Evaluating the Performance and Scalability of MapReduce Applications on X10*

Chao Zhang, Chenning Xie, Zhiwei Xiao, and Haibo Chen

Parallel Processing Institute, Fudan University

Abstract. MapReduce has been shown to be a simple and efficient way to harness the massive resources of clusters. Recently, researchers propose using partitioned global address space (PGAS) based language and runtime to ease the programming of large-scale clusters. In this paper, we present an empirical study on the effectiveness of running MapReduce applications on a typical PGAS language runtime called X10. By tuning the performance of two applications on X10 platforms, we successfully eliminate several performance bottlenecks related to I/O processing. We also identify several remaining problems and propose several approaches to remedying them. Our final performance evaluation on a small-scale multicore cluster shows that the MapReduce applications written with X10 notably outperform those in Hadoop in most cases. Detailed analysis reveals that the major performance advantages come from a simplified task management and data storage scheme.

1 Introduction

The continuity of Moore's law in the multicore era provides an increasing number of resources to harness. However, this also creates grand challenges to programmers, as the increasing number of CPU cores requires programmers to understand and write parallel programs. However, previous evidences indicate that parallel programming is notoriously hard. Hence, an ideal scheme would be providing a high-level programming model for average programmers and hiding them from complex issues such as managing parallelism and data distribution using a runtime library.

MapReduce [1], proposed by Google, is a model to program large-scale parallel and distributed systems. Mostly, programmers only need to abstract an application into a Map and a Reduce phases, while letting the underlying runtime manage parallelism and data distribution. Due to its elegance and simplicity, MapReduce has been shown as a simple and efficient way to program many applications.

To provide programmers with more fine-grained control over parallelism and data distribution, yet still maintaining good programmability, *partitioned global address space (PGAS)* [2] based languages and runtime have long been the research focus

* This work was funded by IBM X10 Innovation Faculty Award, China National Natural Science Foundation under grant numbered 61003002, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100, a research grant from Intel, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 46–57, 2011.

and have been developed in various languages such as Unified Parallel C, Fortress and Chapel. X10 [3] is one in the PGAS families that aims at providing high productivity and scalability for parallel programming. It supports a variety of features such as place, activity, clock and finish. X10 has also provided its own constructs to write data-parallel applications such as MapReduce. However, currently, there is little study on the performance and scalability of MapReduce applications written with X10 on clusters.

In this paper, we evaluate two MapReduce applications, WordCount and DistributedSort, on X10 platforms. By analyzing the performance of the two applications, we eliminate several performance bottlenecks related to I/O processing. During our study, we also identify some remaining problems and propose several approaches to remedying them.

We also compare the two applications written with X10 with those in Hadoop. In our evaluation environment, we use a small-scale 7-node cluster, with each node 24 cores and 64 GB memory. The X10 MapReduce applications and the Hadoop benchmarks are both evaluated on this cluster. Our evaluation results show that the two MapReduce applications with X10 notably outperform those in Hadoop. Detailed analysis reveals that the major performance advantages come from simplified task management and data storage schemes.

The rest of the paper is organized as follows. Section 2 presents the background information about X10's main features, its basic grammar and an overview of Hadoop. In section 3, we use X10 to implement two MapReduce applications, and the section 4 discusses the optimization of the two applications. A detailed performance evaluation of the two applications is presented in section 5, as well as a comparison with those of Hadoop's applications. In section 6, we present several optimization suggestions to remedy the remaining problems uncovered based on our study for improvement of performance and scalability of X10 applications. Section 7 discusses the related work and finally we conclude the paper in section 8.

2 Background

This section mainly introduces the fundamentals of X10 languages, as well as its runtime and collecting-finish framework, and the simple overview of the implementation of Hadoop.

