RedLib: A Lightweight Reduction Library for Java, Technical Report

Mostafa Mehrabi,
mmeh012@aucklanduni.ac.nz
Supervisor: Dr. Oliver Sinnen
Co-supervisor: Dr. Nasser Giacaman
Electrical and Computer Engineering Department
The University of Auckland

January 26, 2015

List of Figures

List of Tables

Abstract

Reduction in computer science is the process of combining two or more elements into one. This process is widely used by network based applications for integrating results from different computers of a network. It also seems reasonable to use the same mechanism in shared memory applications that run on a single computer. That is, reducing the results that are obtained from different threads in a computer into the final result. However, there are not many libraries that facilitate reduction on single computers, as the main focus has been on network based applications thus far.

Considering the benefits of reduction for improving performance on shared memory applications, developing assistant libraries in this scope is quite worthwhile. In this paper we have introduced an extensive reduction library that has been developed for Java. Moreover, the object oriented considerations of the design have been explained, and it has been clarified how users benefit from them. Also, we have compared the features provided by our design with a few others that are available in this field. Further examples in this paper help with clearer understanding of the logic of our design.

Key words: Reduction, ParallelIterator, Shared Memory

1 Introduction

Fast technological growth has improved the accuracy of computations substantially due to the large amount of detailed data that we are able to collect now. Subsequently, the calculations that are performed on data are now more expensive in terms of resources. That is, there are considerably more resources required for performing computations than it used to be. Therefore, speed of computation (i.e. performance) is limited by the boundaries set by our resources. On a single computer, the number of CPU cycles can only be increased up to a certain extent due to physical limitations. Because of the problem thereof, many applications are moving on cloud where a network of computers can work on different parts of a task; however there are still limitations regarding the number of machines and data storage provided by a cloud network.

The limitations mentioned above encourage endeavors for more efficient use of resources in order to improve performance as much as possible, and that is where parallel processing becomes important. That is, parallel contribution of different processors (on a single computer), or different computers (on a cloud network) to a problem speeds the performance of a computational task up. In this procedure each of the parallel components provides a partial result, which needs to be integrated with that of other components in order to figure out the final result. The stage of integrating the results from different components of a computation task is called *reduction*.

The concept of parallel processing has been vastly used in network based applications, but has recently become an interest for shared memory applications (i.e. on single computers). A basic search about reduction on internet provides us with numerous articles that are mostly focused on network applications (e.g. Yang et al. [June 2007], Abouzeid et al. [August 2009]). For example, search based algorithms used by Google and Hadoop exploit reductions extensively in order to integrate the search results that are returned from several nodes (i.e. computers) in their network into one final result (Lammel [2008], Hadoop [2014]).

However, as the potential for using reduction on shared memory applications is increasingly growing, the demand for rich assistant libraries has not been fulfilled reciprocally for different programming languages. OpenMP is one library which has been providing this feature for C/C++ and Fortran programming languages. Nevertheless, its support for reduction is limited about which we have elaborately explained in later sections of this paper. Moreover, lack of such a library is even more obvious for programming languages such as Java, as Java support for reduction is very primitive (Java [2014]).

We have integrated complex reduction approaches that are normally used by network applications with simple trivial reductions, and have provided an extensive reduction library that can be used for

both shared memory and network cases. Moreover, we have considered object oriented principles in order to let programmers flexibly extend the reduction functionality and exploit code reusability.

In Section 2 we have discussed some of the common reduction approaches and their use-cases. In Section 3 we have overviewed some of the related works, and have figured out the important principles in developing map-reduce operations. In Section 4 the reduction functionality provided by OpenMP has been discussed. Our design approach has been explained elaborately in Section 5, and few examples have been provided in order to clarify its advantages. In Section 6 the benchmarks that we implemented to examine the efficiency of our design have been discussed. Finally, in Section 7 we have wrapped up conclusions by listing the outcomes of this paper.

2 Common Reduction Paradigms

Reductions are mainly used for finding the answers to search or query tasks. In most reduction algorithms the input and output are in the form of <Key, Value> pairs. Reduction always comes with a mapping algorithm, where a big task is split into a set of smaller tasks. The smaller tasks are performed in parallel via mappers, then the results from mappers are sent to reducers for integration (HighlyScalableBlog [February 2012]). It is important to mention that reduction operations must be commutative and associative (Lammel [2008], Hadoop [2014]). In the following paragraphs some of the common reduction approaches have been discussed.

2.1 Counting

This approach is exploited for figuring out the total number of occurrence of a specific item or pattern across multiple elements. In this approach different mappers inspect different documents/elements in parallel; furthermore the summation of the results returned from the mappers are calculated in the reducers (Lin et al. [April 30 2010]).

2.2 Collating

This approach involves grouping all values that are associated to the same key in the <Key, Value> pairs that are sent from mappers to reducers. One of the most common use-cases of this approach is in the concept of *inverted indices*, where a data structure (e.g. a map) relates certain data to the locations in which it can be found (e.g. web page, database etc.). This approach is one of the most important reduction methods that is widely used by large scale search engines such as Google (Lammel [2008], Lin et al. [April 30 2010]).

The implementations for this approach are mainly application specific. That is, enterprises such as Google or Hadoop implement this approach for their own applications. However, a generic implementation is yet not provided by any of the programming languages (Lammel [2008], OpenMP [July 2013]). We have provided generic classes in our library to facilitate this method, which has been discussed in details in Section 4.

2.3 Filtering

This approach is a more specific version of **collating**, where values that are associated to a key in a <Key, Value> pair, are records from several documents that meet certain conditions (Afrati and Ullman [2010], Lin et al. [April 30 2010]).

2.4 Iterative Message Passing

In some networks (e.g. a network of computers), the status of each entity depends on the properties of its adjacent entities. Therefore, frequent messages are passed to each network entity iteratively. Each message is a set of <Key, Value> tuples, where the IDs of network entities are used as **keys**, and the status of each entity is the **value** for its corresponding key. Receiving nodes group the messages on arrival, and send them to reducers. Subsequently, the reducers calculate and update the state of the corresponding node, using the information that they receive regarding the adjacent entities.

This approach is commonly used in applications that are related to graph analysis and web indexing scopes. For example, in a graph the existence of a parent node may depend on the existence of its children nodes. Therefore, once all children are removed, the parent node must be removed consequently. Thus, frequent updates about the status of the children nodes must be performed on the parent node (HighlyScalableBlog [February 2012]).

2.5 Distributed Task Execution

In this approach big tasks are broken down into smaller tasks, and small tasks are performed in parallel. Furthermore, the results are sent to reducers to figure out the final result. This method is widely used in physical/engineering simulation, performance testing and mathematical computation application domains.

Reducers normally use map data structures, thus the input data to the reducers are in the form of <Key, Value> pairs. Therefore, the operations that are performed on the input data mainly involve union, intersection, subtraction and selection on the values in maps (Lin et al. [April 30 2010]). This method is one of the main focuses of our implementation, thus the operations thereof are also provided as part of the remarkable contributions of our library, about which we have discussed in details in Section 4.

Some algorithms suggests that reducers keep maps sorted based on their keys for more efficient performance. However, performance improvement is conditional to the data being large enough, so that the overhead of insertion sort is compensated when accessing data from the results that are returned by reducers (HighlyScalableBlog [February 2012]).

3 Related Work

Applications that exploit reduction operations can be divided into two major categories of *distributed-memory* and *shared-memory*.

