# Evaluating SPLASH-2 Applications Using MapReduce

Shengkai Zhu, Zhiwei Xiao, Haibo Chen, Rong Chen, Weihua Zhang, and Binyu Zang

Parallel Processing Institute, Fudan University

**Abstract.** MapReduce has been prevalent for running data-parallel applications. By hiding other non-functionality parts such as parallelism, fault tolerance and load balance from programmers, MapReduce significantly simplifies the programming of large clusters. Due to the mentioned features of MapReduce above, researchers have also explored the use of MapReduce on other application domains, such as machine learning, textual retrieval and statistical translation, among others.

In this paper, we study the feasibility of running typical supercomputing applications using the MapReduce framework. We port two applications (Water Spatial and Radix Sort) from the Stanford SPLASH-2 suite to MapReduce. By completely evaluating them in Hadoop, an open-source MapReduce framework for clusters, we analyze the major performance bottleneck of them in the MapReduce framework. Based on this, we also provide several suggestions in enhancing the MapReduce framework to suite these applications.

## 1 Introduction

MapReduce [1], advocated and popularized by Google, has been prevalent for data-parallel applications due to its simplicity yet still powerful processing capability. It has been widely deployed in Google's own clusters and used for various applications such as web-search, indexing and log analysis.

Though Google's implementation detail is fairly secretive for the public domain, Apache has provided Hadoop [2], an open-source implementation of the MapReduce framework. It has gained significant popularity recently due to its practicality, cost-effectiveness and openness. Thus, it has been widely adopted in various application domains such as statistical machine translation [3], textual retrieval [4] and machine learning [5].

The elegance of MapReduce, the readily availability of the cost-effective Hadoop implementation would also open opportunities to run many parallel or supercomputing applications on commodity clusters. Running parallel or supercomputing applications on MapReduce, if applicable, would make the power of solving many difficult scientific problems ubiquitously accessible at a very low cost. Bryant [6] has recently discussed the possibility of running some data-intensive supercomputing applications such as genomic sequences and earthquake modeling on commodity clusters. Unfortunately, there are currently few studies on the performance characteristics of parallel applications on commodity clusters with multi-core.

In this paper, we port and evaluate two parallel applications from the SPLASH-2 [7] benchmark suite which originally run in large shared-memory multiprocessors to a

Y. Dou, R. Gruber, and J. Joller (Eds.): APPT 2009, LNCS 5737, pp. 452–464, 2009.

small-scale commodity clusters with multi-core, aiming at studying the performance characteristics of these applications on commodity clusters.

We have conducted a detailed evaluation on the performance characteristics of these applications. Our evaluation results in a 17 dual-core cluster (1 master node, 16 slave nodes) show there are some performance bottlenecks and we further summarize the key causes of the slowdown. With a detailed and complete analysis, we also present several potential optimization opportunities.

The rest of the paper is organized as follows. The next section presents the necessary background knowledge on MapReduce and Hadoop. In section 3, we port two typical scientific applications from SPLASH-2 suite to run on Hadoop and illustrate the major issues associated with the porting. Section 4 presents a detailed performance evaluation of Hadoop on a commodity cluster. Section 5 discusses several optimization opportunities to improve the performance of MapReduce for supercomputing applications on commodity clusters. Section 6 discusses the related work and section 7 concludes this paper.

## 2 Background

This section presents the necessary background information on the general MapReduce programming model and the design and implementation of Hadoop.

### 2.1 MapReduce Programming Model

The programming model of MapReduce is inspired by the functional programming primitives such as *Map* and *Reduce*. MapReduce processes the input and intermediate data in a Single Program Multiple Data (SPMD) fashion. The *Map* processes the input data and generated a set of $\langle key, value \rangle$ pairs, while the *Reduce* aggregates all $\langle key, value \rangle$ pairs according to the key.

