PHP Cloud Computing Platform

by

Arvind Kalyan

A PROJECT REPORT

submitted to

Oregon State University

in partial fulfillment of

the requirements for the degree of

Master of Science

Presented December, 2008

Commencement June, 2009

AN ABSTRACT OF THE PROJECT OF

Arvind Kalyan for the degree of Master of Science in Computer Science presented on _____.

Title: PHP Cloud Computing Platform

Abstract Approved: _____
Dr. Bella Bose

Motivation for cloud computing applications are listed. A Cloud Computing framework – MapReduce – is implemented. A document indexing application is built as an example MapReduce application on this framework. Focus is given to ease of job submission and scalability of the underlying network.

TABLE OF CONTENTS

1. Introduction

Most applications that run on massively-parallel hardware are written on frameworks or libraries that abstract the issues associated with communication, data transmission, fault-detection/recovery and related concerns. Fault tolerance for example is only handled partially by the framework and the application code itself has to deal with failure, recovery, load distribution and other non-trivial situations adding to the overall complexity of the software application.

Since most of the applications within the enterprise would be duplicating code needed to handle the platform, it would be beneficial to have a platform that takes care of the complex and repetitive error handling, failure/recovery, and load balancing mechanisms. PHPCloud is an attempt to make such tasks easier to handle. We then consider a programming model that allows the developer to use their code over this infrastructure – the MapReduce model. This project implements a simpler programming model than the original MapReduce model without compromising on the advantages. Finally, we run an example application over this cluster.

1.1. Existing Systems

Parallel processing applications use varied approaches to lessen the communication overhead, which also comprises of network (transport layer and below) level error detection and correction. This overhead is apart from the application level synchronization and error handling. Open Source implementations such as OpenMPI implement the Message Passing Interface [4] (MPI) specification and aim to provide ways to efficiently communicate between processes; both on same machine and across the network. OpenMPI has become the standard among distributed memory applications. Though MPI specification promises to handle communication failures, there is the strong requirement that the user's program should co-operatively communicate amongst each

other – this requirement effectively makes the application code handle the logistics by itself and makes it very difficult to scale the infrastructure later on.

1.2. Motivation

Considering the size of the Internet as of 2007, there are approximately 20 billion web-pages. Assuming an average of 20 KB of data per page, the total size of the Internet is approximated to 400 TB. This is a rough approximation, without considering the overhead of required metadata. To merely read this size of data, with a single computer reading at 14MB/second, would take around 330 days.

By distributing the processing over a 1000 node network, and by eliminating or reducing amount of code that can not be run in parallel, we can process the same data in 0.3 days – about 7 to 8 hours.

But each and every project will not have resources to maintain their own grid or cluster of computers to run this processing. More over, the resources will be idle when this application is not using it. So there is a need to separate the processing logic from the platform needed to run the grid or cluster. Once separated, the processing can then be changed or added vey easily on the platform. This implies that development costs are drastically reduced to create a new application or modify an existing one.

1.3. Cloud Computing

Cloud computing is an emerging *style* of computing. Cloud here merely refers to the fact that Internet is being used to connect the various nodes in the computing network. By basing the architecture on Internet, the solutions become very scalable so as to allow addition and removal of machines very simple and straightforward. Cloud computing itself does not have any architecture. The underlying architecture can be made up of a grid or cluster of computers. The management of these computers is done through a framework that would facilitate such elasticity. One such framework – mapreduce – is discussed in a later section.

Cloud computing itself can mean different things based on the following dimensions:

1. Layers of abstraction
    a. Infrastructure as a service
    b. Platform as a Service
    c. Software as a Service
2. Types of deployment
    a. Public cloud
    b. Private cloud
    c. Hybrid cloud – i.e., partially private
3. Domain of application
    a. High performance computing
    b. Increase web throughput

The technologies under the cloud are the key factors that enable such scalability offered by cloud computing:

1. virtualization technology – VMWare, Sun xVM, etc.,
2. distributed filesystems
3. architecture patterns to provide solutions to common problems
4. new techniques to handle structured/unstructured data, such as MapReduce

1.4. MapReduce

Massively parallel applications that deal with petabytes of data usually distribute the processing load over thousands of nodes. MapReduce is a framework introduced by Google which tries to define what such an application should look like. Programmers then implement appropriate interfaces and plug-in their code onto the mapReduce framework. This greatly simplifies the task of scaling later on since the framework is already known to work with well over 20,000 nodes in the network. New well-defined techniques like these designed to handle large amounts of data are making it easier for organizations to clearly know what is the required structure of data that can be processed on a cloud.

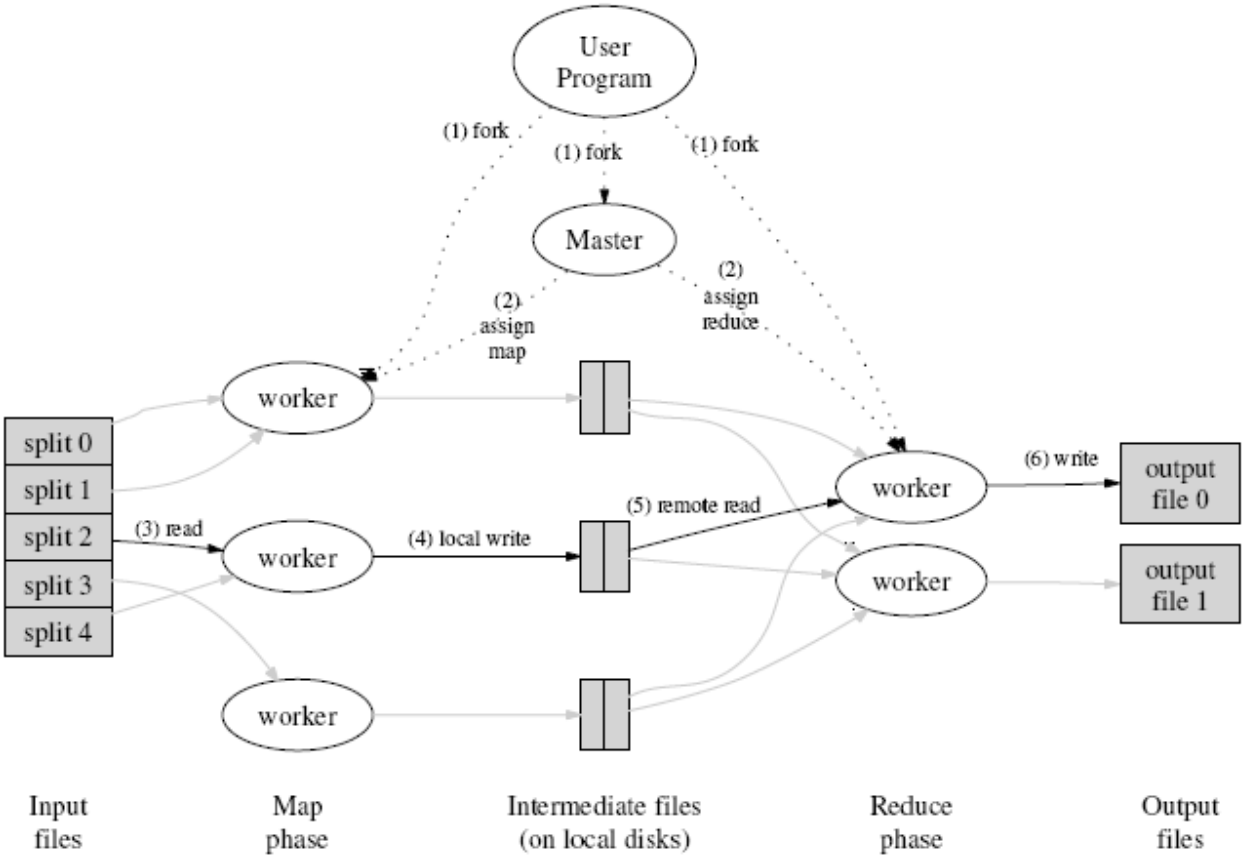

Fig 1. overview of the mapreduce execution (taken from original paper)

Advantages of the MapReduce model:

Library handles all parts of cloud computing infrastructure:

1. Main idea borrowed from Functional Programming
2. The Map components are automatically parallelized
3. Dynamic load balancing – machines can go on and off the network
4. Handles all network failures, recovery, etc

Disadvantages:

5. Data split is done at arbitrary boundaries

- This makes some records useless (first and last ones)
- There is no logical grouping of such splits and ‘related’ data can be present on multiple machines

6. User’s application needs to handle any sub-tasks by itself and this introduces a strong coupling between each ‘map’ step
7. Users application has to be specifically written with complete prior knowledge of all the components involved, this is because they need to implement the necessary interfaces for the mapReduce framework to understand their relationships
8. Data is from a single reader.

