A Hierarchical Framework for Cross-Domain MapReduce Execution

Yuan Luo[1], Zhenhua Guo[1], Yiming Sun[1], Beth Plale[1], Judy Qiu[1], Wilfred W. Li [2]
[1] School of Informatics and Computing, Indiana University, Bloomington, IN, 47405
[2] San Diego Supercomputer Center, University of California, San Diego, La Jolla, CA, 92093
{yuanluo, zhguo, yimsun, plale, xqiu}@indiana.edu, wilfred@sdsc.edu

ABSTRACT
The MapReduce programming model provides an easy way to execute pleasantly parallel applications. Many data-intensive life science applications fit this programming model and benefit from the scalability that can be delivered using this model. One such application is AutoDock, which consists of a suite of automated tools for predicting the bound conformations of flexible ligands to macromolecular targets. However, researchers also need sufficient computation and storage resources to fully enjoy the benefit of MapReduce. For example, a typical AutoDock based virtual screening experiment usually consists of a very large number of docking processes from multiple ligands and is often time consuming to run on a single MapReduce cluster. Although commercial clouds can provide virtually unlimited computation and storage resources on-demand, due to financial, security and possibly other concerns, many researchers still run experiments on a number of small clusters with limited number of nodes that cannot unleash the full power of MapReduce. In this paper, we present a hierarchical MapReduce framework that gathers computation resources from different clusters and run MapReduce jobs across them. The global controller in our framework splits the data set and dispatches them to multiple "local" MapReduce clusters, and balances the workload by assigning tasks in accordance to the capabilities of each cluster and of each node. The local results are then returned back to the global controller for global reduction. Our experimental evaluation using AutoDock over MapReduce shows that our load-balancing algorithm makes promising workload distribution across multiple clusters, and thus minimizes overall execution time span of the entire MapReduce execution.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems – *Distributed applications*.

General Terms: Design, Experimentation, Performance

Keywords: AutoDock, Cloud, FutureGrid, Hierarchical MapReduce, Multi-Cluster

1. INTRODUCTION
Life science applications are often both compute intensive and data intensive. They consume large amount of CPU cycles while processing massive data sets that are either in large group of small files or naturally splittable. These kinds of applications ideally fit in the MapReduce [2] programming model. MapReduce differs from the traditional HPC model in that it does not distinguish computation nodes and storage nodes so each node is responsible for both computation and storage. Obvious advantages include better fault tolerance, scalability and data locality scheduling. The MapReduce model has been applied to life science applications by many researchers. Qiu et al. [15] describe their work to implement various clustering algorithm using MapReduce.

AutoDock [13] is a suite of automated docking tools for predicting the bound conformations of flexible ligands to macromolecular targets. It is designed to predict how small molecules of substrates or drug candidates bind to a receptor of known 3D structure. Running AutoDock requires several pre-docking steps, e.g., ligand and receptor preparation, and grid map calculations, before the actual docking process can take place. There are desktop GUI tools for processing the individual AutoDock steps, such as AutoDockTools (ADT) [13] and BDT [19], but they do not have the capability to efficiently process thousands to millions of docking processes. Ultimately, the goal of a docking experiment is to illustrate the docked result in the context of macromolecule, explaining the docking in terms of the overall energy landscape. Each AutoDock calculation results in a docking log file containing information about the best docked ligand conformation found from each of the docking runs specified in the docking parameter file (dpf). The results can then be summarized interactively using the desktop tools such as AutoDockTools or with a python script. A typical AutoDock based virtual screening consists of a large number of docking processes from multiple targeted ligands and would take a large amount of time to finish. However, the docking processes are data independent, so if several CPU cores are available, these processes can be carried out in parallel to shorten the overall makespan of multiple AutoDock runs.

Workflow based approaches can also be used to run multiple AutoDock instances; however, MapReduce runtime can automate data partitioning for parallel execution. Therefore our paper focuses on extending the MapReduce model for parallel execution of applications across multiple clusters.

Cloud computing can provide scalable computational and storage resources as needed. With the correct application model and implementation, clouds enable applications to scale out with relative ease. Because of the "pleasantly parallel" nature of the MapReduce programming model, it has become a popular model for deploying and executing applications in a cloud, and running multiple AutoDock jobs certainly fits well for MapReduce. However, many researchers still shun away from clouds for different reasons. For example, some researchers may not feel comfortable letting their data sit in shared storage space with users worldwide, while others may have large amounts of data and computation that would be financially too expensive to move

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ECMLS'11, June 8, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0701-4/11/06...$10.00.

RIGHTSLINK

into the cloud. It is more typical for a researcher to have access to several research clusters hosted at his/her lab or institute. These clusters usually consist of only a few nodes, and the nodes in one cluster may be very different from those in another cluster in terms of various specifications including CPU frequency, number of cores, cache size, memory size, and storage capacity. Commonly a MapReduce framework is deployed in a single cluster to run jobs, but any such individual cluster does not provide enough resources to deliver significant performance gain. For example, at Indiana University we have access to IU Quarry, FutureGrid [5], and Teragrid [17] clusters but each cluster imposes limit on the maximum number of nodes a user can uses at any time. If these isolated clusters can work together, they collectively become more powerful.