The following pseudo-code in Figure 1 shows the *Word Count* application written using the MapReduce programming model, which counts the number of occurrences of each word in a document. The *Mapper* function emits a $\langle word, 1 \rangle$ pair for each *word* in document, and the *Reducer* function counts all occurrences of a *word* as the output.

```
//input = a document
//pairs: key = word, value = 1
Mapper (input){
   for each word in input:
      emit_inter( word, 1 );
}
```

```
//key = word
//values = a set of value
Reducer (key, values){
   int sum = 0;
   for each value in values:
      sum += value;
   emit( key, sum );
}
```

**Fig. 1.** *Mapper* and *Reducer* of *Word Count* in MapReduce

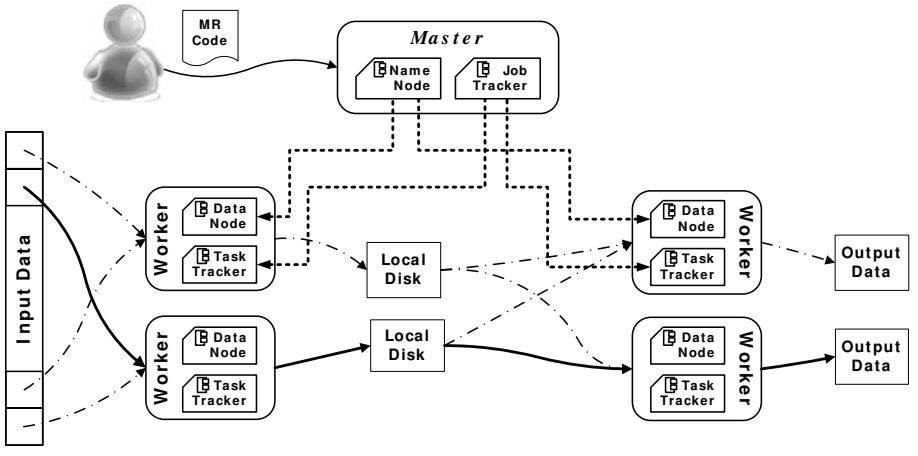


**Fig. 2.** MapReduce Execution Flow

### 2.2 The Hadoop Design and Implementation

Hadoop is an open-source implementation of the MapReduce framework. Hadoop uses a distributed file system, namely Hadoop Distributed File System (HDFS) to store the input and the final results. HDFS manages a number of local disks owned by the nodes in a cluster and maps them to a single file system. The HDFS resembles the Google File System in the fashion of handling storage failures using several replicas of the same data. One of the key principles in Hadoop is that "moving computation is much cheaper than moving data". Thus, Hadoop schedules the MapReduce tasks to the node near the data storage to minimize the data transfers.

An overview of the architecture and the execution flow of Hadoop are shown in Figure 2, which uses a master-slave mode. There is a master node that runs the *Job Tracker* for task allocation and scheduling, and *Name Node* for HDFS metadata management. To run a MapReduce task, the *Job Tracker* allocates the *Task Trackers* on the slave nodes to run the *map* or *reduce* tasks. Each slave node may also be the data node, which stores the data blocks of file in HDFS. The task tracker consults the *Name Node* to get the specific *Data Node* to get the data for a file.

## 3 Implementing SPLASH-2 Applications with Hadoop MapReduce

This section introduces the SPLASH-2 benchmark suite and how two of them are ported to the MapReduce framework.

### 3.1 SPLASH-2 Suite

The SPLASH-2 suite consists of a set of complete applications and computational kernels. The programs represent a variety of computation workloads in scientific, graphics

computing and engineering. The suite is designed to facilitate the study of centralized and distributed shared address-space multiprocessors. We choose *Water Spatial*, a water molecule simulation system and *Radix Sort*, an integer radix sort kernel for porting and evaluation. We believe that these two typical scientific and engineering programs cover the major characteristics of supercomputing with MapReduce.

Water Spatial is an N-body molecules dynamics application that evaluates the forces and potentials which occur over time in a cluster of water molecules in a liquid state. It is improved from the program water in SPLASH [8]. In an initial state, configurable number of water molecules are scattered in a cubical space. They are generated globally with random coordinate and velocity. Also many other physical and system parameters are carried by each molecule. Most of them will be updated several times during the whole life time of the application. Further documentation and details of the Water Spatial models can be found in [9, 10, 11].

