SEDGE: A SELF EVOLVING DISTRIBUTED GRAPH PROCESSING ENVIRONMENT

User Guide, version 0.10.0, 05/23/2012

Shengqi Yang

Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106-5110, USA
sqyang@cs.ucsb.edu

Overview

Sedge is a software framework that supports large scale graph processing. Essentially, Sedge is inspired by Google's Pregel. In Sedge, the input graph is usually split into several non-overlapping parts which are distributed to workers and then processed in a distributed manner. In general, the applications running in Sedge consists of several supersteps, in each of which a vertex can receive messages sent in the previous iteration, execute user-defined functions and send messages to other vertices. This vertex centric approach has been shown to be comprehensive and efficient enough to express a broad set of graph algorithms.

Installation

Pre-requisite

- **Java 1.6.x** or greater
- **ssh** should be installed and well configured among all the nodes so that Sedge can access any node without entering password.

Download Sedge

Please download our latest version from our website.

Configuration

To start using Sedge, you need first to personalize the following configuration files with respect to your cluster environment.

conf/sedge-env.sh

Parameter	Value	Notes
JAVA_HOME	The complete path of Java SDK	Required
SEDGE_CLASSPATH	Java classpath	Optional

SEDGE_HEAPSIZE	Java heap size	Optional (default 1000m)

conf/config.properties	
Parameter	**Value**
HOST	Name of master node
SEDGE_ROOT_DIR	The complete path of Sedge root directory
SEDGE_TEMP_DIR	Root path where Sedge store temporary files
SEDGE_DATA_DIR	Path where Sedge store temporary data
SEDGE_JARS_DIR	Path where Sedge store temporary jars
MAX_VTX_ID	Maximum vertex id of the graph
MAX_VTX_DUPS	Maximum number of duplicate copies for each vertex
DATA_PORT	Data transmission port
JOB_PORT	Job scheduling port
MSG_PORT	Message transmission port
SEARCH_PORT	Search service port
MONITOR_PORT	Monitoring service port
LINE_SPTOR	Line separator for input graph file

conf/workers	
List of workers	One worker per line

Sedge installation

Once you have properly configured the above files, installing Sedge is pretty easy.

```
[user@master sedge-0.10.0]$ bin/install
```

Notice that you should install Sedge from the master node. In order to execute the command above, you ***don't*** need a root authority.

Sedge commands

All the functional routines in Sedge can be executed via **bin/sedge** command. To see what routines Sedge provides, you can run script **bin/sedge** without any arguments.

```
[user@master sedge-0.10.0]$ bin/sedge
  Usage: sedge COMMAND
  where COMMAND is one of:
    version          print the version
    jar              run a jar file
    stop             stop the current job
    get              get the result from workers
    ls               list the content in Sedge data repository
    less             view the dst file in Sedge data repository
```

put	copy single/multiple srcs from local to Sedge
rm	delete files
dist	distribute file/dir to workers
clean	clean the Sedge data repository in workers
gen	graph generation
part	graph partitioning
job-info	job information

version

print the version and new updates

Usage: sedge version

put

copy file or dir from local file system to the Sedge data repository.

Usage: sedge put [option] <src> <dst>

Options:

-s, put graph splits

-m <map>, put graph according to partition map

ls

Lists the file/dir in the temporary data directory

Usage: sedge ls <arg>

less

view a file in Sedge data repository

Usage: sedge less <arg>

rm

Deletes files specified as args

Usage: sedge rm <dst>

send

send local file/directory to workers

Usage: sedge send <src> <dst>

clean

Cleans all the data in sedge temporary data directory

Usage: sedge clean

jar

Runs a jar file.

