SIGMOD23Experiments
This is the read me for reproducing experimental results in our SIGMOD paper
"Generalizing Bulk-Synchronous Parallel Processing: From Data to Threads and Agent-Based Simulations".

Systems
In our paper, we reviewed existing BSP-like systems, including Spark, GraphX, Giraph, and Flink (Gelly), and compared their performance with our system CloudCity.
We proposed a set of simulation workloads selected from areas where simulations are prevalent: population dynamics (gameOfLife), economics (stockMarket), and epidemics,
and implemented these workloads on each system.
You can find the scripts and experimental results for each system in the corresponding folder, whose names are self-explanatory. The input graphs for each of these systems can be generated from `ExperimentGraphs`.

Setup
We used a cluster of servers running CentOS 7, with Java 8 installed. Each server has 24 cores (two Intel Xeon E5-2680, 48 hardware threads), 256GB of RAM, and
400GB of SSD. Each server connects to routers through two 10Gb Ethernet network interface cards.

(Can also be found in the corresponding folder in the source code)
Spark
The source code for implementing each of these simulations in as a vertex program can be found in `/src/main/scala/simulations/`.

Benchmark
Our paper included the following benchmark experiments in Spark: tuning, scaleup, scaleout, communication frequency, and computation interval. For each of the benchmark, you can find its corresponding configuration in `conf/`, which is necessary for launching a benchmark. The input graphs are not included here due to their size, but you should be able to easily generate them based on our description in the paper.

The driver script for starting a benchmark is `/bin/bench.py`. Prior to running a benchmark, you need to compile and assemble the vertex programs. Our benchmark script automates this for you by passing `-a`. In short, to compile and assemble the vertex program and then running a benchmark named {test}, you should enter the following command (tested with python3.9, but you can easily adjust the driver script to use other versions of Python):

```python3 bin/bench.py -t test```.

As a reference, you can checkout our measured performance in `benchmark/`.

Remark
Before running a benchmark, you should already have the Spark cluster up and running. Our experiments were done using Spark 3.3 on CentOS 7. We used a cluster of servers, each with 24 cores (two Xeon processors, 48 hardware threads, supporting hyper-threading), 256GB of RAM, and 400GB of SSD. To obtain the best performance, you need to tune the configuration of Spark in `conf/spark-defaults.conf` (this is defined in Spark, *not* part of our benchmark configuration).

GraphX
The source code for implementing each of these simulations in GraphX as a vertex program can be found in `/src/main/scala/simulations/`.

Benchmark
Our paper included the following benchmark experiments in GraphX: tuning, scaleup, scaleout, communication frequency, and computation interval. For each of the benchmark, you can find its corresponding configuration in `conf/`, which is necessary for launching a benchmark. The input graphs are not included here due to their sheer size, but you should be able to easily generate them based on our description in the paper.

The driver script for starting a benchmark is `/bin/bench.py`. Prior to running a benchmark, you need to compile and assemble the vertex programs. Our benchmark script automates this for you by passing `-a`. In short, to compile and assemble the vertex program and then running a benchmark named {test}, you should enter the following command (tested with python3.9, but you can easily adjust the driver script to use other versions of Python):

```python3 bin/bench.py -a -t test```.

Remark
Before running a benchmark, you should already have the Spark cluster up and running. Our experiments are done using Spark 3.3.0, tested on CentOS 7 using a cluster of Xeon servers. Each server has 24 cores and 220 GB RAM. For the best performance, you need to tune the configuration of Spark (this is *not* part of our benchmark configuration). You can find some of our tuning setup in `/src/main/scala/simulations/Simulate.scala`.

Giraph
The source code for implementing each of these simulations in GraphX as a vertex program can be found in `/giraph-examples/src/main/java/org/apache/giraph/examples/*`. Test files are in `/benchmark/`.