Radix Sort is a small computational kernel performing sorting on integers using an iterative algorithm. Its implementation is based on [12].

### 3.2 Implementing Water Spatial(WS) and Radix Sort(RS) in MapReduce

**Data Structures.** In typical supercomputing applications, lots of mathematical, physical and system parameters are involved during the whole computation. Arrays and matrices are the most commonly adopted data structures. And in a system with large amounts of elements, the items are also kept in a list-based structure.

Due to well-defined partition methods and synchronization mechanisms, access to shared data is not difficult in a shared address-space environment. However, in a cluster environment, data structures need to be serialized into the distributed storage system for remote access. In Hadoop, the HDFS (Hadoop Distributed File System) is deployed to hold the data.

As a result of heavy network communications, access to shared data turns to be a significant source of overhead. Hence, the data partition policy in MapReduce determines the efficiency of the parallel algorithm implemented. A well-designed data structure and partition method could avoid a lot of unnecessary network communications, which could be the bottleneck in many cases.

Data updates in MapReduce can be done in different approaches. For fields owned by each basic element, its information can be refreshed through a direct update in the map/reduce phase. A global aggregative variable is usually updated in a synchronization point, accomplished by a MapReduce job with single reduce task. Data movements are the most complicated and common cases in typical supercomputing applications, resulting from changes of inter-data relation, which in turn forces a reconstruction of data partitions.

**Computation Steps.** Many supercomputing applications can be divided into several computation steps, often with a number of iterations doing a series of calculation. Between two consecutive steps, global data synchronization is performed to ensure the correctness of succeeding computing.

In MapReduce, unlike the shared-memory environment, data synchronization can only be performed after completion of a job and is costly. Usually, the number of global barriers defines a lower bound of the number of MapReduce jobs.

According to different behaviors of MapReduce jobs, jobs composing a typical supercomputing can be classified into three categories: element-update jobs, global-variable aggregation jobs and mixed jobs. The mixed job performs the element-update and does aggregation for a global variable in the same phase.

During a MapReduce computation, each map/reduce task works on their local copy of data. Data updates on global storage have to be performed after each MapReduce job. The distributed file system significantly affects the efficiency of the data-sharing. There is also consistency problem associated with it. The computation in each phase is thus required to dump the updated data into distributed storage with a specific format, which can be recognized and read effectively by the next worker. Usually the formats are designed specifically for each situation.

**A Walkthrough for Water Spatial.** At the beginning of a Water Spatial instance, random input data is generated and stored in HDFS, with only append operation allowed. Thus, the data file has to be reconstructed after each update.

The storage format of the basic element, water molecule, is shown in Figure 3. It consists of coordinates and other parameters holding the force and energy information. However, these two parts of parameters are rarely modified simultaneously in the same phase. This makes it unnecessary to hold these two parts in the same chunk of storage. During the data reconstruction, accesses to the part not involved in computation would unnecessarily increase the network load. Taking this into consideration, we store the coordinates of molecule and other physical information in two separated chunks. Each molecule will be assigned a unique identifier used as an index to refer to the both parts.

Water Spatial consists of a series of complex computations, with all three kinds of MapReduce jobs involved. The detail computation flow is shown in the Figure 4. Three phases can be transformed as mixed jobs, while the rest perform only element update.

