Reduction

We are aiming at extending the features provided by the current *Reduction* class provided in ***ParaTask*** library. Nowadays the technology of CPUs has reached a stage that increasing CPU clock cycles would be expensive or to some extent impossible. Therefore, new approaches for splitting tasks into sub-tasks and performing multiple tasks concurrently and in parallel are proposed for exploiting the CPU power more efficiently!

Consequently, splitting tasks into sub-tasks and computing them separately in parallel requires subsequent operations to be done on the results in order to integrate them into the single final result. This mechanism is called *reduction,* where a set of inputs are reduced into a single output, after a set of calculations take place.

Most of the current reduction approaches are limited to primitive types. Reduction computations could be done in sequential and parallel approaches. The parallel approach is sometimes more efficient than the sequential approach; however in some instances context switching and thread safe operations could be costly enough to overweight the advantage of parallel reduction over sequential reduction. In this project we are focused on the reduction implementation provided by ***ParaTask*** library.

In ParaTask, tasks are classified into three types of *one-off, multi* and *I/O* tasks. The multi-tasks are considered for breaking down big tasks into smaller sub-tasks which will be executed by multiple threads asynchronously. ParaTask allows developers to develop their own customized reduction using the provided interface. However, providing built in reduction functionalities which have proved an optimized performance is always considered as an advantage that is rarely provided by other libraries.

The current reduction implementations are mainly focused on trivial operations on primitive types; therefore, as an extension to ParallelTask’s current reduction library we would like to add more complex types of reductions that exploit more complex merging operations, and are implemented on more complex types or data containers (e.g. using collections such as ***HashMap, Collection, Set*** and ***List*** as containers). Moreover, the current reduction approaches assume that both sides of the reduction are of the same type, and reduction operations are *commutative and associative*.

The three features mentioned above could be considered as limitations that are imposed on reduction operations. Some of the reductions that have been considered for adding to the current ParaTask library so far could be named as: Concatenation/Combination, Counting the number of occurrence for a specific element, Sorted or Priority Combination etc. One of the remarkable proposals by [1] suggests using a cache for saving the results of a reduction which could be used by further reduction with a high probability. However, saving those results in a cache still requires searching and matching which could impose the same over head as searching through the final result (the result of the reductions that has taken place so far).

Another remarkable proposal by [1] is avoiding the creation of new collections each time that we perform a reduction, such that the result of the reduction will be kept in one container, and the others are merged into that. However, we should consider that if the reduction requires elements to follow a specific order, then pushing a new element in a collection and shifting the elements on its side back and forth could impose its own overhead.

Parallelizing the reduction process is another potential approach for increasing the reduction performance, given that the number of threads that are in charge of implementing reduction conforms the optimum number of CPU cores; otherwise the system performance drops rather than improving. Reduction allows calculations to be done separately, and the results to be saved in a private attribute, which can finally merger and deliver the final result. Without reduction these operations would require one shared thread-safe container to which access would be time-consuming.

Before progressing further we need to figure out or answer three questions in order to enlighten our path and clarify the direction in which we are going to move. The questions are as follows.

1. In which aspects is reduction being used at this stage?
2. What are the possible improvements that could be done to reduction?
3. What are the best approaches for implementing the improvements?

# Reduction Usage

Reductions are mainly used for figuring out an answer to a specific search or query. Reduction always comes with a mapping algorithm where a set of problems are performed separately (possibly in parallel), and the results from mappers are integrated into one final result. Some of the Map-Reduce algorithms are as follows.

## Counting and Summing

This algorithm can be used for calculating a total number of occurrences of a specific item in a number of documents. Mappers that count the number of occurrences of a specific item in different documents, return their lists to the reducer. The reducer will then add the number of occurrences of different items in each list, and provides one final list with the number of occurrences for each item in all documents.

## Collating

This algorithm can save all the items that would return the same value as a result of performing a specific function into one file, or alternatively performs another operation that requires all those items to be grouped. One of the most common uses of this algorithm is in *inverted indices,* where an indexed data structure maps some contents, such as work, text etc. to their corresponding location, such as a database, webpage etc. In this algorithm the mappers implement the function on each item and save the result of the function as a key in a map which associates the key (i.e. the result of the function) with the item itself. The reducer will then group items by their function values using the maps that are returned from mappers.

