DevelopBSP

This is a simple tutorial how to write your own BSP, from the algorithm idea to a real working BSP in distributed mode. It is assuming that you have Hama-0.5.0 running (see GettingStarted), either as deamons or in Eclipse (or any other favorable IDE) as a localrunner.

The idea

A common task with Apache Hama is the realtime processing. You probably have some kind of API or data source that gets notified when new data is available or you poll for new data. For processing you normally decouple the producer (your data source) from the consumer (the processing) to parallize and distribute your workload. Have a read on the wikipedia article to familiarize yourself with the topic.

In our specific example case, we want to produce random numbers in a specific intervall and want to distribute the work across our available task.

A possible way is to first prototype a working BSP with the LocalBSPRunner in Eclipse IDE and then start the task in a real distributed environment.

The LocalBSPRunner

The LocalBSPRunner or shortly referred as localrunner is a threaded emulator of a Hama cluster. So instead of processes (also called tasks or peers) threads are used, making it possible for you on a workstation to easily debug and develop BSP applications right from your IDE. It offers the same capability as a real Hama cluster, so you can use it also to crunch small datasets on your local machine. In everycase there is a small divergence between the (pseudo)-distributed mode and the local mode, which is declared later in this section.

The localrunner is activated by default if no configuration value for the key *"bsp.master.address"* is set or if it is set to local.

Start off by first creating a new class, let's call it *RealTime* and add a main method for it to be an entry point. Let's add the skeleton for the job submission to it:

```
public static void main(String[] args) throws IOException, InterruptedException, ClassNotFoundException {
  HamaConfiguration conf = new HamaConfiguration();
  BSPJob job = new BSPJob(conf);
  // set the BSP class which shall be executed
  job.setBspClass(RealTime.class);
  // help Hama to locale the jar to be distributed
  job.setJarByClass(RealTime.class);
  // give it a name
  job.setJobName("Producer Consumer Test");
  // use 4 tasks
  job.setNumBspTask(4);
  // output path must be defined
  job.setOutputPath(new Path("/tmp/realtime-test"));
  job.setOutputFormat(NullOutputFormat.class);
  // submit the job to the localrunner and wait for its completion, while outputting logs
  job.waitForCompletion(true);
}
```

You should notice that it does not compile because RealTime is no BSP class. Which leads us to the next section.

My First BSP

In the last section you noticed that we don't have compilable code in the main method, let's fix it by introducing the BSP API.

What you have to do now is let the class *RealTime* inherit from BSP by writing:

```
public class RealTime extends BSP<NullWritable, NullWritable, NullWritable, NullWritable, IntWritable> {
```

The generic types are explained in more detail here: BSPModel. To make it short, the first four types are for KeyInput, ValueInput, KeyOutput, ValueOutput and MessageType.

In our case we don't want to use the input and output system provided by Hama, rather than let it launch several tasks. For our messaging we need *Integers* because we want to distribute some randoms. Integers serialized form in Hadoop is *IntWritable* so we use it as our message type.

Eclipse should now complain again and want you to override the bsp method, which should look like this:

```
    @Override
    public void bsp(
            BSPPeer<NullWritable, NullWritable, NullWritable, NullWritable, IntWritable> peer)
            throws IOException, SyncException, InterruptedException {
       // our code here
    }
```

What our first try should be now is to setup a "master-task" which is our producer. Therefore you can override the *setup()* function from the BSP class. In Eclipse you can do a right-click on your source -> Source -> Override/implement methods -> Choose setup. Setup is called right at the beginning of a task after it spawned.

In this case we want to determine if the task is the master task, by simply check if its index is 0 in the peer lookup table each peer holds. This works as follows by also introducing a private field:

```
    private boolean isMaster;

    @Override
    public void setup(
            BSPPeer<NullWritable, NullWritable, NullWritable, NullWritable, IntWritable> peer)
            throws IOException, SyncException, InterruptedException {

        if (peer.getPeerName().equals(peer.getPeerName(0))) {
            // I'M THE MASTER!
            isMaster = true;
        }
    }
```

You have to imagine that each task holds it's own BSP instance and therefore has different states of the variable *isMaster*. So try to not use static fields besides constants in your code, as it introduces strange behaviour in local runner and does not work in distributed mode either.