2.1 An Overview of X10

X10 is a new language and runtime to program Non-Uniform Cluster Computing (NUCC) [3] systems, with the goal of both good programmability and performance scalability. It supports the *asynchronous partitioned global address space (APGAS)* programming model [2], which provides a global shared address space but partitions the address space into different portions according to system topology. The data is distributed into the partitioned address spaces called Places. Remote accesses to data in a different place are through one-directional communication primitives. To execute a series of instructions, a program declares many activities, which can be either synchronous or asynchronous. The activities can also be spawned on remote places. The

basic structure of X10 is shown in Figure 1. The number of places remains the same during the lifetime of a program, which means no place is dynamically added once a program begins to run. To schedule numbers of activities, the X10 runtime provides a work-stealing algorithm that balances the execution of activities on different workers. These allow control of parallelism, data accesses of non-uniformed memory, reducing of the cost for remote accesses and optimal task scheduling.

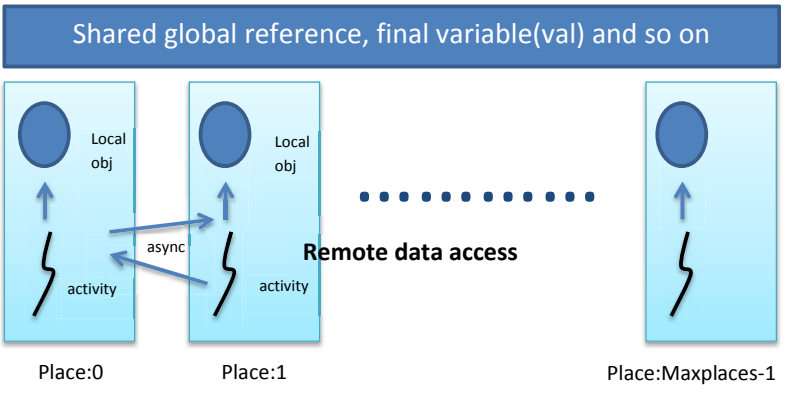

Fig. 1. Basic structure of X10 places, activities and remote data accesses

Many programming models can be subsumed under the APGAS model. For example, the MapReduce programming model can be implemented using the collecting-finish framework in X10. The followings list some important abstraction in X10 [3]:

* An ***at(p) S*** statement assigns a specific place p to execute statement S. The current activity is blocked until S terminates.
* An ***async S*** statement launches an activity and executes statement S without blocking the current activity.
* A ***finish S*** statement is used to block the current activity and wait for all activities in S to terminate.
* ***Collecting-finish*** provides an interface of Reducible[T], of which T can be any reduce unit that is to be merged by the reducer according to the method operator *this()* implemented by user (an example is shown in Figure 2). The method accepts two reduce unit and returns the reduced output in the same format. User also needs to implement an *zero()* method to define the default empty result. Meanwhile, user should also program in the finish (reducer) {} way, which may launch a number of activities and each activity can offer the intermediate value in the format of the reduce unit. The collecting-finish scheduler collects the data from the body of the finish (reducer) {} on each place. This process resembles the Reduce option in MapReduce.

```
val r = new Reducer();
      val result = finish(r){
      //do the data collecting process
      ......
      offer(partResult:T);
}
```

```
class def Reducer extends Reducible[T]{
  def zero(){
    return an instance of T that is empty.
  }
  def operator this(x:T,y:T){
    ..... // do the merge option
  return (z:T);
  }
}
```

Fig. 2. An abstraction frame of collecting-finish

2.2 Hadoop

Hadoop [4] is an open-source implementation of MapReduce on clusters. A typical Hadoop deployment consists of a master and multiple slaves. The master schedules MapReduce jobs, while slaves perform the actual computation. Hadoop stores both input and output files on Hadoop Distributed File System (HDFS). HDFS stores files as blocks and replicates them to multiple nodes. Hadoop could gain block locations from HDFS and thus assign tasks to nodes storing the input data. A MapReduce job is divided into a number of Map and Reduce tasks in Hadoop. The Map task loads data from HDFS, performs the map function and flushes the intermediate data onto the local disk. The Reduce task fetches its portion of the results of all other Map tasks, performs the reduce function and saves the final results back to HDFS.

