BSPModel

Overview

In Apache Hama, you can implement your own BSP method by extending from `org.apache.hama.bsp.BSP` class. Apache Hama provides in this class a user-defined function `bsp()` that can be used to write your own BSP program.

The `bsp()` function handles whole parallel part of the program. (So it just gets called once, not all over again)

There are also `setup()` and `cleanup()` which will be called at the beginning of your computation, respectively at the end of the computation.

`cleanup()` is **guranteed** to run after the computation or in case of failure. (In 0.4.0 it is actually not, we expect this to be fixed in 0.5.0).

You can simply override the functions you need from BSP class.

Basically, a BSP program consists of a sequence of supersteps. Each superstep consists of the three phases:

- Local computation
- Process communication
- Barrier synchronization

NOTE that these phases should be always sequential order.

In Apache Hama, the communication between tasks (or peers) is done within the barrier synchronization.

BSP Function

The "bsp()" function is a user-defined function that handles the whole parallel part of the program. It only takes one argument "BSPPeer", which contains an communication, counters, and IO interfaces.

Input and Output

General Information

Since Hama 0.4.0 we provide a input and output system for BSP Jobs.

We choose the key/value model from Hadoop, since we want to provide a conherent API to widely used products like Hadoop MapReduce (SequenceFiles) and HBase (Column-storage).

Input

Configuring Input

When setting up a BSPJob, you can provide a InputFormat and a Path where to find the input.

```
  BSPJob job = new BSPJob();
  // detail stuff omitted
  job.setInputPath(new Path("/tmp/test.seq");
  job.setInputFormat(org.apache.hama.bsp.SequenceFileInputFormat.class);
```

Another way to add input paths is following:

```
SequenceFileInputFormat.addInputPath(job, new Path("/tmp/test.seq"));
```

You can also add multiple paths by using this method:

```
SequenceFileInputFormat.addInputPaths(job, "/tmp/test.seq,/tmp/test2.seq,/tmp/test3.seq");
```

Note that these paths must be separated by a comma.

In case of a `SequenceFileInputFormat` the key and value pair are parsed from the header.

When you use want to read a basic textfile with `TextInputFormat` the key is always `LongWritable` which contains how much bytes have been read and `Text` which contains a line of your input.

Using Input

You can now read the input from each of the functions in BSP class which has `BSPPeer` as parameter. (e.G. setup / bsp / cleanup)

In this case we read a normal text file:

```
@Override
 public final void bsp(
     BSPPeer<LongWritable, Text, KEYOUT, VALUEOUT, MESSAGE_TYPE> peer)
     throws IOException, InterruptedException, SyncException {

     // this method reads the next key value record from file
     KeyValuePair<LongWritable, Text> pair = peer.readNext();

     // the following lines do the same:
     LongWritable key = new LongWritable();
     Text value = new Text();
     peer.readNext(key, value);
 }
```

Consult the docs for more detail on events like end of file.

There is also a function which allows you to re-read the input from the beginning.

This snippet reads the input five times:

```
 for(int i = 0; i < 5; i++){
   LongWritable key = new LongWritable();
   Text value = new Text();
   while (peer.readNext(key, value)) {
      // read everything
   }
   // reopens the input
   peer.reopenInput()
 }
```

You must not consume the whole input to reopen it.

Custom Inputformat

You can implement your own inputformat. It is similar to Hadoop MapReduce's input formats, so you can use existing literature to get into it.

Output

Configuring Output

Like the input, you can configure the output while setting up your BSPJob.

```
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(DoubleWritable.class);

    job.setOutputFormat(TextOutputFormat.class);

    FileOutputFormat.setOutputPath(job, TMP_OUTPUT);
```

As you can see there are 3 major sections.

The first section is about setting the classes for output key and output value.

The second section is about setting the format of your output. In this case this is TextOutputFormat, it outputs key separated by tabstops ('\t') from the value. Each record (key+value) is separated by a newline ('\n').

The third and last section is about setting the path where your output should go. You can use the static method in your choosen Outputformat as well as the convenience method in BSPJob:

```
job.setOutputPath(new Path("/tmp/out"));
```

If you don't provide output, no output folder or collector will be allocated.

