**A Hierarchical Framework for Cross-Domain MapReduce Execution**

Yuan Luo1,2, Zhenhua Guo1,2, Yiming Sun1,2, Beth Plale1,2, Wilfred Li 3,4

1 School of Informatics and Computing, Indiana University, Bloomington, IN, 47405

2 Pervasive Technology Institute, Indiana University, Bloomington, IN, 47408

3 San Diego Supercomputer Center, University of California, San Diego, La Jolla, CA, 92093

4 National Biomedical Computation Resource, University of California, San Diego, La Jolla, CA, 92093

{yuanluo, zhguo, yimsun, plale}@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 execution usually consists of a very large number of docking processes from multiple ligands that 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 does load-balancing by assigning tasks in accordance to the capabilities of each cluster and of each node, then return back to the global controller to finish the global reduction. Our experimental evaluations using AutoDock over MapReduce show that our load-balancing algorithm makes promising workload distribution of each clusters, and thus minimizes overall execution time span of the entire MapReduce execution.

**Categories and Subject Descriptors**

**C.2.4 [Distributed Systems]**: MapReduce Architecture - *Hierarchical MapReduce for Life Science Applications.*

**General Terms**

Performance, Experimentation

**Keywords**

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

# 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 programming model.

AutoDock [1] 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. The process of application of 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) [1] and BDT [5], 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 input parameter file (dpf). The results can then be summerized interactively using the desktop tools such as AutoDockTools or with a python script.

A typical AutoDock run 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 length of AutoDock run duration.

Cloud computing can provide virtually unlimited 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 dominant model for deploying and executing applications in a cloud, and AutoDock 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 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. If these isolated clusters can work together, they collectively become a more powerful resource. Unfortunately, users cannot simply throw a MapReduce framework such as Hadoop on top of these clusters to form a single larger MapReduce cluster because in most cases, the internal nodes of a cluster are not directly reachable from outside, but MapReduce requires that the master node to be able to directly communicate with any slave node, and this is also a part of the reason why MapReduce frameworks are usually deployed within a single cluster. Therefore, one challenge for us is to find a way to let MapReduce work across different domains, so these clusters can act collaboratively to more efficiently run MapReduce jobs, at least for certain classes of MapReduce jobs.

By observing the amount of time MapReduce jobs have spent in each Map and/or Reduce phase, Kavulya et al. [17] characterize MapReduce jobs into four pattern categories: map-only jobs, map-mostly jobs, shuffle-mostly jobs, and reduce-mostly jobs. Their study shows that 91% of the MapReduce jobs fall into the map-only and map-mostly jobs categories. MapReduce jobs that fit these two patterns can be distributed across multiple clusters to perform map-intensive computation and the outputs can then be easily combined in the global nodes.

We propose a hierarchical MapReduce framework that can distribute map-only and map-mostly jobs across different cluster domains, and thus effectively gathers the isolated resources from the individual domains into a more unified and powerful one. 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 work. Section 3 gives an overview of our hierarchical MapReduce framework. Section 4 presents more details on the application of AutoDock using MapReduce. Section 5 gives experiment setup and result analysis. Conclusion is given in Section 6.

# RELATED WORKS

With respect to the application of AutoDock, our earlier efforts presented at National Biomedical Computation Resource [2] Summer Institute 2009, address the performance issue of massive docking processes by distributing the jobs to the grid environment. We used the CSF4 [3] meta-scheduler to schedule docking jobs to heterogeneous clusters cross different virtual organizations. A MPI-based implementation of AutoDock4 [4] allows simultaneous and independent docking of multiple compounds on up to thousands of central processing units (CPUs). This MPI version of AutoDock reads a single binary file containing precalculated grids representing the protein–ligand interactions, and needs only two input parameter files for the entire docking run.

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 [6], SGE [7], PBS [8], LSF [9], etc, aim to provide highly optimized resource allocation, job scheduling, load balancing, within a single cluster environment. On the other hand, Grid Broker and Metascheduler, eg., Condor-G [10], CSF [3], Nimrod/G[11], GridWay [12], 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.