1.5. PHPCloud

PHPCloud is an implementation of MapReduce approach. All parts of the following ‘original goals’ have been implemented as part of PHPCloud and details are described in the later sections of this document. Each sub-section under Functional Components corresponds to each of these goals and describes my implementation in greater detail.

In summary, PHPCloud tries to address the following issues.

1. High availability large-file storage
    1. Should be able to handle data unavailability
    2. Should allow data level parallelism
2. User configurable file-splitting algorithm.
    1. Since meaning of data might be lost by random splits, user should be able to configure on what basis records are grouped and split into multiple buckets
    2. If logic is too complicated to be a simple expression, user should be able to write a separate program to decide the split logic
3. Easy job submission to process those files. Job consists of task(s) with custom readers/writers.
    1. User should not have to write code to glue components together

    2. Reading and writing should be handled by user application so that format of input/output can be flexible
    3. Framework only responsible for ensuring stdin and stdout are properly interconnected between modules
4. Automatically run each task on ‘appropriate’ node.
    1. Determine the list of machines where data is available
    2. Launch task on machine with lowest load
5. Consolidate results from node(s), run a filter. User configurable ‘consolidate’, and ‘filter’ rules.
    1. Each node stores the partial results. Reduce node obtains partial results from these nodes and reduces all the partial results into one consolidated result.
    2. Multiple operations can be chained at this level too to filter and or translate the results into a suitable format
6. Implement such that programmer has flexibility in linking with other languages
    1. All the computation that needs to be performed can be programmed in any language
    2. Framework should rely on operating system level primitives to communicate with process
    3. No new constraints or API should be used so that developer is not forced to change their programs
7. Ability to monitor what is going on in the cloud
    1. Simple tools that can visually show the cloud activity that can be used for monitoring

Fig. 2 shows an overview of the PHPCloud.

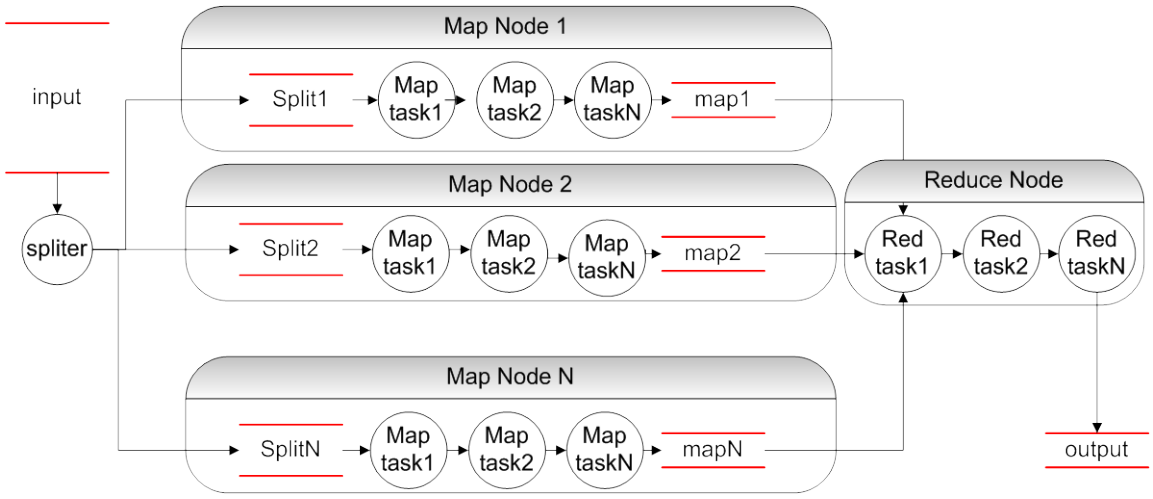

Fig 2. Execution overview for PHPCloud

2. Functional Components

1. Data Storage

1.1. Background

A distributed file system (DFS) is basically a setup with multiple machines that store data, and such data is accessed over the network using a standard protocol (such as nfs) making the file location irrelevant for the purpose of the user or program. A distributed data store (DDS), is a collection of machines that replicate data amongst themselves to achieve some goal (examples are p2p networks, high availability storage networks, etc).

Cloud applications use some form of DFS – example Google File System is used by Hadoop – to store their input and output data. Alternatives to this are approaches by Amazon's S3 service (Simple Storage Service - http://aws.amazon.com/s3/) and Microsoft's SQL Server Data Service [http://msdn.microsoft.com/en-us/sqlserver/dataservices/default.aspx] which are essentially highly scalable data-storage applications that serve and store data on-demand over the Internet.

1.2. PHPCloud implementation

Given the size of these datasets, one of the biggest drawbacks of these approaches is the network latency. Even with current network speeds of Gigabit Ethernets, the time taken to transfer a few GB of data over Internet is a big bottleneck when performance is of importance.

PHPCloud addresses this situation by making use of the available *compute* nodes themselves for data-storage. This has a very big advantage since we completely avoid network latency if all tasks were reading and writing local data.

To achieve this benefit, following is being done in PHPCloud

1. Use a slightly modified/restricted distributed data store (DDS) approach.
2. Maintain a master node which has a map to know what data is present where
3. Replicate the data on multiple machines to increase its availability

The underlying file-system used in PHPCloud is the linux based file-system as maintained by the host operating system on each of the nodes in the cloud. To achieve the high-availability aspect of the requirement, each of the split data is duplicated among *d* machines; where *d* is typically between 2 and 4. When data is to be processed later on, the framework identifies which machine has the data using its internal index.

The main idea behind this is to avoid any single point of failure. Figure 2-1 shows how PHPCloud's splits might be duplicated across multiple compute nodes. In the example shown, there are d = 2 duplicates for each split file. Since the failure probabilities of each of these *d* nodes are statistically independent, this setup is comparable to RAID level 1 setup which is the best RAID setup for reliability among RAID levels 1 through 5.

