Using BigGrph in your projects

Setup

Two possibilities exist to use BigGrph:

1. The first one is to get the sources of BigGrph and import the projects in your Eclipse workspace, create your own project and add the BigGrph projects as dependencies of your project. See the file Contribute to know how to get the sources.
2. The second one is to build a JAR containing BigGrph and all its dependencies using the Maven build described in the document Contribute. This produces two JAR files one that contain the binaries of BigGrph and one that contains the sources. Create a project in your Eclipse workspace and put the BigGrph JAR as a dependency in the classpath.

The first method is currently recommended because it is simple, gives you easy access to the BigGrph sources and eventually allows you to become a contributor to BigGrph if you wish so.

Overall structure of a simple BigGrph program

A BigGrph program is written as an ordinary java program, using a `static main()` function declared in a public class. Running a BigGrph program requires several (5) steps that are executed from the `main()` function:

- Create and start a *cluster*
- Associate a graph file with a *dataset*
- Instantiate a *distributed graph* from the *dataset* by loading the associated file
- Execute algorithms on this graph
- Stop the cluster.

All these steps are launched from the main() function using BigGrph API. The concepts of *cluster*, *dataset* and *distributed graph* are explained below.

Clusters

Master and slaves nodes

With BigGrph a *cluster* is made of *nodes*: one of this node is the *master* node, all the others are the *slave* nodes. The master node is the one that starts the execution of the main program, it is in charge of starting the slave programs on all the slave nodes, sending computations and tasks to the slave nodes, gathering the results, and finally stopping all the program running on the slave nodes.

The master node uses SSH to connect to the slave nodes and launch the slave programs.

Creating clusters

Clusters are represented in BigGrph using an instance of the class `BigGrphCluster`. There are various ways to create such objects, the simplest is to create it from the list of hostnames of the slave nodes in the cluster:

```
import jacaboo.NodeNameSet;
import biggrph.BigGrphCluster;

public class MyBigGrphProgram
{
    // ...

    public static void main(String[] args)
    {
        // Suppose that the String slaveNames contains a comma-separated
        // list of hostnames
        String slaveNames = "host1.mydomain,host2.mydomain";

        NodeNameSet nodeNames =  new NodeNameSet();
        for (String name : slaveNames.split(","))
        {
            nodeNames.add(name);
        }

        BigGrphCluster cluster
            = BigGrphCluster.workstations(System.getProperty("user.name"), nodeNames, 1);

        // ...

    }
}
```

The previous code is part of the main program and runs on the master node of the cluster.

The first argument to the `BigGrphCluster.workstations()` function is the username that will be used for SSH connections from the master node to the slave nodes. There is no provision made for automatically entering a password or a passphrase, so it is advised to setup SSH using either temporary keys without passphrase or else ssh-agent or equivalent tools.

It is possible to create more that one slave process on each cluster nodes by setting the last argument of `BigGrphCluster.workstations()` to a value greater than 1. Take care to not use more than the available memory, in order to not loose a possible benefit of this form of parallelism.

Returning from the `BigGrphCluster.workstations()` function, the cluster is actually not yet running. The only program that currently runs is the main program on the master node.

Other static functions exist in the `BigGrphCluster` class to create clusters. One useful function is `BigGrphCluster.localhost(int nProcess, boolean useMainProcess)` that allows to create a cluster where each node is a distinct java process on the local host, the total number of processes (and also the number of slave nodes) is given as the first argument `nProcess` of the function. The second argument, a boolean, if true, runs one of the slaves inside the java process running the master node. These kind of setups are used for some unit tests, for debugging, and can be used to run BigGrph programs on one host, without distribution if the `nProcess` argument is set to 1. It can be useful to assess performance of some algorithms on one node, in order to compare with a distributed execution over several nodes.

Starting and Stopping clusters

Right after the creation of the `BigGrphCluster` object, nothing is actually running except the main program that created this object. The cluster setup is complete only when slave nodes are executing the slave program, ready to execute computations received from the master node.

The function that actually complete the initialization and that starts the slave program is `BigGrphCluster.start()`. This function is called on the created `BigGrphCluster` instance.

At the end of the processing, it is also necessary to call the `BigGrphCluster.stop()` function to stop the slave nodes, release allocated resources on all nodes, and allow the main program on the master node to exit in clean condition.

```
BigGrphCluster cluster
    = BigGrphCluster.workstations(System.getProperty("user.name"), nodeNames, 1);

cluster.start();

// Do all the processing ...

cluster.stop();
```

Customizing clusters

Cluster can be customized before they are started. Two parameters can be changed using either API functions, environment variables or Java properties: the heap memory of slave Java processes and the remote debugging ability of the slave processes.

