Apache Hama Programming

Bulk Synchronous Parallel Programming Model

Introduction

Hama is a distributed computing framework based on BSP (Bulk Synchronous Parallel) [1] computing technique for massive scientific computations (e.g., matrix, graph, network, etc) designed to run on massive data-sets stored in Hadoop Distributed File System (HDFS). It is currently an incubator project by the Apache Software Foundation.

Apache Hama leverages BSP computing techniques to speed up iteration loops during the iterative process that requires several passes of messages before the final processed output is available. It provides an easy and flexible programming model, as compared with traditional models of Message Passing [2].It is compatible with any distributed storage (e.g., HDFS, HBase, etc), so you can use the Hama BSP on your existing Hadoop clusters. Finding shortest paths, value of PI etc. are some of the problems tackled by Hama today.(See http://wiki.apache.org/hama/Benchmarks-Random Communication Benchmark" results).

This document explains in some detail how bulk synchronous programming model works. It uses a simple example for understanding. **All explanations are biased to the way BSP model is expressed in Hama**. **This document does not intend to describe BSP introduced in 1990. You are encouraged to read the paper for the same.** Once different nuances of BSP are explained, we delve into how Hama takes a shot at this using a flexible programming model on scalable and huge data-set. Hama also provides a BSP implementation based API for graph operations. It also introduces on how Hama can be used for stream processing.

Bulk Synchronous Parallel Programming Model

In Bulk Synchronous Parallel Model (BSP), a solution is described as a series of units of computation running in parallel. This unit of computation is termed as a *superstep*. These tasks or, more appropriately, the task-executor or the process running the task also get referred to as *peer*. In this document, we would be occasionally using both the terms interchangeably. BSP model also accounts for the communication between them. This is done by defining a global synchronization step at the end of each superstep. Let's understand this with the help of the figure below.

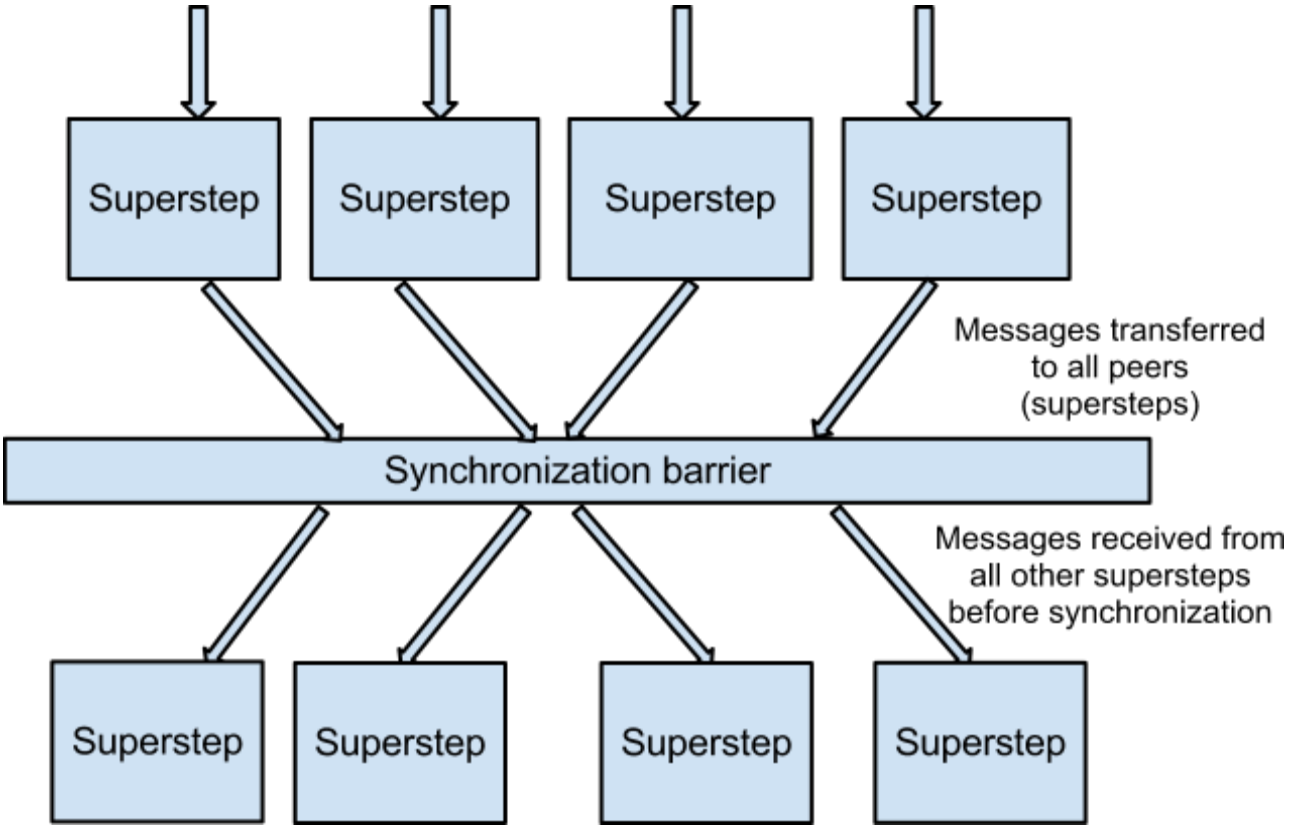


Figure - Snapshot of four supersteps running in parallel. The synchronization barrier implies the state where every superstep waits for other parallely running superstep to finish and enter the barrier.

The figure shows four supersteps running in parallel. All the four supersteps on completion of their computation unit enter a mode called *synchronization barrier.* Before entering the phase, every peer sends output messages to other peers if needed.