3 Implementation of Two MapReduce Applications Using X10

In this section, we discuss the design and implementation of the two X10 applications using the MapReduce programming model.

3.1 Execution Overview

Typical MapReduce applications consist of the following phases: split, map, reduce and combine or merge. The two main processing parts are the map and reduce functions. The map function emits a set of $\langle key, value \rangle$ pairs and the reduce function processes all the pairs to get the sum of values according to the key.

The two X10 MapReduce applications are written in this programming style. Figure 3 shows the overall workflow of the applications. During the program's lifetime, it goes through the following steps:

1. The splitter splits the input data into N pieces according to the number of places. The size of each piece of data is evenly divided and may be dynamically adjusted.
2. Each split of the data is assigned to a place, in which there are some pre-allocated workers. The input data is partitioned again according to the number of workers as done in step 1.
3. Once a worker gets the input data from the splitter, it scans the content and begins the map work. It produces a set of $\langle key, value \rangle$ pairs and passes them to the reducer.

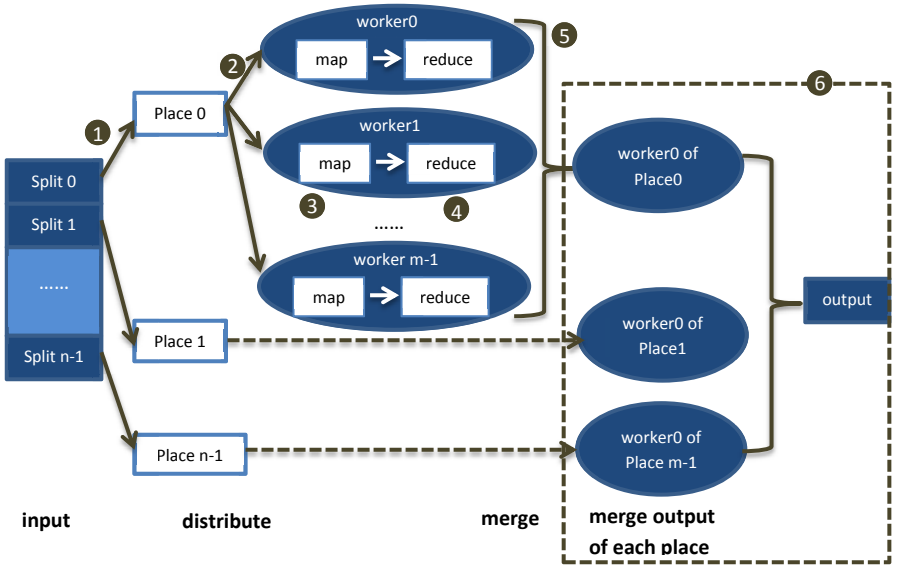

Fig. 3. Execution overflow of the two X10 MapReduce applications

4. The reduce worker gets the input pairs and aggregates the value according to the keys. After the entire map and reduce tasks complete, there are M*N pieces of output data, where M is the number of works and N is the number of places.
5. The merge work is divided into two steps. Firstly, in each place, there are M pieces of output data from the reduce worker. M/2 workers are spawned to do the merge work in parallel. At the end of this step, the intermediate value containing all the data processed by this place is ready.
6. This step collects data from all N places. After all the steps successfully completed, we can get the final result of a list of $\langle key, values \rangle$ pairs in place 0.

3.2 Data Structures

For simplicity, we use arrays to store the data. Every map and reduce worker holds an array of the $\langle key, value \rangle$ pairs. Thus, the computation can be done in parallel without synchronization since the arrays are independent. The keys are kept in the alphabetical order for simple management and binary search is used to scan arrays.