Clouds give users virtually unlimited, on-demand resource for computation and storage. Attributed to its ease of executing pleasingly 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) [13] uses the TORQUE resource manager [Sta06] to provision and manage independent MapReduce and HDFS instances on shared physical nodes. The authors of [14] have identified some fundamental performance limitation issues in Hadoop and 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 [15] 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 from the conventional MapReduce model to accomplish common relational algebra operations over distributed heterogeneous data sets [16]. 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. Our hierarchical MapReduce model, aims to enable map-only and map-most jobs to be run across a number of isolated clusters, so these isolated resources can collectively provide a more powerful resource for the computation.

# 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 supplied 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 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 output to perform a final reduction using the global reducer 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 [16], 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. 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. 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.

## 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. 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 roughly 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 have them deployed 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.



**Figure 1. Hierarchical MapReduce Architecture**

## 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. Users of our framework implement the three functions as a Mapper, a Reducer, and a Global Reducer, and submit the jar to the global controller. 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 Producer are executed on local clusters. The Global Reducer is also a conventional Reducer, except that it 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.

**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 | $$(k^{m},[v\_{1}^{m},…,v\_{n}^{m}])$$ | $$(k^{r},v^{r})$$ |
| Global Reduce | $$(k^{r},[v\_{1}^{r},…,v\_{n}^{r}])$$ | $$(k^{o},v^{o})$$ |

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 set key/value pairs as the 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.



**Figure 2. Programming Model**

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 levels by turning the non-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.

Our model brings the opportunity for solving problems with extreme large data sets which can’t be practically fitted in a single layer MapReduce setup. One observation is that a user may have access to several clusters. For example, we have access to IU Quarry, FutureGrid, and Teragrid clusters. Each cluster imposes limit on maximum number of nodes a user can uses at any time. If the user needs more nodes, he/she must manually split the job and run separate jobs on different clusters. This is time consuming and error-prone. Another observation is that usually computation nodes within a cluster cannot be accessed from outside.

## Job Scheduling and Data Partitioning

The challenges for our work are (1) how to partition the data based on the MapReduce application; (2) how to balance the workloads among each Local MapReduce Clusters; etc.

The input datasets for a particular MapReduce job may either sit on the user side and pass all the way down to the local clusters during execution, or pre-exist on the local clusters and expose the metadata to the user who runs the MapReduce job. The scheduler on the Global Controller takes into consideration the data locality when partition the datasets and schedule the job.

Assume all the map tasks of a MapReduce application are compute-intensive and take approximately same amount of time to run, which can be achieved by AutoDock MapReduce Implementation explained in the next section. The scheduling algorithm we use for our framework is as follows. Let $MaxMapper[i]$ be the max number of Mapper that can be running 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, $NumCore[i]$ be the total number of CPU Cores on Cluster[i], and $ρ\left[i\right]$ defines how users set the $MaxMapper[i]$ using $NumCore[i]$, where i $\in \{1,..., n\} $is the Cluster number, so that,

 $MaxMapper\left[i\right]=ρ[i]×NumCore[i]$ (1)

where we set $ρ\left[i\right]=1$ in the local MapReduce clusters for compute intensive jobs, so we get (2) ,

 $MapperAvail\left[i\right]=MaxMapper\left[i\right]-MapperRun[i]$ (2)

The weights of each sub-jobs can be got from (3) where factor $θ[i]$ is the computing power of each cluster, eg, the CPU speed, memory size, storage, etc. The actual $θ[i]$ varies depending on the character of the jobs, whether they are compute intensive or I/O intensive, etc.