Memory

Heap memory can be set using two functions to call on the `BigGrphCluster` object: `setMaxMemorySizeInGigaBytes(int n)` and `setMaxMemorySizeInMegaBytes(int n)`. The first uses Gigabytes as unit, and the second uses Megabytes. If needed, these functions must be called before calling the `start()` function of the cluster object.

The same setting can be used using either the environment variable `BGRPH_MEMORY_MAX`, or the `biggrph.memory.max` property. The value of both is given in Megabytes.

In all cases, the setting is used for both the `–Xms` and `–Xmx` argument to the JVM process so that no heap resizing occur during execution. During the development and testing of BigGrph, it was sometimes encountered longer execution times when the heap was enlarged with large size in hundreds of Gigabytes.

Debugging

Remote debugging can be enabled on all the slave nodes, in order to ease development and testing. Three ways are possible:

- Call the `enableDebugging(int port)` function on the cluster object, before calling `start()`. The argument is a TCP port number that will be used to give each slave a different debugging port: `port` for the first slave, `port+1` for the second and so on.
- Set the `BGRPH_DEBUGPORT_BASE` environment variable to the TCP port number used as the base for setting the slaves port.
- Set the Java property `biggrph.debugPort.base` in the same way.

Other options for slave nodes

The java property `biggrph.jvm.parameters` can be used to add to the slaves JVM command line arguments and options that do not fit into one of the previous category. If it is defined, its value is added to the JVM launch command of all the slaves.

Using cluster resource managers

BigGrph has a minimal support for executing from a cluster resource manager and scheduler. Two such managers can be used from BigGrph : Torque and OAR. In both cases, it is possible to detect if the program has been launched from Torque or OAR scheduler, and to initialize the nodes of the cluster from the list of nodes assigned by the resource manager.

TORQUE

The class `TorqueContext` in package `jacaboo.TorqueContext` is used for Torque support. The recommended usage is to first instanciate this class, then call the `runFromTorque()` function on this object, and if the result is `true`, to initialize the cluster object from the result of the `TorqueContext.getSlaveNames()`.

```
            BigGrphCluster cluster = null;

            TorqueContext te = new TorqueContext();
            if (te.runFromTorque())
            {
                NodeNameSet nodeNames = te.getSlaveNames();
                cluster = BigGrphCluster.workstations(
                            System.getProperty("user.name"), nodeNames, 1);
            }
```

OAR

OAR support is provided by the `OARContext` class, much in the same way as the Torque support. The usage is the same:

```
            OARContext oc = new OARContext();
            if (oc.runFromOAR())
            {
                NodeNameSet nodeNames = oc.getSlaveNames();
                cluster = BigGrphCluster.workstations(
                                System.getProperty("user.name"), nodeNames, 1);
            }
```

Datasets

Datasets are the primary source of data for graphs. They are created from a file name, which contains the graph data, either as a edge list, or as adjacency lists.

Datasets are created to match the format of the associated file. An edge list file, where each eadge is a pair of numbers (the source and destination vertices) is represented by an instance of `EdgeListDataSet`. An adjacency list file is represented by an instance of `AdjDataSet`.

By default, datasets files are stored in a dedicated directory `~/biggrph/datasets/` under the home directory of the current user. If the dataset file is in this directory the dataset can be created with only the filename as argument to the constructor:

```
String datasetName = " ... ";
BigGrphDataSet dataset = new EdgeListDataSet(datasetName, cluster);
```

or

```
String datasetName = " ... ";
BigGrphDataSet dataset = new AdjDataSet(datasetName, cluster);
```

If the dataset file is not in the default directory, it is necessary to give the full path to this file with an instance of the `RegularFile` class:

```
RegularFile file = new RegularFile(dataFilename);
BigGrphDataSet dataset = new EdgeListDataSet(file, cluster);
```

This last way to create a dataset allows to overcome the dataset directory setting.

Beware that the path of directories and files used for datasets have to be **the same on all the cluster nodes**. It can be a shared or distributed filesystem, or else a replicated filesystem on all the cluster nodes. In any cases, paths are the same on all nodes, and files must be at the same place in the file hierarchy.