Usage: sedge jar <jar file> <main class> <args>

get

Gets results from workers

Usage: sedge get <src> <dst>

stop

Stops current job

Usage: sedge stop

gen

Generates a graph from edges

Usage: sedge gen <graph> <directed> <weighted> <result>

Arguments:

<graph>	input graph (edges)
<directed>	to generate a directed graph (“true”) or not (“false”)
<weighted>	to generate a weighted graph (“true”) or not (“false”)
<result>	graph output dir

part

Partitions a graph

Usage: sedge part <graph> <parts> <balance> <result>

Arguments:

<graph>	input graph
<parts>	number of partitions
<balance>	partition balance (e.g. 0.02)
<result>	graph partition result (vertex-partition map)

job-info

Prints job running information

Usage: sedge job-info

Sedge (Pregel) Tutorial

Input graph file format (Adjacent list format)

The primary input of Sedge is a graph which is stored in a plain file (or multiple files). Given a graph with *n* vertices, the input file/files should contain *n* lines. Each line in the file contains information for each vertex of the graph. A typical input graph file may be like

```
[user@master ~]$ less test.graph
1,2,3,4,5,6,7,8
2,3,4
3,7,10,13
5,1,3,6,10
……
```

Specifically, the first value in each line should be a vertex id, such as 1, 2, 3 and 5. The rest ids in each line are the neighbors of the first id. For example, vertex 3 and 4 in the 2nd line are the neighbors of vertex 2. It can be seen from the above example that the *i*th line

is not necessary to exclusively represent the information of the vertex *i*, such as the 4th line in the above example.

It is worth noticed that the vertex id should be an integer ranging from 1 to *MAX_VTX_ID* defined in *conf/config.properties*. However, the vertex ids are not necessary to be consecutive. Besides neighbors, each vertex (line) can contain other information, such as vertex attributes. The *FileInputer* interface enable users define personalized interpretation of each input line.

Graph generation

Sedge also provides a graph generation routine that can transform a file in the edge list format to the adjacent list format. For instance, a graph file may be as

```
[user@master ~]$ less test2.graph
1,2
1,3
……
5,10
```

To transform the edge list format to the adjacent list format, you can use the following commands

```
[user@master sedge-0.10.0]$ bin/sedge put ~/test2.graph test
[user@master sedge-0.10.0]$ bin/sedge gen test true false result
```

where the “true” and “false” in the second command indicate the generated graph will be a directed unweighted graph.

Sedge user interface

The following content provides the primary user interfaces of Sedge, which help you to implement and configure your own applications in a fine-grained manner.

Vertex

Each vertex of the input graph is referred to as a **Vertex** in Sedge. The Vertex in Sedge is a abstract class. To develop your own application, you can start with defining/subclassing the **Vertex** class. Under the simplest case, the only thing you have to do is to override the abstract **compute()** method. In each superstep, the **compute()** method of active vertex will be invoked by workers. The **compute()** method defined in **Vertex** is as follows

```
public abstract boolean compute();
```

The return value (boolean) is used for voting to be active (*true*) or inactive (*false*) in the next superstep. We encourage you to always return false in your **compute()** method since a vertex can be activated by receiving messages. More detailed interfaces

provided by Vertex can be found in *Appendix A*.

Message

Most graph algorithms and many machine learning problems can be considered as a message passing procedure based on a graph. Sedge supports a sufficient message passing mechanism which enables each vertex to efficiently communicate with other vertices.

To facilitate your implementation, Sedge provides an abstract **Message** class which you can inherit. The **Message** class primarily consists of the source vertex, the destination vertex and the superstep in which the message is sent out. You can easily define your own Message class to incorporate more personalized attributes, such as message value. To make the message class succinct, you'd better **not** extend the Message class by adding logical functions.

Sedge also provides a message **combination** mechanism which groups the messages intended for the same destination. This mechanism proves to be quite effective in reducing the communication overhead. To benefit from the message combination, you have to override the **add()** function defined in Message class. For the details of the **Message** class, please refer to *Appendix A*.

Overall, the Message class, in tandem with the Vertex class, constitutes the primary building blocks of your application.