Thus, when a peer enters the barrier synchronization mode, it is implied that :

- Peer has completely executed the superstep.
- All the messages for each of the others peers are (most of the times reliably) sent out.
- Peer is waiting for all other peers to enter the barrier.

When all the peers enter the synchronization mode, the system is in a global consistent state. In this state the supersteps have completed their tasks and have sent all their messages to other peers. As soon as the last peer enters the barrier synchronization, all the peers start leaving the barrier.

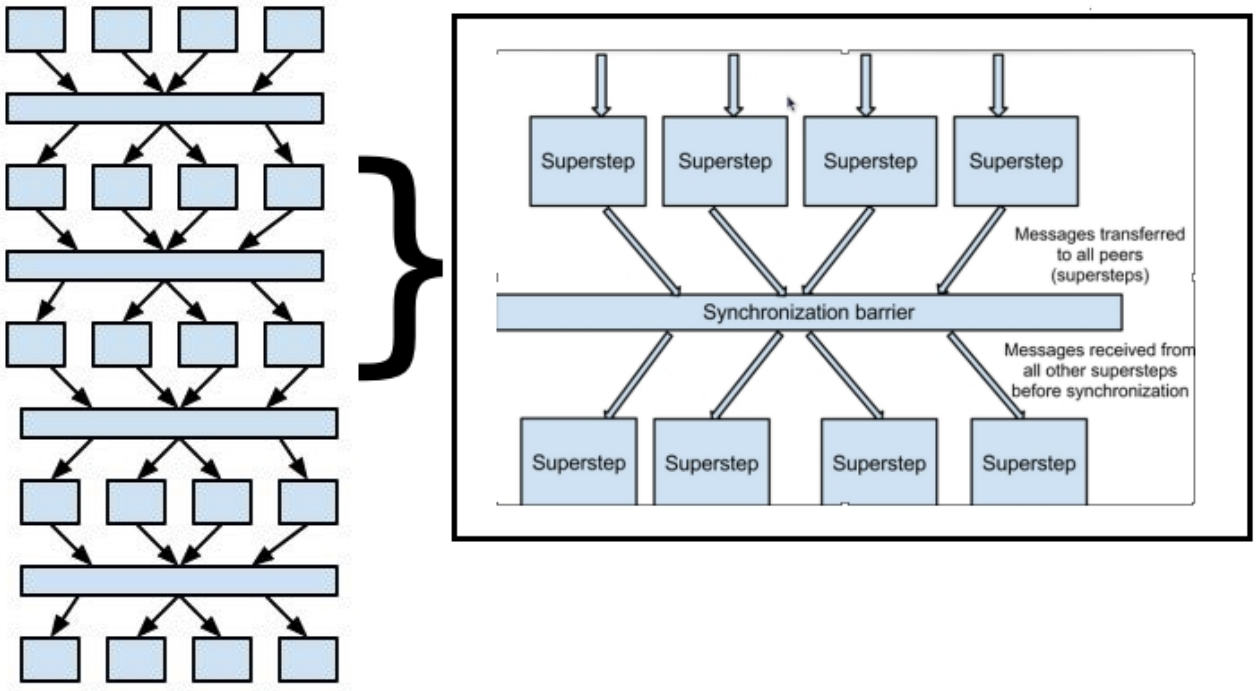
Thus, when a peer leaves a barrier, it is implied that :

- The peer is about to start working on the next superstep.
- There are no other peers in the system working on the previous superstep.
- A peer gets all the messages, sent to it by other peers in the previous superstep, as input for the new superstep execution.

To summarize from above, a BSP task should account for the following three phases:

- Local computation
- Process communication
- Barrier synchronization.

NOTE: These phases should always be in sequential order. The figure below shows how the previous figure was a part of the whole computation.



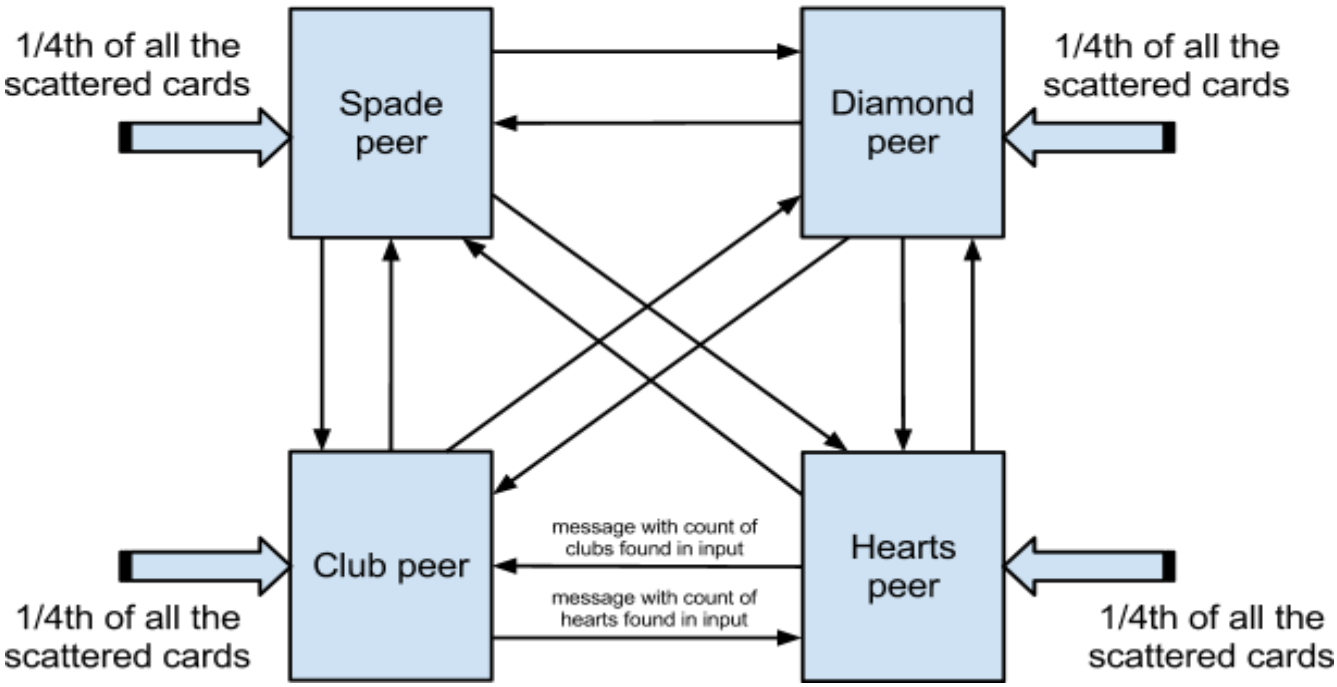
We are going to use a simple example to understand what is expected from each of the phases in this model. Let's take a situation where someone over-engineered the task of finding if his scattered multiple deck of cards have all the cards per suit. He is also interested in knowing how many deck of cards are present from the mess. And also what cards per suit is needed to make up for missing cards. Here is the over-engineering part - he blindly scans all the cards into files in computer. His goal is to find the number of decks of cards that are present and in the process