3.1 Distributed-Memory Applications

Distributed-memory applications are based on computer networks, where the data being processed is scattered through different machines on a cluster of computers. Map-reduce operations are widely used by distributed-memory applications. The following paragraphs introduce a few well-known frameworks that support map-reduce operations for the applications in this category.

3.1.1 Google Map-Reduce

Google widely exploits map-reduce in its daily search-based operations. As a matter of fact, more than hundred thousand map-reduce tasks are performed on Google clusters everyday for operations such as intensive graph processing, inverted indices, graph representation of the structure of a web-page, reports on frequent queries, text processing and summarized reports on the web-pages visited at each host. The calculations that are involved in the operations thereof are simple to a high extent; however

the input and out data is enormously big. Therefore, Google breaks task down into smaller tasks, and splits them across thousands of computers in order to increase performance (Dean and Ghemawat [January 2008]).

Google performs map-reduce tasks in two different phases of *mapping* and *reducing*. During the mapping phase, several documents across different computers or data-bases are processed, and the intermediary <Key, Value> pairs are created. Google sorts intermediary results based on their **keys** for faster grouping and access of data in later stages. During the reducing phase intermediary results are integrated to form the final result. Google divides the reducing phase into two stages of *merging* and *combining*. During the merging stage, different sets of <Key, Value> tuples (i.e. map data-structures) are merged into one set. This stage may include various set operations, such as union, intersection or difference of the sets. During the combining stage, different values that are associated to the same key get combined. Programmers can specify their own merging and combining functions for the stages mentioned above (Lammel [2008]).

3.1.2 Hadoop

Hadoop is mainly focused on processing large amount of data across cloud networks. For this purpose, Hadoop uses parallel data-bases and map-reduce. Parallel data-bases perform well, as the approach is based on *shared-nothing* architecture; therefore there is not a single point of contention between the computers in a system. However, this approach only scales well when there is less than one hundred nodes involved in a process. Moreover, parallel data-bases assume seldom failure in the system, thus the approach does not consider fault tolerance. More importantly, this approach assumes that all computers involved in the parallel processing are homogeneous, whilst this is nearly impossible in a large computer network (Abouzeid et al. [August 2009]).

Because of the limitations mentioned above, Hadoop extends Google's map-reduce approach for processing data on very large networks . Hadoop also exploits Distributed File Systems (DFS) for map-reduce operations in order to minimize data transfer, and the effects of node-failures in a cluster. That is, once a node-failure is detected, the tasks for that node are performed by other computers without any need for transferring data thanks to DFS (Abouzeid et al. [August 2009], Shneider [2014]).

However, one weakness of Hadoop map-reduce approach is that input and output data is read from and written to files. This fact imposes unnecessary overhead on operations for processing data streams, especially for intensive tasks that require working on large numbers of files. Therefore, Hadoop communicates with application servers for direct transmission of data in order to avoid files and improve performance. Instead, the application servers deal with input and output data streams (Shneider [2014]).

Hadoop mad-reduce operations are performed on <Key, Value> pairs, and are done through three different phases of *shuffle*, *sort* and *reduce*. The tasks that are accomplished at each phase can be listed as follows (Hadoop [2014]).

Shuffle. The outputs from all mappers are grouped and partitioned by the framework. Furthermore, each reducer receives its relevant partition via HTTP connections.

Sort. The inputs to a specific reducer are sorted and grouped based on their **keys**. The main purpose for this phase is to ensure that values returned from different mappers associated to the same key are grouped together before the actual reduction process. The first and second phases are done simultaneously.

Reduce. The actual reduction is done in this phase, where values that are associated to the same key get combined based on user specifications. That is, Hadoop provides a *Reducer* interface that must be implemented by the programmer in order to complete the third phase. An instance of *Reducer* provides a method called *reduce* that gets called once for each <Key, ListOfValues>

pair in the grouped input for each reducer. This factor is considered as one of the drawbacks of Hadoop.

Hadoop provides implementations for some common reduction operations, such as chaining multiple mappers and reducers, field selection, long summation and combining values associated to the same key (Hadoop [2014]).

3.1.3 Apache Spark

Apache Spark is a framework that provides high-level APIs for processing large data sets on computer clusters and standalone desktops. Spark is focused on inheriting Hadoop map-reduce strengths and improving the weaknesses of Hadoop that were mentioned in Section 3.1.2. In other words, Spark introduces an enhanced version of Hadoop file system called Resilient Distributed Data-set (RDD) to become faster than Hadoop (Spark [2015e]).

Apache Spark support Java, Scala and Python programming languages, and is able to work with all files storage systems that are supported by Hadoop. Moreover, Spark is the base of four frameworks that utilize parallel processing in different aspects. The frameworks are listed as follows.

Spark SQL. Spark SQL is a framework that facilitates fast parallel processing of data queries over large distributed data sets. For this purpose, Spark uses a query language called HiveQL which is very similar to SQL (Spark [2015f]).

Spark Streaming. Spark Streaming allows programmers to stream tasks by writing batch-like processes in Java and Scala. The framework enables integration between batch jobs and interactive queries (Spark [2015d]).

MLlib. MLlib is a framework for efficient parallelization of machine learning algorithms. The algorithms are optimized and proven to perform fast (Spark [2015b]).

GraphX. GraphX provides efficient mechanisms for analysis and iterative computation of graphs on a single system, as well as a cluster with RDDs (Spark [2015a]).

At the implementation level, Spark performs map-reduce operations on almost every parallel task. Programmers are able to specify their own map and reduce operations using *map*, *reduce* and *reduceByKey* methods. The functions that are supposed to be performed by the methods thereof could be specified by *lambda expressions* in Java8. However, for older versions of Java, the functional interface that is provided by *org.apache.spark.api.java.function* should be used, as they do not support lambda expressions (Spark [2015c]).

3.1.4 Scope

Scope is a framework that has been developed by a team of Microsoft engineers under C#. The main motivation for this project was to develop a model that hides the complexity of the underlying system that is involved in processing SQL queries, but at the same time provide enough flexibility for the programmers to define customized functionalities for the system. The authors believe that their proposed model is specifically helpful with application domains that involve detecting pattern changes over time and detecting fundamental trends in large sets of data.

The model introduces a new data type that is inspired by SQL data rows. Each instance of the proposed data type contains a set of rows, and each row consists of columns of different types. However, the new data type highly resembles the concept of map collections in Java where collections can be used to associate various types to a key, and keys play the role of indices for the rows. Hence, the data model in this framework follows the same <Key, Value> pattern as the ones mentioned earlier.

As it was mentioned, The model provides the flexibility for the programmers to specify their own map-reduce operations. In order to do so, a programmer must provide implementations for *extractor*, *processor*, *reducer* and *combiner* functions. The tasks that each of the functions are responsible for can be listed as follows (Chaiken et al. [August 2008]).

Extractor. Extracting data from different documents, and converting them into data rows that conform the specified data format of the framework.

Processor. Processing the rows of data by different mappers in parallel, and providing the intermediary results.

Reducer. Grouping the intermediary data-rows based on user specifications, and sending the grouped data to reducers.

Combiner. Combining each group of data-rows based on user specifications.

The framework has provided ready-to-use implementations for some of the commonly used map-reduce approaches, such as *projection* and *selection*. The ready-to-use implementations are provided to avoid error-prone user code, and to encourage code reuse. This framework is intertwined with another framework developed by Microsoft, which is called *COSMOS*. COSMOS is specifically used for analyzing large data sets in big data clusters (Chaiken et al. [August 2008]).