## Filtering, Parsing and Validation

This algorithm is used when we have a set of records, and we want to collect all records that meet a certain condition/requirement. For example we can name applications that involve text parsing, value extraction, conversion from one format to another etc.

## Distributed Task Execution

In this algorithm, a big and computationally intensive task is broken down into several sub-tasks, and the results from executing those sub-tasks are combined in order to obtain the final result. The mappers are in charge of performing the computations, and the reducer combines all emitted parts into the final result. Reducer’s job is normally performed on Maps, and it could include the union, intersection, selection or difference of the values in a map.

The main applications of this algorithm are in *physical and engineering applications/simulations* and *numerical analysis* or *performance testing*. In all styles of reduction thereof, sorting and ordering of elements could be an important factor. In Map-Reduce, sorting is normally intended for sorting the key-value pairs by their key. In this regard, sorting during insertion could be more efficient for many cases; however it may not be scalable for very big reductions.

## Interactive Message Passing

This algorithm involves entities/items that are interconnected in a network of entities/items. In this network each entity updates its status based on the status of its adjacent entities. Map reduce jobs are performed iteratively. During each iteration a node sends messages to its adjacent nodes/entities in order to help them update their status.

Tree of categories is one of the examples where this algorithm is vastly used is the tree of categories; where a bigger or a major category in the higher level of the hierarchy is branched into smaller sub-categories in the lower level of the hierarchy. In some instances the existence, or the status of the bigger category depends on the status of its sub-categories (e.g. graph analysis, web indexing).

## Distinct Values

This algorithm identifies the number of unique values of ***“F”*** in a set of records that contain ***“F”*** and ***“G”*** which have the same value for ***“G”***. In this algorithm mappers map the every value of ***“F”*** to their corresponding key ***“G”***; furthermore in the reducer the lists are combined. In the more efficient approach, the first list returned from a mapper is extended, the reducer avoids inserting duplicates and instead their corresponding value is increased in the list.

## Cross Correlation

This algorithm is used when we have a set of tuples of items, and we want to calculate the number of times that a certain tuple of items co-occur. In these cases where cross correlation is required, mappers can group data based on one of the items as a key, and associate an array to each key for which the elements are the corresponding counters for the encountered adjacent items.

*Map Reduce Patterns, Algorithms and Use-cases*

*Lin J. Dyer C. Hirst G. “Data intensive processing mapreduce”,(http://www.amazon.com/Data-Intensive-Processing-MapReduce-Synthesis-Technologies/dp/1608453421/)*

*Join Algorithms using Map/Reduce (*[*http://www.inf.ed.ec.uk/publication/thesis/online/IM100859.pdf*](http://www.inf.ed.ec.uk/publication/thesis/online/IM100859.pdf)*)*

*Optimizing Joins in a MapReduce environment (*[*http://infolab.stanford.edu/~ullman/pub/join-mr.pdf*](http://infolab.stanford.edu/~ullman/pub/join-mr.pdf)*)*

**Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters**

*Hung-chih Yang, Ali Dasdan, Yahoo, CA, USA*

*Ruey-Lung Hsiao, D. Stott Parker, Computer Science Department, UCLA, CA, USA*

This research uses an interface with two functions for *map* and *reduce.* The model facilitates parallel implementation of real-world tasks such as data processing for search engines and machine learning. The model does not support processing multiple related heterogeneous datasets. This shortage could be addressed using the operand<T> in our design. Where an operand may contain heterogeneous types and the method for combining two operands will be specified by the programmer.

The algorithm in this research provides implementation for relational algebra operators as well as several join algorithms. The infrastructure involves processing input data as map and value pairs in order to provide intermediate results in the form of key/value pairs (normally new values associated to the same key). In the reduce phase, all <key/value> pairs are merged in the user defined fashion. The model is best suited for processing homogeneous datasets. One of the most common cases in reduction is *joining large databases*, in order to achieve more comprehensive information.