Unfortunately, users cannot directly deploy a MapReduce framework such as Hadoop on top of these clusters to form a single larger MapReduce cluster. Typically the internal nodes of a cluster are not directly reachable from outside. However, MapReduce requires the master node to directly communicate with any slave node, which is also one of the reasons why MapReduce frameworks are usually deployed within a single cluster. Therefore, one challenge is to make multiple clusters act collaboratively as one so it can more efficiently run MapReduce. There are two possible approaches to address this challenge. One is to unify the underlying physical clusters as a single virtual cluster by adding a special infrastructure layer, and run MapReduce on top of this virtual cluster. The other is to make the MapReduce framework directly working with multiple clusters without needing additional special infrastructure layers.

We propose a hierarchical MapReduce framework which takes the second approach to gather isolated cluster resources into a more capable one for running MapReduce jobs. Kavulya et al. characterize MapReduce jobs into four categories based on their execution patterns: map-only, map-mostly, shuffle-mostly, and reduce-mostly, and also find that 91% of the MapReduce jobs they have surveyed fall into the map-only and map-mostly categories [10]. Our framework partitions and distributes MapReduce jobs from these two categories (map-only and map-mostly) into multiple clusters to perform map-intensive computation, and collects and combines the outputs in the global node. Our framework also achieves load-balancing by assigning different task loads to different clusters based on the cluster size, current load, and specifications of the nodes. We have implemented the prototype framework using Apache Hadoop.

The rest of the paper is organized as follows. Section 2 presents some related works. Section 3 gives an overview of our hierarchical MapReduce framework. Section 4 presents more details on the multiple AutoDock runs using MapReduce. Section 5 gives experiment setup and result analysis. The conclusion and future work are given in Section 6.

2. RELATED WORKS

Researchers have put significant efforts to the easy submission and optimal scheduling of massive parallel jobs in clusters, grids, and clouds. Conventional job schedulers, such as Condor [12], SGE [6], PBS [8], LSF [23], etc., aim to provide highly optimized resource allocation, job scheduling, and load balancing, within a single cluster environment. On the other hand, grid brokers and metaschedulers, e.g., Condor-G [4], CSF [3], Nimrod/G[1], GridWay [9], provide an entry point to multi-cluster grid environments. They enable transparent job submission to various distributed resource management systems, without worrying about the locality of execution and available resources there. With respect to the AutoDock based virtual screening, our earlier efforts presented at National Biomedical Computation Resource (NBCR) [14] Summer Institute 2009, addressed the performance issue of massive docking processes by distributing the jobs to the grid environment. We used the CSF4 [3] meta-scheduler to split docking jobs to heterogeneous clusters where these jobs were handled by local job schedulers including LSF, SGE and PBS.

Clouds give users a notion of virtually unlimited, on-demand resources for computation and storage. Attributed to its ease of executing pleasantly parallel applications, MapReduce has become a dominant programming model for running applications in a cloud. Researchers are discovering new ways to make MapReduce easier to deploy and manage, more efficient and scalable, and also more able to accomplish complex data processing tasks. Hadoop On Demand (HOD) [7] uses the TORQUE resource manager [16] to provision and manage independent MapReduce and HDFS instances on shared physical nodes. The authors of [21] have identified some fundamental performance limitation issues in Hadoop and in the MapReduce model in general which make job response time unacceptably long when multiple jobs are submitted; by substituting their own scheduler implementation, they are able to overcome these limitations and improve the job throughput. CloudBATCH [22] is a prototype job queuing mechanism for managing and dispatching MapReduce jobs and commandline serial jobs in a uniform way. Traditionally a cluster must separate MapReduce-enabled nodes because they are dedicated to MapReduce jobs and cannot run serial jobs. But CloudBATCH uses HBase to keep various metadata on each job and also uses Hadoop to wrap commandline serial jobs as MapReduce jobs, so that both types of jobs can be executed using the same set of cluster nodes. The Map-Reduce-Merge is extended from the conventional MapReduce model to accomplish common relational algebra operations over distributed heterogeneous data sets [20]. In this extension, the Merge phase is a new concept that is more complex than the regular Map and Reduce phases, and requires the learning and understanding of several new components, including partition selector, processors, merger, and configurable iterators. This extension also modifies the standard MapReduce phase to expose data sources to support some relational algebra operations in the Merge phase.

Sky Computing [11] provides end user a virtual cluster interconnected with ViNe [18] across different domains. It aims to bring convenience by hiding the underlying details of the physical clusters. However, this transparency may cause unbalanced workload if a job is dispatched over heterogeneous compute nodes among different physical domains.

Our hierarchical MapReduce framework, aims to enable map-only and map-most jobs to be run across a number of isolated clusters (even virtual clusters), so these isolated resources can collectively provide a more powerful resource for the computation. It can easily achieve load-balance because the different clusters are visible to the scheduler in our framework.

3. HIERARCHICAL MAPREDUCE

The hierarchical MapReduce framework we present in this paper consists of two layers. The top layer has a global controller that accepts user submitted MapReduce jobs and distributes them across different local cluster domains. Upon receiving a user job, the global controller divides the job into sub-jobs according to the capability of each local cluster. If the input data has not been deployed onto the cluster already, the global controller also

RIGHTSLINK

partitions input data proportionally to the sub-jobs, and sends them to these clusters. After the jobs are all finished on all clusters, the global controller collects the outputs to perform a final reduction using the global reducer which is also supplied by the user. The bottom layer consists of multiple local clusters that each receives sub-jobs and input data partitions from the global controller, performs local MapReduce computation and sends results back to the global controller.