3.1.5 Simplified Relational Data Processing

This research project was conducted by Yang et al. [June 2007] in order to propose a more efficient map-reduce approach for parallel processing of large relational data-sets. The authors believe that the conventional map-reduce approach does not ease processing heterogeneous data which is very common in relational data-sets. Furthermore, they argue that considering relational algebraic operations (e.g. union and intersection) helps with providing more comprehensive data, thus speeding up the performance.

Therefore, the research proposes a model that adds a *merging* phase to map and reduce. The main focus of this research is on separation of concerns in order to ease implementation, as well as improving performance. According to this model the tasks for the map, reduce and merge phases can be specified as follows.

Map. User defined logic is applied to <Key, Value> data pairs in order to create the intermediary <Key, Value> pairs.

Reduce. All data pairs with the same **key** are grouped together. Furthermore, each group of data is sent to a reducer.

Merge. At each reducer different values that are associated to the same key are merged in order to form a <Key, ListOfValues> pair as the final result. The model also suggests that the key in the final result can be of a different type from the initial one. However, this definition is in contradiction with the definition that is suggested by most other related works.

3.1.6 Summary

Considering contemporary projects and research studies in this field – from which we mentioned a few in this section – most studies are unanimous about the following principles.

- Using <Key, Value> data pairs in order to allow more flexible, or even heterogeneous, data processing.

- Including relational algebraic operations (e.g. union and intersection of sets) in order to allow more complex reductions.
- Considering separate and independent stages for grouping <Key, Value> data-sets returned by different mappers and combining the values that are associated to the same **key**.
- Providing implementations for common map-reduce operations in order to avoid error-prone programmer code as much as possible.
- Encouraging quality software designs that ease code reuse and code modification.

In the next section we have explained an existing map-reduce approach for shared-memory applications. Furthermore, in Section 4 we have discussed accommodating the principles thereof in our design.

3.2 Shared-Memory Applications

Shared-memory applications run on desktop computers, where memory resources are shared between different components of a system. Applications in this category have to consider concurrency principles (e.g. thread safety) when accessing memory. Therefore, map-reduce operations has recently become an interest in share-memory applications. In this section we have introduced some of the frameworks that support map-reduce operations in this categories.

3.2.1 OpenMP

The map-reduce operations that have been discussed thus far are specifically focused on network-based operations. That is, a big task is broken down into smaller tasks and sent to different computers across a network for execution. Furthermore, the results returned from the computer are integrated in parallel reducers until the final result is achieved.

Considering that parallel processing and concurrent applications on single computers follow the same principle of dividing workload between processors, the concept of map-reduce has recently become an interest on single desktops as well (Lin et al. [April 30 2010]).

Despite the potentials for exploiting map-reduce operations in shared-memory (i.e. single computer) applications, many programming languages have not considered providing adequate support for this concept yet. OpenMP is one of the few APIs that has been supporting reduction as part of its support for parallel processing in Fortran and C/C++ (OpenMP [July 2013]). In the following paragraphs of this section we have have discussed the reduction clauses in OpenMP.

Reduction Clauses

OpenMP provides compiler directives to help programmers to specify blocks of sequential code that they would like to parallelize. Subsequently, OpenMP compiler automatically converts that sequential block into a parallelized region with efficient concurrent handling of the resources that are used in that block.

On the other hand, the results that are returned from parallel threads require a mechanism to integrate them into the final result. Therefore, OpenMP allows programmers to use reduction clauses in their directives in order to specify how the final result should be calculated. Reduction clause in OpenMP have the following syntax (BlaiseBarney [2014]).

```
reduction(operator:list) e.g.  reduction(+:var)
```

In the example above, **var** is a shared variable in a parallelized region. Every thread that executes the parallelized code, receives its own private copy of **var** where it saves the result of its operations. Once all threads have finished execution, OpenMP integrates each thread's copy of **var** using the **operator** specified in the reduction clause (i.e. +), and stores the final result in the global version of

var. However, there are limitations on both the operator and the variables that can be specified in a reduction clause. These limitations can be listed as follows (BlaiseBarney [2014]).

- Variables that are used in a reduction clause must be declared as *shared* for the parallel region.
- Variables must be scalar, and cannot be any type of a data-structure or a reference to an object.
- The operator can only be one of the internally defined operators (e.g. +, -, *, /, &, ^, |, &&, ||).
- Reduction operations can only be used in *work-sharing* regions.

As it has been mentioned above, reduction operations are very limited to basic predefined operators that are provided by the programming language. However, since the release of OpenMP 4.0 the API supports the concept of *reduction identifiers*, which allows programmers to define their customized reduction approach (OpenMP [July 2013]). The following paragraphs explain this feature into more details.

Reduction Identifiers

Since OpenMP 4.0 the API provides a directive for declaring new reduction identifiers which can be used in reduction clauses later in the code. Reduction identifiers act as user-defined operators which use customized reducing approaches. The syntax for defining a new reduction identifier is as follows.

```
#pragma omp declare reduction (reduction-identifier: typeName-list: combiner) [initializer-clause]
```

In the syntax above, **reduction-identifier** is the name of the new reduction operator that is intended to be used in reduction clauses. A reduction-identifier can be either a base language identifier, or any of internally defined operators (e.g. +, -, *, /, &, ^, |, &&, ||).

Furthermore, **typeName-list** is the list of all data-types that can be reduced by this reduction approach. Finally, **combiner** is a logic expression that specifies the reduction approach for combining partial results.

Reduction identifiers have remarkably increased the flexibility of reduction clauses in OpenMP. However, there are still limitations that prevent this feature of OpenMP from being completely versatile. These limitations can be listed as follows (OpenMP [July 2013]).

- A customized reduction identifier can only be used in the same code as it is declared in.
- The data type used by a reduction identifier cannot be a data structure (e.g. list), or a reference to an object.
- Only **omp_in** and **omp_out** variables can be used within the **combiner** expression. Similarly, the **initializer** clause can only use **omp_priv** and **omp_orig** variables.
- Reduction identifiers are still not supported by some C/C++ compilers.

Compensating the limitations mentioned above can strongly encourage programmers to use the feature more often. That is, providing a mechanism through which a programmer can reuse their customized reduction approach will encourage investing more time for optimized implementations. Moreover, being able to use reduction in areas that are not necessarily *work-sharing* will increase the versatility of this feature.

Furthermore, the constraints that exist on the variables that can be used in the combiner and initializer clauses can cause confusion and erroneous implementations. That is, according to OpenMP documentation initializing or modifying variables in an incorrect region can lead to undefined behavior of a custom reduction identifier (OpenMP [July 2013]). Thus, enabling programmers to define and use their own variables in their implementations will ease creating clearer and more understandable codes. In the next section, we have discussed our design, and have explained our considerations for avoiding the limitations that exist in OpenMP.

4 RedLib

Parallel Iterator is an ongoing project conducted by the Parallel and Reconfigurable Computing group at the University of Auckland. Parallel Iterator aims at providing a convenient mechanism for parallelizing java loops and parallel iteration through java collections (see ParallelIT [2014]). Parallel Iterator provides a comprehensive reduction library in order to facilitate integrating the partial results returned from processes (or threads) that run in parallel.

The development of such a library was motivated to fulfill the requisite of map-reduce operations under Java, considering that Java support for reductions is very primitive (Java [2014]). Moreover, the final goal is to implement a reduction library for shared-memory applications that is as powerful as the frameworks available for distributed-memory applications (e.g. Hadoop, Google etc.), considering their quality principles which we mentioned in Section 3.1.6. Furthermore, the object-oriented pattern proposed for our design avoids the limitations of reduction identifiers that were mentioned in Section 3.2.1 due to the procedural nature of OpenMP. In the following paragraphs we have elaborated how these considerations are addressed in our implementation.