3.3 Implementing WordCount and DistributedSort

WordCount: WordCount counts the number of occurrences for each word in files. First, Place0 splits the input data into N pieces according to the number of places as shown in Figure 4. Then, it sends the split information containing the start position and the length of the data to each place. Once a place gets the information, it reads the file from

```
1   //define the reducer for collecting-finish
2   r = new Reducer();
3   result = finish(r){
4   for(p in Place.places()){
5     async at(p){
6       for(t in threads){
7         intermediate =
8            map(data_of_this_thread);
9         offer(intermediate);
10      }
11    }
12  }
```

```
1   def map(arg){
2     read(); // split data into set of <word,1>
3     reduce();//combine value of same word
4   }
5
6   def class Reducer extends Reducible[T]{
7     def apply(x:T,y:T){
8       merge_offered_intermediate(x,y);
9     }
10  }
```

Fig. 4. Splitter, map, reduce and collecting-finish of WordCount

local disks and spawns some workers to do the map and reduce operations (Figure 4(a), Line 7). After the completion of map and reduce phases, merge workers are invoked to produce the intermediate results (Figure 4(a), Line 9). Finally, the place uses the collecting-finish mode to offer the result to the master (Figure 4(a), Line 9) and the master schedules some places to do the reduce work and gets the final output.

DistributedSort: DistributedSort sorts a set of 100-byte records of which 10 bytes are used as keys. Different from WordCount, the keys of DistributedSort are all unique, which means DistributedSort does not reduce the size of input data, and it has to transfer a lot of intermediate records. Therefore, it is challenging for DistributedSort to collect data from all the places when the input data is large. When transferring data from one place to another, we make use of the RemoteArray object (Figure 5, Line 2). However, instances of user-defined types cannot be transferred by methods of RemoteArray object so far. We need to serialize them (Figure 5, Line 9) before offering them to the final reducer who will deserialize them before merging (Figure 5, Line 14).

```
1  store = new Array[Array[T]](place_num);
2  for(each place) remote_array(place_id) = new RemoteArray(store(place_id));
3  finish{
4    for(p in Place.places())
5      async at(p){
6        array = new Array[T](thread_num);
7        for(t in threads) read_and_sort(array(t));
8        output = merge_all_array(array);
9        stdoutput[char] = serialize(output);
10       Array.asyncCopy(stdoutput,remote_array(place_id));//offer data
11     }
12 }
13 data = deserialize(store);
14 result = do_merge_of(data);
```

Fig. 5. Serialization to transfer data in DistributedSort

4 Optimization

In this section, we present some optimizations to make the two applications more efficient.

4.1 Using C++ Native Calls to Process I/O

To our knowledge of the X10 standard library, the I/O performance decreases when processing large files. For C or C++ programming in Linux environment, the general approach to this problem is using the mmap() system call to map the file from disk directly to the user space address. However, there is no similar interface in X10 standard library.

Fortunately, X10 provides interfaces to invoke C++ or Java methods. In this way, we can easily call mmap() or similar methods to process the large files with high efficiency.

Figure 6 shows the processing time of reading files of 10MB, 20MB and 40MB size. The gap is significant between the X10 standard library and mmap(). The X10 standard library spends tens of seconds to read the files into the memory while mmap() only takes less than 1 second. Therefore, the interfaces to invoke the C++ native calls could be an alternative to deal with large files. We build a small library to encapsulate the mmap() system call for simplifying the invocation of C++ native calls and reusing the codes.