Although on the surface our framework may appear structurally similar to the Map-Reduce-Merge model presented in [20], our framework is very different in nature. As discussed in the related work section, the Merge phase introduced in the Map-Reduce-Merge model is a new concept which is different and more complex than the conventional Map and Reduce, and programmers implementing jobs under this model must not only learn this new concept along with the components required by it, but also need to modify the Mappers and Reducers to expose data source. Our framework, on the other hand, strictly uses the conventional Map and Reduce, and a programmer just needs to supply two Reducers – one local Reducer, and one global Reducer – instead of just one for the regular MapReduce. The only requirement is that the programmer must make sure that the formats of the local Reducer output keys/value pairs match those of the global Reducer input key/value pairs. However, if the job is map-only, the programmer does not need to supply any reducers, and the global controller simply collects the map results from all clusters and places them under a common directory.

3.1 Architecture

Figure 1 is a high-level architecture diagram of our hierarchical MapReduce framework. The top layer in our framework is the global controller, which consists of a job scheduler, a data transferer, a workload collector, and a use-supplied global reducer. The bottom layer consists of multiple clusters for running the distributed local MapReduce jobs, where each cluster has a MapReduce master node with a workload reporter and a job manager. The compute nodes inside each of the cluster are not accessible from the outside.

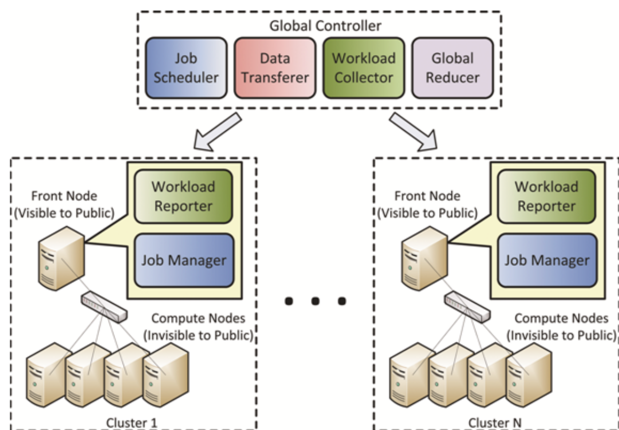

Figure 1. Hierarchical MapReduce Architecture

When a user submits a MapReduce job to the global controller, the job scheduler splits the job into a number of sub-jobs and assigns them to each local cluster based on several factors, including the current workload reported by the workload reporter from each local cluster, as well as the capability of individual nodes making up each cluster. This is done to achieve load-balance by ensuring that all clusters will finish their portion of the job in approximately the same time. The global controller also partitions the input data in proportion to the sub-job sizes if the input data have not been deployed before-hand. The data transferer would transfer the user supplied MapReduce jar and job configuration files with the input data partitions to the clusters. As soon as the data transfer finishes for a particular cluster, the job scheduler at the global controller notifies the job manager of that cluster to start the local MapReduce job. Since data transfer is very expensive, we recommend that users only use the global controller to transfer data when the size of input data is small and the time spent for transferring the data is insignificant compared to the computation time. For large data sets, it would be more efficient and effective to deploy them before-hand, so that the jobs get the full benefit of parallelization and the overall time does not get dominated by data transfer. After the local sub-jobs are finished on a local cluster, if the application requires, the clusters will transfer the output back to the global controller. Upon receiving all the output data from all local clusters, the global reducer will be invoked to perform the final reduction task, unless the original job is map-only.

3.2 Programming Model

The programming model of our hierarchical MapReduce framework is the "Map-Reduce-Global Reduce" model where computations are expressed as three functions: Map, Reduce, and Global Reduce. We use the term "Global Reduce" to distinguish it from the "local" Reducer, but conceptually as well as syntactically, a Global Reducer is just another conventional Reducer. The Mapper, just as a conventional Mapper does, takes an input pair and produces an intermediate key/value pair; likewise, the Reducer, just as a conventional Reducer does, takes an intermediate input key and a set of corresponding values produced by the Map task, and outputs a different set of key/value pairs. Both the Mapper and the Reducer are executed on local clusters. The Global Reducer is executed on the global controller using the output from the local clusters. Table 1 lists these 3 functions and also the input and output data types. The formats of the local Reducers output keys/value pairs must match those of the Global Reducer input key/value pairs.

Table 1. Input and output types of Map, Reduce, and Global Reduce functions

Function Name	Input	Output
Map	(k^i, v^i)	(k^m, v^m)
Reduce	$(k^m, [v_1^m, \dots, v_n^m])$	(k^r, v^r)
Global Reduce	$(k^r, [v_1^r, \dots, v_n^r])$	(k^o, v^o)

Figure 2 uses a tree-like structure to show the data flow sequence among the Map, Reduce, and Global Reduce function. In this diagram, the root node is the global controller on which the Global Reduce takes place, and the leaf nodes represent local clusters that perform the Map and Reduce functions. The circled numbers shown in Figure 2 indicate the order in which the steps occur, and the arrows indicate the directions in which the data sets (key/value pairs) flow. A job is submitted into the system in Step 1, and then the input key/value pairs are passed from the root node (global controller) to the child nodes (local clusters) in Step 2, and also Map tasks are launched at the local clusters where each Map consumes an input key/value pair and produces a set of intermediate key/value pairs. In Step 3, the set of intermediate pairs are passed to the Reduce tasks, which are also launched at the local clusters. Each Reduce task consumes an intermediate key with a set of corresponding values, and produces yet another

RIGHTSLINK

set of key/value pairs as output. In Step 4, the local reduce output are send back to the global controller to perform the Global Reduce task. The Global Reduce task takes in a key and a set of corresponding values that were originally produced from the local Reducers, performs the computation, and produces the output in Step 5.