4.1 Modifiability and Reusability

The library provides a class called **Reducible** which is based on a similar concept to ThreadLocal in *java.lang* (Oracle [2014]). The Reducible class provides each thread with its own copy of the initialized variable to which the final result is submitted. Threads can only manipulate their own copy of the initial value through the provided *get* and *set* methods. Finally, when every thread has finished its tasks the *Reducible* object reduces the results based on the instance of *Reduction* that is passed to its constructor. Therefore, every reduction class provided by the library is an implementation of the base **Reduction** interface. This mechanism ensures that further custom implementations provided by programmers follow the same standard. The following code snippet presents the body of our base interface.

```
public interface Reduction<E>{
   public E reduce (E first, E second);
}
```

This interface allows implementing very basic reduction operations, as well as extending an implementation such that it fulfills complex reduction tasks. That is, simple reductions can be reused within the structure of complex reductions while keeping the implementation understandable, easy to modify and very flexible. We have proposed this feature as **nested reductions** which is discussed into more details in Section 4.5.

As a matter of fact, modifiability is one of the greatest concerns in every software architecture (Paul et al. [2003]). Using low-level reductions inside high-level implementations helps programmers to concentrate on implementing different stages at a time, and easily modify minor parts of the functionality without affecting the main body and vice versa. Moreover, the proposed mechanism encourages programmers to avoid code duplicate by reusing the reduction classes, and design reusable implementations of the interface.

4.2 Ready-to-Use Reductions

RedLib offers a considerably large range of ready-to-use implementations of the Reduction interface. As a matter of fact, implementations in RedLib can be divided into two main categories of *high-level* and *low-level* reductions. Low-level reductions involve operations on primitive data types and containers (i.e. sets and collections). Some of the low-level implementations can be listed as follows.

- sum, subtraction, multiplication, average, minimum, maximum operations for Integer, Long, Short, Float, Double, BigInteger and BigDecimal data types.
- BitWiseAND, BitWiseOR and BitWiseXOR operations for Boolean, Byte, Integer and Short data types.
- AND, OR, XOR operations for Boolean data type.
- Union, Intesection and Difference operations for Collection and Set data types.

High-level reductions involve Union, Intersection and Difference operations on Map data types. Classes in this category implement reductions in two stages. First, grouping elements that are associated to the same key, and second, reducing those elements into the final result. High-level reductions can use low-level reductions for the second stage, and form a comprehensive range of combinations. This concept is proposed as nested reductions, which is discussed further in Section 4.5.

4.3 Support for Various Data Types

Map-reduce operations are implemented on a wide range of tasks; therefore they involve noticeable variety of data types. On the contrary to OpenMP, we suggest using generic types for the base interface as well as all of our implementations, to allow programmers to flexibly fit reduction operations within their codes without any type restrictions.

Moreover, considering that map-reduce operations are dominantly based on search and query tasks (HighlyScalableBlog [February 2012]) we have provided implementations for <Key, Value> data pairs beside the basic implementations.

4.4 Ease of Use

The library is meant to easily integrate with programmers' codes, as well as IDEs such as Eclipse without the complexity of frameworks such as Hadoop. As a matter of fact, running Hadoop map-reduce applications on a single computer requires installing a third party software (i.e. Hadoop single node cluster), and involves a number of configuration steps that could appear confusing in the first approach. The hurdles become even stronger when one intends to develop their Hadoop map-reduce project in an IDE such as Eclipse – which is the prevailing Java IDE – since Hadoop has no official plug-ins, thus it requires programmers to do manual setups (Prakash [July 2014]).

Furthermore, in a Hadoop project programmers need to follow concrete conventions for inputs, outputs, mappers and reducers of a system. For instance, the input to and the output from a Hadoop map-reduce application are always in the form of files (Pandya [December 2013], Prakash [July 2014]). However, in many cases the final result from a group of computational tasks is ideally returned to the main thread and used for further processes.

Reduction classes provided by Parallel Iterator can be used anywhere in a programmer's code, free of the limitations of OpenMP and needless of the preliminary setup procedures of Hadoop. Reduction operations are performed as the result of simple method calls on Reduction objects, without requiring extra classes or boilerplate codes on the contrary to Hadoop and OpenMP (Prakash [July 2014], OpenMP [July 2013]). Moreover, We have strictly avoided data sharing in our implementations of the *Reduction* interface; therefore reduction operations can be used in both sequential and parallel regions as opposed to OpenMP reduction identifiers. However, the specifications thereof may not apply to user-defined implementations of the interface.

Furthermore, following the recommendations in Section 3, the library also provides complex-reduction implementations that involve relational algebraic operations (e.g. union and intersection) on maps, as it was . Complex reductions can nest basic or complex reductions in order to ease reduction

operations that could be quite confusing and error-prone to implement otherwise. The examples in Section 4.5 clarify the advantages of our model.

4.5 Nested Reductions

Most of map-reduce frameworks perform their reductions in two stages. That is, during the first stage, the <Key, Value> pairs returned from mappers are **grouped** based on their keys, and then the values for each group are **combined** into the final result during the second stage (see Section 3). For this purpose we have proposed the concept of **nested reductions**, where a programmer can use an instance of Reduction within another implementation of the Reduction *interface*.

For example, class **IMP** which is an implementation of the Reduction *interface* receives object **R1** (an instance of Reduction) via its constructor. Class **IMP** is in charge of grouping data based on user specifications; meanwhile **IMP** uses **R1** to reduce every two elements that can be grouped, into one element. The process continues until all elements that belong to the same group are reduced.

The frameworks that were discussed in Section 3, require users to implement separate functions or classes for grouping and combining. However, function blocks cannot be reused later, neither using different classes for grouping and combining allows flexible nesting of these stages.

Thus, we have proposed implementing both grouping and combining stages simultaneously, but keeping them (i.e. grouping and combining) as separate concepts at the same time while the entire implementation is based on a single interface (i.e. Reduction interface). Therefore, the simplicity and understandability of the design is improved, as independent objects are exploited for each stage. The following example clarifies the advantages of this approach.

4.5.1 Example 1

MapUnion is one of the implementations provided by the library, which renders the union of two different maps (i.e. sets of <Key, Value> pairs). The specified implementation for this class looks like the following pseudo-code.

```
public MapUnion implements Reduction<Map> {
   private Reduction   reducer;
   MapUnion (Reduction r) {reducer = r;}
   public Map reduce (Map m1, Map m2){
     // for every key in m2, if the same key exists
     // in m1, then reduce V1 and V2 using reducer,
     // and put the result back into m1.
     // Otherwise, add <K2, V2> to m1.
   }
}
```

For instance, in a specific case we would like to find the links in which words *insomnia* and *matador* could be found. For this purpose, we use two parallel threads to search through a bunch of documents separately, and return their own map of <Word, Link> data pairs. Finally, we instantiate an object of MapUnion, and send an instance of SetUnion to its constructor. SetUinon is another instance of Reduction, and provides the union of the values (i.e. sets) that are sent to it (see Figure 1).

Now if we want to count the average number of occurrences for the words *insomnia* and *matador* in the documents that are inspected by the threads, all we need to change is the Reduction instance that we send to the constructor of MapUnion. Thus, for this case we send an instance of FloatAverage to the constructor. FloatAverage is an instance of Reduction which calculates and returns the average of two float values (see Figure 2).