The model adds a *merge* phase to the common map-reduce framework. We were inspired by this approach and provided different approaches for merging maps. The approaches include the union of maps, intersection of maps and subtraction of maps. We also separated the concern of reducing the objects related to the same key, from the concern of merging the elements of maps. The research claims that this approach also makes processing of heterogeneous datasets possible. (another possible improvement, merge two maps in the union way while the final map is sorted as well). In this research authors have considered cases where the final result associated to a key could be of a different type from that of the intermediate result.

However, in this approach the final result associated to a certain key is confined to one value. In our design we allow the final result to be yet a collection of elements (as an extra flexibility we can consider having a different type of outcome as well). The common approach between our method and the method in this research is the aggregation between two tuples with the same key. In this approach the aggregation is confined to sort by key and group by key which can be potentially added to our provided merging approaches.

**Adopting Graph Reduction to Synthesize Parallel Computation Models**

*Shen Chao, Liu Xiaodong, Tong Weiqin, Zhi Xiaoli*

*Scholl of computer engineering and science shanghai university, Shanghai, China*

This research addresses the fact that parallel computing models for different applications are aimed at different types. This project proposes graph reduction where each reduction is corresponding to a local transformation of the graph. For this purpose functional languages are focused on, as they are ideal for parallel programming especially with use of lambda expressions.

**HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads**

*A. Abouzeid, K. B. Pawlikowski, D. Abdi, A. Silberschatz, A. Rasin*

*Yale University – New Haven U.S., Brown University*

Enterprises and development teams are rapidly moving their data from high end servers to cheaper, lower end clouds. Therefore, preserving data is now cheaper and more secure. Therefore, the amount of data that needs to be processed grows exponentially high. For processing large amount of data, parallel databases and MapReduce based systems are proposed. Parallel databases scale well when the system deals with tens of nodes. Each node is a self-sufficient system without a single point of contention across the system. No two nodes share memory or storage.

However, in parallel databases the number of nodes is normally limited to 100 nodes maximum, and the systems do not scale well when there are hundreds of nodes. That is caused by two main factors. First, failure of the nodes increases as the number of nodes is increased. Secondly, parallel databases generally assume a homogeneous array of machines, while it is nearly impossible to achieve homogeneity at higher scales. Parallel database also do not provide a high fault tolerance due to the basic assumption of this approach which indicates failures would rarely happen, and that clusters include tens of nodes and not hundreds.

On the other hand, MapReduce-based systems scale better in higher scales as the original design of this concept was meant for high scalability. For example Hadoop’s map-reduce mechanism is used for administrating 2.5 petabytes of data in Facebook’s data storage. Ideally the scalability of map-reduce can be used along with the performance quality of parallel databases in order to get a hybrid system. Therefore, using Map<key, value> types in map-reduce approaches seems to be necessary.

**Map Reduce: Simplified Data Processing On Large Cluster**

*Jeffrey Dean and Sanjay Ghemawat*

*Communications of the ACM, January 2008, Vol. 51, No. 1*

In a map-reduce procedure, users specify the computation in terms of a *map* and a *reduce* function, and the under-lying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. More than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one-hundred thousand MapReduce jobs are executed on Google’s cluster every day, processing a total of more than 20 petabytes of data per day. More than ten thousand distinct programs have been implemented using MapReduce at Google for large-scale graph processing, text processing, data mining, machine learning, statistical machine translation and etc.

Prior to Google’s recent implementation of MapReduce, Google would use hundreds of ***special-purpose computations*** that processed large amounts of raw data, such as crawled documents, web request logs and etc. to compute various kinds of derived data, such as *inverted indices, various representations of the graph structure of web pages, summaries of the number of pages crawled per host, set of most frequent queries sent*. In most of these cases the computation is straight forward; however the input and output data are enormously large, thus the computations have to be distributed among hundreds of machines in order to finish in a reasonable amount of time.