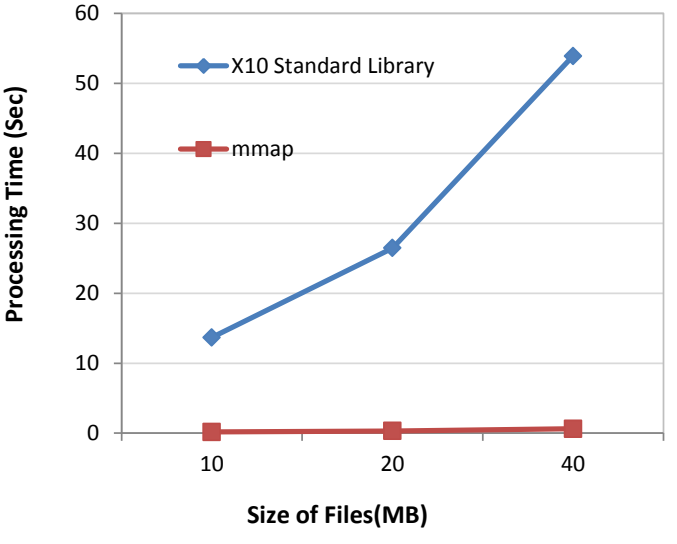

Fig. 6. Comparison of processing time between X10 standard library and mmap

4.2 Improving the Collecting-Finish Framework

We improve two phases of the default collecting-finish framework of X10 as following:

Merge Phase: First, we add filters to local data transfer, to avoid repeated copy of intermediate data on the same place. Second, we rewrite the reduce process to merge every two intermediate key/value pairs concurrently and recursively.

Sharing Data Among Places: In X10, remote data accesses requires some indirect mechanism. The standard collecting-finish framework uses deep copy to transfer data. Specifically, using *async at(p)* statement will cause deep copy, in which both primitive types and user-defined classes will be transferred.

However, when input data become large, deep copy will be very slow because the objects are transferred one by one and it may cause a descriptor error of sockets on Linux platform. Thus, we use the Array.asyncCopy method of X10 standard library to rewrite the data transferring process since Array.asyncCopy can copy a large chunk of memory at a time and provides an efficient data copy. However, it only supports arrays with default types. For user-defined types of data, it can only copy the reference of the object. In order to transfer data as a set of $\langle key, value \rangle$, we implement a simple serialization method to format the intermediate data to a byte array.

For WordCount, we simply use the default collecting-finish mechanism. However, in DistributedSort, which needs a large amount of data copy, we use the optimized implementation of collecting-finish to transfer large data.

Figure 7 shows the processing time of optimized collecting-finish framework and the original one. The suffix of "-opt" stands for our optimized version of collecting-finish. The suffix of "-std" stands for the standard library of X10. From the figure, we can see that the optimized implementation have a notable performance advantage over the original one. The speedup mainly comes from the merge phase, which involves lots of data transferring.

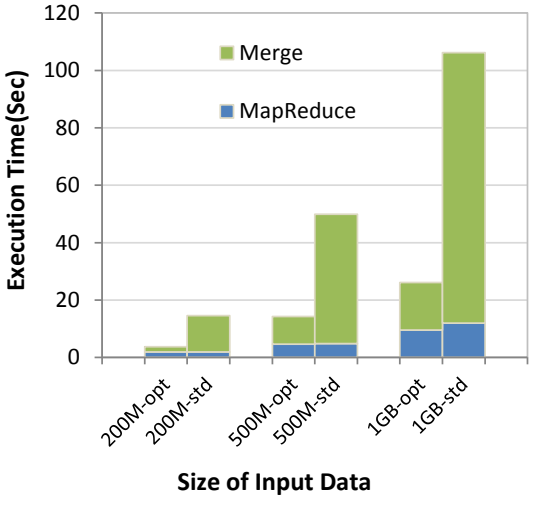

Fig. 7. Comparison of execution time between standard collecting-finish and the optimized one on a single node with 24 workers

5 Evaluation

In this section, we evaluate the performance of WordCount and DistributedSort applications implemented with X10 and compare with their Hadoop version.

5.1 Experiment Setup

The experiments are conducted on a small-scale cluster with 7 nodes. Each node has two AMD Opteron 12-core processors, 64 GB main memory. Every node is connected

to the same switch through a 1GB Ethernet link. The operating system is Debian 6.0 and the version of Hadoop is 0.20.1. We use X10 of version 2.1.1. The input size of the two applications varies from 200MB to 1GB. We evaluate the two applications both on a single node and on the cluster.