Input

Sedge provides the abstract class **FileInputer** which facilitates you to interpret your input data. Please refer to *Appendix A* for the details.

Output

Analogous to input, the abstract class **FileOutputer** enables you personalize your own output format. Please refer to *Appendix A* for the details.

JobConfig

To instantiate your own classes, you should specify the classes in the JobConfig. See *Appendix A* for the details.

FileSplit

Sedge provides 3 built-in classes the facilitate you to split the input graph data into parts and distribute them in the cluster.

Classes	Notes
FileLineSplit	Splits the input graph data into parts with equal amount of lines
FileSizeSplit	Splits the input graph data into parts with equal amount of byte size
GraphPartSplit	Distributes the graph according to the existed partitions

In order to employ the split classes above, you have to specify one of them in the JobConfig (i.e. **setFileSplitClass()**). For example, suppose you want to use **FileLineSplit** and split the graph into 10 parts, a typical configuration would be

```
jobConf.setFileSplitClass(FileLineSplit.class);
jobConf.setSplitNum(10);
```

For distributed applications on large graphs, it is obvious that intensive inter-machine communicate could significantly impact the overall performance. An effective solution for this problem is to partition the graph into small balanced part such that the inter-parts connections are minimized. In the application section, we will show you how to benefit from the graph partitioning routine provided by Sedge. Here suppose we already partitioned the graph into *n* parts and have each of the parts in a separated file named as **split-i**, where *i* is a positive integer. A typical configuration using **GraphPartSplit** would be

```
jobConf.setFileSplitClass(GraphPartSplit.class);
```

Notice that for **GraphPartSplit**, you do **not** need to set split number.

Before starting the job, you have to put all the parts into Sedge. If you have already splits the graph into several split files, you can use,

```
[user@master ~]$ bin/sedge –s parts test
```

where the "*parts*" in the command is the folder where you put your split files. Or you can also use,

```
[user@master ~]$ bin/sedge –m partition.map test.graph test
```

where the "*partition.map*" is a vertex-partition map (in the form of "vtx_Id part_Id") and the "*test.graph*" is the original graph.

Example: PageRank

We next show a comprehensive example which implements the PageRank algorithm by utilizing the user interfaces as we introduced above.

We first define the Message class, **PageRankMessage**, which will not depend on any other classes.

```
package examples.pagerank;

import sedge.graph.Message;

public class PageRankMessage extends Message {
	// the PageRank value
	private double value;

	public PageRankMessage() {
		super();
	}
	public PageRankMessage(int srcId, int dstId, int step) {
		setSrcId(srcId);
		setDstId(dstId);
		setStep(step);
	}
	public void setValue(double value) {
		this.value = value;
	}
	public double getValue() {
		return value;
	}
	public void add(Message msg) {
		value += ((PageRankMessage)msg).getValue();
	}
}
```