Now we know who is the master in the house, let's make our first producer code:

```
@Override
    public void bsp(
            BSPPeer<NullWritable, NullWritable, NullWritable, NullWritable, IntWritable> peer)
            throws IOException, SyncException, InterruptedException {

        if(isMaster){
            Random random = new Random();
            // produces new random integers every second
            while(true){
                int newInt = random.nextInt();
                Thread.sleep(1000);
            }
        } else {
           // if I'm not the master, then I am one of the slaves!
        }

    }
```

Currently we don't have much more than a infinite loop that entertains the garbage collector. So let's introduce a naive partitioning trick to let the slaves do work and consume our sent messages:

```
    @Override
    public void bsp(
            BSPPeer<NullWritable, NullWritable, NullWritable, NullWritable, IntWritable> peer)
            throws IOException, SyncException, InterruptedException {

        if (isMaster) {
            Random random = new Random();
            // produces new random integers every second
            while (true) {
                int newInt = random.nextInt();
                // just send to other not master peers
                peer.send(peer.getAllPeerNames()[Math.abs(newInt % (peer.getNumPeers()-1))+1],
                        new IntWritable(newInt));
                System.out.println("Sending " + newInt);
                peer.sync();
                Thread.sleep(1000);
            }
        } else {
            // if I'm not the master, then I am one of the slaves!
            while (true) {
                peer.sync();
                IntWritable msg = null;
                while ((msg = peer.getCurrentMessage()) != null) {
                    System.out.println(msg.get() + " received!");
                }
            }
        }
    }
```

As you can see, we are now sending our *newInt* to another task, which picks it up and outputs it. We using an naive modulo function to determine to which host the next message should go. Mote that we should use *Math.abs()*, because the random number can be negative and so is the modulo output. Since arrays don't have negative indices this would result in exceptions.

Watch out that only *peer.sync()* is sending the messages to the other peers and not just the send method.

Let's run our first example. In Eclipse you can simply use right-click on source -> Run as Java Application.

If everything was correct you should something like that:

```
12/04/22 15:05:05 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
12/04/22 15:05:05 WARN bsp.BSPJobClient: No job jar file set.  User classes may not be found. See BSPJob#setJar
(String) or check Your jar file.
12/04/22 15:05:06 INFO bsp.BSPJobClient: Running job: job_localrunner_0001
12/04/22 15:05:08 INFO bsp.LocalBSPRunner: Setting up a new barrier for 4 tasks!
Sending -883577075
-883577075 received!
12/04/22 15:05:09 INFO bsp.BSPJobClient: Current supersteps number: 0
Sending -446877068
Sending -1636564077
-1636564077 received!
Sending -2080792393
-2080792393 received!
12/04/22 15:05:12 INFO bsp.BSPJobClient: Current supersteps number: 3
Sending 430229227
430229227 received!
Sending -909671222
-909671222 received!
Sending 1878521166
1878521166 received!
12/04/22 15:05:15 INFO bsp.BSPJobClient: Current supersteps number: 6
```

Don't forget to stop the application, because we have coded an infinite loop.

Let's fix that by adding a configurable amount of iterations to make.

Configure variables from outside

A common task is to add variables or values into computations that aren't known at deploy time. If you are already familiar with Hadoop you should have seen the *Configuration* concept which is also present in Hama.

Configuration is an object which is basically nothing else than a map consisting of key and value strings. It is used everywhere in Hama to configure specific parts of it, therefore it is deserialized at client-side (to XML) once submitting the job and distributed to all tasks, so they have available a snapshot of the configuration the client wants to use.