5.2 Overall Performance on a Single Node with Performance Breakdown

On a single machine, we evaluate the execution time of the two applications with different sizes of input data and different number of workers.

The execution time breakdowns of WordCount in Figure 8 shows the time spent in different phase of the WordCount application running in a single place. From the figure, we can see that the WordCount application has a good scalability. However, when the number of workers exceeds 9, the overhead of concurrency such as work scheduling and data access collision affect the continued improving of performance.

The MapReduce phase takes up most of the time. Currently, we maintain a sorted array to store the intermediate records generated by the map function. Each element contains a unique word and the number of its occurrences. To insert a new word, WordCount should search the word's position and then move all the words after that position backwards, to make room for this new word. The time for searching and moving depends on the number of unique words in the input file.

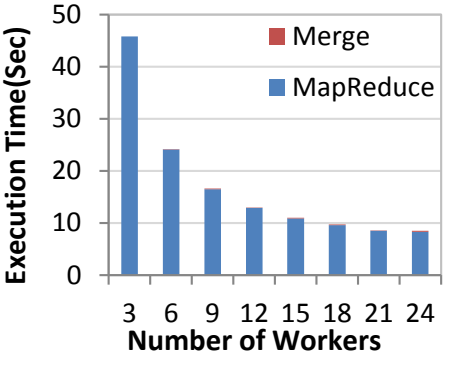

Fig. 8. Execution time breakdown of WordCount on a single node with different number of workers, using 1GB input data

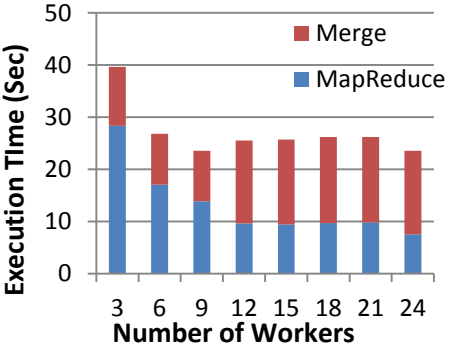

Fig. 9. Execution time breakdown of DistributedSort on a single node with different number of workers, using 1GB of input data

Figure 9 shows the performance trend according to the number of workers in DistributedSort. We can see when the number of workers increases to 9 it reaches quite a good performance. Then, until the worker numbers get to 24, the performance increases very little. This is because map tasks are parallelized well along with the increasing of the worker numbers. However, the reduce tasks, in fact, does not scale when the number of workers increases. By contrast, when the number of workers exceeds the required number of reduce tasks, an excessive amount of workers may cause frequent job stealing, which may slow down the execution of the program. Though the original launched

activity number of first phase is 24, there only needs 12 reducers to merge them, and during merging recursively, number of reducer decreases. Hence, when more workers are involved, the execution time will not necessary decrease. On the other hand, when the number of workers keeps increasing, the reduce time will not increase again, as shown in Figure 9. This helps X10 to keep its performance reasonably efficient without careful consideration on the worker number configuration.

5.3 Overall Performance on the Cluster and Comparison to Hadoop

On the cluster, we evaluate both the X10 MapReduce applications and those in Hadoop under the same environment. In X10, we use 7 places and there are 24 workers in each place. The Hadoop is configured with 7 nodes (one for master node and 6 for slave nodes), 144 map tasks (6 slaves and 24 tasks per slave) and 1 reduce task to merge all the output as in X10.

Figure 10 shows that the performance of WordCount in X10 significantly outperforms its Hadoop counterpart. The speedup mainly comes from the simpler task scheduling and less I/O for fault tolerance in X10. In Hadoop, a MapReduce job should be firstly submitted to the JobTracker running on the master node. Then, JobTracker further splits the job into map tasks and reduce tasks, and schedules them to slave nodes with available computing slots. In X10, when a job comes, it is just split and assigned in average to all workers. What's more, in Hadoop, the intermediate output produced by map tasks should be flushed to local disks before sending to reduce tasks for fault tolerance, which causes lots of I/O. Our X10 MapReduce applications take advantage of PGAS to share the data among all places so that they have reduce time less than the Hadoop ones.