Giraph follows the bulk-synchronous parallel model relative to graphs

```
where vertices can send messages to other vertices during a given
superstep.  Checkpoints are initiated by the Giraph infrastructure
at
user-defined intervals and are used for automatic application
restarts
when any worker in the application fails.  Any worker in the
application can act as the application coordinator and one will
automatically take over if the current application coordinator
fails.

-------------------------------

Hadoop versions for use with Giraph:

Secure Hadoop versions:

- Apache Hadoop 1 (latest version: 1.2.1)

  This is the default version used by Giraph: if you do not
specify a
  profile with the -P flag, maven will use this version. You may
also
  explicitly specify it with "mvn -Phadoop_1 <goals>".

- Apache Hadoop 2 (latest version: 2.5.1)

  This is the latest version of Hadoop 2 (supporting YARN in
addition
  to MapReduce) Giraph could use. You may tell maven to use this
version
  with "mvn -Phadoop_2 <goals>".

- Apache Hadoop Yarn with 2.2.0

  You may tell maven to use this version with "mvn -Phadoop_yarn
-Dhadoop.version=2.2.0 <goals>".

- Apache Hadoop 3.0.0-SNAPSHOT

  You may tell maven to use this version with "mvn
-Phadoop_snapshot <goals>".

Unsecure Hadoop versions:

- Facebook Hadoop releases: https://github.com/facebook/hadoop-20,
Master branch

  You may tell maven to use this version with "mvn
-Phadoop_facebook <goals>"

-- Other versions reported working include:
---  Cloudera CDH3u0, CDH3u1
```

While we provide support for unsecure and Facebook versions of Hadoop
with the maven profiles 'hadoop_non_secure' and 'hadoop_facebook',
respectively, we have been primarily focusing on secure Hadoop releases
at this time.

Building and testing:

You will need the following:
- Java 1.8
- Maven 3 or higher. Giraph uses the munge plugin
  (http://sonatype.github.com/munge-maven-plugin/),
  which requires Maven 3, to support multiple versions of Hadoop. Also, the
  web site plugin requires Maven 3.

Use the maven commands with secure Hadoop to:
- compile (i.e mvn compile)
- package (i.e. mvn package)
- test (i.e. mvn test)

For the non-secure versions of Hadoop, run the maven commands with the
additional argument '-Phadoop_non_secure'.
Example compilation commands is 'mvn -Phadoop_non_secure compile'.

For the Facebook Hadoop release, run the maven commands with the
additional arguments '-Phadoop_facebook'.
Example compilation commands is 'mvn -Phadoop_facebook compile'.

Developing:

Giraph is a multi-module maven project. The top level generates a POM that
carries information common to all the modules. Each module creates a jar with
the code contained in it.

The giraph/ module contains the main giraph code. If you just want to work on
the main code only you can do all your work inside this subdirectory.
Specifically you would do something like:

```
  giraph-root/giraph/ $ mvn verify            # build from current state
  giraph-root/giraph/ $ mvn clean             # wipe out build
files
  giraph-root/giraph/ $ mvn clean verify      # build from fresh
state
  giraph-root/giraph/ $ mvn install           # install jar to
local repository
```

The giraph-formats/ module contains hooks to read/write from various
formats (e.g. Accumulo, HBase, Hive). It depends on the giraph module. This
means if you make local changes to the giraph codebase you will first need to
install the giraph/ jar locally so that giraph-formats/ will pick it up.
In other words something like this:

```
  giraph-root/giraph/ $ mvn install
  giraph-root/giraph-formats $ mvn verify
```

To build everything at once you can issue the maven commands at the top level.
Note that we use the "install" target so that if you have any local changes to
giraph/ which formats needs it will get picked up because it will install
locally first.

```
  giraph-root/ $ mvn clean install
```

Scripting:

Giraph has support for writing user logic in languages other than Java. A Giraph
job involves at the very least a Computation and Input/Output Formats. There are
other optional pieces as well like Aggregators and Combiners.

As of this writing we support writing the Computation logic in Jython. The
Computation class is at the core of the algorithm so it was a natural starting
point. Eventually it is our goal to allow users to write any / all components of
their algorithms in any language they desire.