Split	Node
Green	Node 2

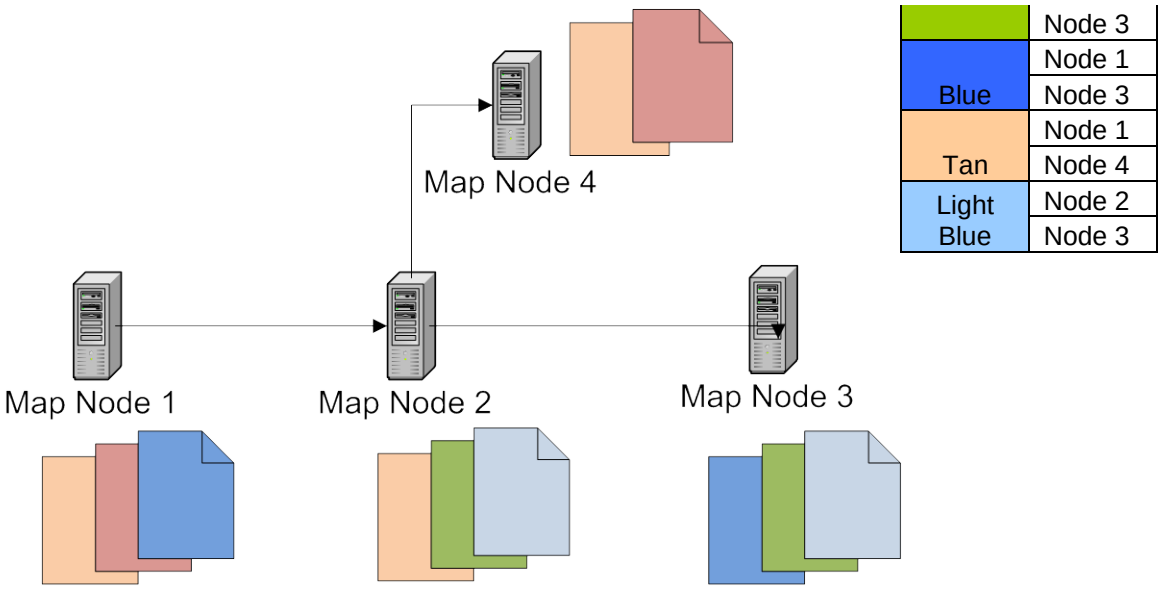

Figure 2-1 Each file split is available in d ≥ 2 compute nodes.

2. Parallelize data

For the nodes to be able to process the data simultaneously without duplicate reads, the input needs to be clearly separated into chunks. Additionally, it would be preferable to reduce any inter process communication over the network. Thus by splitting the input data, the applications do not have to negotiate who needs to process what data.

The number of splits is determined based on 2 conditions, by default:

1. Number of machines available on the network, *n*
2. Total number of records in the input stream, *s*

PHPCloud's default split algorithm takes both the above conditions and determines how many *buckets* to spray the data into; which is some number $m \leq n$. The number of records *s* in the input stream is taken into consideration so that an attempt can be made to equally distribute the size on each machine. It then breaks up the original file into *m* such splits. Each of these *m* splits are then sent to the *m* available machines. A more specialized data splitting algorithm will be discussed in the example section.

Ideally the number of partitions is same as the number of compute nodes available. This helps in optimizing resource utilization since each node can be assigned one partition to process. But in a huge network, we can not expect all the nodes to be up all the time, and depending on when there are more nodes (and less nodes) the assignment of partitions is not always best utilizing the whole network. As a consequence even if more nodes are available later (after the split), they don't get to process any data partitions.

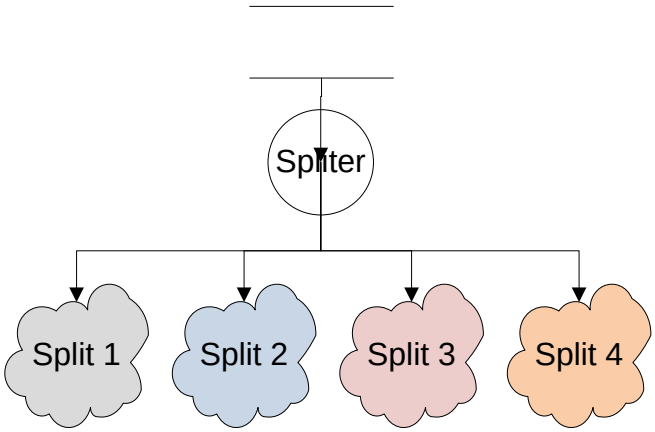

Figure 2-2 Mutually exclusive file splits generated by Spliter

Fig. Spliter used to segregate data that are independent of each other. Each split individually processable

3. Task Submission

As shown in the following example file, it is very straightforward for the user to list out the various stages.

Example task definition file:

[task]
name = arv1001
infile = /data/data/filteredspid.5.25k
spliter = /data/bin/spliter.php -t <TASKFILE>

```
machines        = /data/conf/machines.conf
split_condition = "if ($prev_user != $user) return 1; else return 0;"
stages          = compute,filter,sort,merge
map_timeout = 60
reduce_timeout = 60

; following attribs are shared among all stages.
 [compute]
command         = /data/taskbin/compute.php
type            = map

[filter]
command         = cat
type            = map

[sort]
command         = sort -n
type            = map

[merge]
command         = wc -l
infile          = /data/bin/bucketls.php -t <TASKFILE> -s <PREVSTAGENAME>
type            = reduce
```

I have listed out common unix utilities for “command” options above, to emphasize that the interface between the framework and the application is very minimal and easily pluggable.

4. Distributing load

Compared to the size of the data, the application code is very small. So to avoid transferring all data over to a machine in the network and process, map/reduce

encourages processing locally on where ever the data is available. So PHPCloud does the same – the application is transferred over to the machine where the data is available. The application then processes only the local data and stores results locally.

To identify what machine has what data, PHPCloud uses the index maintained by the split module to know the location of the various splits. An example is shown in Table 1.

Taskname	Split	Machine
task1	1	m1
		m2
	2	m2
		m3
	3	m3
		m4
	4	m4
		m1

Table 1 Data Index generated by Spliter

The load distributor checks the machine availability first by connecting to it. If machine is on the network, it then checks the load on the machine for a certain level of threshold.

After checking if machine is alive and if it can handle more processing load, the masterTaskLauncher invokes localTaskLauncher on that machine to handle the commands in sequence on that machine. But for this to happen, the necessary executable files and environment need to be setup correctly at some point of time before this. Figure 3 shows the control flow in distributing the split files over the cloud.

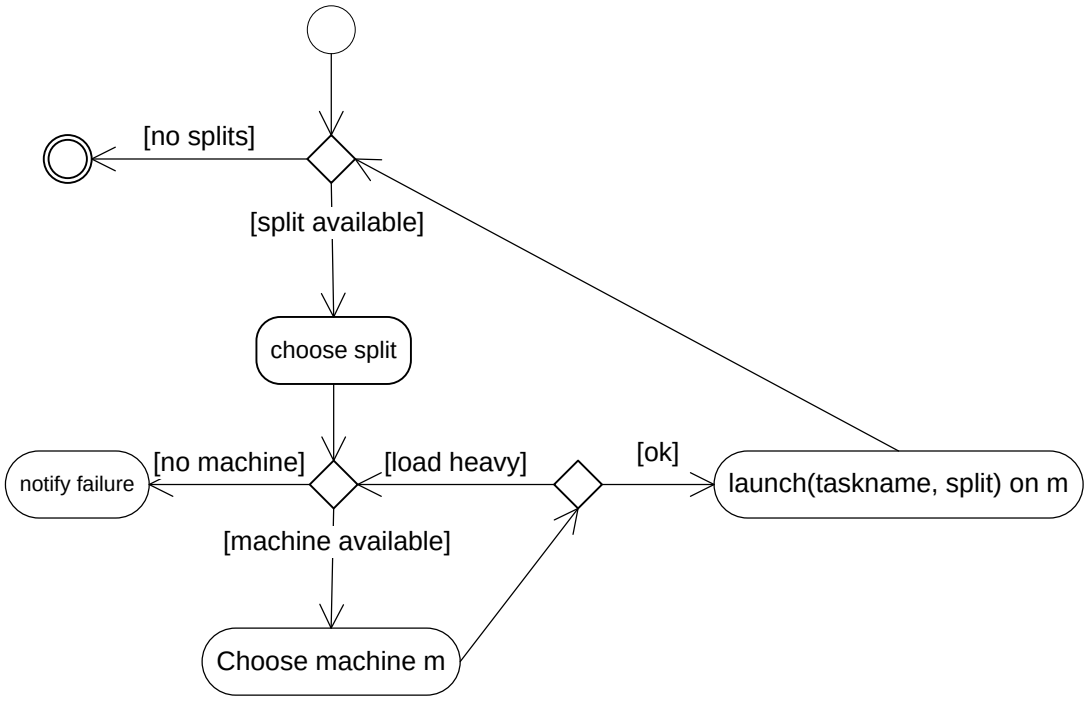

Figure 3 Load distribution activity by master node

Each of the machines then execute a sequence of commands as specified by the user. The localTaskLauncher takes care of interconnecting these tasks together such that the input of each stages the output from previous stage. The typical flow of control is shown in the activity diagram in Figure 4.