In order to use the message combination mechanism, we override the **add()** function in **PageRankMessage**, which simply adds the PageRank value together.

```
package examples.pagerank;

import sedge.graph.Vertex;

public class PageRankVertex extends Vertex {
	 // the PageRank value
	private double value;

	public PageRankVertex(int id, int nghNum, double value) {
		super(id, nghNum);
		this.value = value;
	}
	public double getValue() {
		return value;
	}
	public boolean compute() {
```

```
            // process the messages received from the last step
            if (getStep() > 1) {
                if (hasMsgAt(getStep() - 1) || getStep() == 1) {
                    double sum = 0;
                    while (hasMsgAt(getStep() - 1)) {
                        PageRankMessage prMsg = (PageRankMessage) pollMsg();
                        sum += prMsg.getValue();
                    }
                    value = value * 0.15 + sum * 0.85;
                } else
                    return false;
            }
            // send out messages to the neighbors
            if (getStep() < 20) {
                double ngbValue = value / getNeighbors().length;
                for (int ngbId : getNeighbors()) {
                    PageRankMessage prm = new PageRankMessage(getId(), ngbId,
                            getStep());
                    prm.setValue(ngbValue);
                    sendMessage(prm);
                }
                return true;
            }
            return false;
        }
    }
```

```
package examples.pagerank;

import sedge.graph.Vertex;
import sedge.io.FileInputer;

public class PageRankInput extends FileInputer {

    public int getVtxId(String line) {
        int index = line.indexOf(",");
        return Integer.parseInt(line.substring(0, index));
    }
    public Vertex getVertex(String line) {
        String[] strs = line.split(",");
        PageRankVertex vertex = new PageRankVertex(Integer.parseInt(strs[0]),
                strs.length - 1, 1); // The initial pagerank value is 1
        for (int i = 1; i < strs.length; i++)
            vertex.addNghs(Integer.parseInt(strs[i]), i-1);
        return vertex;
```

```
        }
}
```

```
package examples.pagerank;

import sedge.graph.Vertex;
import sedge.io.FileOutputer;

public class PageRankOutput extends FileOutputer {

        public void printVertex(Vertex vtx) {
                PageRankVertex prVtx = (PageRankVertex)vtx;
                StringBuilder sb = new StringBuilder();
                sb.append(prVtx.getId()).append(" ").append(prVtx.getValue());
                println(sb.toString());
        }
}
```

Finally, we define the PageRank class, which will properly configure the job with regard to the classes we defined above.

```
package examples.pagerank;

import sedge.fs.split.FileSizeSplit;
import sedge.job.JobClient;
import sedge.job.JobConf;

public class PageRank {

        public static void main(String[] args) {
                JobConf jobConf = new JobConf();

                jobConf.setName("Page Rank");

                jobConf.setInputSrc(args[0]);
                jobConf.setOutputDir(args[1]);

                jobConf.setFileSplitClass(FileSizeSplit.class);
                jobConf.setSplitNum(Integer.parseInt(args[2]));

                jobConf.setOutputClass(PageRankOutput.class);
                jobConf.setInputClass(PageRankInput.class);

                jobConf.setCombiner(true);
                jobConf.setMessageClass(PageRankMessage.class);

                JobClient.runJob(jobConf, args);
```

```
        }
    }
```

It can be seen that there are three input arguments in the PageRank class: "*args[0]*" designates the input graph; "*args[1]*" designates the result output directory; "*args[2]*" (integer) is the number of the splits that the **FileSizeSplit** will generate. We will demonstrate another version of this example using graph partitioning (**GraphPartSplit**) in *Appendix B*.

Run the PageRank example as a job

The input graph *test.graph* is as follows

```
[user@master ~]$ less test.graph
1,2,3,4,5,6,7,8
2,3,4
3,7,10,13
5,1,3,6,10
……
```

Put the graph in Sedge

```
[user@master sedge-0.10.0]$ bin/sedge put ~/test.graph test
```

Run the PageRank example

```
[user@master sedge-0.10.0]$ bin/sedge jar sedge-0.10.0-examples.jar examples.pagerank.PageRank test result 10
2011-04-28 16:20:55 INFO - Start master
2011-04-28 16:20:55 INFO - Start Page Rank
2011-04-28 16:20:56 INFO - 10 splits
……
2011-04-28 16:23:41 INFO - Job initializing
2011-04-28 16:24:01 INFO - Start working
2011-04-28 16:24:01 INFO - superstep 1 start
2011-04-28 16:25:14 INFO - superstep 1 done
......
2011-04-28 16:44:11 INFO - superstep 20 start
2011-04-28 16:44:59 INFO - superstep 20 done
2011-04-28 16:45:01 INFO - Job done!
[user@master sedge-0.10.0]$
```

For more system running information, you can refer to the detailed log files in sedge-0.10.0/logs or simply use the command

```
[user@master sedge-0.10.0]$ bin/sedge job-info
```