To use Jython with our job launcher, GiraphRunner, pass the path to the script
as the Computation class argument. Additionally, you should set the -jythonClass

option to let Giraph know the name of your Jython Computation
class. Lastly, you
will need to set -typesHolder to a class that extends Giraph's
TypesHolder so
that Giraph can infer the types you use. Look at page-rank.py as
an example.

How to run the unittests on a local pseudo-distributed Hadoop
instance:

As mentioned earlier, Giraph supports several versions of Hadoop.
In
this section, we describe how to run the Giraph unittests against
a single
node instance of Apache Hadoop 0.20.203.

Download Apache Hadoop 0.20.203 (hadoop-0.20.203.0/
hadoop-0.20.203.0rc1.tar.gz)
from a mirror picked at http://www.apache.org/dyn/closer.cgi/
hadoop/common/
and unpack it into a local directory

Follow the guide at
http://hadoop.apache.org/common/docs/r0.20.2/
quickstart.html#PseudoDistributed
to setup a pseudo-distributed single node Hadoop cluster.

Giraph’s code assumes that you can run at least 4 mappers at once,
unfortunately the default configuration allows only 2. Therefore
you need
to update conf/mapred-site.xml:

```
<property>
  <name>mapred.tasktracker.map.tasks.maximum</name>
  <value>4</value>
</property>

<property>
  <name>mapred.map.tasks</name>
  <value>4</value>
</property>
```

After preparing the local filesystem with:

```
rm -rf /tmp/hadoop-<username>
/path/to/hadoop/bin/hadoop namenode -format
```

you can start the local hadoop instance:

```
/path/to/hadoop/bin/start-all.sh
```

and finally run Giraph’s unittests:

```
mvn clean test -Dprop.mapred.job.tracker=localhost:9001
```

Now you can open a browser, point it to http://localhost:50030 and watch the
Giraph jobs from the unittests running on your local Hadoop instance!

Notes:
Counter limit: In Hadoop 0.20.203.0 onwards, there is a limit on the number of
counters one can use, which is set to 120 by default. This limit restricts the
number of iterations/supersteps possible in Giraph. This limit can be increased
by setting a parameter "mapreduce.job.counters.limit" in job tracker's config
file mapred-site.xml.

CloudCity
The source code for implementing each of these simulations in CloudCity can be found in `/example/src/main/scala/example/*`. Generated programs (compiled) can be found in `/generated/src/main/scala/*`. You can compile all the examples using command `sbt project example; runAll`.

Benchmark
Our paper included the following benchmark experiments in CloudCity: tuning, scaleup, scaleout, communication frequency, and computation interval. For each of the benchmark, you can find its corresponding configuration in `conf/`, which is necessary for launching a benchmark. The input graphs are not included here due to their size, but you should be able to easily generate them based on our description in the paper.

The driver script for starting a benchmark is `/bin/bench.py`. Prior to running a benchmark, the vertex programs need to be compiled and assembled to create a uber jar that will be run on the cluster. Our benchmark script automatically cleans existing jar, if any, and creates a new jar. If undesired ,you can disable this default behavior in `/bin/bench.py`. In short, to compile and assemble the vertex program and then running a benchmark named {test}, you should enter the following command (tested with python3.9, but you can easily adjust the driver script to use other versions of Python):

```python3 bin/bench.py -t test```.

As a reference, you can checkout our measured performance in
`benchmark/`.

Remark
Before running a benchmark, you should already have the CloudCity
cluster up and running. The binary for CloudCity is in the folder
`ch.epfl.data`. You should put it in your local ivy directory,
such as `/username/.ivy2/local/ch.epfl.data/`. Our experiments
were done using CentOS 7. We used a cluster of servers, each with
24 cores (two Xeon processors, 48 hardware threads, supporting
hyper-threading), 256GB of RAM, and 400GB of SSD. To obtain the
best performance, you need to tune the configuration of CloudCity
in `generated/src/main/resources`, see description of Akka cluster
usage.