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.