# Abstract

Timing and cost-effective processing of large datasets has become a critical ingredient for the success of many academic, government and industrial organizations. In recent years, there has been a large increase in the size of computational data distributed across different locations. MapReduce is a programming model and an associated implementation for processing and generating large data sets. It is a software framework for data intensive applications, which provides concurrent and distributed platform for handling computation and storage. The existing MapReduce frameworks are, however, designed with a specific infrastructure; such dependence and tight coupling to specific resource types and technologies is, in a heterogeneous distributed environment is not a optimal solutions. There are numerous scientific applications that either currently utilize, or need to utilize data and resources distributed over vast heterogeneous infrastructures and networks with varying speeds and characteristics; SAGA (Simple API for Grid Applications) is a high level programming interface which provides the ability to develop distributed applications in an infrastructure independent way.

SAGA C++ MapReduce framework was already developed to support heterogeneous distributed environments. However, extracting performance is difficult using the SAGA MapReduce because of the overhead in interoperability provided by SAGA & the network dependency (SSHFS across machines). SAGA C++ MapReduce uses two ways of communications between master and workers; SAGA-Streams and SAGA-Advert server. Often we find machines firewalled for the ports running SAGA Streams. This limits the implementation of SAGA C++ MR to single cluster or domain. SAGA C++ MapReduce in C++ is heavily dependent on boost c++ libraries and it looks very heavy to the users to code it or use it and it is up to the user's to deploy the code and compile the SAGA C++ MapReduce code on different machines which user wants to use.

The limitations of SAGA C++ MapReduce & to incorporate SAGA BigJob & Pilot store (Python Frameworks), a need for python MapReduce aroused. With the combination of these python frameworks better results can be produced. SAGA C++ MapReduce uses SAGA-Streams for communication between worker and master. In SAGA BigJob based Python MapReduce the entire communication between master/worker is based on SAGA-Advert services. But, ultimately SAGA Python MapReduce compromises with the performance because SAGA-Streams are very fast compared to SAGA-Advert.

The objective of this project is to implement wordcount application using SAGA BigJob based Python MapReduce framework. The pilot-store is not incorporated yet. The project involves enhancing the framework suitable to implement wordcount effectively. Further, the results have been shown to implement the framework on clusters and across different machines.

Contents

Abstract 1

Background 3

Previous Implementations 9

Motivation 10

Execution Overview of SAGA BigJob Based Python MapReduce ( SPBMR ) 10

# Background

**MapReduce**

MapReduce is a programming framework that supports applications, which involve parallel and distributed computations on very large data sets. It is extensively used for distributed applications such as genome searching, sorting, page ranking etc.

MapReduce computation is based on key/value pairs. From a set of input key/value pairs a set of output key/value pairs are produced. The programs in the MapReduce are written in functional style. These functions are automatically parallelized and executed on large clusters [3]. This framework typically consists of three functions: Mapper, Reducer, and Partitioner.

The Mapper takes an input pair and produces a set of intermediate key/value pairs. The framework collects all the intermediate values for a key and hands them to the Reducer. The Mapper performs most of the filtering operations like discarding the data, which not required and just picking the data, which is required in the next stage. For example, in a searching operation it just searches for the string in the input line and just emits out the result.

The Reducer gets an intermediate key and intermediate values list, and merges these to form a usually smaller set of values. Typically only a zero or one value is produced for each intermediate key, as the Reducer usually performs an aggregation of data.

The Partitioner is used by the Mapper to determine the partition number for the extracted key value pair. These partitions are assigned to the Reducer to aggregate.