 $Weight\left[i\right]=\frac{θ[i]×MapperAvail\left[i\right]}{\sum\_{i=1}^{N}θ[i]×MapperAvail\left[i\right]} $ (3)

Let $JobMap[x]$ be the number of total Map tasks for a particular job x, which can be get from the key size of the Map() inputs, $JobMap\left[x\right][i]$ be the number of Map tasks to be scheduled to $Cluster[i]$ for job x, where i $\in \{1,..., n\} $ is the Cluster number, so that,

 $JobMap\left[x\right][i] = Weight[i]× JobMap[x] $ (4)

After partitioning the MapReduce job to Sub-MapReduce jobs using equation (4), 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.

# AUTODOCK MAPREDUCE

We apply the MapReduce paradigm to AutoDock application using the hierarchical MapReduce framework to prove the feasibility of our approach. We take the outputs of AutoGrid 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 contains 7 fields showed in Table 2.

**Table 2. AutoDock MapReduce input fields and descriptions**

|  |  |
| --- | --- |
| **Field** | **Description** |
| ligand\_name | Name of the ligand |
| autodock\_exe | Path of AutoDock executable |
| input\_files | Input files of AutoDock |
| output\_dir | Output directory of AutoDock |
| autodock\_parameters | AutoDock parameters |
| summarize\_exe | Path of summarize script |
| summarize\_parameters | Summarize script parameters |

Each input record corresponds to a map task. The job scheduler uses user-implemented InputFormat and RecordReader to go through input file to calculate number of records, so that the job scheduler knows the number of records without incurring additional burden on end users. Combined with cluster weights, global scheduler is able to calculate number of ligands to be processed by each cluster.

1) A Map task takes a ligand to run autodock binary executable against a shared receptor, and then run summarize\_result4.py script to output the lowest energy result using a constant intermediate key.

2) A Reduce task takes all the values corresponding to the constant intermediate key and sort the values by the energy from low to high, and output the sorted results in a file using a local reducer intermediate key.

3) A Global Reduce finally takes all the values of the local reducer intermediate key, sort and combine them into a single file by the energy from low to high.

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 load information we need to external applications. As a result, we must modify Hadoop code to add an additional daemon that collects load data by using Hadoop Java APIs.

# EVALUATIONS

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

MapReduce uses a different model than traditional HPC. It does not distinguish computation nodes and storage nodes. Each node is responsible for both computation and storage. Obvious advantages include better fault tolerance, scalability and data locality scheduling.

To deploy Hadoop to traditional HPC clusters, we first use the built-in job scheduler (PBS) to allocate nodes. To balance maintainability and performance, the Hadoop program is installed in shared directory while data is stored in local directory. The reason is that Hadoop program (Java jar files, etc) is loaded once by Hadoop daemons while 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 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. Testbed specifications are listed in Table 3.

**Table 3. Testbed 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 |

Considering Autodock is a CPU-intensive application, we set $ρ\left[i\right]=1$ in section 3.3 so that the maximum number of map tasks on each node to be the number of cores of 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.



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

In our experiments, we use 6000 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 result is generated becomes. Based on prior experience, the ga\_num\_evals is typically set to 2,500,000 to 5,000,000. We configure it to 2,500,000 in our experiments.

Figure 3 describes 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 number of running map tasks quickly grows to 160 in the beginning and keeps approximately constant for long time. Towards the end of job execution, it drops to small value quickly (roughly 0 - 5). But there is a tail near the end so that node usage ratio is low. At this moment, if new mapreduce tasks come in, the available mapper will be occupied by those tasks.

**Test Case 1:**

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

**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 |



**Figure 4. Local Cluster MapReduce Execution Time based on different number of map tasks.**

As is reflected in Figure 4, total execution time vs.number of map tasks in test case 1 on each cluster are close to linear, regardless of the startup overhead of the MapReduce jobs. The jobs running on Quarry Cluster are approximately 50% slower than running on Alamo Cluster and Hotel Cluster. We guessed the reason is that nodes of Quarry have slower CPUs compared with that of Alamo and Hotel.