Figure 11 shows comparison of DistributedSort between X10 and Hadoop on cluster. We can find that X10 still have efficiency advantages on concurrent computing, especially when data is small. The parallel primitives such as finish() statement detection, async() statement logic control, do not incurs too much overhead.

However, since X10 currently has not completed developing on its remote data transfer, we just implement a simple serializing and deserializing prototype for DistributedSort. So as the input data increases, the overhead of both serialization on map phase and deserialization on reduce phase grows quite quickly, which finally affects the whole computing efficiency of this application. We will try to improve the data transfer efficiency as future work.

6 Further Optimization Opportunities

Large Data Copy among Places: When copying a large amount of data of user-defined types, we suggest adding the serialization and deserialization of an array based on the current implementation of Array copy. We can serialize or even compress the data to be copied to the remote place and get a byte array that can be transferred asynchronous and fast by Array.asyncCopy. The remote place can access the data once the data is deserialized. This will be our future work. Meanwhile, since X10 is using socket-based data transfer, it may be beneficial to optimize the socket transfer structure of X10.

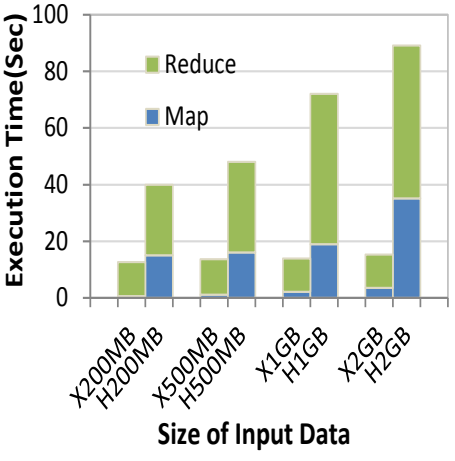

Fig. 10. Execution time of X10 WordCount on the cluster, as well as the time of one in Hadoop. The prefix 'X' represents X10, and 'H' represents Hadoop.

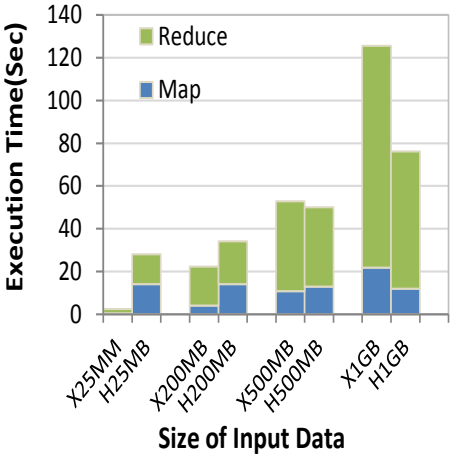

Fig. 11. Execution time of X10 DistributedSort on the cluster, comparing to the one in Hadoop. The prefix 'X' represents X10, and 'H' represents Hadoop.

Collecting-Finish: In the process of collecting-finish, there is only one or two workers involved in the reduce work while others are waiting for I/O, which is a waste of resources. Hence, other idle workers can be used to either for prefetching data or do data compression.

7 Related Work

X10 [3] is a new language under development. In recent years, much research has been done on X10 including optimizing X10 runtime, applying X10 in a variety of fields, etc.

A detailed tutorial about how to write X10 applications on modern architecture was provided by Sarkar et al. [5] and Murthy [6]. Saraswat et al. [7] developed a lifeline-based global load-balancing algorithm to extend the efficiency of work-stealing. Agarwal et al. [8] presented a framework that could statically establish place locality in X10 and offered an algorithm to eliminate runtime checks based on it. Agarwal et al. [9] proposed a lock-free scheduling mechanism for X10 with bounded resources.