Most of computations involve applying a map operation to each logical record in order to produce intermediate *key/value* pairs, and then apply a reduce operation to all *values* that share the same *key* in order to combine them into one final result. Using this functional model with *user-specified* map and reduce operations allows more efficient parallelization of large computations. This parallelization could be either spread across multiple machines in a cluster of connected computers, or across multiple processors of the same computer. The literature introduces two other of main MapReduce implementations, *Hadoop* and *Phoenix* MapReduce systems.

**Programming Model**

The computation takes a set of input *key/value* pairs, and produces a set of *key/value* pairs as input. The *map* function written by the user, takes an input pair and produces a set of intermediate *key/value* pairs. The MapReduce library groups together all intermediate values associated with the ***same key*** value and passes them to the reduce function. The reduce function which is also written by the user, accepts the set of intermediary values associated to the same key, it merges these values together to create a single value, or possibly a smaller set of values. Google’s MapReduce library **is written in C++**, which introduces the potential of adding the same functionality to a Java library.

In our implementations we use java *interfaces* and abstract classes in order to provide programmers with reasonable flexibility for developing their own specified reduce function, and at the same time force them to follow a set of protocols and standards to fit their functions within the pre-defined body of our parallelization system. Moreover, we have provided a vast range of pre-defined implementations for the interfaces and abstract classes thereof that cover most of the commonly used reduce function, such that it relieves users from the burden of focusing on the complications of reduction by exploiting our examined optimized code. We have increased the flexibility of our library and minimized code duplicate by considering Object Oriented principles and separation of concerns in our design.

We embraced the fact explained by most of research projects that most of high level MapReduce operations involve processing ***Map<key, value>*** structures. In these cases the reduction process can be considered of two independent stages. The first stage of the reduction involves specifying how to combine two different maps (i.e. produce their union, intersection, subtraction, etc.), and the second stage of the reduction involves specifying how to combine the values of two different maps that are associated to the same key (i.e. combining function). In our design these two stages are separated. That is, programmers use classes that inherit an abstract class called ***Operand*** as the values that will be associated to *keys* in maps.

The Operand type and its sub-types are generic, and the data type that they wrap in themselves need to be specified by the programmer. An Operand type encapsulates a value of the type specified by a programmer, as well as it requires a programmer to override an abstract method called ***Operate***. The operate method specifies how two its encapsulated value will be combined by another value of the same type. Moreover, the Operand class overrides the *toString* method in java which can be optionally overridden by its sub-classes and specify how the operand should be printed. This function is useful specifically if the Operand type wraps a collection or a set of values, so that the programmer can specify how the elements inside the collection or set are printed.

Consequently, when a programmer tries to specify how two different maps are merged into one, they merely focus on their design for merging, and not focus on the combining method of the values, as that is considered as a different concern in our library. More importantly, it suffices to implement a combining method only once as the corresponding Operand class can be used independently by various reduce functions. As mentioned earlier, we have provided predefined implementations which include four different implementations for the reduction interface (i.e. specifying how different sides of a reduction should be merged), and forty different implementations for the Operand class (i.e. specifying how different values associated to the same key should be combined). Therefore, there are 160 possible combinations of reduction using only forty four classes. This approach helps developers to avoid considerable amount of code duplication, and reuse their implementations efficiently.

Google recommends sorting the intermediary values based on intermediate *keys* and adopts the same approach in its implementation. The reason for sorting intermediary values based on their keys is to ease grouping <key, value> pairs with the same key, because many different keys could be sent to the same reduce task. (Consideration: adding sorted operations on maps in ascending and descending order of their keys).

**OpenMP Reduction Clauses**

*Blaise Barney, Lawrence Livermore National Laboratory*

[*https://computing.llnl.gov/tutorials/openMP/#REDUCTION*](https://computing.llnl.gov/tutorials/openMP/#REDUCTION)

OpenMP is an Application Programming Interface (API) for Fortran and C/C++ programming languages which automates efficient parallelization of a code using compiler derivatives. OpenMP also provides functionalities that facilitate reducing a set of values that are returned from parallelized tasks into a single final result. OpenMp provides the following clause for reduction:

*reduction(operator: list)* for example *reduction(+ : results)*

In the format demonstrated above, the variable “*results”* is a list of values gathered from tasks that were performed in parallel. Furthermore, “+” is the operator specified by the programmer to operate on every two value in the list in order to provide the final result. However, there are limitations to this feature that are listed as follows.