Theoretically, the model we present can be extended to more than just two hierarchical layers, i.e. the tree structure in Figure 2 can have more depth by turning the leaf clusters into intermediate controllers similar to the global controller and each would further divide its assigned jobs and run them on its own set of children clusters. But for all practical purposes, we do not see a need for more than two layers for the foreseeable future, because it could increase the complexity as well as the overhead introduced with each additional layer. If a researcher has a large number of small clusters available, it is most likely more efficient to use them to create a broader bottom layer than to increase the depth.

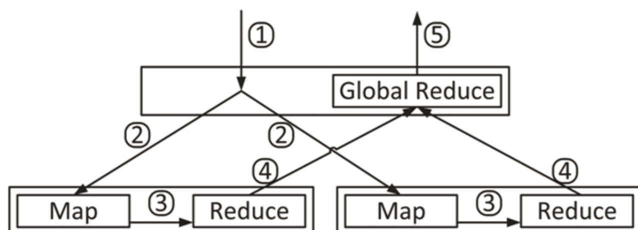

Figure 2. Programming Model

3.3 Job Scheduling and Data Partitioning

The main challenge of our work is how to balance the workloads among each local MapReduce cluster, which is closely tied to how the datasets are partitioned.

The input dataset for a particular MapReduce job may be either submitted by the user to the global controller before execution, or pre-deployed on the local clusters and is exposed via a metadata catalog to the user who runs the MapReduce job. The scheduler on the global controller takes into consideration the data locality when partitioning the datasets and scheduling the job.

In this paper, we focus on the situation where input dataset is submitted by the user. If the user manually split the dataset and run separate sub-jobs on different clusters, it would be time consuming and error-prone. Our global controller is able to automatically count the total number of records in the input dataset using user-implemented InputFormat and RecordReader, and divides the dataset and assigns the correct number of records to each cluster.

We make the assumption that all map tasks of a MapReduce application are computation intensive and take approximately the same amount of time to run – this is a reasonable assumption as we will see in the next section that applying MapReduce to running multiple AutoDock instances displays exactly this kind of behavior. The scheduling algorithm we use for our framework is as follows. Let $MaxMapper_i$ be the maximum number of Mappers that can be run concurrently on $Cluster_i$; $MapperRun_i$ be the number of Mappers currently running on $Cluster_i$; $MapperAvail_i$ be the number of available Mappers that can be added for execution on $Cluster_i$; $NumCore_i$ be the total number of CPU Cores on $Cluster_i$, where i is the cluster number, and i $\in \{1, \ldots, n\}$. We also use ρ_i to define how many map tasks a user assigns to each core, that is,

$$MaxMapper_i = \rho_i \times NumCore_i \quad (1)$$

Normally we set $\rho_i = 1$ in the local MapReduce clusters for computation intensive jobs, so we get

$$MapperAvail_i = MaxMapper_i - MapperRun_i \quad (2)$$

For simplicity, let

$$\gamma_i = MapperAvail_i \quad (3)$$

The weight of each sub-job can be calculated from (4) where the factor θ_i is the computing power of each cluster, e.g., the CPU speed, memory size, storage capacity, etc. The actual θ_i varies depending on the characteristics of the jobs, i.e., whether they are computation intensive or I/O intensive

$$Weight_i = \frac{\gamma_i \times \theta_i}{\sum_{i=1}^{N} \gamma_i \times \theta_i} \quad (4)$$

Let $JobMap_x$ be the total number of Map tasks for a particular job x, which can be calculated from the number of keys in the input to the Map tasks, and $JobMap_{x,i}$ be the number of Map tasks to be scheduled to $Cluster_i$ for job x, so that

$$JobMap_{x,i} = Weight_i \times JobMap_x \quad (5)$$

After partitioning the MapReduce job to Sub-MapReduce jobs using equation (5), we number the data items of the datasets and move the data items accordingly, either from global controller to local clusters, or from local cluster to local cluster.

4. AUTODOCK MAPREDUCE

We apply the MapReduce paradigm to running multiple AutoDock instances using the hierarchical MapReduce framework to prove the feasibility of our approach. We take the outputs of AutoGrid (one tool in the AutoDock suite) as input to the AutoDock. The key/value pairs of the input of the Map tasks are ligand names and the location of ligand files. We designed a simple input file format for AutoDock MapReduce jobs. Each input record, which contains 7 fields shown in Table 2, corresponds to a map task.

Table 2. AutoDock MapReduce input fields and descriptions

Field	Description
ligand_name	Name of the ligand
autodock_exe	Path to AutoDock executable
input_files	Input files of AutoDock
output_dir	Output directory of AutoDock
autodock_parameters	AutoDock parameters
summarize_exe	Path to summarize script
summarize_parameters	Summarize script parameters

For our AutoDock MapReduce, the Map, Reduce, and Global Reduce functions are implemented as follows:

1) *Map*: The Map task takes a ligand to run the AutoDock binary executable against a shared receptor, and then runs a Python script summarize_result4.py to output the lowest energy result using a constant intermediate key.

2) *Reduce*: The Reduce task takes all the values corresponding to the constant intermediate key and sorts the values by the energy

RIGHTSLINK

from low to high, and outputs the sorted results to a file using a local reducer intermediate key.

3) *Global Reduce*: The Global Reduce finally takes all the values of the local reducer intermediate key, sorts and combines them into a single file by the energy from low to high.