The dataset directory setting can be changed in two ways:

1. Using the `biggrph.datasets.directory` property, by setting this property to the absolute path of the directory containing the files.
2. using the `BGRPH_DATASETDIR` environment variable, by setting this variable to the absolute path of the directory containing the files.

For both the property and the environment variable, the main program running on the master test their values if it exists, and propagates to the slave nodes. Inside a cluster, all nodes (master and slaves) have the same path for the dataset directory.

Graphs

Graphs are created from a dataset, which automatically invokes the load of the dataset file according to its format (edge list or adjacency lists) to initialize the structure of the graph on all cluster nodes.

With BigGrph, each vertex is represented by a 64 bits wide signed number. The usage is to limit the range of such vertex ids to positive numbers from 0 to 2^{63}-1.

The internal structure of a graph is built upon distributed adjacency tables. An adjacency table records the neighbors of all vertices in the graph. Depending of the expected usage of the graph, one or two adjacency tables are used:

- If a graph is requested to be bi-directional, it records both the IN and OUT edges of vertices in two adjacency tables, called the IN table and the OUT table.
- The default is to create only the OUT-table, that records the edges with the direction that is found in the dataset file. This direction is recorded as the OUT neighborhood of the vertices.
- If a graph is created as an undirected one, then both directions of all edges are recorded in a single adjacency table, the OUT table.

```
long graphSizeEstimate = 2000000001;  // an upper bound of the number of vertices
GraphTopology topology = new GraphTopology(dataset, false, false, graphSizeEstima
```

The `GraphTopology` constructor is defined this way.

```
public GraphTopology(BigGrphDataSet dataset, boolean biDirectional,
        boolean multigraph, long vertexCountEstimate)
```

An instance of `GraphTopology` embeds one or two instances of `BigAdjacencyTable` to store the actual OUT and IN adjacency tables. From an instance of `GraphTopology` two functions are used to retrieve these tables:

- biggrph.GraphTopology.getOutAdjacencyTable()
- biggrph.GraphTopology.getInAdjacencyTable()

The second function above returns `null` if the graph is not bidirectional.

Algorithms

All the graph algorithms implemented in BigGrph are in the package `biggrph.algo` and its subpackages. Some algorithms are methods of the `GraphTopology` class, others require to instantiate an object with the graph as arguments and call a method on this object to execute the algorithm.

Several algorithms use an instance of the `LongPredicate` interface to allow working on a subpart of the input graph. The interface `LongPredicate`, in package `toools.util` from the `tools` project, defines one unique function `boolean accept(long vertexId)`. Implementation of this function must return true to indicate that the vertex is to be considered during algorithm execution. There is a static instance of this interface `toools.util.LongPredicate.ACCEPT_ALL` that always return true.

Several graph algorithms produce a set of value of various types associated to each vertex of the graph. BigGrph has several implementations of distributed maps where the key is a vertex represented by a Java long number (64 bits) and the value are either byte, 32 bits integer, 64 bits long, and `java.lang.Object`. These classes are named `ByteDHT`, `IntDHT`, `LongDHT` and `ObjectDHT` and are located in the `dht` package from project `ldjo`.

Counting and statistics

Algorithms to count various statistics of a graph are summarized below. These are methods to call on a instance of `GraphTopology` or `BigAdjacencyTable`.

- biggrph.GraphTopology.getNumberOfVertices()
- biggrph.GraphTopology.getNumberOfEdges()
- biggrph.GraphTopology.getOutAverageDegree()
- biggrph.GraphTopology.getOutMaxDegree()
- biggrph.BigAdjacencyTable.getDegreeDistribution()

Some of these function have variants that can be called with an instance of `LongPredicate` to only process part of the graph.

Breadth First Search

The static function `compute()` in the `biggrph.algo.search.bfs.BFS` class can be used to perform a BFS from a unique source in the graph. The exact syntax of this function is `compute(BigAdjacencyTable at, long source, boolean predecessors, LongPredicate filter)` where the arguments are as follows:

- **`BigAdjacencyTable`** *at* is the distributed adjacency table to process. Recall that such tables are retrieved from a `GraphTopology` object using functions `getOutAdjacencyTable()` and `getInAdjacencyTable()`.
- **`long`** *source* is the source vertex of the search.
- **`boolean`** *predecessors* if true will also compute the predecessor of each vertex.
- **`LongPredicate`** *filter* allows to restrict the search to a subset of the graph. Use `LongPredicate.ACCEPT_ALL` to process the whole graph.