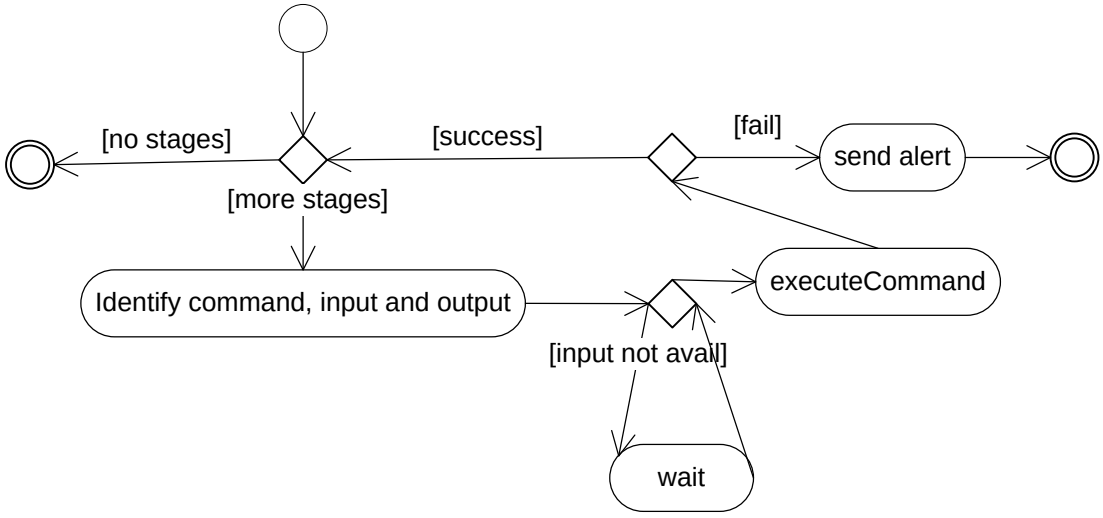

Figure 4 Activity of local master, iterating through list of map/reduce tasks

5. Consolidate and Filter Results

After the distributed tasks have completed, each node would have stored their individual results locally. The reduce component allows the user to run their own algorithm on all of those partial data before merging them.

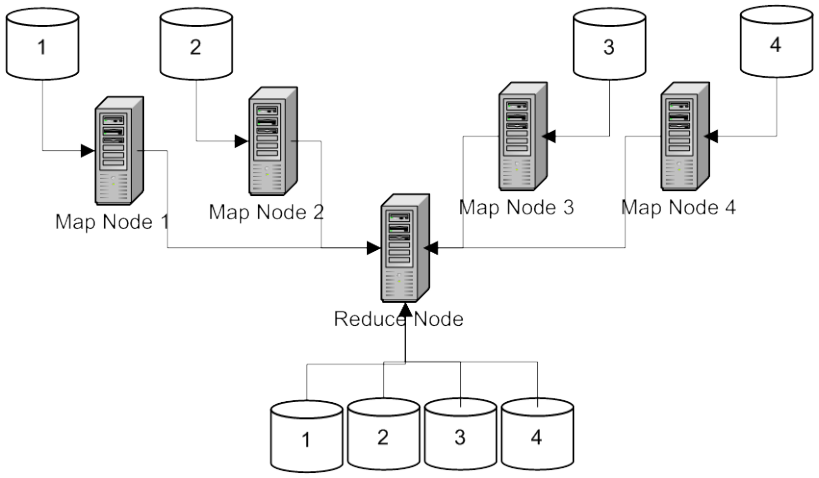

Figure 5 Consolidate all partial results from map node to reduce node

As shown in the Figure 5 the reduce node gets all the computed data once they are available on map nodes. After this it runs a chain of tasks as configured by the user's task definition file. This is depicted in Figure 6.

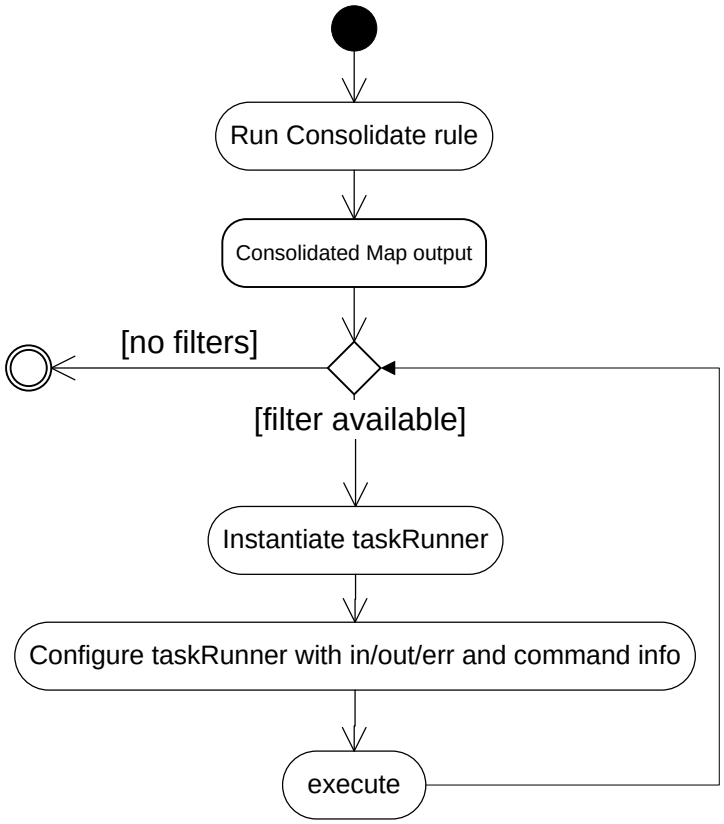

Figure 6 Reduce node iterating through filters

Once the above chain is over, the complete data is present on the reduce node. On most cases such data will be used for further processing by other applications.

6. Programming interfaces

PHPCloud has no programming language constraints. Almost all the other API's have restrictions at application level when it is time to parallelize. The absence of the language requirement is a direct consequence of the simplicity of the framework in that it relies on operating system primitives to establish the necessary communication. Additionally, any data-structure used by the application is maintained by the application itself and the framework does not maintain/manage – i.e., shared memory access and other issues are not done by the framework. This is just another big advantage of making everything local on the machines – it greatly reduces contention for resources.

Before launching any program, the framework determines what are the best files for input, ouput and error file-descriptors. The method of determining what to assign is as follows. First it checks if user has specified any specific file name for this stage for any of the 3 file-descriptors. If yes, the framework uses the specification as-is. At this point, user is also allowed to specify *templates*. The template basically has placeholders for common tokens like taskName, bucketNumber, etc. These tokens get replaced by the framework before being used so it becomes seamless to specify patterns for filenames to be used.

The determined file-descriptors are assigned to the program using standard operating system level redirectors. Since there is no other way the framework interferes with the actual user-program, the user has maximum flexibility for their system design and does not add any level of complexity to their code to achieve this parallelism.

7. Monitoring

PHPCloud lends itself to very easy process monitoring since it relies on basic system tools to accomplish most of its tasks. All parts of the system are completely transparent to anyone who has access to the master node. Starting from system load, we can easily see what machine is doing what activity in the whole cloud. Simple wrapper scripts are part of the framework which lists out the cloud status.

3. Examples

To see the whole framework in action, we will run a data-processing application on the framework.