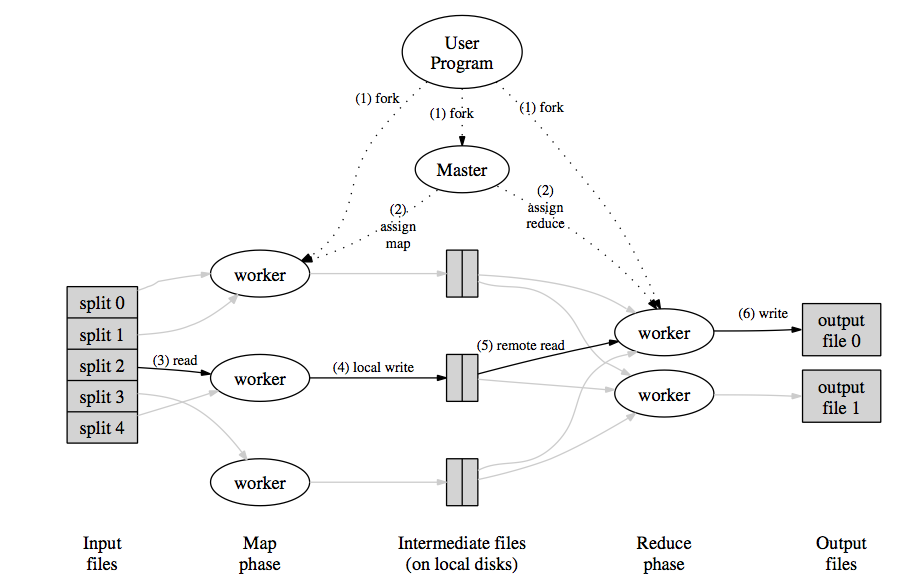
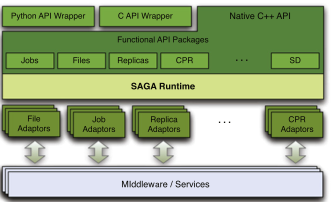


Figure 1: This figure shows the typical MapReduce process. [3]

**SAGA**

SAGA stands for Simple API (Application Program Interface) for Grid Applications. Running distributed applications involving large volumes of datasets and distributed worker machines is a difficult task for computational scientists. SAGA provides a simple Linux based interface for the users with basic programming background. SAGA has many advantages with different packages like a file package, job package, stream package and some other important packages required for grid applications. The SAGA File package handles the file transfers and browsing remote directories. The SAGA Job package handles the submitting, running jobs on remote machine and supercomputers. The SAGA Stream package takes care of the authentications required for performing any applications [2]. SAGA helps the computational scientists to perform large calculations, run distributed applications hiding the inner complexities involved.



**SAGA C++ MapReduce**

SAGA-MapReduce interleaves the core MapReduce logic with explicit instructions on where processes are to be scheduled. The advantage of this approach is that SAGA C++ implementation is no longer bound to run on a system providing the appropriate semantics originally required by MapReduce, and is portable to a broader range of generic systems as well. The drawback is that it is more complicated to extract performance, as some system-level semantics have to be recreated in the application space level. SAGA-MapReduce exposes a simple interface which provides the complete functionality needed by any MapReduce algorithm, while hiding the more complex functionality, such as chunking of the input, sorting the intermediate results, launching and coordinating the workers, etc. — these are generically implemented by the framework. The application consists of two independent process types, a master and worker processes. The master process is responsible for:

• launching all workers for the map and reduce steps, as described in a configuration file provided by the user;

• coordinating the workers;

• chunking of the data;

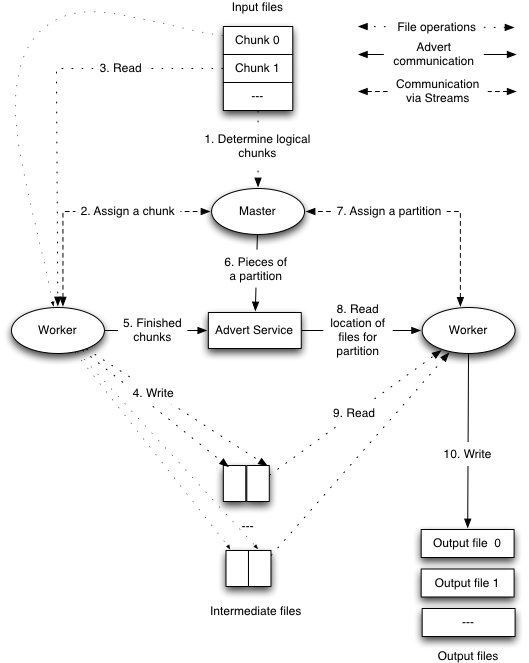
• assigning the input data to the workers of the map step;

• handling the intermediate data files produced by the map step; and

• passing the location of the sorted output files to the workers of

the reduce step.