find what cards are missing per suit i.e. heart, diamond, clubs and spades. (Our main goal is to understand the model and not get personal with the problem with other programming model).

Pre-Processing step

In pre-processing step, we define the number of peers required for the job and then split the initial inputs for each of the peers to run their supersteps. This step is optional considering that there could be other nature of jobs where the initial data is picked up as random number generated. For our example, we define four peers running in parallel. The four suits are distributed among the four peers. The goal of the peer is to hold the count for each face belonging to the suit it is responsible for and by then discover how many decks are available. The input that each peer reads, has random cards with different suits. The job of the peer is to count the cards on reading each faces of cards and notifying other peers on the suits for which each of them are responsible for. The figure below depicts the peer design.

Figure: A spade peer is responsible for counting all the spades, diamond peer counts diamonds, heart peer counts all the hearts and clubs peer counts all club suit cards. The messages incoming to a spade peer implies the count of spades found by other peers from their respctive inputs. Same is the case with inputs coming in to diamond peer, hearts peer and clubs peer.



This is the task structure for the solution. We see that each task has to send messages to other tasks. We can assume here that the four tasks would be executed on four different processors. Each task now reads all records, updates its counters and sends messages to the other peers if needed. In summary, we have just partitioned the input into four and assigned four peers the suit of the card each of them are responsible for. The following is the pseudo-code for the pre-processing step:

```
// Every peer calls itself with an ID assigned by the system.
Input:
```

```
myID - The id of the peer that is running this superstep
Pre-Process (myID):
        Suits={SPADE,CLUB,HEARTS,DIAMOND}
        CardSuit mySuit = Suits[myID]
        numDecks = 0
         input = getSplit(myID)
```

In the next few sections we will get into the main phases of design and use this example to understand the working.

Computation Superstep

A superstep is a unit of computation that would be running in parallel with others. It could be implemented to be aware of other supersteps running in parallel. The input for a superstep could be from messages it receives from peers or from other source of information. The model assures that the messages it reads in the current superstep were to sent to it in the previous superstep. The supersteps running in parallel need not be of the same nature. We can have a task that does multiplication in parallel to task doing division on its inputs. Thus we can implement parallel iterative algorithms by breaking them into supersteps and defining the communication between each of them in a consistent manner. Coming back to the deck of cards example, we now have to design superstep that would count the number of cards per face belonging to a suit the superstep is assigned to. Also, for any other suit read, it has to send a message to the concerned peer. The following pseudo code explains the first superstep for reading the 1/4th split.