5. EVALUATIONS

We evaluate our model by prototyping a Hadoop based hierarchical MapReduce system. The system is written in Java and Shell scripts. We use ssh and scp scripts to finish the data stage-in and stage-out. On the local clusters' side, the workload reporter is a component that exposes Hadoop cluster load information accessed by global scheduler. Our original design was to make it a separate program without touching Hadoop source code. Unfortunately, Hadoop does not expose the load information we need to external applications, and we had to modify Hadoop code to add an additional daemon that collects load data by using Hadoop Java APIs.

In our evaluation, we use several clusters including the IU Quarry cluster and two clusters in FutureGrid. IU Quarry is a classic HPC cluster which has several login nodes that are publicly accessible from outside. After a user logins, he/she can do various job-related tasks, including job submission, job status query and job cancellation. The computation nodes however, cannot be accessed from outside. Several distributed file systems (Lustre, GPFS) are mounted to each computation node for storing input data accessed by the jobs. FutureGrid partitions the physical cluster into several parts, and each of which provides a different testbed such as Eucalyptus, Nimbus, and HPC.

Table 3. Cluster Node Specifications.

Cluster	CPU	Cache size	Memory
Hotel	Intel Xeon 2.93GHz	8192KB	24GB
Alamo	Intel Xeon 2.67GHz	8192KB	12GB
Quarry	Intel Xeon 2.33GHz	6144KB	16GB

To deploy Hadoop to traditional HPC clusters, we first use the built-in job scheduler (PBS) to allocate nodes. To balance maintainability and performance, we install the Hadoop program in shared directory while store data in local directory, because the Hadoop program (Java jar files, etc.) is loaded only once by Hadoop daemons whereas the HDFS data is accessed multiple times.

We use three clusters for evaluations – IU Quarry, FutureGrid Hotel and FutureGrid Alamo. Each cluster has 21 nodes. They all run Linux 2.6.18 SMP. Within each cluster, one node is a dedicated master node (HDFS namenode and MapReduce jobtracker) and other nodes are data nodes and task trackers. Each node in these clusters has an 8-core CPU. The specifications of these cluster nodes are listed in Table 3.

Considering AutoDock being a CPU-intensive application, we set $\rho_i = 1$ per section 3.3 so that the maximum number of map tasks on each node is equal to the number of cores on the node. The version of AutoDock we use is 4.2 which is the latest stable version. The global controller does not care about low-level execution details because our local job managers hide the complexity.

In our experiments, we use 6,000 ligands and 1 receptor. One of the most important configuration parameters is *ga_num_evals* - number of evaluations. The larger its value is, the higher the probability that better results may be obtained. Based on prior experiences, the *ga_num_evals* is typically set from 2,500,000 to 5,000,000. We configure it to 2,500,000 in our experiments.

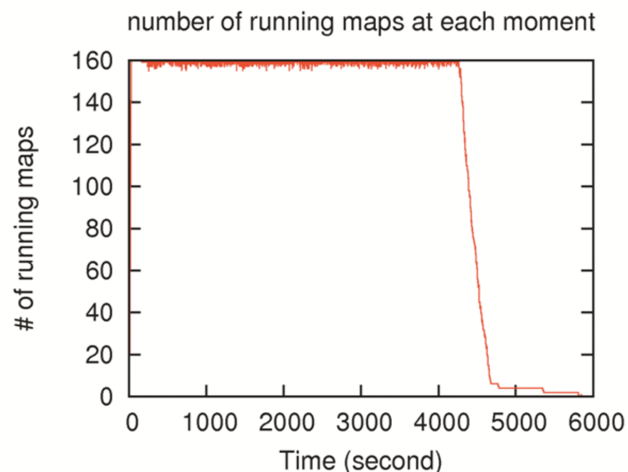

Figure 3: Number of running map tasks for an Autodock MapReduce instance

Figure 3 plots the number of running map tasks within one cluster during the job execution. The cluster has 20 data nodes and task trackers, so the maximum number of running map tasks at any moment is 20 * 8 = 160. From the plot, we can see that the number of running map tasks quickly grows to 160 in the beginning and stays approximately constant for a long time. Towards the end of job execution, it drops to a small value quickly (roughly 0 - 5). Notice there is a tail near the end, indicating that node usage ratio is low. At this moment, if new MapReduce tasks come in, the available mappers will be occupied by those new tasks.

Table 4. MapReduce execution time on different clusters under different number of map tasks.

Number of Map Tasks Per Cluster	Execution Time on Three Clusters		
	Hotel (seconds)	Alamo (seconds)	Quarry (seconds)
100	1004	821	1179
500	1763	1771	2529
1000	2986	2962	4370
1500	4304	4251	6344
2000	5942	5849	8778

Test Case 1:
Our first test case is a base test case without involving the Global Controller to find out how each of our local Hadoop clusters performs under different numbers of map tasks. We ran AutoDock in the Hadoop to process 100, 500, 1000, 1500 and 2000 ligand/receptor pairs in each of the three clusters. See Table 4 for results.

As is reflected in Figure 4, the total execution time vs. the number of map tasks in test case 1 on each cluster is close to linear, regardless of the startup overhead of the MapReduce jobs. The total execution time of the jobs running on the Quarry cluster

RIGHTSLINK

is approximately 50% slower than running on Alamo and Hotel. The main reason is that nodes of the Quarry cluster have slower CPUs compared with that of Alamo and Hotel.