Any specific MapReduce instance is specified by a MR- JobDescription object in which the user specifies Mapper and Reducer classes, input and output paths and data formats. The used InputFormat determines the logical partitions of the input data for the master — that information is then sent to idle workers. A RawRecordReader implementation interprets an InputChunk and provides a record iterator for the Mapper. It is possible to support any kind of data source for which a record oriented view exists, by writing a custom RawRecordReader. The output from the Mapper is further processed by the Partitioner which assigns emitted key/value pairs from the Mapper to Reducers. Finally, a RawRecordWriter writes output data to files. The master process is readily available to the user and needs no modification to execute different map and reduce functions in the worker processes. The functionality for the different steps has to be provided by the user, which means the user has to write the C++ functions implementing the respective MapReduce kernels. Both the master and the worker processes use the SAGA–API as an abstract interface to the infrastructure, making the application portable between different architectures and systems. The worker processes are launched using the SAGA job package, allowing the jobs to launch on any back-end supported by SAGA (such as locally, globus/GRAM, EC2, SSH, Condor etc.). The communication between the master and workers is ensured by using the SAGA advert package, abstracting an information database in a platform independent way, and the SAGA stream package, abstracting streaming data access between network endpoints. The master creates logical partitions of the data (referred to as chunking, analogous to Google’s MapReduce), so the dataset does not have to be split and distributed manually. The input data can be located on any file system supported by SAGA, such as the local file system, or a distributed file system like HDFS or KFS [14]. As with any application which concurrently spans multiple diverse resources or infrastructures, the coordination between the different application components becomes challenging. The SAGA-MapReduce implementation uses the SAGA advert API for that task, and can thus limit the a priori information needed for bootstrapping the application: the compute clients (workers) require (i) the contact URL of the used advert service instance, and (ii) a unique worker ID to register with in that advert service, so that the master can start to assign work items. Both pieces of information are provided by the master via command line parameters to the worker, at start-up time. The master application requires the following additional information: (i) a set of resources to execute the workers on, (ii) the location of the input data, (iii) the target location for the output data, and (iv) the contact URL for the advert service for coordination and communication.



**SAGA BigJob**

The Pilot-Job abstraction has been shown to be an effective abstraction to address many requirements of scientific applications. Specifically, Pilot-Jobs support the decoupling of workload submission from resource assignment; this results in a flexible execution model, which in turn enables the distributed scale-out of applications on multiple and possibly heterogeneous resources. Most Pilot-Job implementations however, are tied to a specific infrastructure. SAGA-based Pilot-Job, which supports a wide range of application types, and is usable over a broad range of infrastructures, i.e., it is general-purpose and extensible, and as we will argue is also interoperable with Clouds.

The Pilot-Job itself is a regular Grid job, which is started through a Grid resource manager, such as the Globus GRAM. Once the batch queue assigned the requested resources to the Pilot-Job, the Pilot-Job circumvents the necessity to queue each individual sub-job and is responsible for managing these and assigning them to so-called sub-jobs. That way queuing times for sub- jobs can be reduced and the predictability for application scenarios can be increased.

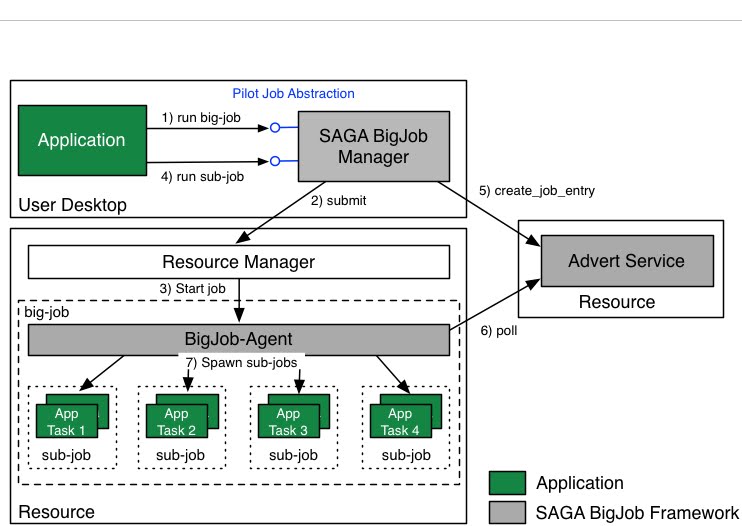
Pilot-Jobs decouple resource allocation from resource binding and allow the efficient utilization of resources. Pilot-Jobs have been used to (i) improve the utilization of resources, (ii) to reduce the net wait time of a collection of tasks, (iii) facilitate bulk or high-throughput simulations where multiple jobs need to be submitted which would otherwise saturate the queuing system, and (iv) as a basis to implement application specific scheduling decisions and policy decisions.

