Apache Hama (v0.2) : User Guide

a BSP-based distributed computing framework

Edward J. Yoon
<edwardyoon@apache.org>

Introduction

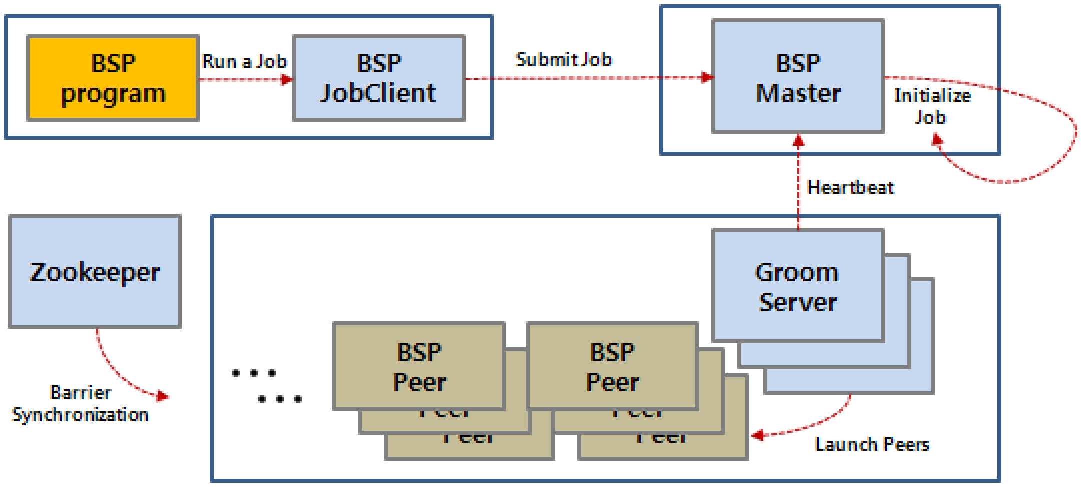
HAMA is a distributed computing framework based on BSP (Bulk Synchronous Parallel) [1] computing techniques for massive scientific computations (e.g., matrix, graph, network, ..., etc), it is currently being incubated as one of the incubator project by the Apache Software Foundation.

The greatest benefit of the adoption of BSP computing techniques is the opportunity to speed up iteration loops during the iterative process which requires several passes of messages before the final processed output is available, such as finding the shortest path, and so on. (See “Random Communication Benchmark” results at http://wiki.apache.org/hama/Benchmarks)

Hama also provides an easy-to-program, as well as a flexible programming model, as compared with traditional models of Message Passing [2], and is also compatible with any distributed storage (e.g., HDFS, HBase, …, etc), so you can use the Hama BSP on your existing Hadoop clusters.

Hama Architecture

Hama consists of three major components: BSPMaster, GroomServers and Zookeeper. It is very similar to Hadoop architecture, except for the portion of communication and synchronization mechanisms.



BSPMaster

BSPMaster is responsible for the following:

- Maintaining groom server status.
- Controlling super steps in a cluster.
- Maintaining job progress information.
- Scheduling Jobs and Assigning tasks to groom servers
- Disseminating execution class across groom servers.
- Controlling fault.
- Providing users with the cluster control interface.

A BSP Master and multiple grooms are started by the script. Then, the BSP master starts up with a RPC server for groom servers. Groom servers start up with a BSPPeer instance and a RPC proxy to contact the BSP master. After it's started, each groom periodically sends a heartbeat message that encloses its groom server status, including maximum task capacity, unused memory, and so on.

Each time the BSP master receives a heartbeat message, it brings the groom server status up-to-date. Then, the BSP master makes use of the groom servers' status in order to effectively assign tasks to idle groom servers and returns a heartbeat response that contains assigned tasks and others actions that a groom server has to do. For now, we have a FIFO [3] job scheduler and very simple task assignment algorithms.

GroomServer

A Groom Server (shortly referred to as groom) is a process that performs BSP tasks assigned by the BSPMaster. Each groom contacts the BSPMaster and takes assigned tasks and reports its status by means of periodical piggybacks with the BSPMaster. Each groom is designed to run with HDFS or other distributed storages. Basically, a groom server and a data node should be run on one physical node.

Zookeeper

A Zookeeper is used to manage the efficient barrier synchronisation of the BSPPeers. (Later, it will also be used for the area of a fault tolerance system.)