Get the results

```
[user@master sedge-0.10.0]$ bin/sedge get result ~/
[user@master sedge-0.10.0]$ ls ~/result/
split-0  split-1  split-2  split-3  split-4  split-5  split-6  split-7  split-8  split-9
[user@master sedge-0.10.0]$ less ~/result/split-0
1 3.464
2 0.543
3 1.511
……
```

In case that you want to stop your job, you can use

```
[user@master sedge-0.10.0]$ bin/stop-all.sh
```

Appendix A: Details of user interface

Vertex

Functions	Notes	Return
getId()	Gets the vertex id	int
getStep()	Gets current superstep number	int
getNeighbors()	Gets the ids of the neighbors	int[]
nghNum()	Gets number of the neighbors	int
hasMsg()	Returns true if there are more messages	boolean
hasMsgAt(int)	Returns true if there are more messages which are receive at the specific step	boolean
pollMsg()	Gets a message from message queue	Message
sendMessage(Message)	Sends a message	void
Output()	Outputs the result	void

Message

Functions	Notes	Return
getSrcId()	Gets the source vertex id of the message	int
setSrcId(int)	Sets the source vertex id of the message	void
getDstId()	Gets the destination vertex id of the message	int
setDstId(int)	Sets the destination vertex id of the message	void
getStep()	Gets the superstep in which the message is sent	int
setStep(int)	Designates the superstep in which the message is sent	void
add(Message)	Combines two messages	void

FileInputer

Functions	Notes	Return
getVtxId(String)	Returns the vertex id	int
getVertex(String)	Returns a vertex instance	Vertex

FileOutputer

Functions	Notes	Return
println(String)	Prints a line to the result	void
printVertex(Vertex)	Prints a vertex to the result	void

JobConfig

Functions	Notes	Return
setName(String)	Sets job name	void
setInputSrc(String)	Sets input graph	void
setOutputDir(String)	Sets output directory	void
setInputClass(FileInputer)	Designates the inputer class	void
setOutputClass(FileOutputer)	Designates the outputer class	void
setVertexClass(Vertex)	Designates the vertex class	void
setMessageClass(Message)	Designates the message class	void
setFileSplitClass(FileSplit)	Designates the split class	void
setSplitNum(int)	Sets the number of splits to be generated	void
setCombiner(boolean)	Uses the message combination or not	void

Appendix B: Graph Partitioning

You can first run script "**bin/sedge part**" to see the help information of the graph partitioning routine in Sedge.

```
[user@master sedge-0.10.0]$ bin/sedge part
 Usage: <graph> <parts> <balance> <result>
   graph          input graph
   parts          number of partitions
   balance        partition balance (e.g. 0.02)
   result         partition result (output)
```

Run the graph partitioning routine.

```
[user@master sedge-0.10.0]$ bin/sedge part 10 0.01 ~/partout
```

Notice that currently Sedge only provides graph partitioning routine on unweighted graphs. Thus the input file should be in the form of adjacent list (see "input graph format" above)

PageRank example revisit

Put the graph into Sedge using the vertex-partition map

```
[user@master sedge-0.10.0]$ bin/sedge put –m ~/result/map ~/test.graph testpart
```

Modify the code in **PageRank** class.

```
package examples.pagerank;

import sedge.fs.split.FileSizeSplit;
import sedge.job.JobClient;
import sedge.job.JobConf;

public class PageRank {

    public static void main(String[] args) {
        ……

        //jobConf.setFileSplitClass(FileSizeSplit.class);
        //jobConf.setSplitNum(Integer.parseInt(args[2]));
        jobConf.setFileSplitClass(GraphPartSplit.class);

        ……
        JobClient.runJob(jobConf, args);
    }
}
```

Run the new job as

```
[user@master sedge-0.10.0]$ bin/sedge jar sedge-0.10.0-examples.jar examples.pagerank.PageRank testpart result
```

Notice that this new version of PageRank class only has two input arguments, i.e. "*testpart*" (as the input) and "*result*" (as the output).