Connected Components

BigGrph includes two connected components implementations: one for connected components, one for strongly connected components.

Connected components computation is done on an undirected graph, using the following code:

```
import biggrph.GraphTopology;
import biggrph.algo.connected_components.AssignConnectedComponents;
import dht.LongDHT;

AssignConnectedComponents pr = new AssignConnectedComponents(topology);
pr.execute();
LongDHT components = pr.getConnectedComponentsAssignments();
pr.delete();
```

Strongly Connected Components are computed on a bidirectional graph, the following code is an example

```
import biggrph.GraphTopology;
import biggrph.algo.connected_components.StronglyConnectedComponents;
import dht.LongDHT;

//...

GraphTopology topology = ...
StronglyConnectedComponents sccPr = new StronglyConnectedComponents(topology);
LongDHT sccIds = sccPr.execute();
```

Both algorithms produce a `LongDHT` that gives for each vertex of the graph an number which is the lowest vertex id of all the vertices in the same component.

Other algorithms

Other algorithms exist in the BigGrph framework. They are implemented in the `biggrph.algo` packages and its subpackages. Related example programs are in the `biggrph.examples` project.

Support for writing BigGrph programs

The class `AbstractBigGrphMain` in package `fr.inria.coati.biggrph.examples` provides some support for writing the `main()` function of BigGrph programs. It allows to:

1. Parse some options on the command line to specify cluster nodes and a dataset.
2. Create the `BigGrphCluster` object either from options and arguments specified on the command line or when running under supervision of Torque and OAR.
3. Create a dataset from command line options and arguments.

Parsing of command line options is based on the `org.apache.commons.cli` package. Processing of all the options described below is done by calling the `AbstractBigGrphMain.processCommonOptions()` static function.

Cluster related options

- `-c` *s* (`--slaves` *s*) where *s* is a comma separated string of hostnames;
- `-l` *n* (`--localhost` *n*) where *n* is the number of slave processes on the local host
- `-p` *n* (`--process` *n*) where *n* is the number of processes per host

After these options are processed, it is possible to call the `AbstractBigGrphMain.getCluster()` static function to get a `BigGrphCluster` object.

Dataset related options

- `-d` *name* (`--dataset` *name*) where *name* is the file name inside the dataset directory.
- `-f` *path* (`--file` *path*) where *path* is the absolute path of a file.
- `-e` (`--edgelist`) indicates that the dataset refers to a file with edge list format.
- `-a` (`--adjlist`) indicates that the dataset refers to a file with adjacency table format.
- `-z` *n* (`--size` *n*) specifies an estimate of the number of vertices in the graph loaded from the dataset. The number *n* cab be suffixed (without space) with letter K, M or G to multiply it by 1000, 1000 * 1000 or 1000 * 1000 * 1000.

After these options are processed, it is possible to call the `AbstractBigGrphMain.getDataset()` static function to get a dataset object from which the graph will be loaded.

Other options

- `-v` (`--verbose`) set the variable `verbose` to `true`.

Usage

The following is a working example of a java program that uses the `AbstractBigGrphMain` to build its `main()` function. With respect to the common set of option described above, it adds a new option `-s` (`--save`) to initialize a boolean variable, with the intended meaning that the program will use it to save some result of the processing after it is done.

```
import java.io.IOException;

import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.DefaultParser;
import org.apache.commons.cli.HelpFormatter;
import org.apache.commons.cli.Options;
import org.apache.commons.cli.ParseException;

import biggrph.BigGrphCluster;
import biggrph.GraphTopology;
import biggrph.dataset.BigGrphDataSet;
import biggrph.examples.AbstractBigGrphMain

public class MyBigGrphProgram extends AbstractBigGrphMain
{
    private static final Options options;

    static
    {
        options = getCommonOptions();
        options.addOption("s", "save", false, "write the final results");
    }
```

```
public static void main(String[] args)
{
    boolean saveResults = false;
    CommandLineParser parser = new DefaultParser();
    CommandLine cmd;

    try
    {
        cmd = parser.parse(getOptions(), args);
    }
    catch (ParseException e)
    {
        System.err.println("Command line parsing failed: " + e.getMessage());
        new HelpFormatter().printHelp(
                "java [jvm args] " + MyBigGrphProgram.class.getName(), options);
        return;
    }

    processCommonOptions(cmd);

    if (cmd.hasOption("s"))
    {
        saveResults = true;
    }
    if ( ! checkCommonOptions())
    {
        return;
    }

    BigGrphCluster cluster = null;
    try
    {
        cluster = getCluster();
    }
    catch (IOException e)
    {
        e.printStackTrace();
        return;
    }

    cluster.start();

    BigGrphDataSet dataset = getDataset(cluster);

    System.out.println("Graph size estimate=" + graphSizeEstimate + " vertices.");

    GraphTopology graph = new GraphTopology(dataset, false, false, true,
            graphSizeEstimate);
        // ... processing

        if (saveResults)
        {
            // ... write results
        }

        cluster.stop();
    }
}
```