The task creation and its termination detection in X10 will cause some overhead [10]. J. Zhao et al designed an efficient framework to eliminate it. Rajkishore Barik [11] proposed an optimized access way to the shared memory in parallel program both from low-level and high-level. The high-level optimizations are approaches to Side-Effect analysis and May-Happen-in-Parallel analysis in concurrent programs. The low-level ones mainly introduced the optimizations of compilation. The research of X10 also covers a Hierarchical Place Trees model which is as a portable abstraction for task parallelism and data movement [12], a compiling optimization of work-stealing in X10 [13], an optimized compiler and runtime of X10 improving the efficiency of executing instructions among places [14] and so on.

8 Conclusion and Future Work

MapReduce has been shown as a simple and efficient way to harness the massive resources of clusters.In this paper, we evaluated two MapReduce applications on the X10 platform, which was a member of languages based on partitioned global address space (PGAS). Though it is not an apple-to-apple comparison for X10 with Hadoop (as X10 currently does not support fault tolerance), our result showed that X10 is a powerful programming language to run MapReduce-style applications on clusters. By our study and analysis of X10 and its runtime, we eliminated some performance bottlenecks such as I/O processing. We also identified some remaining optimization opportunities. As future work, we are interested in improving the data transferring between places as well as the worker scheduling on multicore.

References

1. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51, 107–113 (2008)
2. Saraswat, V., Almasi, G., Bikshandi, G., Cascaval, C., Cunningham, D., Grove, D., Kodali, S., Peshansky, I., Tardieu, O.: The asynchronous partitioned global address space model. In: Proceedings of Workshop on Advances in Message Passing (2010)
3. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proc. OOPLSA, pp. 519–538 (2005)
4. Bialecki, A., Cafarella, M., Cutting, D., Omalley, O.: Hadoop: a framework for running applications on large clusters built of commodity hardware, `http://lucene.apache.org/hadoop`
5. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: concurrent programming for modern architectures. In: Proc. PPoPP, pp. 271–271 (2007)
6. Murthy, P.: Parallel computing with x10. In: Proceedings of the 1st International Workshop on Multicore Software Engineering, pp. 5–6 (2008)
7. Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D., Krishnamoorthy, S.: Lifeline-based global load balancing. In: Proc. PPoPP, pp. 201–212 (2011)
8. Agarwal, S., Barik, R., Nandivada, V.K., Shyamasundar, K., Varma, P.: Static detection of place locality and elimination of runtime checks. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 53–77. Springer, Heidelberg (2008)
9. Agarwal, S., Barik, R., Bonachea, D., Sarkar, V., Shyamasundar, R.K., Yelick, K.: Deadlock-free scheduling of x10 computations with bounded resources. In: Proc. SPAA, pp. 229–240 (2007)
10. Zhao, J., Shirako, J., Nandivada, V.K., Sarkar, V.: Reducing task creation and termination overhead in explicitly parallel programs. In: Proc. PACT, pp. 169–180 (2010)
11. Barik, R.: Efficient optimization of memory accesses in parallel programs (2009), `www.cs.rice.edu/~vsarkar/PDF/rajbarik_thesis.pdf`
12. Yan, Y., Zhao, J., Guo, Y., Sarkar, V.: Hierarchical place trees: A portable abstraction for task parallelism and data movement. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 172–187. Springer, Heidelberg (2010)
13. Raman, R.: Compiler support for work-stealing parallel runtime systems. M.S. thesis, Department of Computer Science, Rice University (2009)
14. Bikshandi, G., Castanos, J.G., Kodali, S.B., Nandivada, V.K., Peshansky, I., Saraswat, V.A., Sur, S., Varma, P., Wen, T.: Efficient, portable implementation of asynchronous multi-place programs. In: Proc. PPoPP, pp. 271–282 (2009)