# MapReduce Programming Model for .NET-based Distributed Computing

*Chao Jin and Rajkumar Buyya*

Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
Email: {chaojin, raj}@csse.unimelb.edu.au

## Abstract

*Recently many data center scale of computer systems are built in order to meet the high storage and processing demands of data-intensive and compute-intensive applications. MapReduce is one of the most popular programming models designed to support the development of such applications. It is initially proposed by Google for simplifying the development of large scale web search applications in data centers and has been proposed to form the basis of a "data center computer". This technical report presents a realization of MapReduce for .NET-based data centers, including the programming model and runtime system. The design and implementation of MapReduce.NET are described and its performance evaluation is presented.*

## 1. Introduction

Recently several organisations are building data center scale of computer systems to meet the increasing demands of high storage and processing requirements of data-intensive and compute-intensive applications. On the industry front, companies such as Google and its competitors have constructed large scale data centers to provide stable web search services with high quality of response time and availability. Although Google does not disclose the size of their server infrastructure, it is reported to be around 450,000 to several million commodity machines dispersed across about 25 data centers [10]. Microsoft also presents the result of a research project, called Dryad [11], using over 1,800 processors to perform computations accessing up to 10 terabytes of data.

On the other side, many scientific research works increasingly rely on large scale data sets and powerful processing ability provided by super computer systems, commonly referred to e-Science [17].

The high demanding requirements on data centers are also reflected by the increasing popularity of cloud computing [3][13]. With cloud, IT-related capabilities can be provided as service, which is accessible through Internet or World Wide Web. Representative systems for cloud computing include Google App Engine, and Amazon Elastic Compute Cloud (EC2). Google App Engine provides a service to web developers for accessing the infrastructure of Google. Users do not need to maintain physical servers, while the infrastructure can automatically scale up to meet the requests of users. Amazon EC2 allows customers to rent computers on which to run their own applications through a commercial web service. The scalable deployment of applications is facilitated by requesting an arbitrary number of Virtual Machines through the web service interface.

Although the popularity of data centers is increasing, it is still a challenge to provide a proper programming model which is able to support convenient access to the large scale data for performing computations while hiding all low level details of physical environments. Within all the candidates, MapReduce is one of most popular programming models designed for data centers. It was originally proposed by Google to handle large-scale web search applications [9] and has been proved to be an effective programming model for developing data mining, machine learning and search applications in data centers. Especially, it can improve the productivity for those junior developers without required experiences of distributed/parallel development. Moreover, since MapReduce was proposed in 2004, there are many efforts trying to support MapReduce on other architectures [4][12] and exploring various ways to make it more suitable for wider applications [2][15]. Recently, MapReduce has been proposed to form the basis of 'data center computer' [5].

.NET is the standard platform of Windows applications and it has been extended to support parallel computing applications. For example, the parallel extension of .NET 4.0 supports the Task Parallel Library and Parallel LINQ, while MPI.NET [6] implements a high performance library for the message passing interface. Therefore, it is reasonable to expect that .NET should be an indispensable component for Windows-based data centers. This technical report presents an implementation of MapReduce for the .NET platform, called MapReduce.NET. The realization of MapReduce for .NET faces several challenges. First, we expect the MapReduce.NET does not only support search related applications, it can also facilitate a much wider variety of applications, even including some compute-intensive applications. Second, to handle large scale data, .NET lacks a distributed storage facility, like Google File System used by Google MapReduce. Third, a more efficient scheduling algorithm is expected to handle various types of applications. We cannot address all of the above challenges within this technical report. Instead, we summarize the details of one basic implementation of MapReduce.NET. In particular, this technical report describes the following matters:

- MapReduce.NET: a MapReduce programming model designed for the .NET platform with the C# programming language.
- A runtime system of MapReduce.NET deployed in an Enterprise Grid environment by the assistance of Aneka [18].
- A distribute storage system, called WinDFS, which can support a distributed storage service required by MapReduce.NET.

The remainder of this technical report is organized as follows. Section 2 reviews the MapReduce programming model and Aneka. Section 3 discusses the related work. Section 4 presents the architecture of MapReduce.NET. Section 5 describes the performance evaluation of the system. Section 6 presents our conclusions.

## 2. Background Overview

MapReduce is triggered by *map* and *reduce* operations in functional languages, such as Lisp. This model abstracts computation problems through two functions: map and reduce. All problems formulated in this way can be parallelized automatically.

Essentially, the MapReduce model allows users to write Map/Reduce components with functional-style code. These components are then composed as a dataflow graph with fixed dependency relationship to explicitly specify its parallelism. Finally, MapReduce runtime system can transparently explore the parallelism and schedule these components to distributed resources for execution.

All data processed by MapReduce are in the form of key/value pairs. The execution happens in two phases. In the first phase, a map function is invoked once for each input key/value pair and it can generate output key/value pairs as intermediate results. In the second one, all the intermediate results are merged and grouped by keys. The reduce function is called once for each key with associated values and produces output values as final results.

### 2.1. MapReduce Model

A map function takes a key/value pair as input and produces a list of key/value pairs as output. The type of output key and value can be different from input key and value:

$map :: (key_1, value_1) \Rightarrow list(key_2, value_2)$

A reduce function takes a key and associated value list as input and generates a list of new values as output:

$reduce :: (key_2, list(value_2)) \Rightarrow list(value_3)$

### 2.2. MapReduce Execution

A MapReduce application is executed in a parallel manner through two phases. In the first phase, all map operations can be executed independently with each other. In the second phase, each reduce operation may depend on the outputs generated by any number of map operations. However, similar to map operations, all reduce operations can be executed independently.

From the perspective of dataflow, MapReduce execution consists of *m* independent map tasks and *r* independent reduce tasks, each of which may be dependent on *m* map tasks. Generally the intermediate results are partitioned into *r* pieces for *r* reduce tasks.

The MapReduce runtime system schedules map and reduce tasks to distributed resources. It handles many tough problems: parallelization, concurrency control, network communication, and fault tolerance. Furthermore, it performs several optimizations to decrease overhead involved in scheduling, network communication and intermediate grouping of results.

### 2.3. Aneka

Aneka [18] is a .NET-based enterprise Grid software platform, which allows the creation of enterprise Grid environments, and it is used to simply the deployment of MapReduce.NET in distributed environments. Each Aneka node consists of a configurable container, hosting mandatory and optional services. The mandatory services provide the basic capabilities required in a distributed system, such as communications between Aneka nodes, security, and membership. Optional services can be installed to support the implementation of different programming models in Grid environments. MapReduce.NET is implemented as optional services of Aneka.

## 3. Related Work

Since MapReduce was proposed by Google as a programming model for developing distributed data intensive applications in data centers, it has received much attention from the computing industry and academy. Many projects are exploring ways to support MapReduce on various types of distributed architecture and for wider applications. For instance, Hadoop [1] is an open source implementation of MapReduce sponsored by Yahoo!. Phoenix [4] implemented the MapReduce model for the shared memory architecture, while M. Kruijf and K. Sankaralingam implemented MapReduce for the Cell B.E. architecture [12].

A team from Yahoo! research group made an extension on MapReduce by adding a merge phase after reduce, called Map-Reduce-Merge, to perform join operations for multiple related datasets. Dryad supports an interface for composing a DAG (Directed Acyclic Graph) for data parallel applications, which can facilitate much more complex components than MapReduce.

Other efforts are trying to find out ways for making MapReduce support wider applications. For instance, MRPSO [2] utilizes the Hadoop implementation of MapReduce to parallelize a compute-intensive application, Particle Swarm Optimization. Researchers from Intel currently work on making MapReduce suitable for performing earthquake simulation, image processing and general machine learning computations. DISC (Data-Intensive Scalable Computing) [14] started to explore suitable programming models for data-intensive computations with using MapReduce as a start point.

MapReduce is used for the education purpose. For example, several companies have plans to make computing resources available to universities for teaching the MapReduce programming model.

## 4. Architecture

There are several MapReduce implementations, respectively for data centers [1][9], shared memory multi-processor [4] and the Cell architecture [12]. MapReduce.NET resembles Google's design with special emphasis on the .NET and Windows platform.

The design of MapReduce.NET aims to reuse as many existing Windows components as possible. Fig. 1 illustrates the architecture of MapReduce.NET. Our implementation is assisted by several distributed component services from Aneka [18].

Besides Aneka, WinDFS supports MapReduce.NET with a distributed storage service over the .NET platform. WinDFS organizes the disk spaces on all the available resources as a virtual storage pool and provides an object based interface with a flat name space, which is used to manage data stored in it. To process local files, MapReduce.NET can also directly talk with CIFS or NTFS.

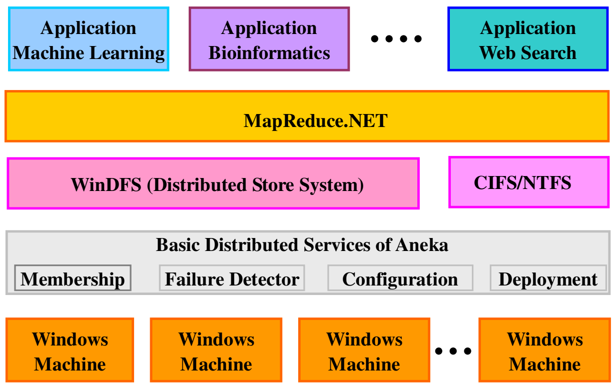

**Fig. 1 Architecture of MapReduce .NET**

The remainder of this section presents details on the programming model and runtime system.

### 3.1. MapReduce APIs

The implementation of MapReduce.NET exposes similar APIs as Google MapReduce. Fig. 2 and 3 illustrate the interface presented to users in C# language. To define Map/Reduce functions, users need to inherit from *Mapper* or *Reducer* class and override corresponding abstract functions. To execute the MapReduce application, user first needs to create a MapReduceApp class, as illustrated in Fig. 4, and set it with corresponding *Mapper* and *Reducer* classes. Then, input files should be configured before starting the execution, as illustrated in Fig. 3. The input files can be local files or files in the distributed store.

```
abstract class Mapper
{
    abstract void Map(object key, object value)
}
```

**Fig. 2 API for Map Function**

```
abstract class Reducer
{
    abstract void Reduce(IEnumerator values)
}
```

**Fig. 3 API for Reduce Function**

The input data type to the Map function is the *object*, which is the root type of all types in C#. For Reduce function, the input is organized as a collection and the data type is *IEnumerator*, which is an interface of supporting an iteration operation on the collection. The data type of each value in the collection is also *object*.

```
class MapReduceApp
{
    void RegisterMapper (Type mapper)
    void RegisterReducer(Type reducer)
    void SetInputFiles(list input)
    list GetOutputFiles()
    bool Execute()
}
```

**Fig. 3 Execution API for Applications**

With *object*, any type of data, including user defined or system build-in type, can be accepted as input. However, for user defined types, users need to provide methods to extract their data from a stream, which may locate in memory or disk.

## 3.2. Runtime System

The execution of a MapReduce computation in .NET environments consists of 5 major phases: Map, Partition, Sort, Merge and Reduce. The overall flow of execution is illustrated in Fig. 4. The execution starts with the Map phase. It iterates the input key/value pairs and invokes the map function defined by users on each key/value pair. The results generated by the Map phase are passed to the Partition, Sort and Merge phases, which perform sorting and merging operations to group the values with identical keys. The result is an array, each element of which is a group of values for each key. Finally, the Reduce phase takes the array as input and invokes the reduce function defined by users on each element of the array.

The execution of MapReduce.NET is orchestrated by a scheduler. The scheduler is implemented as a MapReduce.NET Scheduler service in Aneka, while all the major 5 phases are implemented as a MapReduce.NET Executor service. With Aneka, the MapReduce.NET system can be deployed in cluster or data center environments. Typically, the runtime system consists of one master machine for a scheduler service and multiple worker machines for executor services. As a normal setting illustrated by Fig. 6, each worker machine is configured with one instance of executor and the master machine is configured with the scheduler instance.

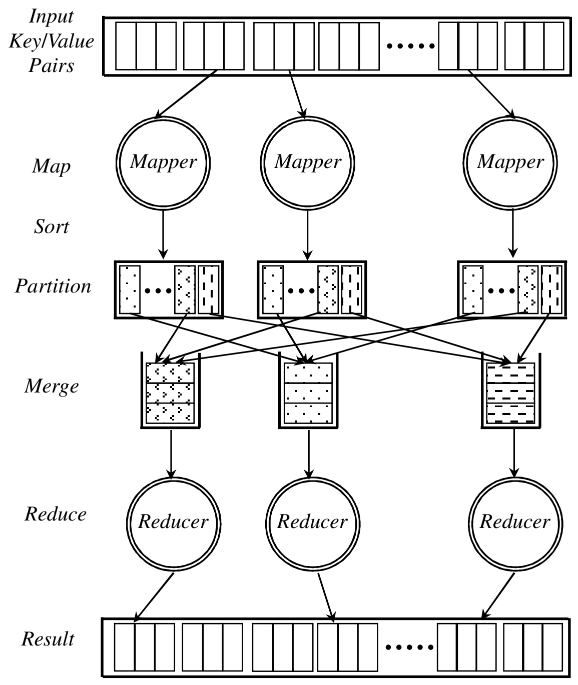

**Fig. 5 Overall Flow of MapReduce.NET Execution**

After users submit MapReduce.NET applications to the scheduler, it deploys the scheduling policy from configuration to map sub tasks to different resources. During the execution, it monitors the progress of each task and takes corresponding task migration operation in case some nodes are much slower than others due to heterogeneity or interference of dominated users.

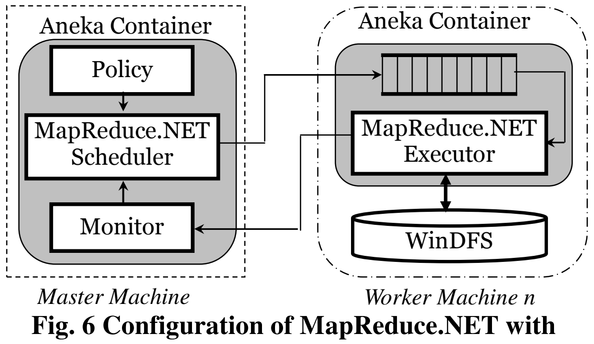

**Fig. 6 Configuration of MapReduce.NET with Aneka**

In the following, we discuss the details of each major phase on the executor of MapReduce.NET.

**3.2.1. Map Phase.** The executor extracts each input key/value pair from the input file. For each key/value pair, it invokes the map function defined by users. The result generated by the map function is first buffered in the memory. The memory buffer consists of many buckets and each one is for different partition. When the size of all results buffered in the memory reaches a predefined maximal threshold, they are sent to the sort phase and written to the disk to save space for holding intermediate results of next round of map invocations.

**3.2.2. Partition Phase.** Partition of the results generated by map functions is achieved in two places: in memory and on disk. In the Map phase, the results generated by map function are first buffered in memory, where there is one bucket for each partition. The generated result determines its partition through a hash function, which may be defined by users. Then the result is appended to the tail of bucket of its partition. When the size of buffered results exceeds the maximal threshold, each bucket is written to disk as an intermediate file. After one map task finishes, all the intermediate files for each partition are merged into one partition.

**3.2.3. Sort Phase.** Before the buffered results are written to disk, elements in each bucket are sorted in memory. They are written to disk by the sorted order, maybe ascending or descending. The sort algorithm we adopt is quick sort [16]. On average, the complexity of this algorithm is $O(n \cdot log(n))$. We choose it because it is always reported faster than other sort algorithms.

**3.2.4. Merge Phase.** To prepare inputs for the Reduce phase, we need to merge all the intermediate files for each partition. First, the executor fetches intermediate files, which are generated in the Map phase, from neighbor machines. Then, they are merged to group values with same key and at the same time, sort keys by a predefined order. Since all the key/value pairs in the intermediate files are already in a sorted order, we deploy a heap sort to achieve the group operation. Each node in the heap corresponds to one intermediate file. Repeatedly, we pick the key/value pair on the top node, and then adjust the shape of the heap to sift the heap node with the biggest key up to the top position. At the same time, we group the values associated with same key.

**3.2.5. Reduce Phase.** In our implementation, the Reduce phase is combined with the Merge phase. During the process of heap sort, we combine all the values associated with same key and then invoke the reduce function defined by users to perform reduction operation on these values. All the results generated by reduce function are written to disk according the order by which they are generated.

## 3.3. Memory Management

Managing memory efficiently is critical for the performance of applications. On each executor, the memory consumed by MapReduce.NET mainly includes memory buffers for intermediate results, memory space for quick sort and buffers for input and output files.

In configuration, administrator can specify a maximal value for the size of memory used by MapReduce.NET. This size is normally determined by the physical configuration of machines and the memory requirement of applications. The memory management is illustrated by Fig. 7.

According to this maximal memory configuration, we set the memory buffer used by intermediate results and input/output files. Our default buffer size for input/output files is 16MB. The input and output files are from the local disk. Therefore, we use *FileStream* in .NET to control the access to local files, including configuration of the size of file buffer.

The memory buffer for intermediate results is implemented by *MemoryStream* of .NET, which is actually a stream in memory. All the results generated by map function are translated into byte array and append to the tail of the stream in memory. An array of indices is used to facilitate accessing each element in this stream. Indices in this array record the position of each intermediate value in the stream. When the size of the stream in memory plus the size of index array exceeds the predefined maximal value, quick sort is invoked to sort all the buffered intermediate values and then write them to disk.

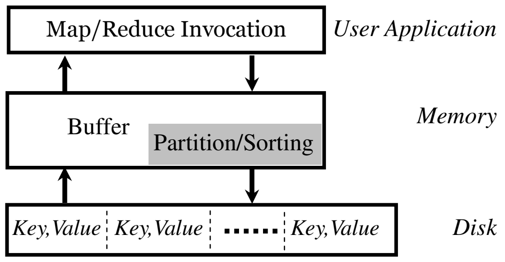

**Fig. 7 Memory Management of MapReduce.NET**

### 3.4. WinDFS

In order to provide a distributed storage system MapReduce.NET, we designed and implemented WinDFS using the C# programming language. WinDFS can be deployed in a dedicated cluster environment or a shared Enterprise Grid environment. Every machine running a WinDFS instance can contribute a certain amount of disk space. All the contributed disk spaces are organized as a virtual data pool. WinDFS provides an object based interface with a flat name space for that data pool. The object can also be taken as a file. Each object contained in WinDFS is identified by a unique name, which is actually a GUID in .NET. WinDFS supports *put* and *get* operations on objects.

The runtime system of WinDFS consist of an *index server* with a bunch of *object server*. Objects are distributed to object servers, while the location information for each object is maintained by the index server. The index server also is responsible for keeping the reliability of objects in the system.

As a representative configuration, the instance of object server runs on each worker machine for managing local objects, while the meta server can be on the master machine.

## 4. Schedule Framework

This section describes the scheduling model for coordinating multiple resources to execute MapReduce computation. The scheduling is conducted by the MapReduce.NET scheduler.

The major 5 phases of MapReduce.NET are grouped into two tasks: Map task and Reduce task. The Map task executes 3 phases: map, partition and sort, while the Reduce task executes merge and reduce. Given a MapReduce.NET job, it consists of *m* Map tasks and *r* Reduce tasks. Each Map task has an input file and generates *r* result files. Each Reduce task has *m* inputs files, which are generated by *m* Map tasks.

Normally the input files for Map tasks are ready in WinDFS prior to execution and thus the size of each Map input file can be determined before scheduling. During the execution, Map tasks dynamically generate output files, the size of which is difficult to determine prior to job execution.

The system aims to be deployed in an Enterprise Grid environment, which essentially organizes idle resources within a company or department as virtual super computer. Normally, resources in Enterprise Grid are shared by two categories of users. The first one is the owner of resources, who has priority to use their resources; the second one is the users of idle resources, who should not disturb the normal usage of resource owner. Therefore, with Enterprise Grid, besides the traditional problems of distributed system, such as complex communications and failures, we have to face a new challenge: *soft failure*. Soft failure stands for the resource involved in MapReduce execution has to quit computation due to domination by its owner.

Due to the above dynamic features of MapReduce.NET application and Enterprise Grid environments, we did not choose a static scheduling algorithm. On the contrary, we deploy a just in time scheduling policy for mapping Map and Reduce tasks to distributed resources in an Enterprise Grid.

The scheduling algorithm for the MapReduce.NET applications starts with scheduling Map tasks. Specifically, all Map tasks are scheduled as independent tasks. The Reduce tasks, however, are dependent on the Map tasks. Whenever Reduce task is ready, i.e. all its inputs are generated by Map tasks, it will be scheduled according to status of resources. The scheduling algorithm aims to optimize the execution time for MapReduce.NET, which is achieved by minimizing the execution of Map and Reduce phases respectively.

During execution, each executor waits task execution commands from the scheduler. For a Map task, normally its input data locates locally. Otherwise, the executor needs to fetch input data from neighbors. For a Reduce task, the executor has to fetch all the input and merge them before execution. Furthermore, the executor monitors the progress of executing task and frequently reports the progress to the scheduler.

## 5. Performance Evaluation

We have implemented the MapReduce.NET system, including the programming model, runtime system and scheduling framework. It has been deployed on desktop machines of several student laboratories in Melbourne University. This section reports the performance evaluation for the runtime system based on two real applications: word count and distributed sort.

All the experiments are executed in an enterprise Grid consisting of 33 nodes drawn from 3 student laboratories. For distributed experiments, one machine was set as master and the rest were configured as worker machines. Each machine has a single Pentium 4 processor, 1GMB of memory, 160GB IDE disk (10GB is dedicated for WinDFS storage), 1 Gbps Ethernet network and runs Windows XP.

### 5.1. Samples Applications

The two sample applications, word count and distributed sort, are benchmarks used by Google MapReduce and Phoenix systems. To implement the Word Count application, users just need to split words for each text file in the map function and sum the appearance number for each word in the reduce function. For sort application, users do not have to do anything within map and reduce functions, while the MapReduce runtime system performs sorting automatically.

### 5.2. System Overhead

MapReduce can be taken as a parallel design pattern, which trades performance to improve the simplicity of programming. Essentially, the Sort and Merge phases of MapReduce runtime system introduce extra overhead. However, the sacrificed perform cannot be overwhelming. Otherwise, it is not acceptable for users. In this section, we evaluate the overhead of MapReduce.NET with local execution. During local execution, the input is from local disk and all 5 major phases of MapReduce.NET executes sequentially on single machine. This is called a local runner and can be used for debug purposes.

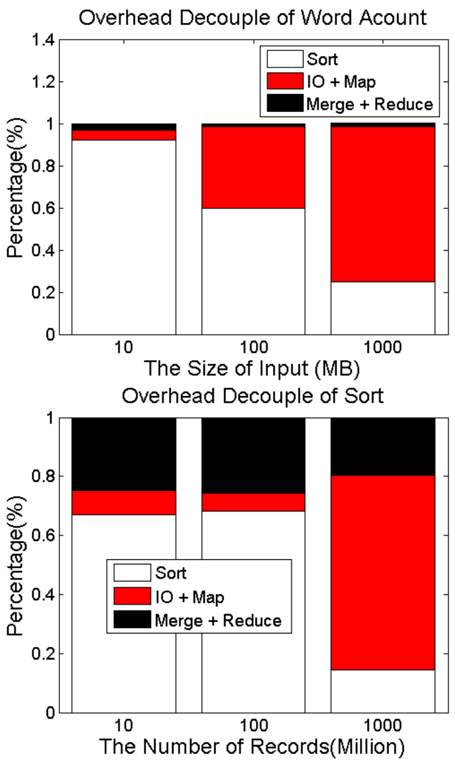

Fig. 8 Overhead Decouple of MapReduce.NET

For local execution, both sample applications were configured as follows:

- The Word Count application took the example text files used by Phoenix [4], with 3 settings of input sizes of raw data: 10MB, 100MB and 1GB respectively.
- The Sort application sorts a number of records. Each record consists of a key and a value. Both the key and value are random integers. Three configurations of input size were adopted: 10 million, 100 million and 1,000 million records respectively. Correspondingly, the sizes of raw data are about 15MB, 150MB and 1.48GB.

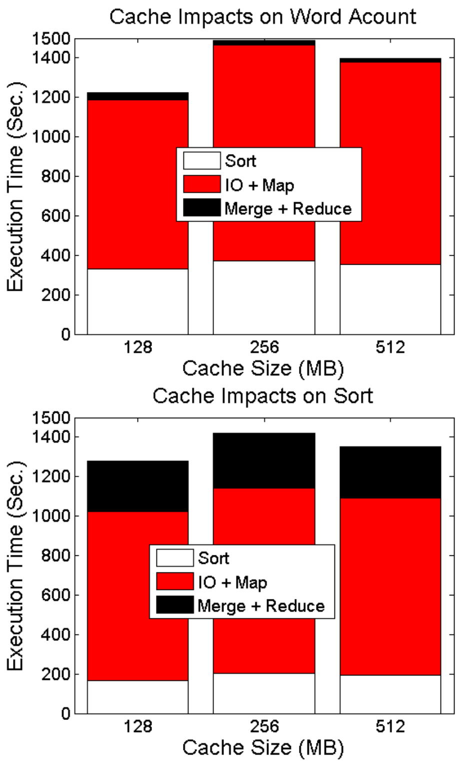

Fig. 9 Cache Impacts of MapReduce.NET

The performance result is split into 3 parts: *sort*, *IO+Map* and *Merge+Reduce*. The sort part is the execution consumed by the sort phase, while the time consumed by the rest of Map task is recorded by IO+Map part, which includes the time consumed by reading input file, invoking map functions and writing partitions of intermediate results to disk. The Merge+Reduce part is the execution time of the Reduce task. Fig. 8 illustrates the percentage of these 3 parts for executing Sort and Word Count applications respectively. We can see that different types of application have different percentage distribution for each part. For Word Count, the time consumed by the reduce and merge phases can even be ignored. The reason is the size of results of Word Count is comparatively small. Differently from Word Count, the reduce and merge phases of Sort application still takes an important percentage. For both applications, as the growth of problem size, the percentage of IO+Map part is correspondingly increasing. Since the map and reduce function of both applications just executed very simple tasks, actually the time consumed by the IO+Map part mainly consists of the contributions from IO operations.

Next, we check the impact of buffer size on the execution time of applications. In particular, the experiments were executed with the different sizes of memory buffer for intermediate results. The results are illustrated in Fig. 9. In the experiments, the size of memory buffer was set to be 128MB, 256MB and 512MB respectively and the results for both applications under each configuration are illustrated.

Different from our expectation, increasing the size of buffer does not have a big impact on the execution time for Word Count and Sort applications. One interesting phenomena is the performance with 256M and 512M buffer is even worse than that with 128M buffer. One reasonable explanation is that a bigger memory buffer can keep more intermediate results, which involves extra overhead during performing quick sort. At the same time, increasing the size of buffer can save the number of IO operations, because the possibility of combining records with same key is increasing. This explains why the performance with 512M buffer is better than with 256M buffer.

## 5.3. Overhead Comparison with Hadoop

This section compares the overhead of MapReduce.NET with Hadoop, the open source MapReduce implementation with Java language. Hadoop is supported by Yahoo and aims to work as a general purposed distributed platform. The DISC project [14] is using Hadoop as the first step for exploring suitable programming models for data intensive scalable computing. The stable release of Hadoop, version 0.16.4 was adopted for comparison.

To compare the overhead, we run the local runner of Hadoop and MapReduce.NET respectively with same size of input for Word Count and Sort applications. The buffer size was configured to be 128MB for both implementations. The input for Sort consists of 1,000 million records with 1.48GB raw data, while for Word Count the size of raw input data is 1GB. The results are illustrated in Fig. 10. MapReduce.NET performs worse on the Word Count application than Hadoop, while outperforming Hadoop on the Sort application. Specifically, for Sort application, the sort phase of Hadoop consumes longer time than the MapReduce.NET, while its IO processing is more efficient. Similar phenomenon happens for the Sort application. However, the reduce and merge phases of Hadoop took comparatively longer time than our implementation.

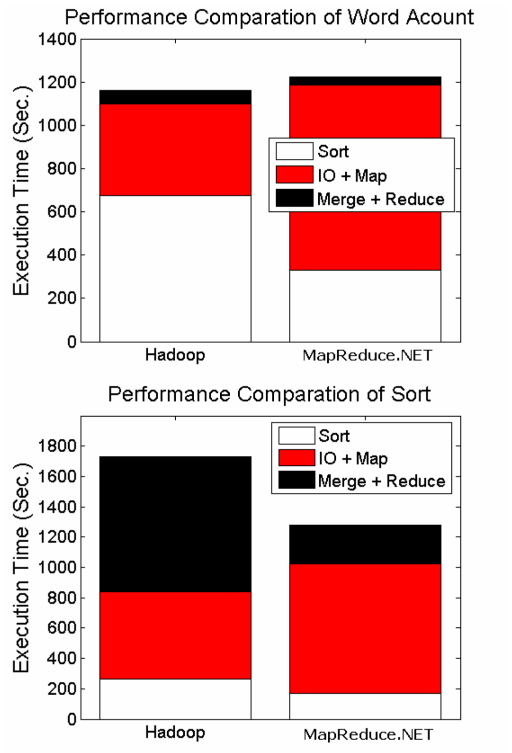

Fig. 10 Overhead Comparison of Hadoop and MapReduce.NET.

## 5.3. System Scalability

In this section, we evaluate the scalable performance of MapReduce.NET in the distributed environment. Since Hadoop does not have a parallel version on Windows platform, we did not compare the parallel performance with Hadoop.

Applications were configured as follows:

- Word Count: takes the example text files used by Phoenix [4]. We duplicated the original text files to generate an example input with 6GB raw data, which is split into 32 files.
- Distributed Sort: sorts 5,000 million records in an ascending order. The key of each record is a random integer. The total raw data is about 7.6GB,

which is partitioned into 32 files.

Fig. 11 illustrates the scalable performance result of the Word Count application. In the figure, the execution time of Map phase consists of the time from starting execution to the finish of all Map tasks, while the Reduce execution time consists of merge phase plus invoking reduce functions on all the work machines. From the results, we can see map, sort and partition phases dominated the whole execution and the performance increased as more resources were added into the computation.

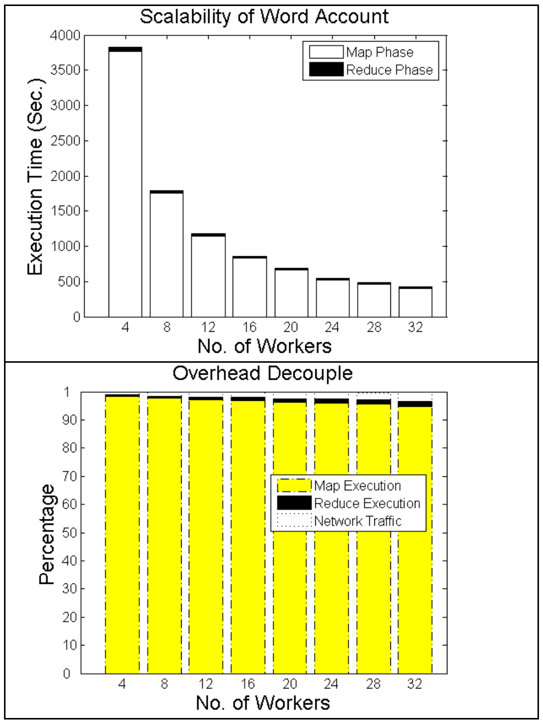

Fig. 11 Scalable Experiment of Word Count

Different from the Word Count application, the Distributed Sort application has a nearly uniform distribution of execution time for Map and Reduce tasks, as illustrated in Fig. 12. However, this does not effect the nearly linearly speedup while adding more resources. The network traffic also takes an important percentage of the whole execution, because the intermediate result of distributed sort is actually same as the original input data.

Based on the experiments of the above tow typical MapReduce applications, MapReduce.NET is shown to provide a scalable performance within homogenous environments during the number of computation machines increases.

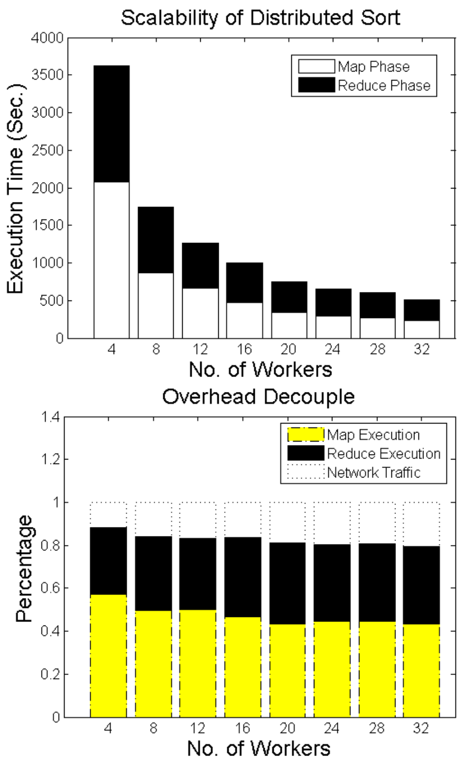

Fig. 12 Scalable Experiment of Distributed Sort

## 6. Conclusion

This technical report presents MapReduce.NET, an implementation of MapReduce over .NET platform. The model and runtime system assemble Google's implementation. We evaluated the overhead of our implementation and compared it with Hadoop, the open source implementation. Besides the comparatively small overhead, the system can also support a scalable performance in distributed environments. The results prove that our implementation can support reasonable performance and is practical for usage as a generally purposed platform for data-intensive applications.

## Acknowledgements

This work is partially supported by research grants from the Australian Research Council (ARC) and Australian Department of Industry, Innovation, Science and Research (DIISR). We thank Christian Vecchiola and Jemal Abawajy for their comments on improving the quality of the paper.

## 10. References

[1] Apache. Hadoop. *http://lucene.apache.org/hadoop/*.

[2] A. W. McNabb, C. K. Monson, and K. D. Seppi, *Parallel PSO Using MapReduce*, In Proceedings of the Congress on Evolutionary Computation (CEC 2007), Singapore, 2007.

[3] A. Weiss. *Computing in the Clouds*. netWorker, 11(4):16-25, Dec. 2007.

[4] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, *Evaluating MapReduce for Multi-core and Multiprocessor Systems*, Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA), Phoenix, AZ, February 2007.

[5] D. A. Patterson, *Technical perspective: the data center is the computer*, Communications of the ACM, 51-1, 105, January 2008.

[6] D. Gregor and A. Lumsdaine, *Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure*, Principles and Practice of Parallel Programming, pp. 133-142, Feb. 2008, ACM.

[7] H. Sutter, J. Larus, *Software and the Concurrency Revolution*, ACM Queue, Vol. 3, No. 7, pp 54–62, 2005.

[8] H. C. Yang, A. Dasdan, R. L. Hsiao, and D. S. P. Jr. *Map-reduce-merge: simplified relational data processing on large clusters*, Proceedings of SIGMOD, 2007.

[9] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI), San Francisco, CA, Dec., 2004.

[10] J. Markoff and S. Hansell. *Hiding in plain sight, Google seeks more power*, New York Times, June 14, 2006.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, *Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks*, European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March, 2007.

[12] M. Kruijf and K. Sankaralingam. *MapReduce for the Cell B.E. Architecture*, TR1625, Technical Report, Department of Computer Sciences, The University of Wisconsin-Madison, 2007.

[13] R. Buyya, C. S. Yeo, and S. Venugopal, *Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities*, Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008), Sept., 2008, Dalian, China.

[14] R. E. Bryant, *Data-Intensive Supercomputing: The Case for DISC*, CMU-CS-07-128, Technical Report, Department of Computer Science, Carnegie Mellon University, May, 2007.

[15] S. Chen, S. W. Schlosser. *Map-Reduce Meets Wider Varieties of Applications*, IRP-TR-08-05, Technical Report, Intel Research Pittsburgh, May, 2008.

[16] T. H. Cormen , C. E. Leiserson , R. L. Rivest , C. Stein, *Introduction to Algorithms*, Second Edition, The MIT Press, Massachusetts, USA.

[17] T. Hey and A. Trefethen. *The data deluge: an e-Science perspective*. In F. Berman, G. C. Fix, and A. J. G. Hey, editors, Grid Computing: Making the Global Infrastructure a Reality, pp. 809–824.Wiley, 2003.

[18] X. Chu, K. Nadiminti, J. Chao, S. Venugopal, and R. Buyya, *Aneka: Next-Generation Enterprise Grid Platform for e-Science and e-Business Applications*, Proceedings of the 3rd IEEE International Conference and Grid Computing, Bangalore, India, Dec. 10-13, 2007.