Input data in Water Spatial is partitioned on the molecules according to their coordinates. In the shared-memory version of SPLASH-2, molecules with the same

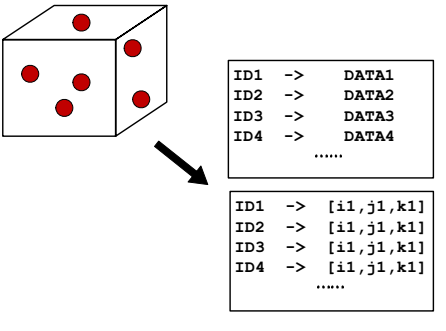


**Fig. 3.** Data Format of Molecules

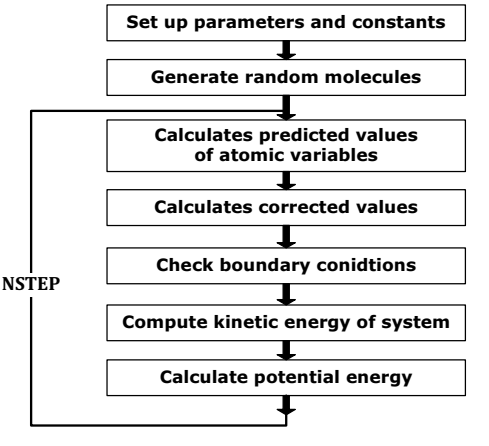


**Fig. 4.** Execution Flow of Water Spatial

```
Mapper (input){
  for each molecule in input:
    local_sum = Collect(molecule );
    emit_inter( id, local_sum );

}
```

```
Reducer (key, values){
  global_sum = 0;
  for each value in values:
    global_sum += value;
  emit( key, global_sum );
}
```

**Fig. 5.** Mapper and Reducer for Aggregation Job

```
Mapper (input){
  for each molecule in input:
    Update( molecule );
    emit_inter( NULL, molecule );
}
```

**Fig. 6.** Mapper for Update-Only Job

```
Mapper (input){
  for each molecule in input:
    if ( isFake(molecule) )
      molecule.value = local_val;
    else
      local_val += Update( molecule );
    emit_inter( NULL, molecule );
}
```

**Fig. 7.** Mapper for Mix Job

coordinates are processed in a single thread together. We keep such a design here but hold their coordinates and other parameters in different chunks.

An aggregation-only job can be performed intuitively by MapReduce. The *Mapper* in Figure 5 collects information from all molecules. The *Mapper* calculates concerned value from some fields and emits it to the intermediate key-value pair. Each key in these pairs represents the different global variables aggregated.

Most update-only jobs can also be processed easily in MapReduce. The fields of each element are modified through the computation. The *Mapper* described in Figure 6 needs only to update the molecule passed in and then bounces it to the *Reducer*.

Since the output key-value pair should always be the same type during a computation phase, the *Mapper* designed for a mixed job in Figure 7 is much more complicated. In our implementation, the *Molecule* is taken as our output key-value pair type. Besides, we use fake molecules to carry the aggregation values.

The *Reducer* for all kinds of job can be easily set to the *IdentityReducer*, which is built in the Hadoop framework, simply doing sorting on the map outputs. In some aggregation jobs, the number of reduce workers has to be set to one. Otherwise, a routine out of the framework should do the aggregation for *Reducer*.

There are two special phases in Water Spatial which compute inter-water forces and their potential energy. The computation needs to calculate the molecules with all their neighbors within effective radius. While the radius is larger than a single data partition block, this *Mapper* would process much more molecules than regular cases. Further, the inter-neighbor communication in a 3D space significantly increases the network load for these redundant transmissions.

**A Walkthrough for Radix Sort.** Data involved in Radix Sort is a list of integers to be sorted. The input set can be partitioned intuitively and will not be modified during the computation. This makes its storage format much simpler than that of Water Spatial.

Radix Sort consists of iterative histogram computing. The computation involved in the program is a simple histogram performing for each radix *r* digits. The number of the MapReduce jobs is determined by the iteration number, which is in turn determined by the max integer provided by users. The *Mapper* for Radix Sort simply ranks the entire integer passed in on the specific *r* digits for each iteration, forming a histogram with $2^r$ buckets, whose indices range from 0 to $2^r$-1. Each integer is processed from the least significant *r* digits to the most significant *r* digits through the loop.

All the local histograms constructed in the map tasks will be merged into a single global histogram. Thus, the number of reduce tasks is required to be set to one. According to the global histogram, a partial sorting on those *r* digits can be performed correctly. After the last iteration, these integers come to an ordered state.