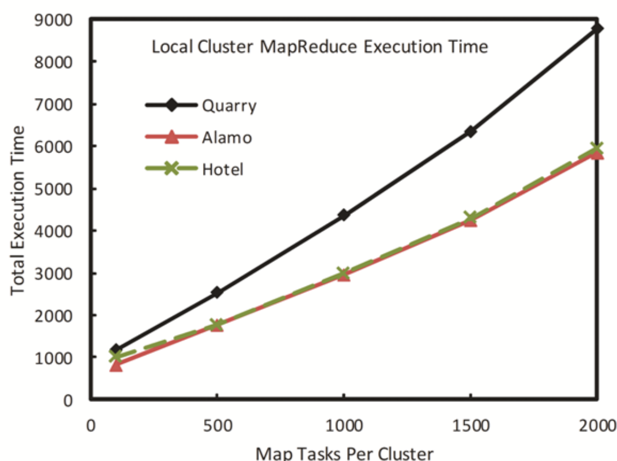

Figure 4. Local cluster MapReduce execution time based on different number of map tasks.

Test Case 2:
Our second test case shows the performance of executing MapReduce jobs with γ-weighted partitioned datasets on different clusters, which is based on the following parameters setup. For equation (4) from section 3.3, we set $\theta_i = C$, where C is a constant, and $i \in \{1,\ 2,\ 3\}$ for our three clusters. Our calculation shows $\gamma_1 = \gamma_2 = \gamma_3 = 160$, given no MapReduce jobs are running beforehand. Therefore, the weight of map tasks distribution on each cluster is $Weight_i = 1/3$. We then equally partition the dataset (apart from shared dataset) into 3 pieces, stage the data together with the jar executable and job configuration file to local clusters for execution in parallel. After the local MapReduce execution, the output files will be staged back to the global controller for the final global reduce. Figure 5 shows the data movement cost in the stage-in and stage-out contexts.

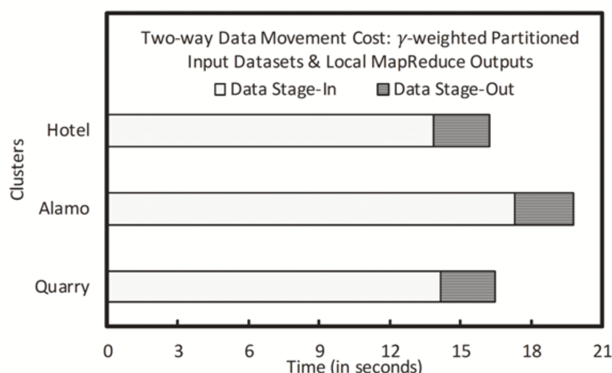

Figure 5. Two-way data movement cost of γ-weighted partitioned datasets: local MapReduce inputs and outputs

The input dataset of AutoDock contains 1 receptor and 6000 ligands. The receptor is described as a set of approximately 20 gridmap files totaling 35MB in size, and the 6000 ligands are stored in 6000 separate directories, each of which is approximately 5-6 KB large. In addition, the executable jar and job configuration file together has a total of 300KB in size. For each cluster, the global controller creates a 14MB tarball containing 1 receptor file set, 2000 ligands directories, the executable jar, and job configuration files, all compressed, and transfers it to the destination cluster, where the tarball is decompressed. We call this global-to-local procedure "data stage-in." Similarly, when the local MapReduce jobs finish, the output files together with control files (typically 300-500KB in size) are compressed into a tarball and transferred back to the global controller. We call this local-to-global procedure "data stage-out." As we can see from Figure 5, the data stage-in procedure takes 13.88 to 17.3 seconds to finish, while the data stage-out procedure takes 2.28 to 2.52 seconds to finish. The Alamo cluster takes a little longer to transfer the data but the difference is insignificant compare to the relatively long duration of local MapReduce executions.

The time it takes to run 2000 map tasks on each of the local MapReduce clusters varies due to the different specification of the clusters. The local MapReduce execution makespan, including data movement costs (both data stage-in and stage-out) is shown in Figure 6. The Hotel and Alamo clusters take similar amount of time to finish their jobs, but the Quarry cluster takes approximately 3,000 more seconds to finish, about 50% more than Hotel and Alamo. The Global Reduce task is only invoked after all the local results are ready in the global controller, and it takes only 16 seconds to finish. Thus, the relatively poor performance on Quarry becomes the bottleneck on the current job distribution.

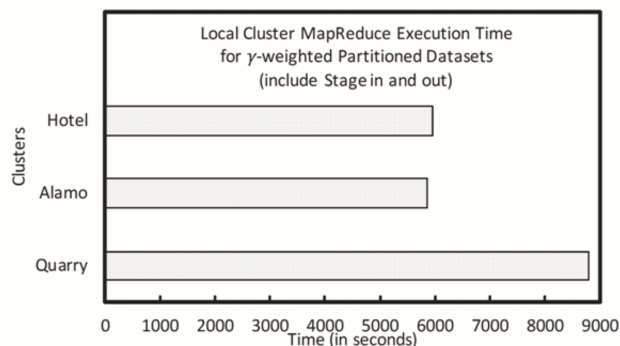

Figure 6. Local MapReduce turnaround time of γ-weighted datasets, including data movement cost