1. Problem definition

Given a very large input containing data with attributes, split the data into clusters and extract a pre-defined set of metrics out of that data

2. Implementation

This is a typical data mining problem, where we first use some logic to group the data points into various clusters, and then extract information out of it. For the purpose of this example, we will use a k-means algorithm to group the data points into *k* clusters.

2.1 Spliting by k-means

PHPCloud by default allows the user to specify a very simple split expression. But, the user can choose to implement their own data splitting mechanism. As mentioned, we will go with the k-means algorithm for data splitting:

The most frequently used function for partitional clustering methods is the squared error function:

$$V = \sum_{i=1}^{k} \sum_{x_j \in S_i} (x_j - \mu_i)^2$$

For k clusters C_i, $1 <= i <= k$. and μ_i is the centroid for each of those clusters.

The k means algorithm uses this as its objective. For our purposes, we will use data attributes as co-ordinates and use the Euclidian distance for distance calculation and fit them in the above equation.

Following is overview of the k-means algorithm:

While (stopping criteria not met):

Assign next data point to one of the clusters

Re-calculate centroids by taking Euclidian distance between points for distance measure

For our purposes, the stopping criteria is basically when the centroid converges to a constant and does not change, and there are no more data points.

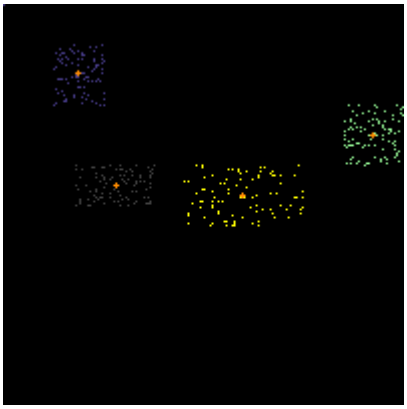

Fig. 4-cluster generated by this spliter

Above figure shows k-means algorithm generating a k=4 clusters. The data points associated with these data points are then fed into the PHPCloud for map/reduce operations.

2.2 Map operations

Each of the clusters identified in the previous step is basically a split of data that can be processed in parallel. The master task launcher then sends the appropriate binaries over to the machine that has each of these clusters and launches local task launcher. Then the local task launcher runs the sequence of map tasks. The output of this extraction tasks is stored locally on all 4 machines in this case.

For now, the example map tasks that I have written are to identify what kind of words and their count are present in this cluster. Listing 1 Example map operation shown below shows the simple code that the end-user has to write. There is no library dependency or restrictions placed on what the user program can do.

```
#!/usr/bin/php
<?php


$in = fopen("/dev/stdin", "r");

$path = array();
$user = '';
```

```
$prev_user = '';

$written = FALSE;
while ($in != FALSE && !feof($in)) {
    $buff = trim (fgets($in));

    if ($buff == '') continue;

    $items = explode(chr(1), $buff);
    $user = $items[0];
    if ($prev_user != '' && $prev_user != $user) {
        showUserPath($prev_user, $path);
        $path           = array();
        $path[]         = $items[1];
        $prev_user      = $user;
        $written        = TRUE;
        continue;
    }
    $path[]         = $items[1];
    $written        = FALSE;
    $prev_user      = $user;
}

if (!$written) showUserPath($prev_user, $path);


function showUserPath($prev_user, $path)
{
    $path = removeDups($path);
    $cnt = count ($path);
    if ($cnt > 0) {
        $path_str = implode(',', $path);
        print "$cnt:$path_str:1\n";
    }
}

function removeDups($path)
{
    $pp = '';
    $p  = '';
```

```
        $ret = array();
        for($i = 0; $i < count($path); $i++) {
            $p = $path[$i];
            if ($pp != '' && $pp == $p) continue;
            $pp = $p;
            $ret[] = $p;
        }
        return $ret;
}

?>
```

Listing 1 Example map operation

Another interesting example map application is a distributed crawler. The split data identifies the clusters of URLs and each node processes the documents fetched from those set of URLs and indexes those documents.

2.3 Reduce operations

Since the data generated in map tasks needs to be aggregated, the reduce task is basically going to gather all 4 partial results and massage them into a suitable format for downstream processing. To showcase the simplicity of the framework, I am currently utilizing standard nix tools – this emphasizes the fact that existing user code does not have to be re-written to be used on the framework.

4. Code Listings

1. Spliter using k-means algorithm

```
<?php

error_reporting(E_ALL);
ini_set("display_errors", 1);
class Cluster
{
        public $points;
```

```
	public $avgPoint;
	public $nochange;

	function Cluster()
	{
		$this->points = array();
		$this->avgPoint = FALSE;
		$this->nochange = FALSE;
	}
	function calculateAverage($maxX, $maxY)
	{
		if (count($this->points)==0)
		{
			$this->avgPoint->x = 10;
			$this->avgPoint->y = 20;
			return;
		}
		$xsum = 0;
		$ysum = 0;
		foreach($this->points as $p)
		{
			$xsum += $p->x;
			$ysum += $p->y;
		}

		$count = count($this->points);
		$tmp_x = $xsum / $count;
		$tmp_y = $ysum / $count;

		if ($this->avgPoint->x == $tmp_x && $this->avgPoint->y
== $tmp_y) {
			$this->nochange = TRUE;
		}
		$this->avgPoint->x = $tmp_x;
		$this->avgPoint->y = $tmp_y;
	}
}

class Point
{
	public $x;
	public $y;
	public $data;

	function Point($x, $y, $d)
	{
		$this->x = $x;
		$this->y = $y;
		$this->data = $d;
	}

	function getDistance($p)
```

```
	{
		$x1 = $this->x - $p->x;
		$y1 = $this->y - $p->y;
		return sqrt($x1*$x1 + $y1*$y1);
	}
}

function kmeans($k, $arr)
{
	$maxX = 0;
	$maxY = 0;
	foreach($arr as $p)
	{
		if ($p->x > $maxX)
		$maxX = $p->x;
		if ($p->y > $maxY)
		$maxY = $p->y;
	}
	$clusters = array();
	for($i = 0; $i < $k; $i++)
	{
		$clusters[] = new Cluster();
		$tmpP = new Point(rand(0,$maxX),rand(0,$maxY), '');
		$clusters[$i]->avgPoint = $tmpP;
	}
	//1. deploy points to closest center.
	//2. recalculate centers
	$converged = FALSE;
	for ($a = 0; FALSE === $converged; $a++)
	{
		foreach($clusters as $cluster)
		$cluster->points = array(); //reinitialize
		foreach($arr as $pnt)
		{
			$bestcluster=$clusters[0];
			$bestdist = $clusters[0]->avgPoint-
>getDistance($pnt);
			foreach($clusters as $cluster)
			{
				if ($cluster->avgPoint->getDistance($pnt) <
$bestdist)
				{
					$bestcluster = $cluster;
					$bestdist = $cluster->avgPoint-
>getDistance($pnt);
				}
			}
			$bestcluster->points[] = $pnt;//add the point to
the best cluster.
		}
		//recalculate the centers.
		$converged = TRUE;
```

```
            foreach($clusters as $cluster) {
                  $cluster->calculateAverage($maxX, $maxY);
                  $converged = ($converged && $cluster->nochange);
            }
      }
      return $clusters;
}

$datapoints = array();
$l = file_get_contents($_GET['fname']);
foreach (explode ("\n", $l) as $line)
{
      $items = explode (",", $line);
      if (count ($items) < 2) continue;
      $p = new Point($items[0], $items[1], $items[2]);
      $datapoints[] = $p;
}

$p = kmeans(4, $datapoints);
if ($_GET['type'] === "image") {
      $gd = imagecreatetruecolor(200, 200);

      $colors = array(
      imagecolorallocate($gd, 255, 0, 0),
      imagecolorallocate($gd, 69,139,116),
      imagecolorallocate($gd, 0,0,255),
      imagecolorallocate($gd, 97,97,97)
      );
      $white = imagecolorallocate($gd, 255, 255,255);

      $i = 0;
      foreach ($p as $n => $cluster)
      {
            $c = $colors[$i];
            $i++;
            foreach ($cluster->points as $pnt)
            {
                  imagesetpixel($gd, round($pnt->x),round($pnt->y),
$c);
            }
            imagesetpixel($gd, round($cluster->avgPoint-
>x),round($cluster->avgPoint->y), $white);
            imagesetpixel($gd, round($cluster->avgPoint-
>x)+1,round($cluster->avgPoint->y), $white);
            imagesetpixel($gd, round($cluster->avgPoint-
>x),round($cluster->avgPoint->y)+1, $white);
            imagesetpixel($gd, round($cluster->avgPoint-
>x)+1,round($cluster->avgPoint->y)+1, $white);
      }


      header('Content-Type: image/png');
```

```
        imagepng($gd);
} else {
        foreach ($p as $n => $cluster)
        {
                $lines = array();
                foreach ($cluster->points as $pnt) {
                        $line = "$pnt->x,$pnt->y,$pnt->data";
                        $lines[] = $line;
                }
                print_r($lines);
        }
}

?>
```