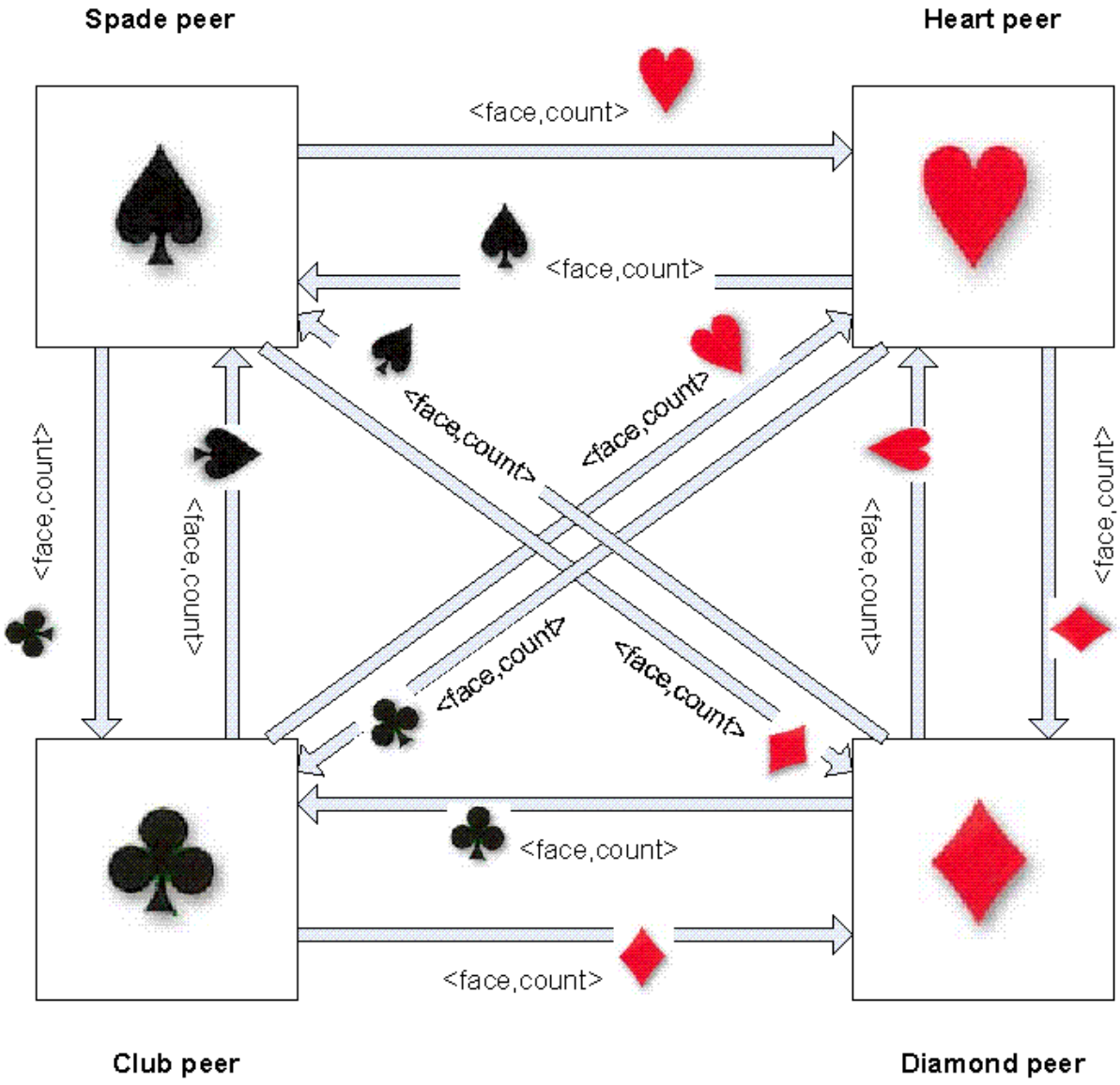
```
// map for holding counts per suit per face.
Map<CardFace, count> spadeCountMap  // for counting faces in spades
Map<CardFace, count> diamondCountMap // for counting faces in diamonds
Map<CardFace, count> heartsCountMap // for counting faces in hearts
Map<CardFace, count> clubCountMap // for counting faces in spades
Input:
input - the split of scattered cards assigned to this peer.
peers - the list of all peers
messagingService - Messaging Service to send messages to other peers.


InputReadingSuperstep (Inputsplit input, Peers[] peers, InterPeerMessaging messagingService):

        while(input.hasNextCard()) do
                Card card = input.nextCard()
                CardSuit suit = card.getSuit()
                CardFace face = card.getFace()

                if suit is SPADE
                        spadeCountMap.incrementCountForFace(face)
                if suit is CLUB
                        clubCountMap.incrementCountForFace(face)
                if suit is DIAMOND
                        diamondCountMap.incrementCountForFace(face)
```

```
                if suit is HEARTS
                        heartsCountMap.incrementCountForFace(face)

        end
        // Send counts to peers that is not my responsibility
        for each CardSuit not equal to mySuit do
                // say choose spadeCountMap for SPADE
                // send <face, count> tuple to the peer counting spades
                for each face in the map for the suit. do
                        messagingService.send tuple<face,count> to peers[CardSuit ]
                end
        end

        // Synchronize with other peers
        barrier_sync()

End of Superstep
```

The above pseudo-code explains just the first part of the solution. Every peer maintains a map for each face. Each of this map holds a <key,value> pair for the face and the number of occurrences of the face. In this superstep, a superstep reads all the cards in its input split. For each card, it updates the counter for the face in the map of that suit. Say a Spade King card would increment the count in the map "**spadeCountMap"** for the face "**King**". Once all the cards are read, a peer sends all the counts accumulated to other peers. Thus a spade peer would send all the diamond suit related counts to the Diamond peer, Hearts counts to Heart peer and Club face counts to Club peer. The messages are sent in the tuple format <face, count> where count is the number of occurrences of face in the map. In the end of the superstep, we see that we call *barrier_sync()* to synchronize the results with all the peers. In the next two sections we will look into the message passing and synchronization process in detail.

Communication

Inter-peer communication in BSP model is consistent and predictable. The messages are sent reliably. Reliability here implies that if the send operation completes successfully, the sender peer is assured that the messages were sent successfully to a location such that the intended peer would be able to read them in the next superstep. The sender as well as the receiver maintains a queue for the messages for each peer. It is up to the superstep to detect the origin of the message in its computation. Every peer sends messages to every other peers directly. So in the example, a diamond suit peer sends counts of faces in the spade suit, encountered in the spade peer directly. The figure below shows the communication between peers. As an example, we can see that all the heart suit <face, count> tuples are sent to the heart peer by all the other peers. The heart peer thus receives counts for its faces from all the peers. These messages are read only after the synchronization process. We get into more details in the next section.



Synchronization

This is the main characteristic of the model that differentiates itself from others models of parallel-computation like PRAM. All tasks enter the barrier synchronization mode after executing a superstep and sending messages to other peers. Just like a typical barrier implementation, all the peers are blocked until the last task has entered the barrier. As soon as the last peer have entered the barrier, every other peer is sure about the global system of the state. Every peer is guaranteed to have all the messages that it gets from its peers in the last superstep. These messages are read in the next superstep. With this confidence on the global state, the peers leave the barrier synchronization to continue working on the next superstep.