Test Case 3:
In our third test case, we evaluate the performance of executing MapReduce jobs with $\gamma\theta$-weighted partitioned datasets on different clusters, which is based on the following setup. From test cases 1 and 2, we have observed that although all clusters are assigned the same number of compute nodes and cores to process the same amount of data, they take significantly different amount of time to finish. Among the three clusters, Quarry is much slower than Alamo and Hotel. The specifications of the cores on Quarry, Alamo and Hotel are Intel(R) Xeon(R) E5410 2GHz, Intel(R) Xeon(R) X5550 2.67GHz, and Intel(R) Xeon(R) X5570 2.93GHz, respectively. The inverse ratio of CPU frequency and that of processing time match roughly. So we hypothesize that the difference in processing time is mainly due to the different core frequencies, therefore, it is not enough to merely factor in the number of cores for load balancing, and the computation capabilities of each core are also important. We refine our scheduling policy to add CPU frequency as a factor to set θ_i. Here we set $\theta_1 = 2.93$ for Hotel, $\theta_2 = 2.67$ for Alamo, and $\theta_3 = 2$ for Quarry. As is for test case 2, we again have calculated

RIGHTSLINK

$\gamma_1 = \gamma_2 = \gamma_3 = 160$, given no MapReduce jobs are running beforehand. Thus, the weights are $Weight_1 = 0.3860$, $Weight_2 = 0.3505$, and $Weight_3 = 0.2635$ for Hotel, Alamo, and Quarry respectively. The dataset is also partitioned according to the new weight. Table 5 shows how the dataset is partitioned.

Table 5. Number of Map Tasks and MapReduce Execution Time on Each Cluster

Cluster	Number of Map Tasks	Execution Time (Seconds)
Hotel	2316	5915
Alamo	2103	5888
Quarry	1581	6395

Figure 7 shows the data movement cost in the weighted partition scenario. The variations in the size of tarball different number of ligands sets are quite small, which is smaller than 2MB. As we can see from the graph, the data stage-in procedure takes 12.34 to 17.64 seconds to finish, while the data stage-out procedure takes 2.2 to 2.6 seconds to finish. Alamo takes a little bit longer to transfer the data but the difference is also insignificant given the relatively long duration of local MapReduce executions as in the previous test case.

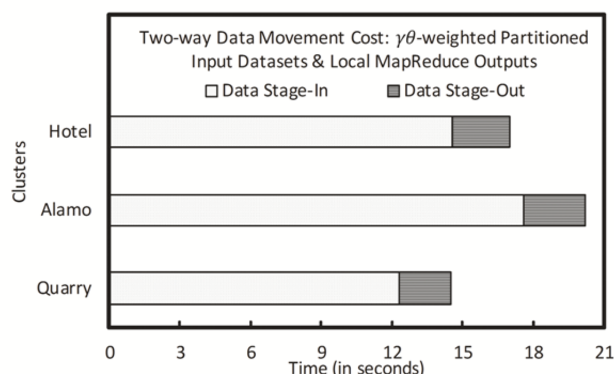

Figure 7. Two-way data movement cost of $\gamma\theta$-weighted partitioned datasets: local MapReduce inputs and outputs

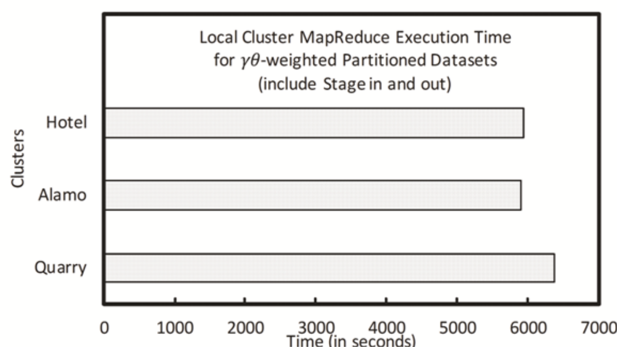

Figure 8. Local MapReduce turnaround time of $\gamma\theta$-weighted datasets, including data movement cost

With weighted partition, the local MapReduce execution makespan, including data movement costs (both data stage-in and stage-out) are shown in Figure 8. All three clusters take similar amount of time to finish the local MapReduce jobs. We can see that our refined scheduler configuration improves performance by balancing workload among clusters. In the final stage, the global reduction combines partial results from lower-level clusters and sorts the results. The average global reduce time taken after processing 6000 map tasks (ligand/receptor docking) is 16 seconds.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented a hierarchical MapReduce framework that can gather computation resources from different clusters and run MapReduce jobs across them. The applications implemented in this framework adopt the "Map-Reduce-Global Reduce" model where computations are expressed as three functions: Map, Reduce, and Global Reduce. The global controller in our framework splits the data set and maps them onto multiple "local" MapReduce clusters to run Map and Reduce functions, and the local results are returned back to the global controller to run the Global Reduce function. We use resource capacity-aware algorithm to balance the workload among clusters. We use multiple AutoDock runs as a test case to evaluate the performance of our framework. The result shows that the workloads are well balanced and the total makespan is kept in minimum.

There are several potential improvements we will address in our future work. Based on the compute-intensive nature of the application, our scheduling algorithm only takes consideration of the CPU specifications. It will not be the case when an application has larger data sets that data movement becomes significant. Other scheduling metrics such as disk I/O and network I/O need to be considered. The remote job submission and data movement in our current prototype are built upon the combination of ssh and scp, which may not work well in a heterogeneous environment. However, they can be replaced by other solutions. One possible solution for remote job submission is to integrate our framework with a meta-scheduler, e.g., CSF and Nimrod/G. Data movement can also be switched to solutions that are more scalable and work well in heterogeneous environments, such as gridftp. As an alternative to transferring data explicitly from site to site, we will also explore the feasibility of using a shared file system to share data sets among global controller and local Hadoop clusters.