BSP Programming Model

Hama BSP is based on the Bulk Synchronous Parallel Model (BSP), in which a computation involves a number of supersteps, each having many distributed computational peers that synchronize at the end of the superstep. Basically, a BSP program consists of a sequence of supersteps. Each superstep consists of the following three phases:

- Local computation
- Process communication
- Barrier synchronization

NOTE that these phases should be always be in sequential order. (To see more detailed information about “Bulk Synchronous Parallel Model” - http://en.wikipedia.org/wiki/Bulk_synchronous_parallel)

Hama provides a user-defined function “bsp()” that can be used to write your own BSP program. The bsp() function handles the whole parallel part of the program. (This means that the bsp() function is not a iteration part of the program.)
In the 0.2 version, it only takes one argument, which is a communication protocol interface. Later, more arguments such as input or reporter could also be included.

Communication

Within the bsp() function, you can use the powerful communication functions for many purposes using BSPPeerProtocol. 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)	Sends a message to another peer.
put(BSPMessage msg)	Puts a message to local queue.
getCurrentMessage()	Returns a received message.
getNumCurrentMessages()	Returns the number of received messages.
sync()	Barrier synchronization.
getPeerName()	Returns a peer’s hostname.
getAllPeerNames()	Returns all peer’s hostname.
getSuperstepCount()	Returns the count of supersteps.

The send(), put() and all the other functions are very flexible. For example, you can send a lot of messages to any of processes in bsp() function:

```
public void bsp(BSPPeerProtocol bspPeer) throws IOException,
                KeeperException, InterruptedException {
        for (String otherPeer : bspPeer.getAllPeerNames()) {
                String peerName = bspPeer.getPeerName();
                BSPMessage msg =
                        new BSPMessage(Bytes.toBytes(peerName), Bytes.toBytes(“Hi”));
                bspPeer.send(peerName, mgs);
        }
        bspPeer.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 “Hi” 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 sequentally:

```
        public void bsp(BSPPeerProtocol bspPeer) throws IOException,
                        KeeperException, InterruptedException {
                for (int i = 0; i < 100; i++) {
                        ….
                        bspPeer.sync();
                }
        }
```

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

Example

Let’s take one more practical example. We may want to estimate PI using Hama BSP.
The value of PI can be calculated in a number of ways. In this example, we are estimating PI by using the following method:

- Each task locally executes its portion of the loop a number of times.

```
iterations = 10000
circle_count = 0

do j = 1,iterations
        generate 2 random numbers between 0 and 1
        xcoordinate = random1
        ycoordinate = random2
        if (xcoordinate, ycoordinate) inside circle
        then circle_count = circle_count + 1
```

```
end do

PI = 4.0*circle_count/iterations
```

- One task acts as master and collects the results through the BSP communication interface.

```
PI = pi_sum / n_processes
```

1) Each process computes the value of Pi locally, and 2) sends it to a master task using the send() function. Then, 3) the master task can recieve the messages using the sync() function. Finally, we can calculate the average value of the sum of PI values from each peer, shown below:

```
public void bsp(BSPPeerProtocol bspPeer) throws IOException,
                KeeperException, InterruptedException {
        int in = 0, out = 0;
        for (int i = 0; i < iterations; i++) {
                double x = 2.0 * Math.random() - 1.0, y = 2.0 * Math.random() - 1.0;
                if ((Math.sqrt(x * x + y * y) < 1.0)) {
                        in++;
                } else {
                        out++;
                }
        }

        byte[] tagName = Bytes.toBytes(bspPeer.getPeerName());
        byte[] myData = Bytes.toBytes(4.0 * (double) in / (double) iterations);
        BSPMessage estimate = new BSPMessage(tagName, myData);

        bspPeer.send(masterTask, estimate);
        bspPeer.sync();

        if (bspPeer.getPeerName().equals(masterTask)) {
                double pi = 0.0;
                int numPeers = bspPeer.getNumCurrentMessage();
                BSPMessage received;
                while ((received = bspPeer.getCurrentMessage()) != null) {
                        pi += Bytes.toDouble(received.getData());
                }

                pi = pi / numPeers;
                writeResult(pi);
        }
}
```

BSP Job Configuration

The BSP job configuration interface is almost the same as the MapReduce job configuration:

```
HamaConfiguration conf = new HamaConfiguration();
BSPJob bsp = new BSPJob(conf, MyBSP.class);
bsp.setJobName("My BSP program");
bsp.setBspClass(MyEstimator.class);
…
bsp.waitForCompletion(true);
```

If you are familiar with MapReduce programming, you can skim this section.

Getting Started With Hama

Requirements

Current Hama requires JRE 1.6 or higher and ssh to be set up between nodes in the cluster:

- hadoop-0.20.x for HDFS (distributed mode only)
- Sun Java JDK 1.6.x or higher version

Startup Scripts and Run examples

The {$HAMA_HOME/bin} directory contains some script used to start up the Hama daemons.

- start-bspd.sh - Starts all Hama daemons, the BSPMaster, GroomServers and Zookeeper.

To start the Hama, run the following command:

```
$ bin/start-bspd.sh
```

This will startup a BSPMaster, GroomServers and Zookeeper on your machine.
To run the examples, run the following command:

```
$ bin/hama jar build/hama-0.2.0-dev-examples.jar pi or test
```

Then, everything is OK if you see the following:

```
[root@test1 hama-trunk]# bin/start-bspd.sh
test1: starting zookeeper, logging to /usr/local/src/hama-trunk/bin/../logs/hama-root-zookeeper-test1.out
starting bspmaster, logging to /usr/local/src/hama-trunk/bin/../logs/hama-root-bspmaster-test1.out
test2: starting groom, logging to /usr/local/src/hama-trunk/bin/../logs/hama-root-groom-test2.out
test3: starting groom, logging to /usr/local/src/hama-trunk/bin/../logs/hama-root-groom-test3.out
test4: starting groom, logging to /usr/local/src/hama-trunk/bin/../logs/hama-root-groom-test4.out
test1: starting groom, logging to /usr/local/src/hama-trunk/bin/../logs/hama-root-groom-test1.out
[root@test1 hama-trunk]#
[root@test1 hama-trunk]#
```

```
[root@test1 hama-trunk]# bin/hama jar hama-0.2.0-dev-examples.jar pi
11/02/15 15:49:44 INFO bsp.BSPJobClient: Running job: job_201102151549_0001
11/02/15 15:49:51 INFO bsp.BSPJobClient: The total number of supersteps: 1
Estimated value of PI is 3.1445
Job Finished in 6.683 seconds
[root@test1 hama-trunk]# bin/hama jar hama-0.2.0-dev-examples.jar test
11/02/15 15:51:50 INFO bsp.BSPJobClient: Running job: job_201102151549_0002
11/02/15 15:52:17 INFO bsp.BSPJobClient: The total number of supersteps: 4
Each task printed the "Hello World" as below:
Tue Feb 15 15:51:51 KST 2011: Hello BSP from 1 of 4: test1:61000
Tue Feb 15 15:52:00 KST 2011: Hello BSP from 2 of 4: test2:61000
Tue Feb 15 15:52:06 KST 2011: Hello BSP from 3 of 4: test3:61000
Tue Feb 15 15:52:11 KST 2011: Hello BSP from 4 of 4: test4:61000
```

(To see more detailed information about “Getting Started With Hama” - http://wiki.apache.org/hama/GettingStarted)

Command Line Interfaces

In previous section, the jar command has introduced. In addition to this command, Hama provides several command for BSP job administration:

> Usage: hama job [-submit <job-file>] | [-status <job-id>] | [-kill <job-id>] | [-list [all]] | [-list-active-grooms] | [-kill-task <task-id>] | [-fail-task <task-id>]

Command	Description
-submit <job-file>	Submits the job.
-status <job-id>	Prints the job status.
-kill <job-id>	Kills the job.
-list [all]	-list all displays all jobs. -list displays only jobs which are yet to be completed.
-list-active-grooms	Displays the list of active groom server in the cluster.
-list-attempt-ids <jobId> <task-state>	Displays the list of tasks for a given job currently in a particular state (running or completed).
-kill-task <task-id>	Kills the task. Killed tasks are NOT counted against failed attempts.
-fail-task <task-id>	Fails the task. Failed tasks are counted against failed attempts.

References

1. Bulk Synchronous Parallel, http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

2. Message Passing, http://en.wikipedia.org/wiki/Message_Passing
3. FIFO, http://en.wikipedia.org/wiki/FIFO

* *Thanks to reviewers (Tommaso, Franklin, Filipe, and all the Hama developer team members).*