When the function call *barrier_sync* returns, every peer is sure about the state of other peers. The heart peer is assured that all the messages required to update its count is available in its

input queue. Think of a situation where the spade peer never got a heart suit in its input. Then, there will not be any messages sent to the heart peer by spade peer. However, the heart peer does not have to wait to hear from all the peers. If a message is not received from the spade peer, it could be only because the spade peer did not get any heart suit cards. (We are not considering failure scenarios here.) This confidence could be achieved only because of the synchronization process.

In the next superstep, the peers update their counts from other peers. Our task is not over yet. We have to find the number of missing cards, we should know the number of deck of cards that were present in the first place. We assume that the cards could only go missing and no duplicate card was present in the mess. The peer should send the maximum among the counts it encountered after updating all the counters for each faces. So say, even if a peer has three cards of all the faces except for a King whose count was four, we should believe that there were four pack of cards in the initial mess. Before the next synchronization, all peers should now exchange information among them on the maximum count they found. The pseudo-code for the next superstep is as below.

Input:
myMap- could be spadeCountMap, diamondCountMap, heartsCountMap, clubCountMap depeding on the id of the peer and the card face it is assigned
peers - the list of all peers
messagingService - Messaging Service to send messages to other peers.

FindingMyMaxSuperstep(Map<CardFace, count> myMap, Peers[] peers, InterPeerMessaging messagingService):

```
    while(messagingService.hasNextMessage()) do
            Message message = messagingService.getMessage()
            CardFace face = message.getFace()
            Count count = message.getCount()
            myMap.incrementCountForFace(face, count)
     end

     // Find the maximum count from myMap and send it to all other peers
     max = max(myMap.counterValues)


     for each peerID in peers that is not equal to myID
            // send <max> to the peer

            messagingService.send max to peers[peerID ]
     end

     // Synchronize with other peers
     barrier_sync()
```

End of Superstep

We could have avoided this superstep by each peer sending all the peers the counts for all the suits it encountered. This would have resulted in more messages sent among peers. It

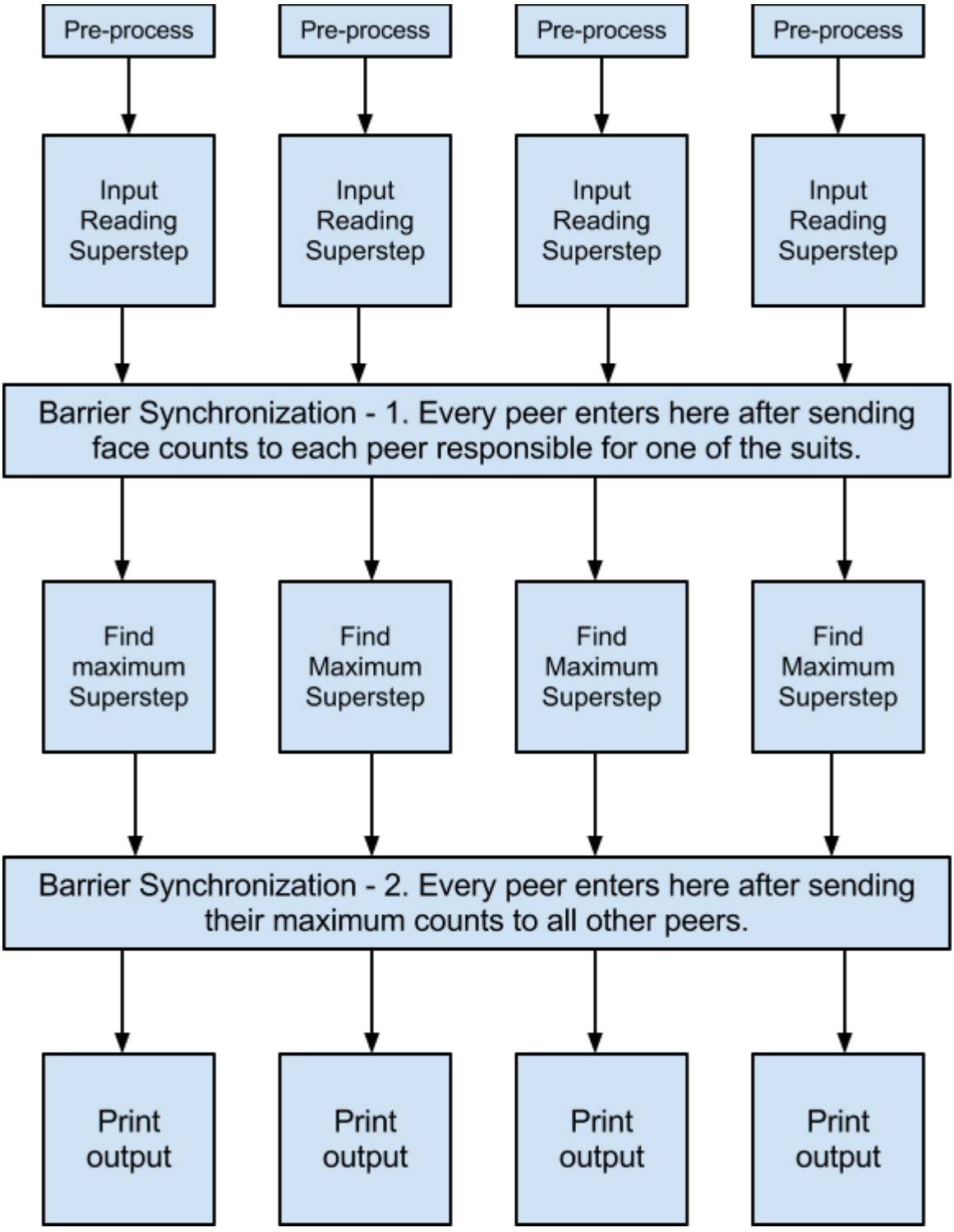
would have been a more complicated example. Now in the next superstep every peer reads the maximum count from all other peers and find the maximum among themselves. This would be helpful in finding the missing cards.