Running BigGrph programs

Running example programs

Example programs from the biggrph.examples project can be executed using the JAR file created by maven in the `target/` directory of this project. This JAR contains BigGrph, all its dependencies and the example programs.

There are some graphs in the project `biggrph.tests` in the directory `resources/`. Besides simple graphs used for testing, two graphs `CA-GrQc.txt` and `CA-HepTh.txt` can be used as sample to run BigGrph programs. These files come from the Stanford Large Network Dataset Collection.

For example, to run a BFS on a graph stored in the file CA-HepTh.txt located in the `$HOME/biggrph/datasets` directory, use the following command:

```
java -cp biggrph.examples/target/biggrph.examples-1.0-SNAPSHOT.jar \
     -Dbiggrph.memory.max=3000 -Dbiggrph.datasets.directory=$HOME/biggrph/datasets \
     fr.inria.coati.biggrph.examples.BigGrphBFSExample --dataset CA-HepTh.txt -z 9877 -e
     --localhost 2 -s 24325
```

This runs a BFS starting at vertex 24325, on the graph stored in the edge-list file `CA-HepTh.txt`. The cluster is actually a two slaves clusters, both running on the local host. At the end of the processing it prints the distance distribution from the source vertex:

```
15:33:50    Duration of the BFS search: 1816ms
15:33:50    Distance distribution for graph CA-HepTh.txt/out (distance #node)
15:33:50    0   1
15:33:50    1   3
15:33:50    2   86
15:33:50    3   743
15:33:50    4   2255
15:33:50    5   3017
15:33:50    6   1781
15:33:50    7   557
15:33:50    8   127
15:33:50    9   40
15:33:50    10  19
15:33:50    11  8
15:33:50    12  1
```

The command line arguments and options that this example program understands are the one described above plus the `-s` or `--source` to specify the source vertex of the search.

Running a BigGrph program in command line

You need both a JAR file containing all BigGrph (see Contribute to get one), and the compiled form of your program, either in a JAR file or in the form of the `bin/` directory automatically created by the Eclipse IDE when building Java projects. These two elements are to be given as the classpath of the JVM.

Suupose that your project is called `myProject` and the main class is `MyMainClass`. One way to launch this BigGrph program is to use the following command:

```
java -cp biggrph.examples/target/biggrph.examples-1.0-SNAPSHOT.jar:myProject/bin \
     -Dbiggrph.datasets.directory=$HOME/biggrph/datasets \
     -Dbiggrph.graph.localdirectory=$HOME/biggrph/datasets/local \
     MyMainClass ...
```

Add to this command all the arguments and options needed for the execution (dataset, cluster nodes, etc.)

Running and Debugging in Eclipse

It is possible to launch a BigGrph program from the Eclipse IDE. In this setup, it is better to have all the BigGrph projects in your Eclipse workspace, together with your own projects and programs. This will allow you to debug your programs, provided you use one of the ways described above to enable debugging of the slave processes.

First step is to create a launch configuration of your program, using the “Run As” or “Debug As” menu item on the java file of your main program. Choose the “Run Configurations” or “Debug Configurations” to create the actual launch configuration. In the interface, create a new “Java Application”, name it accordingly and fill the JVM arguments and Arguments tabs with the same information described above. Use the “Apply” button to save the configuration and the “Run” or “Debug” button to launch your program.

If you have selected the “Debug As” menu item, then the main process, which runs the master node, can be debugged as usual with the Eclipse IDE. To debug the slaves, create a “Remote Java Application” debug configuration and set the host and port corresponding to the slave you want to debug. Recall that the debug port used by the slaves starts from the number given in the `enableDebugging()` function (or the `biggrph.debugPort.base` property, or the `BGRPH_DEBUGPORT_BASE` environment variable) and increments by 1 for each slave node.