Instead of only ranking the $2^r$ buckets of each iteration, our implementation of Radix Sort takes the whole integer as the element for histogram in computation and collects those integers with the same *r* digits value in the corresponding bucket. This implementation ensures that an ordered sequence can be reached just in the time of processing each reduce phase. No more efforts for permutation are needed in the client end that starts the job, which is necessary if *Reducer* only ranks the buckets. In that case, a large amount of data transmission would occur on this single node to copy all integers required at each permutation computing.

**Table 1.** Line of Code in MapReduce version

| Application | Components | Line of Code |
|---|---|---|
| Water Spatial | Original code | 1984 |
| | Append code | 2002 |
| | Interface and Framework | 1226 |
| | Data storage and communication | 776 |
| Radix Sort | Original code | 705 |
| | Append code | 318 |
| | Interface and Framework | 301 |
| | Data storage and communication | 17 |

**Porting Effort.** Table 1 shows the porting effort to translate these two programs onto the Hadoop framework. The original amount of code in MapReduce is 1984 lines. By reusing most of the computation code, we still need to append other 2002 lines of code. The major part of the extra code for Water Spatial is for the interface and framework supports required by Hadoop-0.19.1. Data partition and distributed file system operations code also result in some extra code. The large number of code for framework is caused by many similar routines to setup MapReduce jobs with different configurations.

Data sharing in memory are replaced with network communication on the MapReduce cluster, which is also a source of extra code. As much more inevitable data dumping and loading occur in this condition, the code for defining formats to store the data

structure is also needed in this non-sharing address-space environment. However, the data format code for Water Spatial can be well reused by other scientific applications. In contrast, Radix Sort needs almost no data communication between the consecutive permutation steps. Most of its code is for *Mapper* and *Reducer*, making its porting quite easy. By using a same algorithm design but different coding in Java, Radix Sort does not reuse many original code, which are programmed mostly on dealing with memory in C.

## 4 Evaluation

In this section, we present and analyze the experimental results of Water Spatial and Radix Sort.

### 4.1 Experiment Setup

We conduct our experiments on a cluster consisting of 1 master node and 16 slave nodes. We have single master node running *Job Tracker* and *Name Node*. All slavers run as both *Task Trackers* and *Data Nodes*. Each slave machine has a dual-processor, 2GB main memory and a SATA disk. Network connectivity is by 100M/sec Ethernet links connecting into the campus local network.

In our experiments for Water Spatial and Radix Sort, we evaluate the performance characteristics of our MapReduce implementation. Its core computation algorithms are the same as the original ones in SPLASH-2. We use Hadoop-0.19.1, the most recent version of Hadoop and Java SE Runtime Environment 1.6 as our experiment platform. The input size of Water Spatial experiments varies from $18^3$ to $57^3$, indicating the number of molecules. The size of input data file for Radix Sort varies from 12.5MB to 100MB.

### 4.2 Overall Performance

Figure 8 and Figure 9 show the overall performance of these two applications. In both applications, the scalability with input size demonstrated on cluster is poor.