Input:
myMap- could be spadeCountMap, diamondCountMap, heartsCountMap, clubCountMap depending on the id of the peer and the card face it is assigned
myMax - the max found in the previous superstep
peers - the list of all peers
messagingService - Messaging Service to send messages to other peers.

```
PrintMissingCardsSuperstep(Map<CardFace, count> myMap,  Count myMax, Peers[] peers,
                                        InterPeerMessaging messagingService):
        max = myMax
        while(messagingService.hasNextMessage()) do
                Message message = messagingService.getMessage()
                Count peerMax  = message.getCount()
                if (peerMax > max)
                   max = peerMax
         end

         For each face in myMap do
                 Count count = myMap.get(face)
                 if(max - count > 0)
                       print "<max - count> cards missing for <face> in suit <mySuit>
         end
```

End of Superstep

In this way we used four peers and three supersteps to find all the missing cards. The flow of execution is summarized in the figure below. You are free to verify the features of this model in every course of computation.



In the next section, you will see how Apache Hama is developed to support such a computation model and the APIs available for programmers to implement their BSP solutions.

Hama BSP API

In this section we look into details of how Apache Hama provides APIs to define BSP tasks.

Expressing BSP Task

A BSP program consists of a sequence of supersteps. Each superstep consists of a program. It only takes one argument "BSPPeer", which contains communication, counters, and IO interfaces.

A BSP Task can be implemented in two ways:

1. Inheriting from *org.apache.hama.bsp.BSP* class.
2. Defining a chain of sub-classes of org.apache.hama.bsp.Superstep (experimental).

In the first case, 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. It gets called once, not all over again. There are also *setup()* and *cleanup()* which will be called at the beginning and end of your computation respectively. *cleanup()* is guaranteed to run after the computation or in case of failure. To express your BSP solution, you can simply override the functions you need from the BSP class. To express different nature of the supersteps executing in parallel, one can define the nature of task as per the ID of the peer. It is also common to elect one of the peer as a leader and co-ordinate the results from other peers.

```
public static class MyClass extends
      BSP<K1, V1, K2, V2, M extends Writable> {
    @Override
    public void setup(..){
    }

    public void cleanup(..){
    }

    public void bsp(peer){
      if(peerId == ){
      . …..
      ….... // BSP logic 1
      }
      else {
      . …..
      ….... // BSP logic 2
      }
      peer.sync()
    }

}
```

The second way of expressing BSP programs is by using the *org.apache.hama.bsp.Superstep* class. This is still in an experimental stage. In this method, sub-class from the *Superstep* class and define the BSP task in the *compute* member function. The framework does the synchronization for you after the function returns.

```
protected static class MySuperstep
      extends
      Superstep<K1, V1, K2, V2, M extends Writable> {

    @Override
    protected void compute(peer){
       // Implementation
       // No peer.sync required it is done by the
       // framework after the compute function exits.
    }
}
```

Input and Output

General Information

Since Hama 0.4.0 we provided an input and output system for BSP jobs. We choose the key/value model from Hadoop to provide a coherent API to widely used products like Hadoop MapReduce (SequenceFiles) and HBase (Column-storage).

Input

Provide an InputFormat and a Path where to find the input to specify the input for the jobs.

```
   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 as follows:

```
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: These paths must be separated by a comma. In the case of a *SequenceFileInputFormat*, the key and value pair are parsed from the header. When you 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. 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> 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 documentation for more details 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 may not consume the whole input to reopen it. You can implement your own input format. It is similar to Hadoop MapReduce's input formats, so you can use existing literature to get into it.

Output

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

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

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

You can see there are three 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 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 do not provide output, no output folder or collector will be allocated. The following code shows an example of setting the output path:

```
@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: You can always output, even from Setup or Cleanup methods. 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 the 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> 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, 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.

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> 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 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). You can use the counters in the following way:

```
 // 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
    }
```

Counters are in 0.4.0 not usable for flow controls, since they are not synced during sync phase. Watch HAMA-515 for details.

Setup and Cleanup

Since 0.4.0 you can use the *Setup* and *Cleanup* methods in your BSP code. They can be inherited from the 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 the 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 1,000 and sum all received integer messages from all processors.