* The variables in the list must be scalar, and cannot be of type *array, list* or any other *structured type.*
* The operator is limited to a few internally defined operators (e.g. +,\*,-,/,&,^,|,&&,||) and cannot be overloaded
* The reduction operation can only be used in a work-sharing region with a specific syntax format (e.g. ***x op expr***)
* The reduction operation on real numbers may not be associative (i.e. a \* (b \* c) may not be equal to ( a \* b) \* c).

We have addressed these limitations in our design using the following approaches.

* Extending the *Operand* abstract class allows programmers to flexibly wrap any data type for their operation thanks to the **generic types**.
* Overriding the abstract *operate* method of the *Operand* class, enables programmers to specify their own combination function without any limitation, or alternatively use the pre-defined combine functions that are already provided in the library.
* Specifying an independent library for *reduction* removes any confinement regarding the regions in which the reduction objects can be used, such that programmers can even use it for their sequential code.

However, OpenMP 4.0 allows programmers to specify their own customized reduction identifiers using the ***declare reduction*** clause. The ***declare reduction*** directive declares a reduction identifier that can be used later in the code in a reduction clause. The syntax of declaring a reduction identifier is as follows.

**#pragma omp declare reduction (reduction-identifier: typeNames-list: combiner)**

The “typeName-list” lists the name of the types that can be used (reduced) by the custom reduction identifier. The custom reduction remains limited to the type names initially declared in the declaration phase. Therefore, if new types are required the declaration needs to be modified. In our design variety of variable types has been accommodated as the focus is on generic types, and the types can be flexibly specified by the programmer.

The “combiner” specifies how partial results should be reduced. Combiner only uses variable identifiers called **omp\_in** and **omp\_out**, which are of the type of the variables declared to be reduced. The latter refers to the storage that holds the result of the reduction. The **omp\_out** must be initialized, and the initial value will be copied for every instance of the reduction identifier. In our design we have separated this concern from the actual reduction of the results. In our design, the elements that are being reduced are independent objects of class *Operand*, which must be necessarily initialized at the time of creation by passing the value to the constructor. The result of the reduction will replace the value of the first *Operand* object. If further reductions take place depending on the program specifications, the value of the first *Operand* object holds the result and is returned at the end of the process. This mechanism saves memory and avoids unnecessary copying of contents into new memory space.

Each ***#declare reduction*** directive is used within a C++/Fortran code, and is usable within the same code. However, object oriented principles favors designs which allow reusing the code in different programs. Providing reducers as independent classes allows using them independently of the programs. OpenMP 4.0 handbook introduces some restriction of its reduction functionality that can be summarized as follows.

There are restrictions on the scope in which variables are accessible, ad for incorrect modifications the behavior of the reducer could be unspecified. In our design *Operand* objects provide “getter” and “setter” methods that are initially inherited from the parent “Operand” class, but can be overridden to further control access and modification of a value.

Secondly, a reduction declaration can at most have one initializer clause for initializing the variables that are used by the combiner function. Furthermore, the type name in a ***declare reduction*** directive cannot be a *function type,* an *array type,* or a *reference type.* This restriction is addressed in our design where generic types allow programmers to flexibly decide about the type of objects being reduced.

**Hadoop Reduction API**

[*https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/mapred/Reducer.html*](https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/mapred/Reducer.html)

Hadoop framework also provides a ***Reducer*** interface that can be implemented by programmers in order to specify their custom reductions, as well as some of the pre-defined implementations of the interface that could be summarized as follows.

1. **FieldSelectionMapReduce** – this reducer receives two different lists of fields. The first list forms the keys of the output map, and the other list forms the values of the output map. The reducer performs the union of the lists in order to provide a <key, value> set for the corresponding output map.
2. **ChainReducer** – chains multiple mappers after a certain reducer within a reduction task. The advantage of this approach is that the order of mappers can be flexibly changed without any of the mappers needing to know the mapper before and after them.
3. **ValueAggregatorCombiner** – combines the values that are associated to the same key.