**Test Case 2:**

When there is no MapReduce jobs running on our clusters, MapperAvail[i] and MaxMapper[i] have the same value -160. In equation (3) in section 3.3, we set $ θ\left[i\right]=C$, where C is a constant, and i $\in \{1, 2, 3\} $ reflects the number of our three clusters. We can have the weight of map tasks distribution on each cluster Weight[i]=1/3. We evenly 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 for triggering the local cluster executions. After the local MapReduce execution, the outputs will be staged back to the global nodes for final global reduce. Figure 5 shows the data movement cost in the global->local and local->global contexts. The input dataset of AutoDock contains 1 receptor and 6000 ligands. The receptor is described as a set of gridmap files (approximately 20 files with the total size of 35MB), which together with jar executable and job configuration file (300KB), are transferred onto every local clusters.. The 6000 ligands are stored in 6000 separate directories, each of which is approximately 5-6 KB large. So, for each cluster, we compress on the global controller 1 receptor file set and 2000 ligands directories, together with jar executable and job configuration files, to a total size of 14MB tarball and transfer to the destination cluster, where the tarball is decompressed. This global-to-local procedure, namely data stage-in, is described in the graph tagged with “Global Controller -> Local Cluster”. Alternatively, when the local MapReduce jobs finish, the outputs together with control files (normally with the total size of 300-500KB) are compressed into a tarball and transfered back to the global controller. This local-to-global procedure, namely data stage-out, is described in the graph tagged with “Local Cluster -> Global Controller”. As we can see from the graph, 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. Alamo Cluster takes a little bit longer to transfer the data but the difference is ignorable regarding the relatively long-run of local mapreduce executions.



**Figure 5. Data Movement Cost of Equally Partitioned Datasets and their local mapreduce outputs**

Running 2000 map tasks on each of the local MapReduce clusters, the execution time vary with respected to different specification of the clusters. The local mapreduce execution makespans, including data movement costs (both data stage-in and stage-out) are showed in Figure 6. The Hotel and Alamao Clusters takes similar amount of time to finish the jobs, but Quarry Cluster takes extra 50% amount of time, which is approximately 3000 seconds in number than Hotel and Alamo to finish. The global reduce procedure which only takes 16 seconds will only be invoked after all the local results are ready in the global controller site. Thus, the relatively poor performance on Quarry Cluster becomes the bottleneck for current job distribution.



**Figure 6. Local MapReduce turnaround time for Equally Partitioned Datasets, including Data movement Cost.**

**Test Case 3:**

From test case 1 and 2, we can clearly see that the computing power varies among different clusters, although the clusters are under the same number of compute nodes and cores. As shown in results above, considering number of cores is not enough. Capabilities of cores are also important. All three clusters have the same number of cores. But to process the same amount of data takes significantly different time. Quarry is much slower than Alamo and Hotel. Specs of cores of cluster 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. Ratio of CPU frequency and that of processing time match roughly. So we consider that difference in processing time mainly results from different core frequencies. We refined our scheduling policy to add CPU frequency as a contribution to set $ θ\left[i\right]$. Here we set $θ\left[i\right]=2.93$ to Hotel Cluster, $θ\left[i\right]=2.67$ to Alamo Cluster and $θ\left[i\right]=2$ to Quarry Cluster. Thus, the weights are weight[1]=0.3860, weight[2]=0.3505, and weight[3]=0.2635 for Hotel, Alamo and Quarry Cluster respectively. The dataset is weighted partitioned. Table 5 shows how the dataset is partitioned.

Figure 7 shows the data movement cost in the weighted partition scenario. The variations of tarballs size with different number of ligands sets are quite small, which is smaller than 2MB. So the tarballs for transferring fit the range of 12MB-14MB. 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 Cluster takes a little bit longer to transfer the data but the difference is also ignorable regarding the relatively long-run of local mapreduce executions as the previous test case.

**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. Data Movement Cost of Weighted Partitioned Datasets and their local mapreduce outputs**

With weighted partition, the local mapreduce execution makespans, including data movement costs (both data stage-in and stage-out) are showed 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 sort the results. The average time taken to process 6000 map tasks (ligand/receptor docking) is 16 seconds.