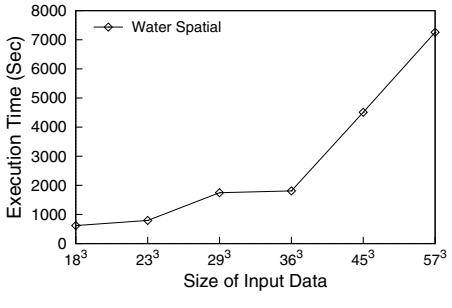


**Fig. 8.** The overall execution time of WS on a commodity cluster

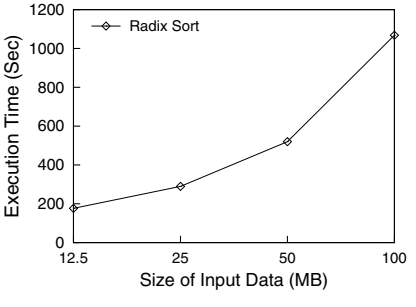


**Fig. 9.** The overall execution time of RS on a commodity cluster

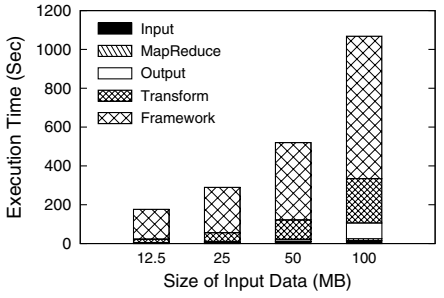


**Fig. 10.** The execution time breakdown of RS on a commodity cluster

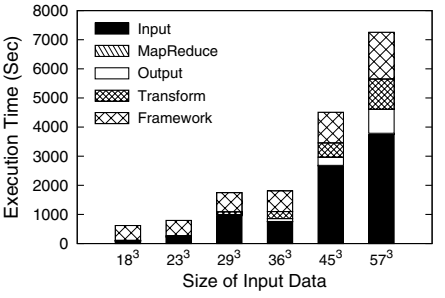


**Fig. 11.** The execution time breakdown of WS on a commodity cluster

### 4.3 Performance Breakdown

The time breakdown of Water Spatial in Figure 11 presents the execution time spent in different components of Hadoop. The overall time is divided into five parts. The **Input** and **Output** parts count for the time spent for data reading/writing from/to HDFS for *Mapper*/*Reducer*. The **MapReduce** part here stands for the time spent in the computation inside *Mapper* and *Reducer*, which execute as the core computation algorithms. The **Transform** part denotes the data transformation time from the output of a MapReduce job to the required format in the next step. The last part, **Framework** time, is the time spent in MapReduce job creation, Hadoop scheduling and the map/reduce task initialization. The intermediate data transmissions are also accounted for the **Framework** part.

From the time breakdown, we notice that time for computation in *Mapper* and *Reducer* is negligible. This part does the major computation for the Water Spatial simulation. And compared with the overall execution time in shared-memory environment, time in this part has a speedup more than 2x. And this speedup shows the advantage of parallel-computing in MapReduce for the general application without heavy loads of data communication. On the other hand, the MapReduce framework causes much more negative side-effects for the scientific application. From the figure, we notice the time in **Input** part increases rapidly with the data input size. This is because our implementation has to scan through some data files more than one time in certain phases, such as the inter-molecule phase and potential energy phase. Thus the network loads for this part can increase much more quickly than that of the **Output** and **Transform** parts. The molecule layout at the input size $29^3$ is a little irregular. It can not be scanned sequentially well and thus suffers a significant amount of HDFS cache miss. We can see that the execution time at $29^3$ is only sightly shorter than that at $36^3$. The **Framework** time grows with input size because of the increased transmission of intermediate data. As Figure 10 illustrates, Radix Sort also spends most of its execution time on data communication. Because of its less data synchronization and simpler storage format, the **Input** and **Output** time of Radix Sort is insignificant compared with the **Framework** time.