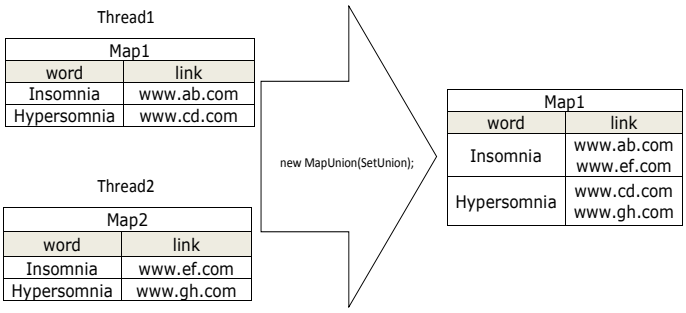

Figure 1: MapUnion with SetUnion as its nested Reduction

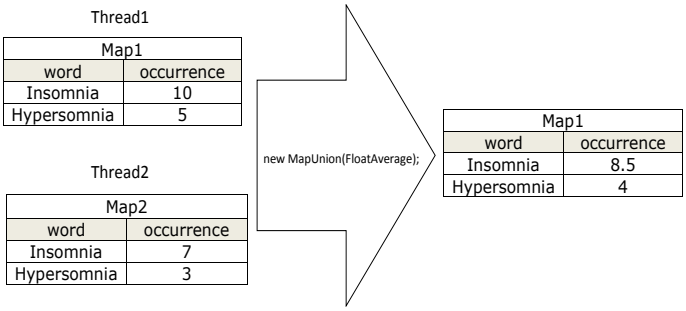

Figure 2: MapUnion with FloatAverage as its nested Reduction

The main advantage of nesting reductions is that, very complex reductions which may require reduction operations in multiple levels, can be implemented easily while the probability of confusion and error is minimized. Furthermore, the number of reduction classes can be kept at minimum, while their combinations result in many different reduction approaches.

For example, *MapUnion* and *MapIntersection* are implemented only once; however their combinations with lower-level reductions (e.g. SetUnion, FloatAverage etc.) can form a wide range of complex reduction operations. Similarly, MapUnion or MapIntersection can be nested into higher-level implementations for reducing map of maps, or even more complex operations which are application-specific; therefore we have not provided implementations for them.

4.5.2 Example 2

Lets consider a use-case in which two threads are engaged to explore a graph separately. Each thread explores the connection between the nodes (i.e. edges), and the cost for transferring between one node to the other (i.e. weights of edges). At the end the results returned from the threads are merged, and the edges that are common between both results take the minimum weight. Figure 3 demonstrates the process involved in this use-case.

A closer look at the implementation of this use-case suggests using map of maps as the data structure which relates each node to its edges and the weights of those edges. That is, the outer map relates each node to an inner map that relates an edge to its weight. In order to figure out the final result, the maps must be reduced through three steps (see Figure 4).

Implementing this procedure in one stage can be cumbersome, error-prone and hard to modify. However, breaking the implementation down into separate stages makes it clearer and more flexible.

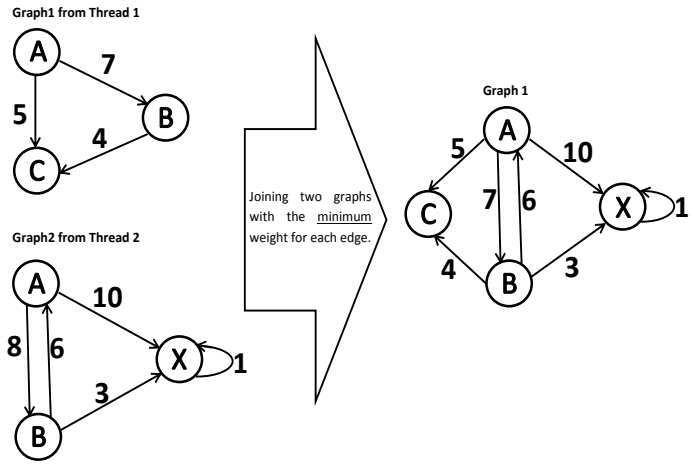

Figure 3: Merging two graphs into one

For this example, the first stage unifies the outer maps, such that all nodes explored by both threads are included once in the final map. The second stage unifies the inner maps, such that all edges of a node explored by both threads are included once. Finally, the third stage takes the minimum weight for the edges that are commonly explored by both threads.

The reduction approach in ParallelIterator enables implementing these stages by nesting three different instances of Reduction. In this approach the latter reductions are nested in the former reductions. That is, an instance of *IntegerMinimum* is nested in an instance of *MapUnion<String, Integer>*. IntegerMinimum reduces two integer values into the minimum one, and is used to reduce the weights that are associated to the same edge when the union for two instances of *Map<edge, weight>* is calculated. Furthermore, this instance of MapUnion is nested into an instance of *MapUnion<String, Map<String, Integer>>* to reduce the maps of edges that are associated to the same node when unifying two instances of *Map<node, map-of-edges>*.

Thus, if we need to take the maximum value for an edge in the third stage, or to calculate the intersection of the edges associated to the same node in the second stage, we only need to change the instances of nested reductions. Therefore, the behavior of the reducer can be flexibly changed without requiring any specific changes to the actual implementations. Figure 4 clarifies the steps involved in this procedure.

4.6 Performance

Beside the implementation aspects discussed in this section, performance principles have been considered in the design as well. That is, reduction classes in ParallelIterator occupy the minimum possible memory space, because the classes do not use state variables or shared data. For the same reason, a reduction instance can be created only once, and used by different threads without concerns about thread safety.

Moreover, in our approach objects are passed to reducers only once, whereas in other approaches such as Hadoop reducer methods are called once for every pair of data in a collection (Hadoop [2014]). Therefore, applications such as Hadoop map-reduce impose considerable overheads on the system for method calls and transferring and casting data objects. Furthermore, as we discussed in Section 3.1.2 Hadoop applications need to process input and output data in the form of file streams. These inefficiency factors become even more noticeable when the amount of data to deal with grows bigger, and become discouraging enough for using Hadoop in shared-memory applications.

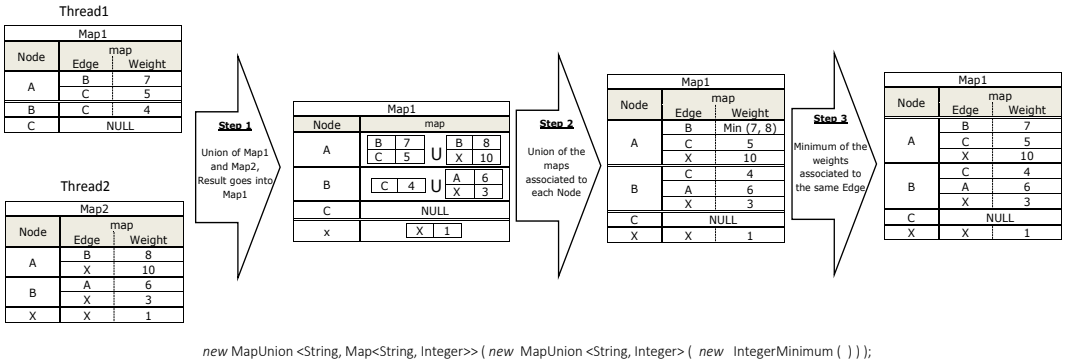

Figure 4: Unifying map of maps using nested reductions

Thus, avoiding the inefficiency factors of Hadoop and OpenMP (see Section 3.2.1) alongside other concepts such as nested reductions and the Reducible class make our reduction library lightweight and scalable. That is, the library will not impose extra overhead on the implementation; therefore it will not degrade performance as the complexity of implementation increases. In the next section we have discussed the benchmarks that we implemented in order to empirically examine the theoretical arguments that we mentioned in this section.