2. File distribution and indexing

```
#!/usr/bin/php

<?php


$taskname = '';
$taskFile = '';

$options = getopt("t:n:");

if (isset ($options['t'])) {
    $taskFile = $options['t'];
}
if (isset ($options['n'])) {
    $taskname = $options['n'];
}

$ini_array = parse_ini_file($taskFile,true);
$indexfile = $ini_array['task']['index'];
$indexfile = str_replace('<TASKNAME>', $taskname,
$indexfile);

$flist = file ("/data/split-files/" . $taskname);
$machines_f =
parse_ini_file($ini_array['task']['machines'],true)
;
```

```
$machines = array_keys($machines_f['map']);
$nrmachs = count($machines);

$dups = 2;

$dist_data = array();
for ($start = 0; $start < $dups; $start++) {
    for ($i = 0; $i < count($flist); $i++) {
        $f = trim ($flist[ ($i + $start) %
count($flist) ]);
        $m = trim ($machines[$i]);
        $cmd = "scp $f $m:$f";
        $dist_data[$f][$m]['status'] = 1;
        $dist_data[$f][$m]['cmd'] = $cmd;
    }
}

$f_done = array();
$f_failed = array();
foreach ($dist_data as $file => &$tasks) {
    foreach ($tasks as $m => &$task) {
        $bdir = dirname($f);
        system("ssh $m mkdir -p $bdir", $ret);
        system($task['cmd'], $ret);
        $task['status'] = $ret;
        if ($task['status'] == 0) {
            $f_done[$file][] = $m;
        } else {
            $f_failed[$file][] = $m;
        }
    }
    if (isset ($f_done[$file]) &&
count($f_done[$file]) > 0)        $f_done[$file]
= implode(','
    , $f_done[$file]);
    if (isset ($f_failed[$file]) &&
count($f_failed[$file]) > 0)    $f_failed[$file]
= implode(','
    , $f_failed[$file]);
}
```

```
//print_r($dist_data);
print_r($f_done);


save_index($f_done);

$donedir = $ini_array['task']['done'];
$donedir = str_replace('<TASKNAME>',
$ini_array['task']['name'], $donedir);
touch ("$donedir/distIndex");


function save_index($f_done)
{
     global $indexfile;
     $d = dirname ($indexfile);
     if (!file_exists($d))
     mkdir ($d, 0777, true);
     $if = fopen($indexfile, "w");
     fwrite ($if, "[split]\n");
     foreach ($f_done as $f => $m) {
          fwrite ($if, "$f = $m\n");
     }
     fclose($if);
}


?>
```

3. Application transfer

```
#!/usr/bin/php
<?php

$machines = parse_ini_file
("/data/conf/machines.conf");
```

```
$dirs = $argv[1];
if ($dirs == '') {
    $dirs =
'/data/bin,/data/bin/inc,/data/conf,/data/taskbin';
}
$dirs = explode (',', $dirs);

foreach ($dirs as $dir)
{
    $files = trim(`ls -1 $dir/*.php $dir/*.sh
$dir/*.conf $dir/*.ini 2>/dev/null| tr '\n' ','`);
    foreach (explode (',', $files) as $f) {
        $f = trim($f);
        if ($f == '')
        continue;
        foreach (array_keys($machines) as $m)
        {
            $m = trim ($m);
            $thismachine = trim (`hostname`);
            if ($thismachine == $m)
            continue;
            $c = "ssh $m mkdir -p `dirname $f`;
scp $f $m:$f";
            print "$c\n";

            system ($c, $ret);
            if (is_executable($f)) {
                system("ssh $m chmod a+x $f");
            }

        }
    }
}
?>
```