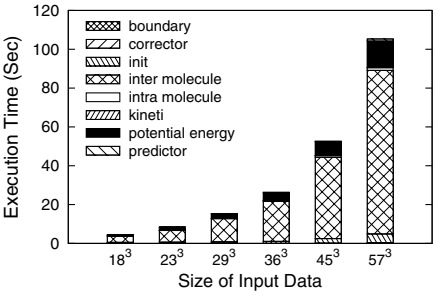


**Fig. 12.** The execution time breakdown of WS on a single machine

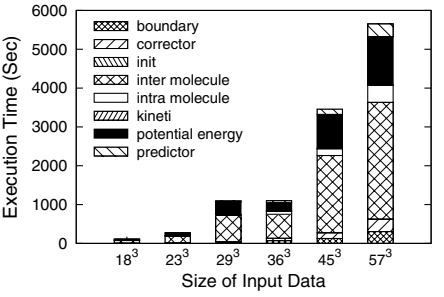


**Fig. 13.** The execution time breakdown of WS on a commodity cluster

### 4.4 Affects with Application Characteristics

With the breakdown time in Figure 12 and Figure 13, we further investigate the distribution of time spent in different computation phase of Water Spatial. The computation phases here are divided according to the program flow described in Figure 4. The time of Water Spatial is mainly spent in inter-water force computing and potential energy calculating. These are the only two phases involving the neighbor partitions of data. Thus they are dominant in a whole run since their greater computation loads. The inter-water force computing time is longer since it works as a mixed job in MapReduce while potential energy calculating is an update-only job. And in the MapReduce version with a large input size, time of other phases start to become obvious. This happens because a large number of data communications is raised.

## 5 Optimization Opportunities

**Storage System.** The HDFS used in Hadoop framework is designed to work with regular data-intensive applications. Files in the HDFS can only be appended at the tail. The in-place data updating cannot be supported or worked out with a simple alternative using existing approaches. By studying Water Spatial and Radix Sort, we found the matrices and multi- or many-dimension arrays are the most common structure used to hold data. Frequent update operations on these kinds of structures result in quiet a lot of overhead in reconstruction. The dumping and loading from arrays or matrices force other extra efforts to be done. The large proportion of HDFS time showed in our experimental results of Water Spatial just verifies that. Considering the native characteristics of supercomputing applications, a specific lower-level storage system is necessary. And good support to distributed arrays and matrices access will lead to a great improvement on data communication. The general-purpose distributed storage system or sequential file system cannot work well with those structures.

**Output Directing.** In supercomputing applications, work completion by multiple MapReduce jobs causes another performance problem. The output from previous job

needs to be dumped onto HDFS. Such data is then read by tasks from the next job. The indirect data transmission costs a large fraction of execution time. Actually, before dumping the output, tasks for the coming jobs have already been scheduled. Allowing output written directly to its destination can save the time writing to HDFS. This avoids a great waste of network resource and also saves the time significantly.

**Simple Aggregation Function.** Simple aggregation operation in a supercomputing application, like sum of variables, is also a common kind of computation. For a global aggregation function computed, the number of reduce worker has to be one, which forms a bottleneck during the processing. However, the overhead from passing a variable to sum is negligible compared to the creation time of a heavy reduce worker. Note that, a functional enhancement should be augmented. It should allow pass the variables for simple aggregation directly to the application submission end and skip the unnecessary reduce phase. Thus quite a lot of time caused by framework during the reduce phase and its corresponding HDFS operations could be saved.

**Multi-Phase in Single Pass.** The mixed job we introduced often leads to a difficult situation for programming. Thinking of the element update and variable sum in the same pass in Water Spatial, fake elements are created for carrying aggregated variable. In most cases, several operations on the same partition of data are independent during the same pass. Due to lack of support from the MapReduce framework, the operations have to be separated apart or programmed with a bad understandability. A multi-functional *mapper* for MapReduce can improve the working efficiency greatly. Furthermore, since MapReduce cannot ensure the tasks processing the same partition to be assigned to the same nodes, which causes many avoidable data communication.

## 6 Related Work

The evolvement of the MapReduce programming model, MapReduce, invented and popularized by Google, has been widely deployed into the production systems inside Google. Outside Google, Apache has designed and implemented Hadoop, an open-source alternative of Google's MapReduce, which is implemented using Java and built upon the Hadoop Distributed File System (HDFS). Due to the simplicity of MapReduce, the database community also extends the MapReduce programming model by adding an additional stage, called Merge, to support the joint of two tables [13].

There have been a lot of efforts in trying MapReduce to other domains other than the web-search domain. Chu et al. [14] proposed using MapReduce to run machine learning algorithms on multi-core. Dyer et al. [3] also built the statistical machine translation using MapReduce. Besides, Ekanayake et al. [15] applied MapReduce for scientific data analysis. Specifically, they evaluated MapReduce with High Energy Physics data analysis and K-Means clustering. Our work differs from the above ones in that we studied in another domain of applications , supercomputing, on commodity clusters and provided a more detailed study on their performance characteristics.