BigJob is a SAGA-based Pilot-Job implementation. BigJob natively supports parallel applications (e.g. based on MPI) and works independent of the underlying Grid infrastructure across different heterogeneous backend, e.g. Grids and Cloud, reflecting the advantage of using a SAGA-based approach.

SAGA BigJob comprises of three components:

1. The BigJob Manager provides the pilot job abstraction and manages the orchestration and scheduling of BigJobs (which in turn allows the management of both BigJob objects and subjobs)
2. The BigJob-Agent that represents the pilot job and thus, the application-level resource manager on the respective resource, and
3. The advert service that is used for communication between the BigJob Manager and Agent.

Before running regular jobs, an application must initialize a BigJob object. The BigJob Manager then queues a pilot job, which actually runs a BigJob Agent on the respective resource. For this agent a specified number of resources is requested. Subsequently, sub-jobs can be submitted through the BigJob Manager using the jobID of the BigJob as reference. The BigJob Manager ensures that the subjobs are launched onto the correct resource based upon the specified jobID using the right number of processes. Communication between the BigJob Agent and BigJob Manager is carried out using the SAGA advert service, a central key/value store. For each new job, the BigJob creates an advert entry. The agent periodically polls for new jobs. If a new job is found and resources are available, the job is dispatched, otherwise it is queued.

****

**Pilot store ( Need to complete it …… )**

**Application ( Currently wordcount / Need to test with Bfast application)**

Wordcount has a well understood run-time and scaling behavior, and thus serves us well for focusing the tests on the used frameworks and middlewares. For example, a file containing the words (‘bread’ ‘bee’ ‘bee’ ‘honey’) would produce the result (‘bread 1’, ‘bee 2’, ‘honey 1’).

# Previous Implementations

MapReduce framework was implemented in several platforms in recent years. For example, Google has developed its own MapReduce framework in Google file system [3] while Yahoo has implemented MapReduce sorting in Hadoop [4]. However, each of them was tied their respective file systems and the implementations were limited to single cluster which could be as large as possible within the file system. [1]

The SAGA MapReduce framework concentrates on processing large-scale dataset tasks implementable in heterogeneous environments supported by SAGA, so the application is not tied to any infrastructure [1]. Therefore, SAGA-MapReduce can work with different job scheduling systems like Globus, Condor and distributed file systems like HDFS, KFS and can be utilized in combination for performing MapReduce tasks [6].

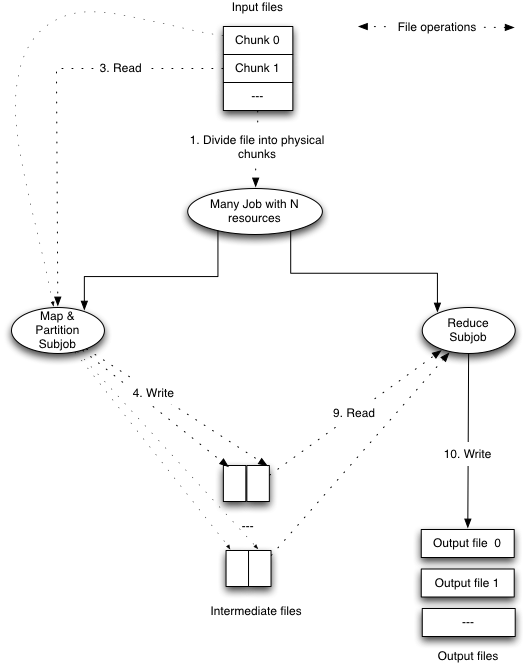
However, extracting performance is difficult using the SAGA MapReduce because of the overhead in interoperability provided by SAGA & the network dependency (SSHFS across machines). SAGA C++ MapReduce uses two ways of communications between master and workers; SAGA-Streams and SAGA-Advert server. Often we find machines firewalled for the ports running SAGA Streams. This limits the implementation of SAGA C++ MR to single cluster or domain. SAGA C++ MapReduce in C++ is heavily dependent on boost c++ libraries and it looks very heavy to the users to code it or use it and it is up to the user's to deploy the code and compile the SAGA C++ MapReduce code on different machines which user wants to use.

