sends messages to its neighbors, receive messages sent in the previous iteration, and update its internal state. This vertex-centric approach applies to varies practical applications. Pregel is for synchronous, fault-tolerant, and distributed implementations and is easy to program.

Giraph is an iterative graph processing framework designed for high scalability. Giraph is an important counterpart to Pregel with several more features beyond Pregel, including master scheduling and edge-oriented computation. Giraph is currently used at Facebook to process and analyze their social network graphs.

The GraphLab framework is for parallel and iterative implementations. GraphLab was first designed for machine learning tasks, but it has been widely used in varies areas related to data-

mining. The implementation of GraphLab usually concern sparse data with local dependencies, iterative algorithms, and asynchronous execution.

MapReduce and other related framework introduced above store intermediate results in Hadoop Distributed File System [21], which requires much longer time to access and update data. Therefore, MapReduce framework are not suitable when implementations need to visit intermediate results frequently. In order to solve this problem, many new frameworks are proposed, such as Piccolo [22] and Maiter [9], which store intermediate data in memory instead of disk.

Piccolo is a data-centric programming framework for writing parallel in-memory applications on clusters. It allows computation running on different machines to share distributed and mutable state via an in-memory key-value table. The Piccolo framework is designed for efficient application implementations. It also provides fault-tolerance and is easy to program.

Maiter is also a data-centric framework that is designed and implemented by modifying Piccolo. It adopts accumulative iterative update algorithm, which accelerates normal large-scale iterative computations. Maiter framework contains a master and a certain number of works. The master coordinates the workers and monitors the status of workers. Those workers run in parallel and communicate with each other through MPI [23].

Although the implementations of such systems can process single graph-processing job efficiently, they required high cost when process multiple concurrent jobs. The reason is that these programs do not allow multiple jobs to share the in-memory graph data, which results in each job needs to maintain their own separate graph data in memory.

There are also many new frameworks proposed to solve this problem. For example, Seraph [5] is designed and implemented based on a decoupled data model, which allows multiple concurrent jobs to share graph structured data in memory. Seraph supports good fault-tolerance. Specifically, it

adopts a copy-on-write semantic to isolate the data change of concurrent jobs and uses snapshots to maintain a consistent graph for jobs submitted at different time.

However, Seraph is an independent framework, the implementations of Maiter cannot use it to achieve in-memory data sharing. Therefore, many implementations of Maiter, such as the fast approach for top-k path-based relevance query [6] and the index-free approach for top-k star queries [10], cannot adopt Seraph to realize query-level parallelism.

CHAPTER 7

CONCLUSION

Query system refers to an implementation that adopts a data processing algorithm to answer queries on a dataset. Many existing query systems are good when processing a single query, but still has problem when processing multiple queries. That is, they do not allow multiple jobs to share the in-memory graph data, which results in large memory usage and longer job completion time.

This thesis aims to design and implement a memory-efficient system, ParQ, which could be adopted by query systems to realize query-level parallelism. The main idea includes constructing a common memory block for maintaining sharable data. By sharing data, ParQ is able to process multiple queries concurrently while reducing memory usage and running time.

ParQ has the ability to work in distributed environment. We also offer a standard interface. It provides the ability to automatically process data in different formats as long as the formats conform to the system's requirements, which allows ParQ to be easily used by many query systems with a little manual work.

REFERENCES

[1] Knowledge Graph: https://en.wikipedia.org/wiki/Knowledge_Graph. Retrieved 11 June 2016.

[2] Knowledge Graph: http://www.google.com/insidesearch/features/search/knowledge.html. Retrieved 11 June 2016.

[3] Freebase: https://en.wikipedia.org/wiki/Freebase. Retrieved 28 May 2016.

[4] Freebase Data Dumps: https://developers.google.com/freebase/data. Retrieved 28 May 2016.

[5] Xue, J., Yang, Z., Qu, Z., Hou, S., & Dai, Y. (2014, June). Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing* (pp. 227-238). ACM.

[6] Khemmarat, S., & Gao, L. (2016). Fast top-k path-based relevance query on massive graphs. IEEE Transactions on Knowledge and Data Engineering, 28(5), 1189-1202.

[7] PageRank: https://en.wikipedia.org/wiki/PageRank. Retrieved 29 May 2016.

[8] Facts about Google and Competition: http://web.archive.org/web/20111104131332/http://www.google.com/competition/howgooglesearchworks.html. Retrieved 12 August 2016.

[9] Zhang, Y., Gao, Q., Gao, L., & Wang, C. (2012, June). Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date* (pp. 13-22). ACM.

[10] Jin, J., Khemmarat, S., Gao, L., & Luo, J. (2014, December). A distributed approach for top-k star queries on massive information networks. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)* (pp. 9-16). IEEE.

[11] Cheng, J., Zeng, X., & Yu, J. X. (2013, April). Top-k graph pattern matching over large graphs. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (pp. 1033-1044). IEEE.

[12] Yin, J., & Gao, L. (2014, November). Scalable Distributed Belief Propagation with Prioritized Block Updates. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management* (pp. 1209-1218). ACM.

[13] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K., & Currey, J. (2008, December). DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI* (Vol. 8, pp. 1-14).

[14] Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D. (2007, March). Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review* (Vol. 41, No. 3, pp. 59-72). ACM.

[15] MapReduce: https://en.wikipedia.org/wiki/MapReduce. Retrieved 21 August 2016.

[16] Lam, C. (2010). *Hadoop in action*. Manning Publications Co.

[17] Apache Hadoop: https://en.wikipedia.org/wiki/Apache_Hadoop. Retrieved 12 August 2016.

[18] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010, June). Pregel: a system for large-scale graph processing. *In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (pp. 135-146). ACM.

[19] Martella, C., Shaposhnik, R., & Logothetis, D. (2014). *Giraph in Action*. Manning.

[20] Low, Y., Gonzalez, J. E., Kyrola, A., Bickson, D., Guestrin, C. E., & Hellerstein, J. (2014). Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*.

[21] Mackey, G., Sehrish, S., & Wang, J. (2009, August). Improving metadata management for small files in HDFS. In *2009 IEEE International Conference on Cluster Computing and Workshops* (pp. 1-4). IEEE.

[22] Power, R., & Li, J. (2010, October). Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI* (Vol. 10, pp. 1-14).

[23] MPI: https://www.open-mpi.org. Retrieved 21 August 2016.