**Figure 8. Local MapReduce turnaround time for Weighted Partitioned Datasets, including Data movement Cost.**

# 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 “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 return back to the global controller to run Global Reduce function. We use resource capacity aware algorithm to balance the workload among clusters. We use AutoDock application as a test case to show the performance of our framework. The results show that the workloads are well balanced and the total makespan is kept in minimum.

Our future work will address the data locality issue of the input datasets, especially for data-intensive and data-sensitive computing. Instead of move data to the computation, we will investigate moving computation to data, which might be in large volume and/or privacy sensitive.

# ACKNOWLEDGMENTS

Our special thanks to Geoffrey Fox for providing us early access to FutureGrid resources and valuable feedback on our work. We also thank Judy Qiu and Chathura Herath for discussions.

# REFERENCES

1. Garrett M. Morris, Ruth Huey, William Lindstrom, Michel F. Sanner, Richard K. Belew, David S. Goodsell, Arthur J. Olson: AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility. Journal of Computational Chemistry 30(16): 2785-2791
2. National Biomedical Computation Resource, http://nbcr.net
3. Zhaohui Ding, Xiaohui Wei, Yuan Luo, Da Ma, Peter W. Arzberger and Wilfred W. Li, 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
4. Collignon, B., Schulz, R., Smith, J. C. and Baudry, J. , Task-parallel message passing interface implementation of Autodock4 for docking of very large databases of compounds using high-performance super-computers. Journal of Computational Chemistry, n/a. doi: 10.1002/jcc.21696
5. Montserrat Vaqué, Anna Arola, Carles Aliagas and Gerard Pujadas. BDT: an easy-to-use front-end application for automation of massive docking tasks and complex docking strategies with AutoDock. In Oxford Journals of Bioinformatics, May 2006.
6. Michael J. Litzkow, Miron Livny, Matt W. Mutka: Condor - A Hunter of Idle Workstations. ICDCS 1988:104-111
7. Wolfgang Gentzsch: Sun Grid Engine: Towards Creating a Compute Power Grid. CCGRID 2001:35-39
8. Robert L. Henderson: Job Scheduling Under the Portable Batch System. JSSPP 1995:279-294
9. Songnian Zhou, Xiaohu Zheng, Jingwen Wang, Pierre Delisle: Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. Softw., Pract. Exper. (SPE) 23(12):1305-1336 (1993)
10. James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. 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
11. Rajkumar Buyya, David Abramson, Jonathan Giddy: Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid CoRR cs.DC/0009021 (2000)
12. Eduardo Huedo, Rubén S. Montero, Ignacio Martín Llorente: A framework for adaptive execution in grids. Softw., Pract. Exper. (SPE) 34(7):631-651 (2004)
13. Hadoop On Demand, [http://hadoop.apache.org/common/docs/r0.17.2/hod.html](http://www.google.com/url?q=http%3A%2F%2Fhadoop.apache.org%2Fcommon%2Fdocs%2Fr0.17.2%2Fhod.html&sa=D&sntz=1&usg=AFQjCNFadAdZVB-18LwyW-ErH-Ic-J-TOQ)
14. Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker and Ion Stoica, Job Scheduling for Multi-User MapReduce Clusters, Technical Report UCB/EECS-2009-55, University of California at Berkeley, April 2009.
15. Cheng Zhang and Hans De Sterck, “CloudBATCH: A Batch Job Queuing System on Clouds with Hadoop and HBase”, in Proceedings of the Second International Conference on Cloud Computing (CloudCom 2010), November, 2010.
16. Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. 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](http://www.google.com/url?q=http%3A%2F%2Fdoi.acm.org%2F10.1145%2F1247480.1247602&sa=D&sntz=1&usg=AFQjCNFp7pJFzecXEjCJXDaPpEXOEXSVQA)
17. Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, Priya Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," Cluster Computing and the Grid, IEEE International Symposium on, pp. 94-103, 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010
18. [Sta06] Staples, Garrick, “TORQUE resource manager”, in proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC’06), 2006.