The prevalent of heterogeneous multi-core systems open opportunities to run MapReduce originally for clusters in a signal machine. Ranger et al. [16] recently provide a MapReduce implementation, namely Phoenix, which runs on multi-core platforms. Their implementation indicates that applications written using MapReduce, are

comparable in performance and scalability to their pthread counterparts. The popularity of MapReduce is also embodied in running MapReduce on other heterogeneous environments, such as on GPUs [17] and Cell [18].

## 7 Conclusion

MapReduce has been prevalent for running data-parallel applications without effort for non-functionality parts. The features mentioned make the programming model popular in various domains like textual retrieval and machine learning. In this paper, we ported and evaluated two typical scientific applications from the SPLASH-2 suite using MapReduce. Based on a detailed study and analysis, we identified that the requirement of frequent data communication in the kind of application results in a huge overhead on network. Based on our experience, we also proposed several potential enhancements on the MapReduce framework to make the model much more suitable for supercomputing applications.

## References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
2. Bialecki, A., Cafarella, M., Cutting, D., O'Malley, O.: Hadoop: a framework for running applications on large clusters built of commodity hardware (2005), `http://lucene.apache.org/hadoop`
3. Dyer, C., Cordova, A., Mont, A., Lin, J.: Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In: Proceedings of the Third Workshop on Statistical Machine Translation at ACL, pp. 199–207 (2008)
4. Elsayed, T., Lin, J., Oard, D.W.: Pairwise document similarity in large collections with mapreduce. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics, pp. 265–268 (2008)
5. Wolfe, J., Haghighi, A., Klein, D.: Fully distributed EM for very large datasets. In: Proceedings of the 25th international conference on Machine learning, pp. 1184–1191. ACM, New York (2008)
6. Bryant, R.: Data-intensive supercomputing: The case for DISC (2007)
7. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proc. ISCA (1995)
8. Singh, J.P., Gupta, A., Levoy, M.: SPLASH: Stanford parallel applications for shared memory. Computer Architecture News 20(1), 5–44 (1994)
9. Lie, G., Clementi, E.: Moleculear-dynamics simulation of liquid water with an ab initio flexible water-water interaction potential. Physical Review A33, 2679–2693 (1986)
10. Matsuoka, O., Clementi, E., Yoshimine, M.: CI study of the water dimer potential suface. Journal of Chemical Physics 64(4), 1351–1361 (1976)
11. Barlett, R., Shavitt, I., Purvis, G.: The quartic force field of $H_2O$ determined by many-body methods that include quadruple excitation effects. Journal of Chemical Physics 71(1), 281–291 (1979)
12. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: A comparison of sorting algorithm for the connection machine CM-2. In: Proc. SPAA (1991)
13. Yang, H., Dasdan, A., Hsiao, R., Parker, D.: Map-reduce-merge: simplified relational data processing on large clusters. In: Proc. SIGMOD (2007)

14. Chu, C., Kim, S., Lin, Y., Yu, Y., Bradski, G., Ng, A., Olukotun, K.: Map-reduce for machine learning on multicore. In: Advances in Neural Information Processing Systems: Proceedings of the 2006 Conference, p. 281. MIT Press, Cambridge (2007)
15. Ekanayake, J., Pallickara, S., Fox, G.: MapReduce for Data Intensive Scientific Analyses. In: IEEE Fourth International Conference on eScience, 2008. eScience 2008, pp. 277–284 (2008)
16. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: Proc. HPCA (2007)
17. He, B., Fang, W., Luo, Q., Govindaraju, N., Wang, T.: Mars: a MapReduce framework on graphics processors. In: Proc. PACT (2008)
18. de Kruijf, M., Sankaralingam, K.: MapReduce for the Cell BE Architecture. University of Wisconsin Computer Sciences Technical Report CS-TR-2007