7. ACKNOWLEDGMENTS

This work funded in part by the Pervasive Technology Institute and Microsoft. Our special thanks to Dr. Geoffrey Fox for providing us early access to FutureGrid resources and valuable feedback on our work. We also would like to express our thanks to Chathura Herath for discussions.

8. REFERENCES

[1] Buyya, R., Abramson, D., Giddy, J. Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid, in: Proceedings of the HPC ASIA'2000, China, IEEE CS Press, USA, 2000.

[2] Dean, J. and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113. DOI=10.1145/1327452.1327492 http://doi.acm.org/10.1145/1327452.1327492

[3] Ding, Z., Wei, X., Luo, Y., Ma, D., Arzberger, P. W., Li, W. W. Customized Plug-in Modules in Metascheduler CSF4 for Life Sciences Applications, New Generation Computing Volume 25, Number 4, 373-394, 2007, DOI: 10.1007/s00354-007-0024-6 http://dx.doi.org/10.1007/s00354-007-0024-6

RIGHTSLINK

[4] Frey, J., Tannenbaum, T., Livny, M., Foster, I., Tuecke, S. 2002. Condor-G: A Computation Management Agent for Multi-Institutional Grids. Cluster Computing 5, 3 (July 2002), 237-246. DOI=10.1023/A:1015617019423 http://dx.doi.org/10.1023/A:1015617019423

[5] FutureGrid, http://www.futuregrid.org

[6] Gentzsch, W. (Sun Microsystems). 2001. Sun Grid Engine: Towards Creating a Compute Power Grid. In Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID '01). IEEE Computer Society, Washington, DC, USA, 35-39

[7] Hadoop On Demand, http://hadoop.apache.org/common/docs/r0.17.2/hod.html

[8] Henderson, R. L.. 1995. Job Scheduling Under the Portable Batch System. In Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '95), Dror G. Feitelson and Larry Rudolph (Eds.). Springer-Verlag, London, UK, 279-294.

[9] Huedo, E., Montero, R. S., and Llorente, I. M. 2004. A framework for adaptive execution in grids. Softw. Pract. Exper. 34, 7 (June 2004), 631-651. DOI=10.1002/spe.584 http://dx.doi.org/10.1002/spe.584

[10] Kavulya, S., Tan, J., Gandhi, R., and Narasimhan, P. 2010. An Analysis of Traces from a Production MapReduce Cluster. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID '10). IEEE Computer Society, Washington, DC, USA, 94-103. DOI=10.1109/CCGRID.2010.112 http://dx.doi.org/10.1109/CCGRID.2010.112

[11] Keahey, K., Tsugawa, M., Matsunaga, A., and Fortes, J. 2009. Sky Computing. IEEE Internet Computing 13, 5 (September 2009), 43-51. DOI=10.1109/MIC.2009.94 http://dx.doi.org/10.1109/MIC.2009.94

[12] Litzkow, M. J., Livny, M., Mutka, M. W. Condor - A Hunter of Idle Workstations. ICDCS 1988:104-111

[13] Morris, G. M., Huey, R., Lindstrom, W., Sanner, M. F., Belew, R. K., Goodsell, D. S. and Olson, A. J. (2009), AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility. Journal of Computational Chemistry, 30: 2785–2791. doi: 10.1002/jcc.21256

[14] National Biomedical Computation Resource, http://nbcr.net

[15] Qiu, J., Ekanayake, J., Gunarathne, T., Choi, J. Y., Bae, S. Ruan, Y., Ekanayake, S., Wu, S., Beason, S., Fox, G., Rho, M., Tang, H., “Data Intensive Computing for Bioinformatics”, In Data Intensive Distributed Computing, IGI Publishers, 2010

[16] Staples, G. 2006. TORQUE resource manager. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06). ACM, New York, NY, USA, Article 8. DOI=10.1145/1188455.1188464 http://doi.acm.org/10.1145/1188455.1188464

[17] Teragrid, http://www.teragrid.org

[18] Tsugawa, M., and Fortes, J. A. B. 2006. A virtual network (ViNe) architecture for grid computing. In Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06). IEEE Computer Society, Washington, DC, USA, 148-148.

[19] Vaqué, M., Arola, A., Aliagas, C., and Pujadas, G. 2006. BDT: an easy-to-use front-end application for automation of massive docking tasks and complex docking strategies with AutoDock. Bioinformatics 22, 14 (July 2006), 1803-1804. DOI=10.1093/bioinformatics/btl197 http://dx.doi.org/10.1093/bioinformatics/btl197

[20] Yang, H., Dasdan, A., Hsiao, R., and Parker, D. S. 2007. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07). ACM, New York, NY, USA, 1029-1040. DOI=10.1145/1247480.1247602 http://doi.acm.org/10.1145/1247480.1247602

[21] Zaharia, M., Borthakur, D, Sarma, J. S., Elmeleegy, K., Shenker, S., and Stoica, I. Job Scheduling for Multi-User MapReduce Clusters, Technical Report UCB/EECS-2009-55, University of California at Berkeley, April 2009.

[22] Zhang, C., De Sterck, H., "CloudBATCH: A Batch Job Queuing System on Clouds with Hadoop and HBase," Cloud Computing Technology and Science, IEEE International Conference on, pp. 368-375, 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 2010

[23] Zhou, S, Zheng, X., Wang, J., and Delisle, P. 1993. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. Softw. Pract. Exper. 23, 12 (December 1993), 1305-1336. DOI=10.1002/spe.4380231203 http://dx.doi.org/10.1002/spe.4380231203

RIGHTSLINK