5 Benchmarks

Performance has always been the main focus in concurrent and parallel processing applications. However, parallelization comes with its own concerns regarding safe manipulations of shared data. Therefore, these techniques involve critical thread-safety and synchronization mechanisms that impose some overhead, and if not implemented efficiently, can counteract the benefits of parallelization. In RedLib, the implementations that are provided for the Reduction interface have minimized data-sharing areas, and we inspected if our approach would help keeping performance close to its expected level.

Thus, we implemented a PDF processor that given a list of string patterns, it would search through a predefined list of PDF files and would return the number of occurrences for each string pattern. The processing times were measured with different numbers of threads, and were used to estimate the overhead of our implementation having our Reduction classes incorporated in it.

As a matter of fact, using a merely computational task without any I/O processing would provide a clearer vision about the performance of our benchmark. However, we decided to simulate a more practical case, in order to provide more realistic measures for the efficiency of RedLib. The following paragraphs describe the details of our experiments.

5.1 Experimental Setups

The benchmarks were developed under Eclipse Kepler IDE, and the iText framework was used for processing and parsing PDF files. As a matter of fact, there are several frameworks for processing PDF files in java, from which we can name JPedal, Apache Lucene and iText. JPedal is a comprehensive library for processing PDF files and converting them to different formats; however the library is not available for free (JPedal [2014]). Lucene is also a fast and full-featured library provided by Apache for searching and processing texts. The API for Lucene is not easy enough to use, such that Apache has

implemented another framework called Solr, which provides a higher level API, but works with Lucene on the background. However, Apache Solr is a web-based framework, and cannot be used for desktop applications (Grainger and Potter [2014]). On the other hand, iText is a lightweight framework that provides simple and easy-to-use methods, which can be easily incorporated into ordinary java projects, and is ideal for processing PDF files in desktop projects (Lowagie [2011]).

We ran the benchmarks on two systems, both with Linux (Ubuntu) 64-bits as their operating systems, but with different CPU specifications. One of the systems was a 16-Core remote server using four Quad-Core Intel Xeon E7340 processors, with maximum 2.4 GHz CPU cycles and 64 GB of ram memory. The other system was running on an i7-4470 Quad-Core Intel processor, with maximum 3.40 GHz CPU cycles and 8 GB of ram memory. We leveraged the difference between the processors to obtain accurate estimations about the efficiency of our Reduction classes by comparing the actual performances of the systems when using RedLib, with the expected performance for each processor.

That is, the first CPU architecture does not support hyper-threading, thus it cannot run more than one thread on a physical CPU core at a time. On the other side, the second CPU architecture supports the technology; therefore each physical CPU core could be used as two logic cores, and run two threads in parallel at a time (Intel [2015]). However, every two logic cores are still managed by one physical core in the background. Therefore, logic cores introduce their own overhead on the system, and they do not improve performance as much as physical cores do.

Moreover, the overheads from other factors such as converting and streaming PDF files, as well as managing and scheduling tasks would slow the process down. Therefore, speedups were not expected to be analogous to that of absolute computational tasks. Considering the specifications thereof, we expected the performance to increase proportionate to the number of threads, but not in a linear fashion. Because, as the number of threads would increase, the overhead of multi-threading would increase as well. Thus, performance gain would be counteracted by the overheads to some extent. Moreover, switching to the logic cores in the Quad-Core system was expected to result in a smaller speedup due to the overheads that we discussed above.

5.2 Methods

The search operations involved processing 32 PDF files for the 16-Core and 16 PDF files for the Quad-Core, in order to guarantee that the workload for each core will compensate the overheads of PDF streaming and multi-threading. In other words, we performed the benchmarks with 8 PDF files for both systems initially; however the overheads imposed by multi-threading and PDF streaming outweighed the workload for each core, and processing files would take a very short time such that the speedups with respect to the sequential mode were very trivial.

Experiments were performed with single thread, two, four, eight and sixteen threads in the 16-Core system. Similarly, we performed the experiments with single thread, two, four and eight threads in the Quad-Core system. Considering that our experiments were replicated under reasonably constant environmental conditions, we required three to ten replications to achieve accurate measurements (SEMATECH [April 2012]). Therefore, we replicated our experiments for ten times in each system.

The experiments were run in random order of thread-numbers, in order to avoid possible caching effects. For the same reason, we ran each experiment on a separate JVM run to avoid side effects of JVM warmup and garbage collection. The PDF files were identical copies of a PDF document. The size of each document was big enough (i.e. 17MB) to ensure enough workload for each core, such that the overhead of multi-threading and PDF streaming was outweighed. It should be mentioned that we performed the experiments with smaller files (i.e. 2MB) at the fist stage. However, the workload for each core was too small, such that we were not able to observe noticeable difference between the performances under different numbers of threads.

5.3 Results

The results demonstrate that performance becomes approximately two times faster when two threads are used rather than a single thread. Similarly, performance improvement is continued when the number of threads is increased to four, and then to eight. However, using sixteen threads did not make remarkable changes to the performance, even in some cases slight performance degradations can be observed. Tables 1 and 3 clarify close behavior of the system with both 32 and 16 documents. Furthermore the trends shown in Figures 5 and 6 demonstrate the performance has gained slightly better improvements with 32 document. This fact confirms our hypothesis about using bigger files for balancing parallelization overheads with workload.

Moreover, Figures 7 and 8 demonstrate that system performance under the Quad-Core processor continuously improves by proceeding to higher number of threads. However, similar to the 16-Core system, we can observe that once the number of threads has reached the maximum number of cores, performance becomes slower..

5.4 Discussion

The results demonstrated in the previous section confirm that the actual performance of both systems comply with our expectations. Especially, when the number of threads is increased within the boundaries of physical cores, the performance is proportionate to the number of cores. However, switching to logic cores when using hyper-threading on i7 shows very slight improvements. Because, every two virtual cores are still managed by one physical core in the underlying level, thus even though concurrency is improved, the overheads of the logic cores impose some extra processing on the physical cores, such that performance cannot improve proportionately.

It should be remarked that during the experiments we were monitoring the cores in both systems. We observed that while there were free cores which did not have PDF searching tasks assigned to them, processing was faster and smoother. Moreover, we realized that the free cores would still do small and minor jobs which were presumably for scheduling and managing the cores allocated to PDF search tasks. However, when PDF search tasks were assigned to all cores in a system (including logic cores), performance would become slower with more lagging. That is, minor jobs for scheduling and managing threads had to be done by the same threads which were processing PDF files, and this would impose considerable penalties on performance.

Thus, the performance of the processors comply with our theoretically expectations that were discussed in Section 5.1. Therefore, considering that throughout the implementation, Reduction classes were created for each thread, increasing the number of threads would result in more Reduction classes. Therefore, if the reducers had imposed excessive overhead at runtime, the benchmarks would have deviated from the expected performance as the number of threads was increased. However, all attempts that were run under both processors consistently followed our expectation, and confirm the lightweight and effective implementations provided by RedLib. The tables and figures that are provided in the following pages provide the details of our experiments in both systems.