4. Master Task Launcher

```php
<?php

require_once '/data/bin/inc/TaskReader.php';
require_once '/data/bin/inc/ErrorObj.php';
require_once '/data/bin/inc/taskRunner.php';
require_once '/data/bin/inc/Stage.php';

class masterTaskLauncher
{
    private $tf, $tr;

    private $logfile;
    private $e;

    private $allDone;

    function masterTaskLauncher($tf)
    {
        $this->tf = $tf;
        $this->tr = new TaskReader($this->tf);
        $this->e = ErrorObj::getErrorObj($this-
>tf);
        $this->logfile = $this->tr->getLogDir() .
"/" . "master";
    }
    function ping($m)
    {
        foreach ($m as $machine)
        {
            $c = "ssh $machine ls / 2>/dev/null
>/dev/null";
            system($c, $ret);
            if ($ret != 0)
            continue;
            $c = "ssh $machine uptime | tr -s ' '
| cut -d ':' -f 5 | tr -d ' '";
            $o = trim(shell_exec($c));
            $o = explode (',', $o);
            $load = $o[0];
```

```
                print "System $machine load is
$load...";
                if ($load > 20) {
                    print "High\n";
                    continue;
                }
                print "OK\n";
                return $machine;
            }
            return '';
        }
        function waitForFile($f, $timeout, $interval)
        {
            $start = date("U");
            while (!file_exists($f) && ($end =
date("U")) - $start < $timeout) {
                sleep($interval);
            }
            if (file_exists($f)) {
                return TRUE;
            }
            return FALSE;
        }
        function waitForFiles($files, $timeout,
$interval, $hostname = FALSE)
        {
            $start = date("U");
            $notfound = TRUE;
            while ($notfound && ($end = date("U")) -
$start < $timeout) {
                $notfound = FALSE;
                foreach ($files as $f) {
                    if ($hostname == FALSE) {
                        if  (!file_exists($f)) {
                            $notfound = TRUE;
                        }
                    } else {
                        $c = "ssh $hostname 'ls $f'
>/dev/null 2>&1";
                        system($c, $ret);
```

```
                    if ($ret != 0) {
                        $notfound = TRUE;
                    }
                }
            }
            if ($notfound) {
                sleep($interval);
            }
        }

        return ! $notfound;
    }
    function launchReduce()
    {
        $this->logfile = $this->tr->getLogDir() .
"/reduce";
        $c = "date >> $this->logfile";
        system($c, $ret);
        $f = $this->findMapDoneFiles();

        $ret = $this->waitForMapTasks($f);
        if ($ret == FALSE) {
            return FALSE;
        }
        $ret = $this->transferMapoutFiles();
        if ($ret === FALSE)
        {
            $msg['function'] = __FUNCTION__;
            $msg['logfile'] = $this->logfile;
            $msg['msg'] = "Failed copying map out
files to reduce machine";
            $this->e->addError($msg);
            return FALSE;
        }
        $r = $this->tr->getReduceMachine();
        $c = "(ssh $r php
/data/bin/localTaskLauncher.php -r reduce -t $this-
>tf) >/dev/null &";
        print "Launching $c\n";
        system("$c", $ret);
```

```
            if ($ret != 0) {
                $msg['function'] = __FUNCTION__;
                $msg['failedcmd'] = $c;
                $msg['retVal'] = $ret;
                $this->e->addError($msg);
                continue;
            }
            return TRUE;
        }
        function createReduceInput()
        {

        }
        function transferMapoutFiles()
        {
            $l = $this->logfile;
            $ret = $this->findMapOutFiles();
            // print_r($ret);
            // print ($this->tr->getLastMapStage());
            $r = $this->tr->getReduceMachine();

            $name = $this->tr->getTaskName();
            $targetd = "/data/$name/map/";
            $waitfiles = array();
            foreach ($ret as $mapoutfiles) {
                $i = explode (':', $mapoutfiles);
                $m = $i[0];
                $f = $i[1];
                $fname = basename ($f);
                $waitfiles[] = "$targetd/$fname.map";
                $c = "(ssh $r mkdir -p $targetd; ssh
    $m 'scp $f $r:$targetd/$fname.map') >>$l 2>&1 &"
                ;
                print "$c\n";
                system($c, $ret);
                if ($ret === FALSE)
                {
                    $msg['function'] = __FUNCTION__;
                    $msg['cmd'] = $c;
```

```
                    $msg['msg'] = "Failed copying map
out file to reduce machine";
                    $msg['retVal'] = $ret;
                    $this->e->addError($msg);
                    return FALSE;
                }
            }
            $rtimeout =  $this->tr-
>getParam('reduce_timeout', 'task');
            $ret = $this->waitForFiles($waitfiles,
$rtimeout, 1, $r);
            if ($ret === FALSE) {
                $msg['function'] = __FUNCTION__;
                $msg['cmd'] = $c;
                $msg['msg'] = "Timed out waiting for
map files to be available. Too huge? try
 increasing timeout.";
                $msg['retVal'] = $ret;
                $this->e->addError($msg);
                return FALSE;
            }
            return TRUE;
        }
        function findMapOutFiles()
        {
            $dirs = $this->findMapDoneFiles();
            $files = array();
            $lastmap = $this->tr->getLastMapStage();
            $name = $this->tr->getTaskName();
            $dir = "/data/$name/$lastmap/";
            foreach ($dirs as $d) {
                $bucket = basename($d);
                $f = trim(shell_exec("ls $d
2>/dev/null"));
                if ($f != '') {
                    $files[] = "$f:$dir/$bucket";
                }
            }
            return $files;
        }
```

```
    function waitForMapTasks($doneFiles)
    {
        $timeout =  $this->tr-
>getParam('map_timeout', 'task');

        $ret = $this->waitForFiles($doneFiles,
$timeout, 1);
        if ($ret === FALSE)
        {
            $msg['function'] = __FUNCTION__;
            $msg['msg'] = "Timedout ($timeout
seconds) waiting for files ". print_r($doneFiles, t
            rue);
            $msg['retVal'] = $ret;
            $this->e->addError($msg);
        }
        return $ret;
    }
    function findMapDoneFiles()
    {
        $ret = array();
        $files = $this->tr->getSplitFiles();
        foreach ($files as $n=>$file)
        {
            $bucket = basename($file);
            $taskName = $this->tr->getTaskName();
            $ret[$bucket] =
"/data/$taskName/map/$bucket";
        }
        return $ret;
    }
    function launchMaps()
    {
        $this->logfile = $this->tr->getLogDir() .
"/map";
        $c = "date >> $this->logfile";
        system($c, $ret);
        $files = $this->tr->getSplitFiles();
        foreach ($files as $n=>$file)
        {
```

```
                $bucket = basename($file);
                $m = $this->tr->getMachines('split',
$bucket);
                $m = $this->ping($m);
                if ($m != '')
                {
                    $c = "(ssh $m php
/data/bin/localTaskLauncher.php -r map -t $this->tf
-s $fil
                    e) >/dev/null &";
                    print "Launching $c\n";
                    system("$c", $ret);
                    if ($ret != 0) {
                        $msg['function'] =
__FUNCTION__;
                        $msg['failedcmd'] = $c;
                        $msg['retVal'] = $ret;
                        $this->e->addError($msg);
                        continue;
                    }
                }
            }
        }
}

$options = getopt("t:r:");
if (!isset($options['t']) || !isset($options['r']))
{
    print "Usage: -t taskFile -r <map|reduce>\n";
    exit;
}
$taskFile       = $options['t'];
$run            = $options['r'];

$m = new masterTaskLauncher($taskFile);

switch ($run)
{
    case 'map':
        $m->launchMaps();
```

```
            break;
        case 'reduce':
            $m->launchReduce();
            break;
}
?>
```

5. Local Task Launcher

```
<?php

require_once '/data/bin/inc/TaskReader.php';
require_once '/data/bin/inc/ErrorObj.php';
require_once '/data/bin/inc/taskRunner.php';
require_once '/data/bin/inc/Stage.php';

class localTaskLauncher
{
    private $tf, $tr;

    private $e;

    private $allDone;

    function localTaskLauncher($tf)
    {
        $this->tf = $tf;
        $this->tr = new TaskReader($this->tf);
        $this->e = ErrorObj::getErrorObj($this-
>tf);
    }
    function launchMaps($inputbucket)
    {
        $bucketNumber = basename($inputbucket);
        $stages = $this->tr->getStages();
        $lastDone = '/dev/null';
        $taskRunners = array();
        foreach ($stages as $s) {
```

```
            $stage = new Stage($s, $this->tf,
$inputbucket);
            if ($stage->getType() == 'reduce') {
                break;
            }

            $i = $stage->getInputFile();
            $o = $stage->getOutputFile();
            $c = $stage->getCommand();
            $d = $stage->getDoneFile();
            $l = $stage->getLogFile();

            $trunner = new TaskRunner($this->tf,
$i, $o, $l);
            $trunner->setDone($lastDone, $d);
            $trunner->setCommand($c);
            $taskRunners[] = $trunner;
            $lastDone = $d;
            //print "$i\t-> $c\t->$o\n";
        }
        $thishost = trim (`hostname`);
        $taskname = $this->tr->getTaskName();

        $alldone =
"/data/$taskname/map/$bucketNumber/$thishost";
        $this->allDone = array(
                                'machine'       =>
'slack1',
                                'file'          =>
$alldone);

        //print count ($taskRunners) . " items in
taskRunners\n";
        $this->clearAllDoneonMaster();
        foreach ($taskRunners as $t) {
            $t->run();
        }
        $this->touchAllDoneonMaster();
    }
    function sendCompleteMail()
```

```
    {
        mail($this->tr->getOwnerEmail(),
"$taskName completed: ". date("D M j G:i:s T Y"),
"Process C
ompleted.");
    }
    function launchReduce()
    {
        $name = $this->tr->getTaskName();
        $targetd = "/data/$name/map/";
        $c = "cat $targetd/*.map > $targetd/all";
        print "$c\n";
        system($c, $ret);
        if ($ret != 0) {
            $msg['function'] = __FUNCTION__;
            $msg['failedcmd'] = $c;
            $msg['retVal'] = $ret;
            $this->e->addError($msg);
            return;
        }
        $i = "$targetd/all";
        $o = "/data/$name/reduce/all";

        $stages = $this->tr->getStages();
        $lastDone = '/dev/null';
        $taskRunners = array();
        foreach ($stages as $s) {
            $stage = new Stage($s, $this->tf,
'all');
            if ($stage->getType() != 'reduce') {
                continue;
            }
            $c = $stage->getCommand();
            $d = $stage->getDoneFile();
            $l = $stage->getLogFile();

            $trunner = new TaskRunner($this->tf,
$i, $o, $l);
            $trunner->setDone('/dev/null', $d);
            $trunner->setCommand($c);
```

```
        }

        $this->allDone = array(
                                    'machine'       =>
'slack1',
                                    'file'          =>
$d);

        $this->clearAllDoneonMaster();
        $trunner->run();
        $this->touchAllDoneonMaster();
        $this->sendCompleteMail();
    }
    function clearAllDoneonMaster()
    {
        $dfile  = $this->allDone['file'];
        $mach   = $this->allDone['machine'];

        $cmd = "ssh $mach rm -rf $dfile";
        system($cmd, $ret);
        if ($ret != 0) {
            $msg['function'] = __FUNCTION__;
            $msg['failedcmd'] = $cmd;
            $msg['retVal'] = $ret;
            $this->e->addError($msg);
            return;
        }
        //print "Removed $dfile on $mach\n";
    }
    function touchAllDoneonMaster()
    {
        $dfile  = $this->allDone['file'];
        $ddir   = dirname($dfile);
        $mach   = $this->allDone['machine'];

        $cmd = "ssh $mach mkdir -p $ddir";
        system($cmd, $ret);
        if ($ret != 0) {
            $msg['failedcmd'] = $cmd;
            $msg['retVal'] = $ret;
```

```
                $this->e->addError($msg);
            }
            $cmd = "ssh $mach touch $dfile";
            system($cmd, $ret);
            if ($ret != 0) {
                $msg['failedcmd'] = $cmd;
                $msg['retVal'] = $ret;
                $this->e->addError($msg);
            }
            print "Touched $dfile on $mach\n";
        }
        function launchStage($stage, $machine)
        {

        }
        function getStageMachines($stage)
        {
            $c = "/data/bucketls.php -t $this->tf -s
$stage";
            $if = trim(shell_exec($c));
            $if_rows = explode("\n", $if);
            $m = array();
            foreach ($if_rows as $row) {
                $r = explode (':', $row);
                $m[] = $r[0];
            }
            return $m;
        }
}

$options = getopt("t:s:r:");
if (!isset($options['r']) || !isset($options['t']))
{
    print "Usage: -r <map|reduce> -t task file -s
startFile\n";
    exit;
}
$taskFile       = $options['t'];
$run            = $options['r'];
$m = new localTaskLauncher($taskFile);
```

```
switch ($run)
{
    case 'map':
        if (!isset ($options['s'])) {
            print "Usage: -r <map|reduce> -t task
file -s startFile\n";
            exit;
        }
        $startFile      = $options['s'];
        $m->launchMaps($startFile);
        break;
    case 'reduce':
        $m->launchReduce();
        break;
}
?>
```