Let's add a variable where we can control how much iterations to be done: (This is how your final class should look like!)

```
public class RealTime
        extends
        BSP<NullWritable, NullWritable, NullWritable, NullWritable, IntWritable> {

    private boolean isMaster;
    private int maxIterations;

    @Override
    public void setup(
            BSPPeer<NullWritable, NullWritable, NullWritable, NullWritable, IntWritable> peer)
            throws IOException, SyncException, InterruptedException {

        if (peer.getPeerName().equals(peer.getPeerName(0))) {
            // I'M THE MASTER!
            isMaster = true;
        }
        // 2 is the default if "max.iterations" wasn't set
        maxIterations = peer.getConfiguration().getInt("max.iterations", 2);
    }

    @Override
    public void bsp(
            BSPPeer<NullWritable, NullWritable, NullWritable, NullWritable, IntWritable> peer)
            throws IOException, SyncException, InterruptedException {

        if (isMaster) {
            Random random = new Random();
            // produces new random integers every second
            while (true) {
                int newInt = random.nextInt();
                peer.send(
                        peer.getAllPeerNames()[Math.abs(newInt
                                % peer.getNumPeers())], new IntWritable(newInt));
                System.out.println("Sending " + newInt);
                peer.sync();
                Thread.sleep(1000);
                if(maxIterations < peer.getSuperstepCount())
                    break;
            }
        } else {
            // if I'm not the master, then I am one of the slaves!
            while (true) {
                peer.sync();
                IntWritable msg = null;
                while ((msg = peer.getCurrentMessage()) != null) {
                    System.out.println(msg.get() + " received!");
                }
                if(maxIterations < peer.getSuperstepCount())
                    break;
            }
        }
    }

    public static void main(String[] args) throws IOException,
            InterruptedException, ClassNotFoundException {

        HamaConfiguration conf = new HamaConfiguration();

        conf.set("max.iterations", "5");

        BSPJob job = new BSPJob(conf);
        // set the BSP class which shall be executed
```

```
        job.setBspClass(RealTime.class);
        // help Hama to locale the jar to be distributed
        job.setJarByClass(RealTime.class);
        // give it a name
        job.setJobName("Producer Consumer Test");
        job.setNumBspTask(4);
        job.waitForCompletion(true);
    }

}
```

As you can see, we are adding a new integer which is the *maxIterations* and check it after each *sync()*.

Now you should be able to run this on your cluster!

Run it on your cluster

You can use the export jar function in eclipse to get a runnable jar for the class.

Right-click your *RealTime.java* file in the package explorer -> Export -> under Java, pick Runnable Jar File -> Pick the launch configuration you used for local running and a destination -> Finish!

Now you can run this jar on your running Hama cluster via:

```
hama/bin/hama jar your_exported.jar
```

And watch the results 😉

Divergence between local and distributed modes

There are some small differences between the modes which leads to common programming mistakes:

static fields

As also described above, static fields if you want to share state through many tasks are useless in distributed mode. Each task has its own JVM (java virtual machine) process in distributed mode, so you can't access other instances variables. In localmode this is sadly working, but here is the message to you: **don't use static shared state in your BSP!**.

Writables in messages

Messages in the distributed mode must somehow be serialized, to offer optimizations there are these methods called *readFields()* and *write()* in the *Writable* interface you have to implement in order to send messages. Please implement them accordingly to your implementation. And watch out to add a default constructor, because messages are instantiated via reflections and it cannot guess parameters!

JarByClass not set

Or better: Why do I see this message?

```
12/04/22 15:35:37 WARN bsp.BSPJobClient: No job jar file set.  User classes may not be found. See BSPJob#setJar
(String) or check Your jar file.
```

In distributed mode the jar which is containing the execuable code will be distributed to all task hosts. In localmode this is not necessary, because you are always local. So better use the method *BSPJob#setBspClass(your_class.class)* to let the framework determine the jar it needs to distribute properly.

Advanced programmers note

Yes, it is pretty inefficient to barrier sync for every 4 byte integer. Ideally you want to wait until all task have the same amount of messages and are optimal to transfer, say 64M chunks per destination task.

We want to provide a better support for real-time systems by adding remote (a-)synchronous memory access in the future, see HAMA-546 for progress and information on it.

This example would then not have overhead in using a global sync, because it does not use it.

Also when a single master task is used, the throughput and scalability is limited by a single task. You could add more task to serve as a master and we want to definitely want to add convenience functions to make it much more easier to implement these real-time systems in a scalable manner.