No. Threads	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6	Exp7	Exp8	Exp9	Exp10
1	135.8	135	136.15	135.35	135.6	134.9	136.3	139.9	136.31	134.33
2	77.44	72.62	79.62	73.06	73.9	72.2	74.7	72.25	72.5	72.67
4	45	41	46.8	46.55	46.12	40.5	46.52	43.48	46.42	44.84
8	33	32.9	32.5	32.4	31.7	32.71	31.64	31.57	31.76	31.3
16	34.7	34.48	34.74	34.02	33.5	33.9	34.12	33.52	33.21	33.17

Table 1: The Results for 16-Core Processor – 32 Documents (Seconds)

No. Threads	Std. Deviation	Medians	Means	Speedups (Median)	Speedups (Means)
1	1.45	135.7	135.96	1	1
2	2.39	72.86	74.1	1.86	1.83
4	2.21	45.56	44.72	2.98	3.0
8	0.59	32.08	32.15	4.23	4.23
16	0.55	33.96	33.94	4.0	4.0

Table 2: Standard Deviation, Medians, Means and Speedups for 16-Core – 32 Documents

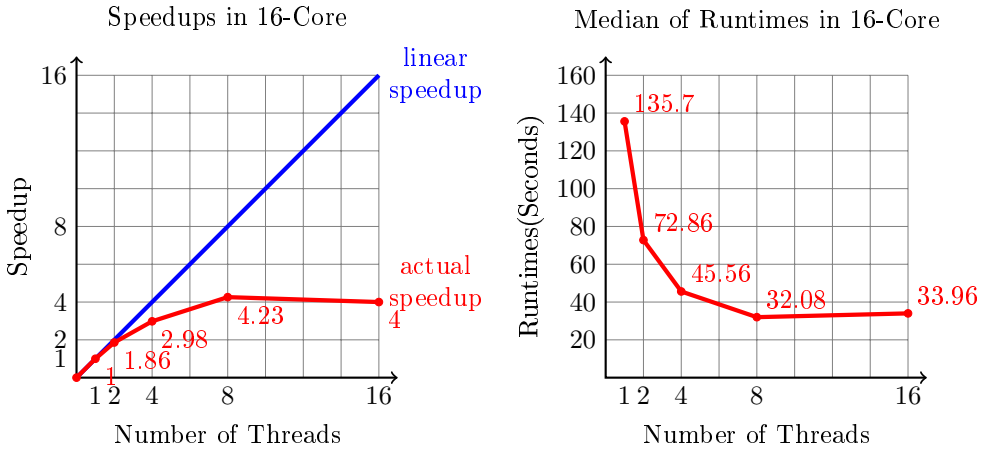

Figure 5: Trends of Speedups and Runtimes in 16-Core – 32 Documents

No. Threads	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6	Exp7	Exp8	Exp9	Exp10
1	71.41	70.8	71.16	71.44	72.18	71.14	70.43	71.19	71.06	71.13
2	38.36	38.61	38.93	38.5	39.43	39.46	43	41.82	39.11	38.7
4	25.02	22.6	25.09	25.11	25.42	25.02	24.8	23.48	22.4	23.1
8	18	17.67	17.97	17.76	17.65	17.83	17.63	17.58	17.5	17.76
16	19.07	18.67	19.2	19.87	19.23	19.41	19.58	19.87	19.2	19.46

Table 3: The Results for 16-Core Processor –16 Documents (Seconds)

No. Threads	Std. Deviation	Medians	Means	Speedups (Median)	Speedups (Means)
1	0.43	71.15	71.19	1	1
2	1.47	39.02	39.6	1.82	1.8
4	1.11	24.91	24.20	2.86	2.94
8	0.15	17.71	17.73	4.0	4.01
16	0.35	19.32	19.36	3.68	3.68

Table 4: Standard Deviation, Medians, Means and Speedups for 16-Core – 16 Documents

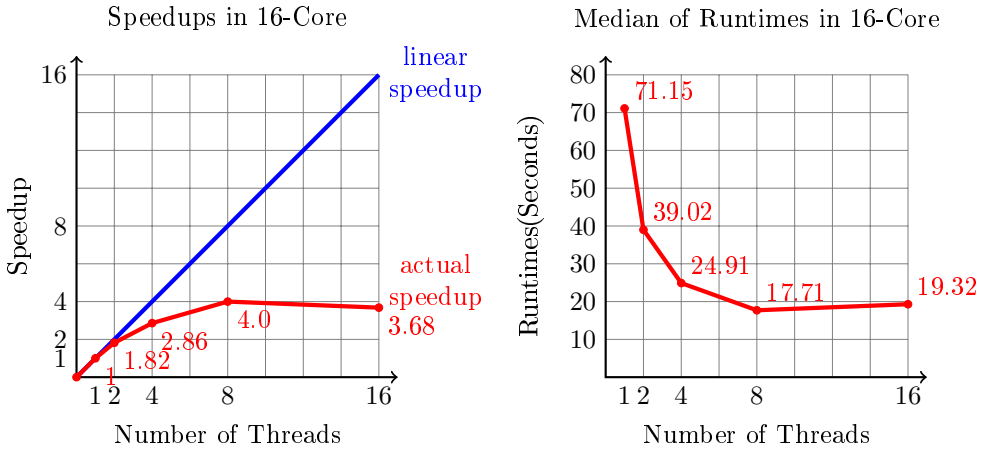

Figure 6: Trends of Speedups and Runtimes in 16-Core – 16 Documents

No. Threads	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6	Exp7	Exp8	Exp9	Exp10
1	27.25	27.62	27.94	27.89	27.83	27.84	27.9	27.5	27.9	27.88
2	15.56	15.52	15.16	15.82	16.16	15.08	16.22	15.91	15.9	15.91
4	11.14	11.01	11.04	11.37	11.21	11.69	11.08	11.19	11.15	11.19
8	10.86	10.86	11.02	10.97	10.83	10.86	10.93	10.96	10.91	10.89

Table 5: The Results for Quad-Core Processor – 16 Documents (Seconds)

No. Threads	Std. Deviation	Medians	Means	Speedups (Median)	Speedups (Means)
1	0.21	27.86	27.56	1	1
2	0.37	15.86	15.72	1.76	1.77
4	0.19	11.17	11.2	2.49	2.48
8	0.06	10.9	10.9	2.55	2.54

Table 6: Standard Deviation, Medians, Means and Speedups for Quad-Core – 16 Documents

No. Threads	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6	Exp7	Exp8	Exp9	Exp10
1	14.84	14.82	14.83	14.63	14.93	14.82	14.53	15.1	14.6	14.72
2	8.44	8.72	8.4	8.7	8.2	8.91	8.98	8.24	9.43	8.78
4	6.57	6.14	6.4	6.4	6.73	6.41	6.06	6.3	6.58	6.31
8	6.05	6.48	6.19	6.4	6.38	6.42	6.42	6.49	6.23	6.23

Table 7: The Results for Quad-Core Processor – 8 Documents (Seconds)

