X10-Enabled MapReduce

Han Dong, Shujia Zhou
University of Maryland Baltimore County
han6@umbc.edu

David Grove
IBM
groved@ibm.us.com

Abstract

The MapReduce framework has become a popular and powerful tool to process large datasets in parallel over a cluster of computing nodes. Currently, there are many flavors of implementations of MapReduce, among which the most popular is the Hadoop implementation in Java. However, these implementations either rely on third-party file systems for across-computer-node communication or are difficult to implement with socket programming or communication libraries such as MPI. To address these challenges, we investigated utilizing the X10 language to implement MapReduce and tested it with the word-count use case. The key performance factor in implementing MapReduce is data moving across different computer nodes. Since X10 has built-in functions for across-node communication such as distributed arrays, a major challenge with MapReduce implementations is easily solved. We tested two main implementations: the first utilizes the HashMap data structure and the second a Rail with elements consisting of a string and integer pair. The performance of these two implementations are analyzed and discussed. In addition, their performances are also compared with Hadoop's.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – concurrent programming structures, data types and structures, frameworks.

General Terms Performance, Design, Experimentation, Languages

Keywords PGAS; MapReduce; across-node-communication

1. Introduction

The MapReduce programming model was introduced by Google as an efficient way to support distributed computing and processing large datasets on a large cluster of computing nodes [1]. The model relies heavily on functional programming functions, map and reduce. The model is mostly embarrassingly parallel since it allows the independent execution of key-value pairs with the map and reduce functions on each node. Only across-node communication is needed when congregate and redistribute intermediate key-value pairs created by map to among nodes for reduce. A notable MapReduce implementation is done in Hadoop, with is written in Java. The Hadoop implementation utilizes its own file system, Hadoop Distributed File System (HDFS) for storing and sorting data across compute nodes. It consists of a map function for data distribution and creates key-value pairs, a combiner function that merges the intermediate maps, then a partitioner that shuffles and sorts the data across different computing nodes [5]. This design is common among other flavors of MapReduce implementations. However, a key performance factor in this design is the cost of shuffling data across different nodes and the reliance on a third party file system to handle data storage. We chose the X10 language to address these key challenges with the implementation of MapReduce and demonstrate it with a word-count use case. The X10 language was chosen since it has benefits such as easy-across-node communication built into the language, which will also help to avoid the use of third party file systems so as to increase portability and provide opportunities for holistic optimization. In our design, we utilized three different implementations. The first implementation utilized a HashMap data structure to store the key-value pairs. In the second implementation, we built a simple Linked List data structure in X10 and used that for storing data. Our final implementation utilizes more inbuilt X10 data structures called Rail to store Pair's that store key, value pairs in a primitive String, Integer format. In this paper, we will investigate these three approaches and discuss the pros and cons of their implementations.

2. Design

The X10 language is the latest addition into the partitioned global address space (PGAS) model. It further extends PGAS with Asynchronous PGAS [3]. Key concepts behind the X10 APGAS model is the advent of *places* and *activities*. A *place* is defined as a "collection of resident objects and activities"; it can also be described purely as a computation unit with threads and a local heap memory. An activity is often described as just a lightweight thread [2]. The X10 language realizes the APGAS model through remote references, global address spaces, and inter- and intra-place operations [4]. The APGAS model allows *activities* to access remote objects at other *places* through functions such as the *at(Place)* place shifting operation. However, this only works through immutable objects/data structures; it is not possible to access a mutable state in a remote *place* [4]. In our three implementations, we intend to minimize the cost of shuffling by taking advantage of X10's APGAS model.

In our X10 enabled MapReduce, we introduce three main functions:

1. Map – this handles the parsing and storing of words into the data structures local to each *place*.
2. Shuffle/Merge – this handles the distribution of data from local to remote *places*.
3. Sort – this handles the final sorting of the data structure and removes duplicates.

The first implementation utilizes the inbuilt X10 data structure HashMap as shown in Figure 1. In this implementation, each *place* consists of its own HashMap data structure which stores each word and its occurrence in a String-Integer: key- value pair. The indexes are hashed with the key. We utilized two methods to shuffle data: the first method copies and stores data in a single

place; the second method also stores the data in a single *place*, however it utilizes the parallel reduction paradigm [6] to achieve this. These two methods are possible because of X10's APGAS model. The design allows simple copying of data from one place to another place with methods that are built into the language. Since HashMap uses a hash function to store the key-value pairs, it is very difficult to implement a sorting function. In this respect, the sorting is done through the copying of the data from the final merged HashMap into an ArrayList, which is then sorted.

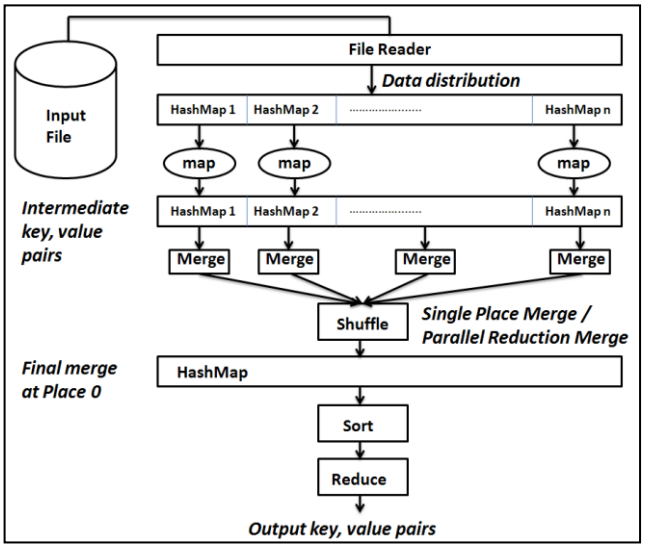

Figure 1. The architecture of MapRduce implementation based on HashMap

The second implementation utilizes a GrowableRail data structure where each element is a pair of String and LinkedList<Int> (see Figure 2). The GrowableRail can be transferred into a ValRail, which is a 1-dimensional sequence of immutable types. The ValRail data structure has built in functions to transfer data from one *place* to another. This is an example of the benefit of X10's library in across computing node communication compared to other serial languages such as C and Java. The X10 language uses MPI or other communication protocols and libraries to do across computing node communication. We wrote a merge sort algorithm to sort the final result after shuffling.

The motivation behind using a linked list to keep track of word occurrences is due to the cost of shuffling. We initially believed that the X10 libraries for across computing node communication would include remote memory accesses, thus, reducing redundant data copying like the previous HashMap example. The key concept here during data shuffling is to copy the head of the linked list to another computing node, which in turn will enable us to easily walk through the rest of the data in the linked list without the need to physically copy the entire structure. However, we discovered later that trying to take advantage of remote memory accesses in X10 is not as easy as we expected. The challenges will be discussed in a later section.

The third implementation utilizes a GrowableRail data structure where an element is a pair of primitive String and Integer. This was chosen due to the difficulties of shuffling in the prior implementation. Since the data structure only contains primitive values now, the difficulty with using ValRail to transfer data to different computing nodes is easily solved. A merge sort algorithm is used to sort the final result. In this design, data distribution to remote *places* is based on a pre-computed data table. Each place is allocated a group of alphabets that the starting character of the word should be copied to. For example, the key-value pairs with the starting characters of the word from A to G are copied to Place 0, H to K are copied to Place 1, etc. The load balancing is not optimal since it is not done dynamically in this implementation. We plan to develop a dynamic load balance scheme which will collect the distribution of key-value pairs during the map operation, analyze communication cost among compute nodes, and optimally dispatch the key-value pairs to the destined compute nodes.

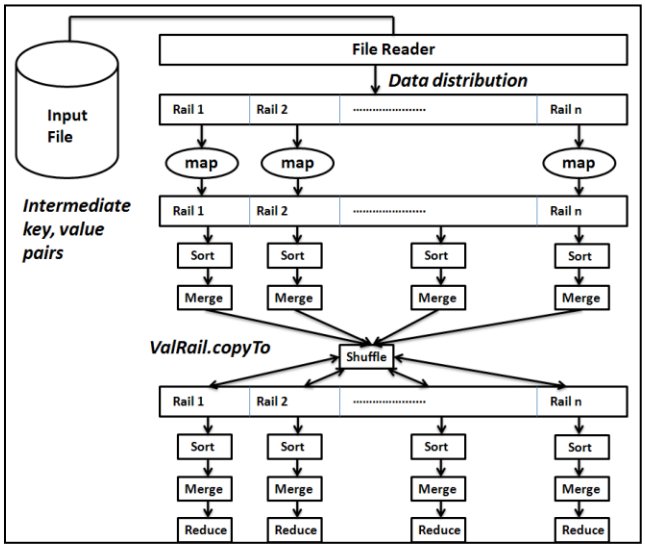

Figure 2. The architecture of MapReduce implementation based on Linked List or Primitive String, Integer.

3. Implementation

In all three implementations, a data structure is allocated to each *place* with the following function:

```
val n = PlaceLocalHandle.make[DataStructure]
(Dist.makeUnique(), ()=>new DataStructure());
```

In the X10 MapReduce implementation, the key performance factor is the time spent in shuffling data across different computing nodes. Specifically in the HashMap implementation, there are two merging functions: the single place merge (Figure 3) and the parallel reduction merge (Figure 4).

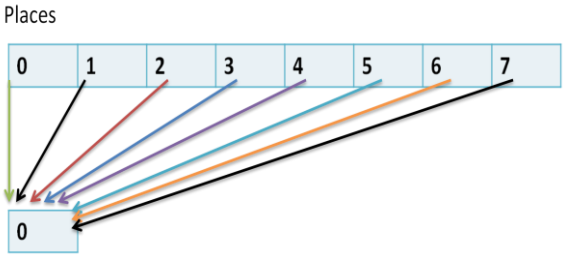

Figure 3. Illustration of single place merge function.

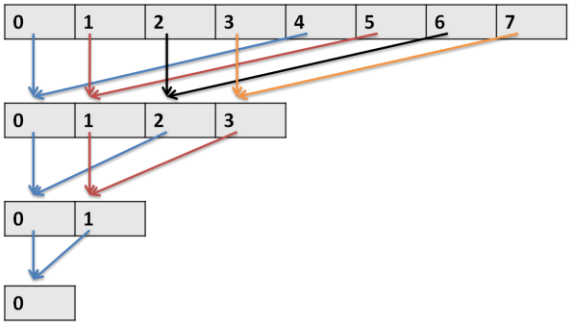

Figure 4. Illustration of parallel reduction merge function.

The single place merge is a simple serial merging function where the data of each HashMap is copied into a single designated *place*.

3.1 Example of single place merge where data from places 1 to n is copied to the HashMap data structure at place 0:

```
for((i) in 1..numPlaces)
{
    async(Place.places(i))
    {
        for(e in hmaps().entries())
        {
            async(Place.place(0))
            {
                hmaps().insert(key, value);
            }
        }
    }
}
```

The parallel reduction merge is a recursive function that merges data structures in parallel with each *activity* spawned handling the merging of two different places. The motivation behind this implementation is that this sorting is naturally parallel and would perform better than the serial single place merge.

3.2 Example of parallel reduction merge where data structures from different places are merged in parallel.

```
val c = nPlaces/2;

for((i) in (c..nPlaces)
{
    async(Place.places(i))
    {
        val j = here.id – c;
        for(e in hmaps().entries())
        {
            async(Place.places(j))
            {
                hmaps().insert(key, value);
            }
        }
    }
}

parallelMerge(hmaps,(nPlaces/2));
```

The second implementation utilizes a GrowableRail data structure, which has simple built in functions for data transfer across remote *places*.

3.3 Example of ValRail copyTo function.

```
for(….)
{
    at(Place.places(..))
    {
        val r = n().gRail.toValRail();
        at(Place.places(..))
        {
            r.copyTo(src,n().rail,rail,num);
        }
    }
}
```

The copyTo function requires the specific boundary of the data structure to copy to. This can be rather complicated if the load balancing of the data distribution was dynamically computed. However, we pre-computed the boundary data, thus making the coding less complex.

The third implementation is very similar to the linked list implementation. The GrowableRail utilizes a pair of String, Integer for each element instead of a pair of String, LinkedList<Int>. The shuffling of data in this implementation is the same as the example code shown in 3.3.

4. Results

To measure the performance of the X10 MapReduce implementations, we tested them on one cluster of four blades in the IBM Watson Lab which is running X10-2.0.4. Each blade has two Quadcore AMD processors (that is, 8 cores in a blade.). Four nodes are connected over a 10Gb Ethernet in the same blade center, so they are on the same internal Ethernet switch and should have fairly good connectivity. We utilized two test files: one was 4 MB (bible) [7] while the other was 50 MB (WordNet) [8]. The word distributions in the two files were statistically uniform since they were intended to be commercially used. We ran the tests using 1 place, 2 places, 4 places, and 8 places, respectively for scalability up to 8 cores. In all three following implementations we used the –O and –NO_CHECKS compiler flags to compile the codes.

We ran the initial test on one blade for the baseline case (Figure 5). Although this was run on only one blade, we can already see that the performance of the parallel reduction merge is worse than that of the others. We do not expect it to improve since the communication cost is greater when more nodes (places) are used due to the transportation of data through sockets.

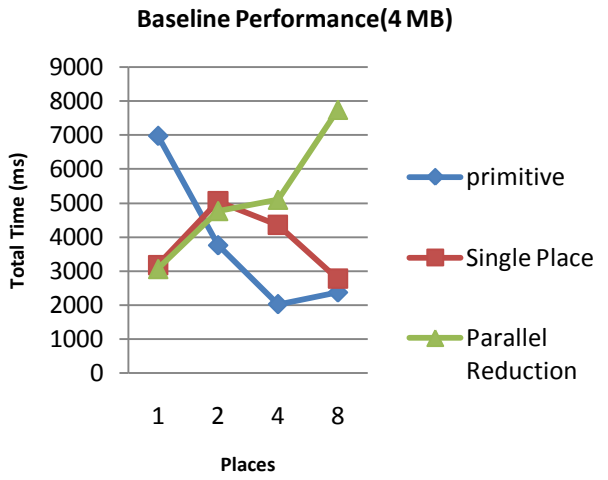

Figure 5. The performance of X10-enabled MapReduce on one node.

The results of processing a 4 MB file on four blades are shown in Figure 6. It can be seen that both the primitive and single place implementations are scalable to eight *places*. This could be the benefit of using X10's APGAS model to transport data across remote *places*. A subsequent graph of the shuffling time is shown in Figure 7.

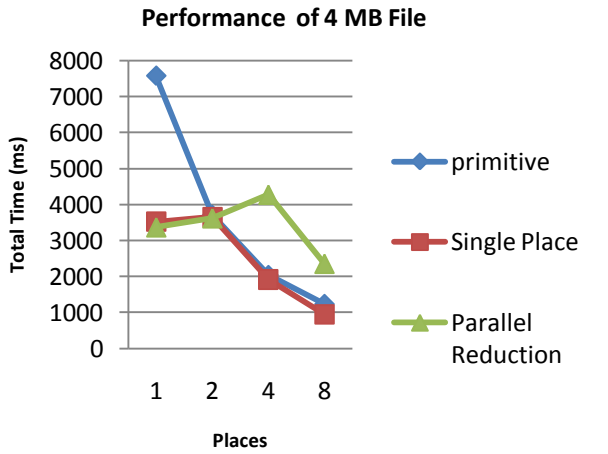

Figure 6. The performance of X10-enabled MapReduce on 4 nodes with a 4 MB test file.

However, for the parallel reduction case, the performance could be explained by the fact that it is a recursive function, which is already naturally slow.

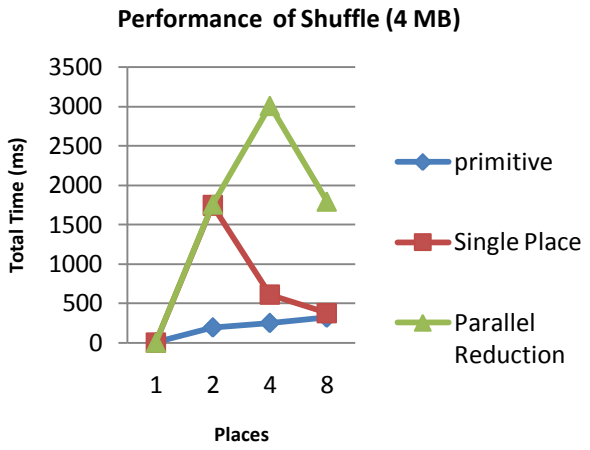

Figure 7. The performance of shuffle with 4MB test file.

As the graph in Figure 7 indicates, the shuffling time for the parallel reduction is the longest out of all three implementations while the primitive implementation performed the best. We can also observe that the cost of shuffling is a major factor in the overall performance of an X10- enabled MapReduce. The primitive implementation with the best scalability is also an indication that it had the best performance in terms of data transportation.

Our next tests were to process a 50 MB file. The results of both the overall time and the shuffling time can be seen in Figures 8 and 9, respectively.

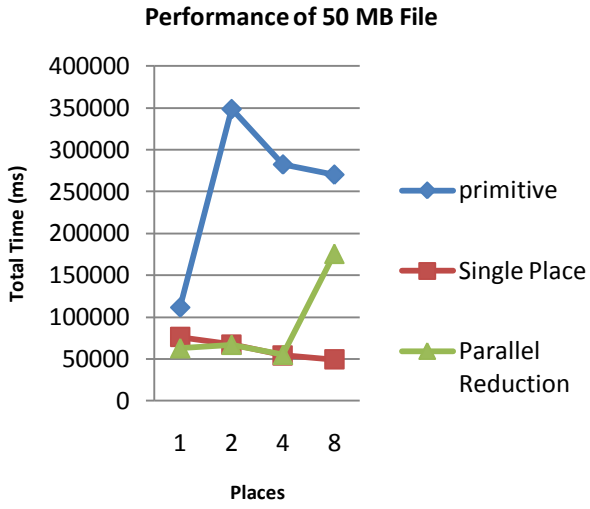

Figure 8. The performance of X10-enabled MapReduce on 4 nodes with a 50 MB test file.

Based on the results, we can see that the primitive implementation actually performed the worst. With a larger file, the HashMap single place merge implementation was still scalable to eight *places*.

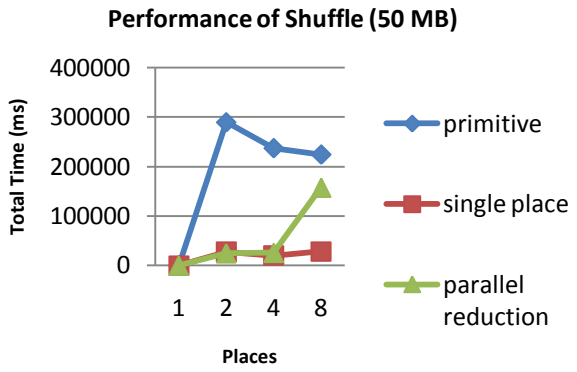

Figure 9. The performance of shuffle with 50 MB test file.

Based on the results in Figure 9, the shuffle operation can be seen to take up the bulk of the processing time. In our analysis, the copying of data by utilizing the *at(Place)* place shifting operator seems to perform better than the *ValRail.copyTo* operator. This is primarily because we were not able to access a mutable object from a remote *place*. In order to utilize the copyTo function, we had to convert the GrowableRail into a ValRail and preprocess the borders of the ValRail to specify the ranges to be copied; this resulted in very complex code, redundant data copying and also a waste of memory (Example 3.3). Whereas the HashMap implementation had no redundant data copying, each data was appended to a pre-existing HashMap (Example 3.1).

We were not able to successfully test the linked list implementation since it was not possible to pass the head node of the list into another place then walk through the rest of the nodes in the list; in other words, it was not possible to do remote memory accesses unless the physical data was copied. This meant that in order to have a working linked list implementation, it was required to have each node in the linked list to be copied over along with the head node. This implementation not only introduces redundant data copying, but its implementation would be similar to HashMap.

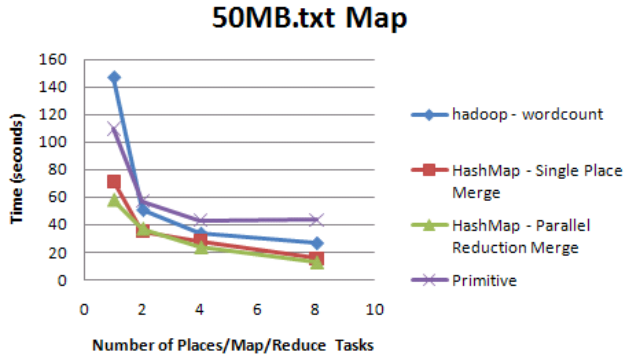

Figure 10. Performance of Map operation between Hadoop and X10 implementation.

We were also able to run some tests on a cluster of blades running Hadoop 0.20.0 to test X10's performance against it. The Hadoop tests were run on a cluster of IBM PowerPC blades connected over a 1Gb Ethernet, each blade with a Dualcore 2.2 Ghz processor chip. We were mainly concerned with the performance of the Map and Reduce tasks. To do a fair comparison between the two implementations, we mapped the number of tasks utilized in the hadoop-example.jar wordcount implementation against the number of *Places* used in X10. We compared the performance of the Map tasks in Hadoop against the Map tasks in X10. In addition, we compared the Reduce tasks in Hadoop against the Shuffle and Sort tasks in X10. The results of the tests are shown in Figure 10 and Figure 11. Based on the results, we can see that X10's performance can be comparable to Hadoop's.

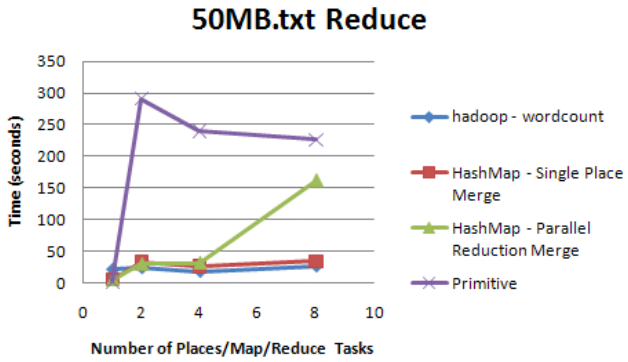

Figure 11. Performance of Reduce operation between Hadoop and X10 implementation.

However, there are a few key points that should be pointed out in terms of the performance. Firstly, Hadoop's HDFS greatly reduces the processing time for large input files. Secondly, Hadoop is designed to be scalable to many clusters of computing nodes while maintaining the same performance level on each node. Thirdly, the cluster of computing nodes for running Hadoop only had 1 Gb Ethernet connections while the IBM compute nodes had 10 Gb Ethernet connections. This would mean that the Hadoop Reduce tasks were at a disadvantage compared to the X10 Shuffle operations. Our tests with X10 indicate that the performance of Shuffle and Sort degrades after 8 *Places*. This can be attributed to shuffling of data among different *Places*. However, adaptive load distribution seems to alleviate this degradation [9].

5. Summary

We have investigated the X10 implementation of MapReduce with three different methods and indentified both the benefits and difficulty of using X10 to implement data shuffling. Our tests show that the X10-enabled MapReduce implementation has a comparable performance with Hadoop's. While the X10 libraries provide very robust functions for easy transportation of data, the lack of remote memory accesses makes the shuffling of data across places time-consuming. Although the APGAS model provides a very powerful way to do across node computing, it could provide better features such as the ability to build a mutable data structure in one place and transport its contents to a data structure in another place.

Acknowledgments

This project is partially supported with the IBM X10 2009 Award.

References

[1] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[2] Saraswat V, Bloom B. Report on the Programming Language X10 Version 2.0.4. June 10, 2010 <http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf>

[3] Ebcioglu K, Saraswat V, Sarkar V. X10: an Experimental Language for High Productivity Programming of Scalable Systems. P-PHEC workshop, HPCA 2005.

[4] Charles P, Donawa C, Ebcioglu K, Grothoff C, Kielstra A, v. Praun C, Saraswat V, Sarkar V. X10: An Object-Oriented Approach to Non-Uniform Cluster Comptuting. OOPSLA 2005

[5] Hadoop MapReduce. <http://hadoop.apache.org/mapreduce/>

[6] Harris M. Optimizing Parallel Reduction in Cuda. <http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf>

[7] Williams F. Smyth. July 14, 2010. <http://www.cas.mcmaster.ca/~bill/strings/english/bible>

[8] Miller, George A. "WordNet - About Us." WordNet. Princeton University. 2009. http://wordnet.princeton.edu

[9] Shujia Zhou and Congchong Liu, private communication