# Motivation

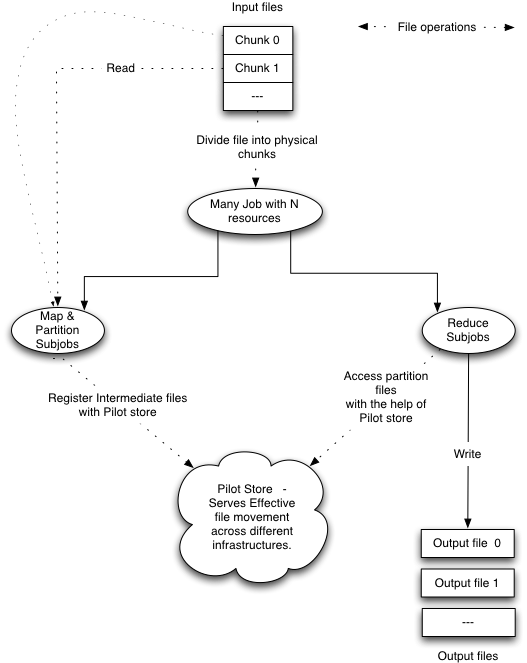
The limitations of SAGA C++ MapReduce & to incorporate SAGA BigJob & Pilot store (Python Frameworks), a need for python MapReduce aroused. With the combination of these python frameworks better results can be produced. SAGA C++ MapReduce uses SAGA-Streams for communication between worker and master. In SAGA BigJob based Python MapReduce the entire communication between master/worker is based on SAGA-Advert services i.e thru BigJob. But, ultimately SAGA Python MapReduce compromises with the performance because SAGA-Streams are very fast compared to SAGA-Advert.

# Execution Overview of SAGA BigJob Based Python MapReduce ( SPBMR )

1. The MapReduce Framework takes a configuration file as input and validates the input parameters and reports if there is any required parameter missing. The required parameters are advert server location, BigJob location, chunk size, input location, temporary location, map script location, reduce script location and the number of workers, wall time.
2. The Framework calculates the number of chunks for the files in the input directory based on the chunk size. If the size of the file is less than the chunk size it leaves the file as it is. If the number of chunk files created is less than the number of subjobs then the user specified chunk size is overridden with the best value of chunk size such that the each subjob will get equal amount of work load. The best value of chunk size is calculated using the formula (Total file size / number of workers). The divided chunks are written to temporary location using SAGA File write operations and chunk list is prepared.
3. After preparing chunks, the MapReduce Framework initialize manyjob object with the parameters provided in the configuration file. A Subjob for each chunk is submitted using Manyjob object to execute Map & partition script. The ManyJob will take care of executing the map & partition script on number of cores requested by each subjob.
4. After the map subjobs are completed the intermediate files are moved to the output directory. The movement of files is required when the MapReduce is executed in distributed mode. Pilot store is not yet implemented in MapReduce Framework but will be used to distribute the files.
5. The MapReduce Framework submits reduce subjobs for each map\_partition output file. The number of reduces are kept equal to number of subjobs to get the optimum performance because every map & partition output file will get assigned to a subjob which leaves no subjobs idle.
6. After the reduce subjobs completed, the output files are created in the output directory and the Manyjob is terminated.



Python MapReduce without pilot store.



Python MapReduce with Pilot Store.

Comparision between SAGA BigJob based Python MapReduce / SAGA C++ MapReduce.

Excel sheet……

C++ MR execution…

1. Word count is started. The mapreduce framework is initilalized . If it is master it does nothing… if it is worker then worker code is executed. For this the parameters are parsed..The file input path is added. The mapper, reducer, partition are set using set\_methods…DistirbutedJobRunner is called.
2. After the distribtuedjobr object is initialized run function is called ( distributedjobrunner.cpp ). In the run there are several functions. A new session is created . This createNewSession\_ will create necessary directories for workers, binaries, chunks , and server address…
3. Added binaries.. to the advert.. The location of each binary file is attached..
4. HandleMaps.cpp – the master assigns chunks /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
5. \* assignMaps is the only public function, it keeps iter-\*
6. \* ating through chunks and assigning them to running \*
7. \* workers until all chunks have been finished mapping. \*
8. \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* issue\_command\_ finds and idle worker or finished \*

\* worker and tries to assign them a chunk. If thye are \*

\* finished, record their results in the finished\_ vector\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

How to debug worker….