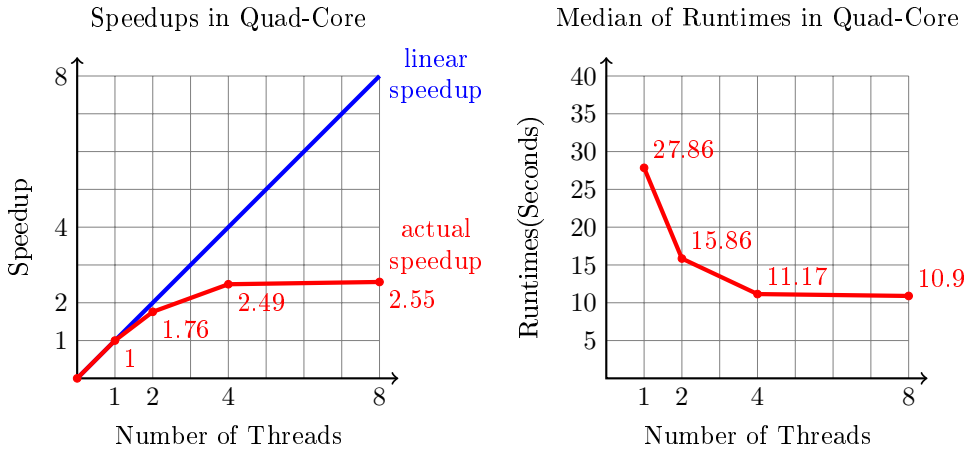

Figure 7: Trends of Speedups and Runtimes in Quad-Core – 16 Documents

No. Threads	Std. Deviation	Medians	Means	Speedups (Median)	Speedups (Means)
1	0.16	14.82	14.78	1	1
2	0.36	8.71	8.68	1.7	1.7
4	0.19	6.4	6.39	2.32	2.31
8	0.14	6.39	6.33	2.32	2.34

Table 8: Standard Deviation, Medians, Means and Speedups for Quad-Core – 8 Documents

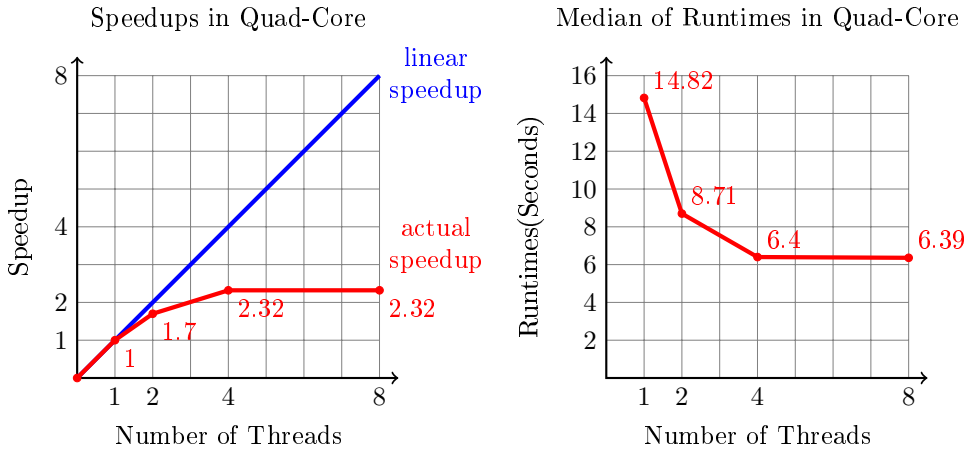

Figure 8: Trends of Speedups and Runtimes in Quad-Core – 8 Documents

6 Conclusion

In this paper we fulfilled the following objectives.

- Summarized some of the commonly used reduction operations and their use-cases.
- Introduced some of the current applications and projects that use map-reduce for their operations, and summarized their common methods and preferences.
- Overviewed OpenMP as a framework that supports map-reduce for desktop applications, and summarized its advantages and its drawbacks.
- Introduced a lightweight reduction library that is implemented by the Parallel and Reconfigurable Lab at the University of Auckland.
- Theoretically discussed the advantages offered by our reduction library, and further explained how the pitfalls of frameworks such as OpenMP and Hadoop are avoided in our implementations.
- Proposed a new concept of flexible and nestable reductions, and clarified its advantages with intuitive examples.
- Provided empirical benchmarks to practically prove our theoretical arguments.
- Consolidated the claim that our reduction library helps with preventing boilerplate and erroneous code, without any negative effects on the performance.

References

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoop db: An architectural hybrid of mapreduce and dbms technologies for analythic workloads, yale university and brown university, u.s. *ACM VLDB '09*, pages 922–933, August 2009.

Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a mapreduce environment, stanford university of u.s. & national technical university of athenes. *ACM EDBT*, 2010.

BlaiseBarney. Openmp reduction clauses, retrieved on Nov. 2014. Lawrence Livermore National Laboratory, available at: `http://computing.llnl.gov/tutorials/openMP/#REDUCTION`, 2014.

Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment 1.2*, 08:1265 – 1276, August 2008.

Jeffrey Dean and Sanjay Ghemawat. Map reduce: Simplified data processing on large cluster. *Communications of the ACM*, 51(1):107–113, January 2008.

Trey Grainger and Timothy Potter. *Solr In Action*. Manning Publications, 2014.

Hadoop. Hadoop reduction api. Available at Hadoop website: `http://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/mapred/Reducer.html`, 2014.

HighlyScalableBlog. Map reduce patterns, algorithms, and use cases. Available at: `http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/`, February 2012.

Intel. Hyper threading processors. Available at: http://ark.intel.com/search/advanced/?s=t&HyperThreading=true, 2015.

Java. The java tutorials – aggregation operations - reduction. Available at: https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html, 2014.

JPedal. Java pdf library. Available at: https://www.idrsolutions.com/java-pdf-library/, 2014.

Ralf Lammel. Google's mapreduce programming model – revisited. *ELSEVIER Sience of Computer Programming*, 70:1–30, 2008.

Jimmy Lin, Chris Dyer, and Graeme Hirst. *Data Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan and Claypool Publisher, April 30 2010.

Bruno Lowagie. *iText In Action, Second Edition*. Manning Publications, 2011.

OpenMP. *OpenMP Application Program Interface*. OpenMP, version 4.0 edition, July 2013.

Oracle. Class threadlocal documentations. Available at: http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html, 2014.

Hardik Pandya. Running hadoop mapredce application from eclipse kepler. Available at: http://letsdobigdata.wordpress.com, December 2013.

ParallelIT. University of auckland, parallel and reconfigurable computing lab. Available at: http://homepages.engineering.auckland.ac.nz/~parallel/ParallelIT/PI_About.html, 2014.

Clements Paul, Rick Kazman, and Mark Klein. *Evaluating Software Architecture*. Tsinghua University Press, 2003.

Praba Prakash. Apache hadoop for windows platform. Available at: http://www.codeproject.com/Articles/757934/Apache-Hadoop-for-Windows-Platform, July 2014.

SEMATECH. Engineering Statics e-Handbook. Available at: http://www.itl.nist.gov/div898/handbook/, April 2012.

Robert D. Shneider. *Hadoop Buyer's Guide*. Ubuntu, 2014.

Apache Spark. Spark GraphX. Available at: https://spark.apache.org/graphx/, 2015a.

Apache Spark. Spark Machine Learning Library. Available at: https://spark.apache.org/mllib/, 2015b.

Apache Spark. Apache Spark Programming Guide. Available at: http://spark.apache.org/docs/latest/programming-guide.html, 2015c.

Apache Spark. Spark Streaming. Available at: https://spark.apache.org/streaming/, 2015d.

Apache Spark. Apache Spark: Lightning-fast Cluster Computing. Available at: https://spark.apache.org, 2015e.

Apache Spark. Spark SQL. Available at: https://spark.apache.org/sql/, 2015f.

Hung-Chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Stott Parke. Map-reduce merge: Simplified relational data processing on large clusters. *ACM*, pages 1029–1040, June 2007.