

**Final Project Description**

DD2356 / Methods in High Performance Computing

## 1 - Introduction

MapReduce has emerged as one of the preferred programming models to hide the complexity of process and data parallelism on HPC. The power of this paradigm resides on the definition of simple **Map()** and **Reduce()** functions, that become highly-parallel operations using complex inter-processor communication.

For the final project of the course, **you are asked to implement a MapReduce framework designed specifically for HPC**. In addition, you are also expected to provide a comprehensive performance evaluation with a large dataset from the Purdue MapReduce Benchmarks Suite (PUMA), using hundred of processes running on Beskow.

The goals of the course final project are:

* Design and implement MapReduce for supercomputing using MPI. To use OpenMP is a plus, but not required.
* Use the implemented MapReduce for Word-Count with a reference benchmark dataset.
* Measure and analyze the performance of the code varying the number of processes.
* Write a report, maximum ten pages, describing the design and implementation of the MPI MapReduce code, together with performance measurements and analysis.

**The final project must be solved in groups of two students**. This corresponds to the final exam and will determine your grade in the course. Please, submit the report to Canvas before the specified deadline and do not forget to include in the document the members of your group. In addition, include a link to a Git repository with the source code of your project.

The rest of this document provides further information on how to implement the MapReduce framework. We also propose you optional features to get a higher grade. The explanations follow part of the insights provided on the following research paper:

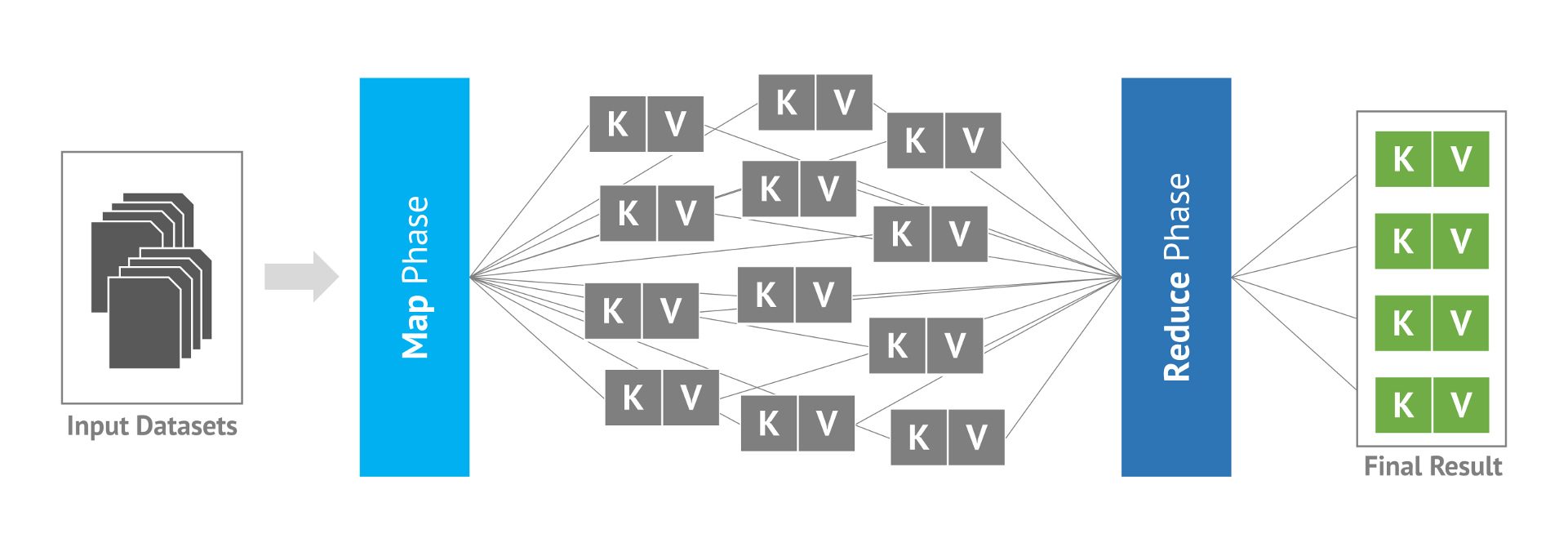
[**Towards Efficient MapReduce Using MPI**](https://htor.inf.ethz.ch/publications/index.php?pub=92) **by T. