MapReduce Programming Model for .NET-based Cloud 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 large scale computer systems are built in order to meet the high storage and processing demands of compute and data-intensive applications. MapReduce is one of the most popular programming models designed to support the development of such applications. It was initially created 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 paper presents a realization of MapReduce for .NET-based data centers, including the programming model and the runtime system. The design and implementation of MapReduce.NET are described and its performance evaluation is presented.

1 Introduction

Recently several organizations are building large scale computer systems to meet the increasing demands of high storage and processing requirements of compute and data-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 fast response and high availability. On the academia front, many scientific research projects increasingly rely on large scale data sets and powerful processing ability provided by super computer systems, commonly referred to as e-Science [15].

These huge demands on data centers motivate the concept of Cloud Computing [9] [12]. With clouds, IT-related capabilities can be provided as service, which is accessible through the Internet. Representative systems include Google App Engine, Amazon Elastic Compute Cloud (EC2), Majrasoft Aneka, and Microsoft Azure. The infrastructure of Cloud Computing can automatically scale up to meet the requests of users. The scalable deployment of applications is typically facilitated by Virtual Machine (VM) technology.

With the increasing popularity of data centers, it is 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 the most popular programming models designed for this purpose. It was originally proposed by Google to handle large-scale web search applications [8] and has been proved to be an effective programming model for developing data mining and machine learning applications in data centers.

Especially, it can improve the productivity of junior developers who do not have required experiences of distributed/parallel development. Therefore, it has been proposed to form the basis of a 'data center computer' [5].

The .NET framework is the standard platform of Microsoft 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 (MPI). Moreover, the Azure cloud service recently released by Microsoft, enables developers to create applications running in the cloud by using the .NET Framework.

This paper presents a realization of MapReduce for the .NET platform, called MapReduce.NET. It not only supports data-intensive applications, but also facilitates a much wider variety of applications, even including some compute-intensive applications, such as Genetic Algorithm (GA) applications. In this paper, we describe:

- MapReduce.NET: A MapReduce programming model designed for the .NET platform using the C♯ programming language.
- A runtime system of MapReduce.NET deployed in an Enterprise Cloud environment, called Aneka [12].

The remainder of this paper is organized as follows. Section 2 gives an overview of MapReduce. Section 3 discusses related work. Section 4 presents the architecture of MapReduce.NET, while Section 5 discusses the scheduling framework. Section 6 describes the performance evaluation of the system. Section 7 concludes.

2 MapReduce Overview

MapReduce is triggered by the *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 to explicitly specify their parallelism. Finally, the MapReduce runtime system schedules these components to distributed resources for execution while handling many tough problems: parallelization, network communication, and fault tolerance.

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:

$$map :: (key_1, value_1) \Rightarrow list(key_2, value_2) \tag{1}$$

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

$$reduce :: (key_2, list(value_2)) \Rightarrow list(value_3) \tag{2}$$

A MapReduce application is executed in a parallel manner through two phases. In the first phase, all map operations can be executed independently from each other. In

the second phase, each reduce operation may depend on the outputs generated by any number of map operations. All reduce operations can also be executed independently similar to map operations.

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 academia. Many projects are exploring ways to support MapReduce on various types of distributed architecture and for a wider range of applications. For instance, Hadoop [2] 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 [11].

A team from Yahoo! research group made an extension on MapReduce by adding a merge phase after reduce, called Map-Reduce-Merge [7], to perform join operations for multiple related datasets. Dryad [10] supports an interface to compose a Directed Acyclic Graph (DAG) for data parallel applications, which can facilitate much more complex components than MapReduce.

Other efforts focus on enabling MapReduce to support a wider range of applications. MRPSO [1] utilizes the Hadoop implementation of MapReduce to parallelize a compute-intensive application, called Particle Swarm Optimization. Researchers from Intel currently work on making MapReduce suitable for performing earthquake simulation, image processing and general machine learning computations [14]. MRPGA [3] is an extension of MapReduce for GA applications based on MapReduce.NET. Data-Intensive Scalable Computing (DISC) [13] started to explore suitable programming models for data-intensive computations by using MapReduce.

4 Architecture

MapReduce.NET resembles Google's MapReduce, but 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. Its implementation is assisted by several component services from Aneka. Aneka is a .NET-based platform for enterprise and public Cloud Computing [12]. It supports the development and deployment of .NET-based Cloud applications in public Cloud environments, such as Amazon EC2. We used Aneka to simplify 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 Cloud environments. MapReduce.NET is implemented as an optional service of Aneka.

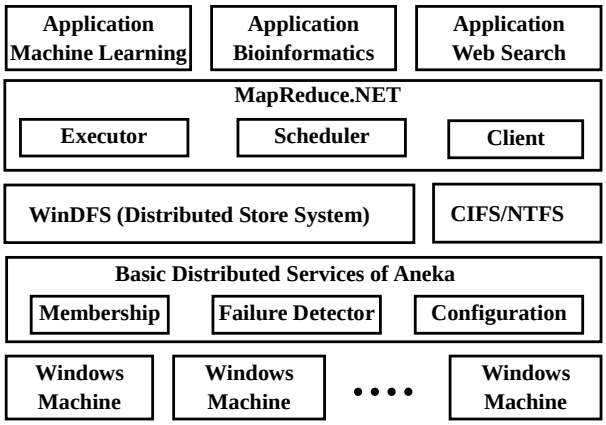

Fig. 1: Architecture of MapReduce.NET.

Besides Aneka, WinDFS provides 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 communicate with CIFS or NTFS. The remainder of this section presents details on the programming model and runtime system.

Table 1: APIs of MapReduce.NET

```
class Mapper
{
    void Map(MapInput < K, V> input)
}

class Reducer
{
    void Reduce(IReduceEnumerator input)
}
```

4.1 MapReduce.NET APIs

The implementation of MapReduce.NET exposes APIs similar to Google MapReduce. Table 1 illustrates the interface presented to users in the 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, the user first needs to create a MapReduceApp class and set it with the corresponding *Mapper* and *Reducer* classes. Then, input files should be configured before starting the execution and they can be local files or files in the distributed store.

The type of input key and value 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 for supporting an iterative operation on the collection. The data type of each value in the collection is also *object*.

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 serialization and deserialization methods. Otherwise, the default serialization and deserialization methods will be invoked.

4.2 Runtime System

The execution of a MapReduce.NET application consists of 4 major phases: Map, Sort, Merge and Reduce. The overall flow of execution is illustrated in Fig. 2. The execution starts with the Map phase. It iterates the input key/value pairs and invokes the map function defined by users on each pair. The generated results are passed to the 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.

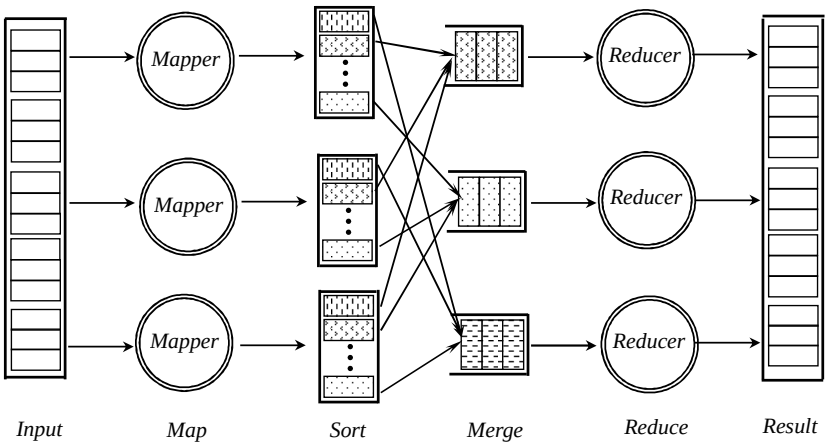

Fig. 2: Computation of MapReduce.NET.

The runtime system is based on the master-slave architecture with the execution of MapReduce.NET orchestrated by a scheduler. The scheduler is implemented as a MapReduce.NET Scheduler service in Aneka, while all the 4 major 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, it consists of one master machine for a scheduler service and multiple worker machines for executor services.

The 4 major phases are grouped into two tasks: Map task and Reduce task. The Map task executes the first 2 phases: map and sort, while the Reduce task executes the last 2 phases: merge and reduce. The input data for the map function is split into even-sized m

pieces to be processed by m map tasks, which are evenly assigned to worker computers. The intermediate results generated by map tasks are partitioned into r fragments, and each fragment is processed by one reduce task.

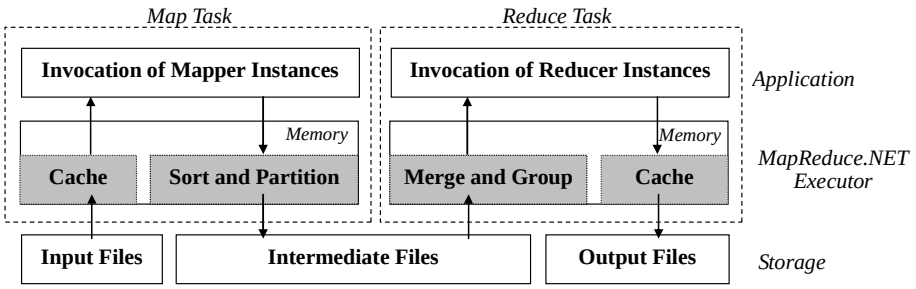

Fig. 3: Dataflow of MapReduce.NET.

The major phases on the MapReduce.NET executor are illustrated in Fig. 3.

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 a different 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 the bucket of its partition. When the size of all the results buffered in the memory reaches a predefined maximal threshold, they are sent to the Sort phase and then written to the disk. This saves space for holding intermediate results for the next round of map invocations.

Sort Phase. When the size of buffered results exceeds the maximal threshold, each bucket is written to disk as an intermediate file. 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, either ascending or descending. The sorting algorithm adopted is quick sort.

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 neighboring machines. Then, they are merged to group values with the same key. 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, the key/value pair on the top node is picked, and simultaneously the values associated with same key are grouped.

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 the same key and then invoke the reduce function defined by users to perform the reduction operation on these values. All the results generated by reduce function are written to disk according to the order by which they are generated.

4.3 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 the sorting algorithm and buffers for input and output files. The memory management is illustrated in Fig. 3.

The system 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.

According to this maximal memory configuration, we set the memory buffer used by intermediate results and input/output files. The default value for read/write buffer of each file is 16MB. The input and output files are from the local disk. Therefore, we use the *FileStream* class to control the access to local files.

The memory buffer for intermediate results is implemented by using the *MemoryStream* class, which is a stream in memory. All the results generated by map functions are serialized and then append to the tail of the stream in memory.

5 Scheduling Framework

This section describes the scheduling model for coordinating multiple resources to execute MapReduce computations. The scheduling is managed by the MapReduce.NET scheduler. After users submit MapReduce.NET applications to the scheduler, it maps Map and Reduce tasks to different resources. During the execution, it monitors the progress of each task and migrate tasks when some nodes are much slower than others due to their heterogeneity or interference of dominating users.

Typically, a MapReduce.NET job 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 input files which are generated by m Map tasks.

Normally the input files for Map tasks are available in WinDFS or CIFS prior to job execution, thus the size of each Map input file can be determined before scheduling. However, the output files are dynamically generated by Map tasks during execution, hence the size of these output files is difficult to determine prior to job execution.

The system aims to be deployed in an Enterprise Cloud environment, which essentially organizes idle resources within a company or department as a virtual super computer. Normally, resources in Enterprise Clouds are shared by the owner of resources and the users of idle resources. The latter one should not disturb the normal usage of resource owner. Therefore, with an Enterprise Cloud, besides facing the traditional problems of distributed system, such as complex communications and failures, we have to face *soft failure*. Soft failure refers to a resource involved in MapReduce execution having to quit computation due to domination by its owner.

Due to the above dynamic features of MapReduce.NET application and Enterprise Cloud environments, we did not choose a static scheduling algorithm. On the contrary, the basic scheduling framework works like the work-stealing model. Whenever a worker node is idle, a new Map or Reduce task is assigned to it for execution with special priority on taking advantage of data locality.

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

6 Performance Evaluation

We have implemented the programming model and runtime system of MapReduce.NET and deployed it on desktop machines of several student laboratories in Melbourne University. This section evaluates its performance based on two benchmark applications: Word Count (WC) and Distributed Sort (DS).

All the experiments were executed in an Enterprise Cloud consisting of 33 machines located in 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 memory, 160GB hard disk (10GB is dedicated for WinDFS storage), 1 Gbps Ethernet network and runs Windows XP.

6.1 Sample Applications

The sample applications (WC and DS) are benchmarks used by Google MapReduce and Hadoop. To implement the WC application, users just need to split words for each text file in the map function and sum the number of appearance for each word in the reduce function. For the DS application, users do not have to do anything within the map and reduce functions, while MapReduce.NET performs sorting automatically.

The rest of this section presents the overhead of MapReduce.NET. First, we show the overhead caused by the MapReduce programming model in a local execution. Then the overhead of MapReduce.NET in a distributed environment is reported.

6.2 System Overhead

MapReduce can be regarded as a parallel design pattern, which trades performance to improve the simplicity of programming. Essentially, the Sort and Merge phases of the MapReduce runtime system introduce extra overhead. However, the sacrificed performance cannot be overwhelming. Otherwise, it would not be acceptable. We evaluate the overhead of MapReduce.NET with local execution. The input files are located on the local disk and all 4 major phases of MapReduce.NET executes sequentially on a single machine. This is called a local runner and can be used for debugging purposes.

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

- The WC application processes the example text files used by Phoenix [1] and the size of raw data 1GB.
- The DS application sorts a number of records consisting of a key and a value, both of which are random integers. The input data includes 1,000 million records with 1.48GB raw data.

The execution time is split into 3 parts: Map, Sort and Merge+Reduce. They correspond to the time consumed by reading inputs and invoking map functions, the time consumed by the sort phase(including writing intermediate results to disk) and the time consumed by the Reduce tasks. In this section, we analyze the impact of buffer size for intermediate results on the execution time of applications. In particular, the experiments were executed with different sizes of memory buffer for intermediate results. The size of memory buffer containing intermediate results was set to be 128MB, 256MB and 512MB respectively and the results for both applications are shown in Fig. 4.

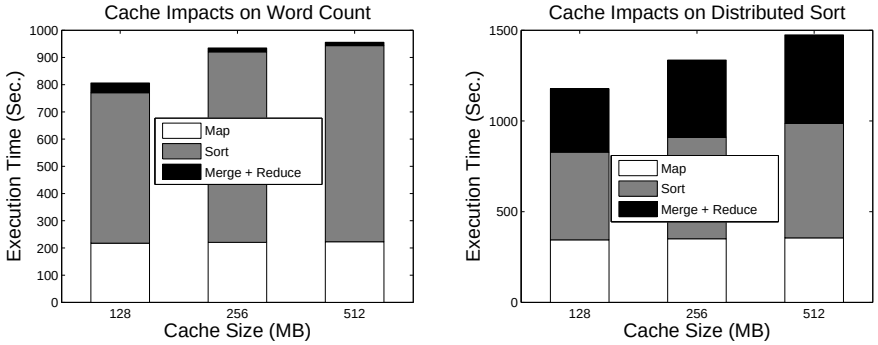

Fig. 4: Cache Impacts of MapReduce.NET.

First, we can see that different types of application have different percentage distribution for each part. For the WC application, the time consumed by the reduce and merge phases can even be ignored. The reason is that the size of results of WC is comparatively small. On the contrary, the reduce and merge phases of the DS application incur a much larger percentage of total time consumed.

Second, out of our expectation, increasing the size of the buffer for intermediate results may not reduce the execution time for both applications. On the contrary, a larger buffer increases the time consumed by sorting because sorting more intermediates at one time needs deeper stack and more resources. A larger memory buffer generates fewer intermediate files, but each is characterized by a larger size. The read/write buffer of each input/output files is configured per file, and the default value is 16MB. Therefore, with a larger buffer for intermediate results in the Map and Sort phase, the Reduce phase consumes longer time because the overall size of of input file buffers is smaller. However, a larger memory buffer does not have significant impacts on the Map phase.

6.3 Overhead Comparison with Hadoop

This section compares the overhead of MapReduce.NET with Hadoop, an open source implementation of MapReduce in Java. Hadoop is supported by Yahoo! and aims to be a general purpose distributed platform. We use the latest stable release of Hadoop (version 0.18.3).

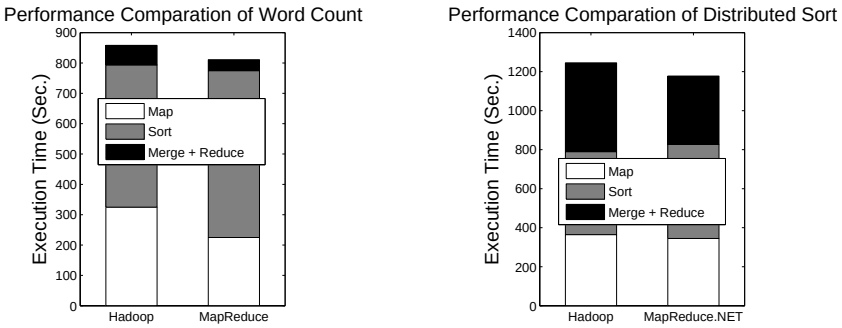

Fig. 5: Overhead Comparison of Hadoop and MapReduce.NET.

To compare the overhead, we run the local runner of Hadoop and MapReduce.NET respectively with the same input size for both applications. The size of buffer for intermediate results was configured to be 128MB for both implementations. The configuration of WC and DS applications are the same as Section 6.2. The JVM adopted in our experiment is Sun JRE1.6.0, while the version of the .NET framework is 2.0. The results are shown in Fig. 5. MapReduce.NET performs better than Hadoop for both applications. Specifically, both *Map* and *Merge+Reduce* phase of MapReduce.NET consumes less time than Hadoop, but more time than Hadoop in the *Sort* phase.

Reasons for this are:(a) the deserialization and serialization operations achieved by MapReduce.NET is more efficient than Hadoop; (b) the Merge phase of Hadoop involves extra IO operations than MapReduce.NET. In particular, for the Map phase, the major overhead of both applications consists of invocation of deserialization of raw input data and map functions combined with reading disk operations. According to our experiments, however, we did not find significant performance difference of disk IO operations by using JRE1.6.0 and .NET 2.0 over Windows XP. In the Merge+Reduce phase, the major overhead includes serialization, deserialization and reading and writing disk. Hadoop splits the large input files into a number of small pieces(32 pieces for WC and 49 pieces for DS) and each piece corresponds to a Map task. Then, Hadoop first has to merge all the intermediate results for the same partition from multiple Map tasks prior to starting the combined Merge and Reduce phase. MapReduce.NET does not require this extra overhead. Therefore it performs better than Hadoop in the *Merge+Reduce* phase.

In the Sort phase, the sorting algorithm implemented by Hadoop is more efficient than its corresponding implementation in MapReduce.NET. Both MapReduce.NET and Hadoop implement the same sorting algorithm, hence identifying the difference in performance between two implementations implies a deep investigation involving the internals of the two virtual machines.

6.4 System Scalability

In this section, we evaluate the scalable performance of MapReduce.NET in a distributed environment. Applications were configured as follows:

- WC: We duplicated the original text files used by Phoenix [1] to generate an example input with 6GB raw data, which is split into 32 files.
- DS: 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.

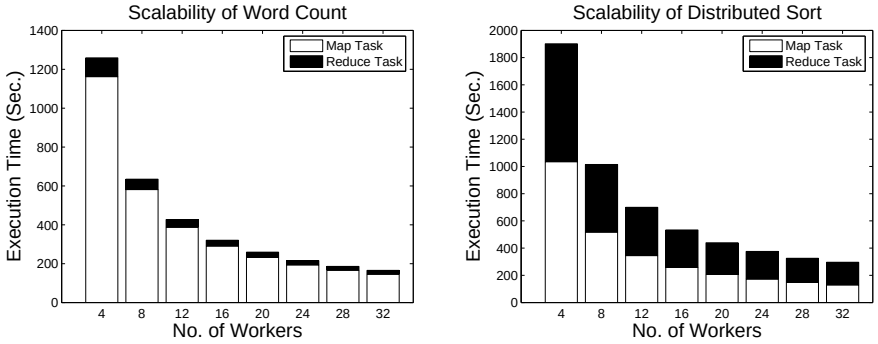

Fig. 6: Scalability of MapReduce.NET in Aneka Environment.

Fig. 6 illustrates the scalable performance result of the WC application. The Map task execution time consists of the map and sort phases which include the time from starting the execution to the end of execution for all Map tasks. The Reduce task execution time consists of overhead of network traffic, the merge phase and the reduce phase invoking reduce functions on all the worker machines. We can see that the map and sort phases dominate the entire execution of the WC application.

Unlike the WC application, the DS application has a nearly uniform distribution of execution time for Map and Reduce tasks, as shown in Fig. 6. The network traffic also incurs a substantial percentage of the entire execution, because the intermediate results are actually the same as the original input data.

For both applications, the performance increases as more resources are added to the computation. Therefore, MapReduce.NET is able to provide scalable performance within homogenous environments when the number of machines increases.

7 Conclusions

This paper presented MapReduce.NET, a realization of MapReduce on the .NET platform. It provides a convenient interface to access large scale data in .NET-based distributed environments. We evaluated the overhead of our implementation on Windows platforms. Experimental results have shown that the performance of MapReduce.NET is comparable or even better (for some cases) than Hadoop on WindowsXP. Furthermore, MapReduce.NET provides scalable performance in distributed environments. Hence, MapReduce.NET is practical for usage as a general purpose .NET-based distributed and Cloud computing model. In the future, we endeavor to integrate MapReduce.NET with the Azure Cloud Platform.

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 would like to thank Chee Shin Yeo, Christian Vecchiola and Jemal Abawajy for their comments on improving the quality of the paper.

References

[1] A. W. McNabb, C. K. Monson, and K. D. Seppi. Parallel PSO Using MapReduce. In *Proc. of the Congress on Evolutionary Computation*, 2007.

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

[3] C. Jin, C. Vecchiola and R. Buyya. MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. In *Proc. of 4th International Conference on e-Science*, 2008.

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

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

[6] D. Gregor and A. Lumsdaine. Design and Implementation of a High-Performance MPI for C♯ and the Common Language Infrastructure. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[7] H. C. Yang, A. Dasdan, R. L. Hsiao, and D. S. P. Jr. Map-Reduce-Merge: simplified relational data processing on large clusters. In *Proc. of SIGMOD*, 2007.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th Symposium on Operating System Design and Implementation*, 2004.

[9] J. Varia. Cloud Architectures. In *White Paper of Amazon, jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf*, 2008.

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

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

[12] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[13] R. E. Bryant. Data-Intensive Supercomputing: The Case for DISC. In *CMU-CS-07-128, Technical Report, Carnegie Mellon University*, 2007.

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

[15] T. Hey and A. Trefethen. The data deluge: an e-Science perspective. *Grid Computing: Making the Global Infrastructure a Reality*, pages 809–824, 2003.