Using Output

From your BSP, you can output like this:

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

    peer.write(new Text("Estimated value of PI is"), new DoubleWritable(3.14));

}
```

Note that you can always output, even from Setup or Cleanup methods!

Custom Outputformat

You can implement your own outputformat. It is similar to Hadoop MapReduce's output formats, so you can use existing literature to get into it.

Communication Model

Within the bsp() function, you can use the powerful communication functions for many purposes using BSPPeer. We tried to follow the standard library of BSP world as much as possible. The following table describes all the functions you can use:

Function	Description
send(String peerName, BSPMessage msg)	Send a message to another peer.
getCurrentMessage()	Get a received message from the queue.
getNumCurrentMessages()	Get the number of messages currently in the queue.
sync()	Starts the barrier synchronization.
getPeerName()	Get the peer name of this task.
getPeerName(int index)	Gets the n-th peer name.
getNumPeers()	Get the number of peers.
getAllPeerNames()	Get all peer names (including "this" task). (Hint: These are always sorted in ascending order)

The send() and all the other functions are very flexible. Here is an example that sends a message to all peers:

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

  for (String peerName : peer.getAllPeerNames()) {
    peer.send(peerName,
      new LongMessage("Hello from " + peer.getPeerName(), System.currentTimeMillis()));
  }

  peer.sync();
}
```

Synchronization

When all the processes have entered the barrier via the sync() function, the Hama proceeds to the next superstep. In the previous example, the BSP job will be finished by one synchronization after sending a message "Hello from ..." to all peers.

But, keep in mind that the sync() function is not the end of the BSP job. As was previously mentioned, all the communication functions are very flexible. For example, the sync() function also can be called in a for loop so that you can use to program the iterative methods sequentially:

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

  for (int i = 0; i < 100; i++) {
    // send some messages
    peer.sync();
  }

}
```

The BSP job will be finished only when all processes have no more local and outgoing queues entries and all processes done or is killed by the user.

Counters

Just like in Hadoop MapReduce you can use Counters.

Counters are basically enums that you can only increment. You can use them to track meaningful metrics in your code, e.G. how often a loop has been executed.

From your BSP code you can use counters like this:

```
// enum definition
enum LoopCounter{
  LOOPS
}

@Override
public void bsp(
    BSPPeer<NullWritable, NullWritable, Text, DoubleWritable, DoubleWritable> peer)
    throws IOException, SyncException, InterruptedException {
  for (int i = 0; i < iterations; i++) {
    // details ommitted
    peer.getCounter(LoopCounter.LOOPS).increment(1L);
  }
  // rest ommitted
}
```

Setup and Cleanup

Since 0.4.0 you can use Setup and Cleanup methods in your BSP code. They can be inherited from BSP class like this:

```
public class MyEstimator extends
     BSP<NullWritable, NullWritable, Text, DoubleWritable, DoubleWritable> {

   @Override
   public void setup(
       BSPPeer<NullWritable, NullWritable, Text, DoubleWritable, DoubleWritable> peer)
       throws IOException {
     //Setup: Choose one as a master
     this.masterTask = peer.getPeerName(peer.getNumPeers() / 2);
   }

   @Override
   public void cleanup(
       BSPPeer<NullWritable, NullWritable, Text, DoubleWritable, DoubleWritable> peer)
       throws IOException {
     // your cleanup here
   }

   @Override
   public void bsp(
       BSPPeer<NullWritable, NullWritable, Text, DoubleWritable, DoubleWritable> peer)
       throws IOException, SyncException, InterruptedException {
     // your computation here
   }
 }
```

Setup is called before bsp method, and cleanup is executed at the end after bsp. You can do everything in setup and cleanup: sync, send, increment counters, write output or even read from the input.

Combiners

Combiners are used for performing message aggregation to reduce communication overhead in cases when messages can be summarized arithmetically e.g., min, max, sum, and average at the sender side. Suppose that you want to send the integer messages to a specific processor from 0 to 1000 and sum all received the integer messages from all processors.

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

      for (int i = 0; i < 1000; i++) {
            peer.send(masterTask, new IntegerMessage(peer.getPeerName(), i));
      }
      peer.sync();

      if (peer.getPeerName().equals(masterTask)) {
        IntegerMessage received;
        while ((received = peer.getCurrentMessage()) != null) {
          sum += received.getData();
        }
      }
    }
```

If you follow the previous example, Each bsp processor will send a bundle of thousand Integer messages to a masterTask. Instead, you could use a Combiners in your BSP program to perform a sum Integer messages and to write more concise and maintainable as below, that is why you use Combiners.

```
public static class SumCombiner extends Combiner {

  @Override
  public BSPMessageBundle combine(Iterable<BSPMessage> messages) {
    BSPMessageBundle bundle = new BSPMessageBundle();
    int sum = 0;

    Iterator<BSPMessage> it = messages.iterator();
    while (it.hasNext()) {
      sum += ((IntegerMessage) it.next()).getData();
    }

    bundle.addMessage(new IntegerMessage("Sum", sum));
    return bundle;
  }

}
```

Implementation notes

Internal implementation details

BSPJobClient

1. Create the splits for the job 2. writeNewSplits() 3. job.set("bsp.job.split.file", submitSplitFile.toString()); 4. Sets the number of peers to split.lenth

JobInProgress

1. Receives splitFile 2. Add split argument to TaskInProgress constructor

Task

1. Gets his split from Groom 2. Initializes everything in BSPPeerImpl