```
public void bsp(BSPPeer<NullWritable, NullWritable, NullWritable,
NullWritable> 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 = null;
        while ((received = (IntegerMessage) 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;
    }
  }
```

Hama Graph API

Apache Hama leverages the aforesaid BSP APIs to provide a Google Pregel like graph computational model. This way we define a BSP job to define a vertex in the graph. The messaging infrastructure between the peers can be envisaged as how vertices in a graph can update the state of their neighbors.

Hama provides *org.apache.hama.graph.Vertex* class to define a vertex in a graph. Overriding the compute function will give you the ability to implement graph algorithms by the “Thinking like a vertex”-model that is described in the Google Pregel Paper [3].

The Vertex has three type arguments, first is the ID type of the vertex (an unique identifier of your vertex), the second is the value type of your vertex (it will be used for messaging) and the third type is the edge value type.

The “Edge” is a tuple of the vertex identifier it points to and the value it holds. All three arguments need to be writable in order to be transferred to the task and between the tasks. In the graph API you will deal with directed graphs, so your vertex has “n”-outgoing edges to other vertices.

For example you can see the types for inlink count:

```
public class InlinkCount extends Vertex<Text, IntWritable, NullWritable>
```

InlinkCount is going to measure how many edges are pointing to a specific vertex, whereas each vertex is defined by a unique Text (a serializable/writable String), the value is a serializable integer to track how many ingoing links we have. The last argument is “null” or NullWritable because we have need a weight or value on the edge between two vertices.

NOTE**:** Make sure that you implement *hashCode()* and *equals()* in your Identifiertype, it is heavily used in Maps. Usually all Writable types that ship with Hadoop and Hama will have these implemented, so you just have to take care on self-implemented types.

You are now able to override the *compute()* function to implement your graph logic. Let’s move forward to the architecture to better understand what works under the hood of this fancy API.

May the graphs be with you!

Architecture

If you read the BSP introduction above, you will be very fine with understanding how the graph model is expressed here.

The BSP that handles a graph API job is called “GraphJobRunner”, it handles all the managing of the vertices namely Input/Messaging/Output.

When submitting a graph job, a partitioner will partition your vertices with a hash function defined in the HashPartitioner on the given Vertex Identifier. Now each launched task has a chunk of your graph expressed by a tuple of a Vertex and its outgoing edges.

At setup time the BSP will load every vertex into main memory and put them into a HashMap of the identifier and the vertex, every outgoing edge is added to the vertex as a list. In mapping phase, the setup function of a vertex is called.

At the initialization step the value of a vertex is always null. So you have to initialize it correctly in the setup function or in the compute function.

If everything is mapped each task begins to iterate over the list of vertices and begins to call the compute function that needs to be overridden by the user’s algorithm.

Once this is all done and the first iteration of the graph algorithm is done, we are entering the real BSP phase. The BSP phase will run until the there are no global updates happening anymore or the defined number of iterations has been done. Global updates in this case are messages being sent, this is a convenience change opposed to the Pregel paper that has a vertex state machine where the user controls when a vertex is considered inactive (via voting).

In each iteration in the BSP phase we are iterating the vertices and calling compute, sending messages, syncing, parsing the messages that are incoming and feeding it back to compute.

At the end of the BSP phase the cleanup is run, in this step all vertices are flushed to the output with a ID-Value mapping. Meaning that if you define a Text identifier for your vertex and a IntWritable for your value, your output will be a <*Text,IntWritable*> pair.

This is of course a bit more complicated (combiners and aggregators are shown later) but for the sake of a tutorial this is enough to know what happens inside.

Example

A very simplistic example is the algorithm called *InlinkCount*, you find it in the examples package under *org.apache.hama.examples.InlinkCount.* Basically this example measures how many edges point to the current vertex by sending messages and reading them back.

```
  @Override
  public void compute(Iterator<IntWritable> messages) throws IOException {

    if (getSuperstepCount() == 0L) {
      sendMessageToNeighbors(new IntWritable(1));
    } else {
      while (messages.hasNext()) {
        IntWritable msg = messages.next();
        this.setValue(new IntWritable(this.getValue().get() + msg.get()));
      }
    }
  }
```

As you can see, in the first superstep (namely 0) you are sending a message to each neighbour with an integer = 1, representing an edge traversal.

In the following superstep, you are reading the messages back and incrementing the value of the vertex accordingly. (We know that this is solved pretty in-efficiently, but it is an example :P)

This example consists of actually two supersteps, in the first you are sending a message to the neighbour representing the edge. In the next superstep you are reading back the messages.
In this example many messages with the same value (1) are received, we can make this a bit more efficient by using a combiner.

Combiner

Combiners are used to combine different messages to a single message to save resources or computation time.

They are called before the compute method and can be constructed by inheriting from the abstract *org.apache.hama.bsp.Combiner* class. There you get an iterable over a couple of incoming messages and at best combine it to a single message.

In our *InlinkCount* example we could use a so called *SumCombiner* that will sum each value of the incoming message.

An example of a *SumCombiner* for *InlinkCount* could be:

```
public class SumCombiner extends Combiner<IntWritable>{
  int sum = 0;
  @Override
  public IntWritable combine(Iterable<IntWritable> messages) {
      for(IntWritable w : messages)
      sum+= w.get();
      return new IntWritable(sum);
  }
}
```

As you can see, we are combining “n”-messages to a single, making the algorithm in the compute function more efficient.

Aggregator

Another great tool in addition to Combiners are Aggregators. Aggregators are a mechanism for global communication, monitoring, or data observation.

That means that the value of a vertex will be observed after the compute function in a given Superstep “S” and globally aggregated after a superstep by a master task. In Superstep “S+1” it will be made available for all vertices.

Just like combiners there is an interface which will let you implement two functions, aggregate and getValue. Let’s look at the *MaxAggregator*:

```
public class MaxAggregator extends AbstractAggregator<IntWritable> {
  int max = Integer.MIN_VALUE;
  public void aggregate(IntWritable value) {
      if (value.get() > max) {
        max = value.get();
      }
  }
  public IntWritable getValue() {
      return new IntWritable(max);
  }
}
```

In this case we are inheriting from the *AbstractAggregator* which is offering us a bit more functions explained later. This function checks each value of a vertex in aggregate and compares it with a local maximum. If an iteration is done, the *getValue()* function is called to exchange this value with other tasks, therefore it needs to be writable just like the value of your vertex.

A master task chosen from the peers is now collecting each aggregated value and is going to successive aggregating observed values from each peer resulting in a final value. This value is being broadcasted again to the tasks and made available through the aggregator again.

We have extended the behavior of Aggregators defined in the Pregel paper by an observation before a computation and afterwards, you can see this in the Pagerank example. In Pagerank the value of a vertex (a double which is the rank of this vertex) is observed before the computation and afterwards to check convergence of the rank.

So at every superstep the aggregator will make a difference between the value before and after a computation, send it to a master task which will apply an average function over all differences of a vertex before and after compute. Afterwards the convergence average will be send back to all the tasks and made available through the aggregator again.

All the functions descsribed are available in the *AbstractAggregator* given more functionality than aggregate. It is also tracking how often something has been aggregated, making it easy for you to implement average functions or other fancy things.

Here is the example for the *AverageAggregator* used by Pagerank:

```
public class AverageAggregator extends AbsDiffAggregator {

  @Override
  public DoubleWritable finalizeAggregation() {
      return new DoubleWritable(getValue().get()
      / (double) getTimesAggregated().get());
  }

}
```

The *AverageAggregator* is just an Absolute Difference Aggregator, but is overriding the *finalizeAggregation* method that is called on the master task after every other aggregated value is consumed, right before it is sent back to the task.
As you can see as described above there is the function getTimesAggregated which is given you the number of calls globally made to aggregate so it can be used for averaging very well.

For additional information, please consult the JavaDoc, as it is pretty much documented in the Aggregator case.

GraphJob Submission

To submit a graph job there are just a few plenty things to take care of.
Let's take a closer look at the submission of our oftentimes-cited inlink count:

```
    // Graph job configuration
    HamaConfiguration conf = new HamaConfiguration();
    GraphJob inlinkJob = new GraphJob(conf, InlinkCount.class);
    // Set the job name
    inlinkJob.setJobName("Inlink Count");
    inlinkJob.setInputPath(new Path(args[0]));
    inlinkJob.setOutputPath(new Path(args[1]));
    inlinkJob.setInputFormat(SequenceFileInputFormat.class);
    inlinkJob.setInputKeyClass(VertexWritable.class);
    inlinkJob.setInputValueClass(VertexArrayWritable.class);
    inlinkJob.setOutputFormat(SequenceFileOutputFormat.class);
    inlinkJob.setOutputKeyClass(Text.class);
    inlinkJob.setOutputValueClass(IntWritable.class);

    inlinkJob.setVertexClass(InlinkCount.class);
    inlinkJob.setVertexIDClass(Text.class);
```

```
    inlinkJob.setVertexValueClass(IntWritable.class);
    inlinkJob.setEdgeValueClass(NullWritable.class);
    inlinkJob.setPartitioner(HashPartitioner.class);


    inlinkJob.waitForCompletion(true)
```

This isn't too far away from a usual BSP job setup, but we are now setting the vertex class instead of a BSP class (the BSP class mentioned earlier to drive the fancy API is set implicitly). At first you see a lot of in- and output methods called, make sure that your graph is consisting of VertexWritable and VertexArrayWritable like told earlier.
The main two things to keep an eye on are to use the appropriate partitioner class (currently only HashPartitioner is available) and to set the classes you have defined in the generics of the vertex. Java sadly can't guess the types at runtime due to type-erasure, so to run your algorithm successfully, you need to provide the classes at compile time.

Additional Functionality

Graph repair

Hama requires a graph to be completed before feeding it to an algorithm. By complete we mean that every vertex that is referenced by an edge must somewhere be a vertex in the graph.

In many cases of leafs this is not always the case, therefore we have added a repair functionality which is traversing the whole graph for leafs and adding them to the vertex structure to prevent algorithms from breaking with *NullPointerExceptions* when it does not find a referenced vertex.

You can turn this feature on by setting it in your configuration like this:
conf.setBoolean(GraphJobRunner.GRAPH_REPAIR, true);

NOTE: This will make your computation slower, so this is turned off by default.

Self referencing vertex

In case of Pagerank every vertex needs an edge from itself to itself. You can either take care that the input is doing that job or let Hama do this for you. Similarly to graph repair you can set this by a configuration value:
conf.set("hama.graph.self.ref", "true");

Pagerank needs that kind of edge, otherwise the algorithm is broken. There are possibly many other algorithms needing it as well, if it is going to be a popular feature we want to add this to the *GraphJob* with a convenience method.

Advanced Features and Future Plans

In future, we intend to make Hama programming and job submission process, more flexible and elegant. Our future work includes:
- Real time Supersteps
- A BSP peer spawning new BSP tasks on the fly.
- Selective syncing where a task can span for more than one global superstep before syncing
- Submitting a graph of BSP jobs
- Providing a superstep and computation model library for users to reuse.
- More machine learning algorithms implemented on Hama
- More graph algorithms implemented on Hama

On the algorithmic part of Hama we want to add a various amount of algorithms ranging from graph theory and maths to machine learning and data mining applications. In the math package we want to include efficient matrix multiplication and transformation algorithms, whereas we want to add constantly new useful graph examples. We also see forward to have graph examples that can not be expressed with the Pregel API forced in frameworks like Giraph or GoldenOrb, rather than needing some kind of advanced supervision of the graph which is only available at the low level BSP primitives. We are looking forward to see parallel SVM (Support vector machine) and ANN (artificial neural network), as well as DBSCAN (**d**ensity **b**ased **s**patial **c**lustering of **a**pplications with **n**oise) implementations with BSP soon.

References

[1] Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990
[2] Message Passing Interface
[3] Large-scale graph computing at Google