6. Task Runner

```
<?php

require_once 'TaskReader.php';
require_once 'ErrorObj.php';


class TaskRunner {
    var $fin, $fout, $ferr;
    var $done_in, $done_out;
    var $cmd, $args;
    var $e;
    function TaskRunner($tf, $i = FALSE, $o =
FALSE, $e = FALSE)
    {
        $this->setInfd($i);
        $this->setOutfd($o);
        $this->setErrfd($e);

        $this->e = ErrorObj::getErrorObj($tf);
    }
```

```
    function setInfd($i) {
        $this->fin      = $i;
    }
    function setOutfd($o) {
        $this->fout     = $o;
    }
    function setErrfd($e) {
        $this->ferr     = $e;
    }
    function setCommand($c, $args = FALSE)
    {
        $this->cmd = $c;
        $this->args = $args;
    }
    function run()
    {
        //print "Executing $this\n";
        return $this->ExecuteCommand($this->cmd,
$this->args);
    }
    function setDone($i,$o) {
        $this->done_in = $i;
        $this->done_out = $o;
    }
    function ExecuteCommand($c, $args = FALSE) {
        $wait_time = 10;
        $upstream_available = $this-
>waitForFile($this->done_in, $wait_time, 1);
        if (FALSE == $upstream_available) {
            $m['msg']               = "Timed out
waiting for done file";
            $m['cmd']               = $c;
            $m['donefile']  = $this->done_in;
            $m['waitPeriod']= $wait_time;
            $this->e->addError($m);
            return;
        }
        $ret = $this->ExecuteCommandVanilla($c,
$args);
        if ($ret == 0) {
```

```
            $this->touchDoneFile($this->done_out);
        } else {
            $msg['msg']     = "Couldn't execute
command [$c]";
            $msg['stderr']  = $this->ferr;
            $this->e->addError($msg);
        }
    }
    /**
     * proc_open way to invoke child
     *
     * pblm is it doesnt return proper exit code
     *
     * @param string $c command to exec
     * @param array $args array of key-val pairs of
arguments
     */
    function ExecuteCommandPHP($c, $args = FALSE) {
        $descriptors = array (
        0       =>      array("file", $this->fin,
"r"),
        1       =>      array("file", $this->fout,
"w"),
        2       =>      array("file", $this->ferr,
"w")
        );
        $cwd = "/data/";
        $env = array();

        $child_process = proc_open($c,
$descriptors, $pipes, $cwd, $env);
        if (is_resource($child_process)) {
            $ret = proc_close($child_process);
            //print "returned $ret\n";
            //print_r($pipes);
        }

    }
    function touchDoneFile($f)
    {
```

```
        $cmd = "mkdir -p `dirname $f`";
        system($cmd, $ret);
        if (0 != $ret) {
            $this->e->addError(array("couldn't
create done directory for [$f]"));
        }
        $ret = touch ($f);
        if (FALSE == $ret) {
            $this->e->addError(array("couldnt
touch file [$f]"));
        }
    }
    function waitForFile($f, $timeout, $interval)
    {
        $start = date("U");
        while (!file_exists($f) && ($end =
date("U")) - $start < $timeout) {
            sleep($interval);
        }
        if (file_exists($f)) {
            return TRUE;
        }
        return FALSE;
    }
    function ExecuteCommandVanilla($c, $args =
FALSE) {
        $in = $this->fin;
        $out = $this->fout;
        $err = $this->ferr;
        /*
         $outdir = dirname ($out);
         $errdir = dirname ($err);

         $cmd = "rm -rf $outdir $errdir";
         system ($cmd, $ret);
         if ($ret != 0) {
         $this->e->addError(array("couldn't remove
[$outdir] and [$errdir] with cmd [$cmd]. re
         turn val is $ret"));
         }
```

```
            if (FALSE == mkdir( $outdir , 0777, true
)) {
            $this->e->addError(array("couldn't create
out directory for [$outdir]"));
            }
            if (FALSE == mkdir( dirname ($errdir) ,
0777, true )) {
            $this->e->addError(array("couldn't create
error/log directory for [$errdir]"));
            }*/

            $cmd = "mkdir -p `dirname $out` `dirname
$err`; $c <$in >$out 2>$err";
            //print "\n\t\t$cmd\n";
            system($cmd, $ret);
            return $ret;
        }
        function __toString()
        {
            //return print_r($this, true);
            $ret = array(
                                'command'       => $this-
>cmd,
                                'args'          => $this-
>args,
                                'infile'        => $this-
>fin,
                                'outfile'       => $this-
>fout,
                                'logfile'       => $this-
>ferr,
                                'done_in'       => $this-
>done_in,
                                'done_out'      => $this-
>done_out
            );
            return print_r($ret, true);
        }
}
```

```
//$tr = new TaskRunner('/data/task.ini',
"/data/head", "/tmp/testout", "/tmp/testerr");
//$tr->setDone('//dev/null', '/tmp/do');
//$tr->ExecuteCommand("cat");

?>
```

5. References

[1] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Operating System Design and Implementation (OSDI), Usenix Assoc.*, 2004

[4] *MPI: A Message-Passing Interface Standard*, http://www.mpi-forum.org/docs/mpi21-report.pdf.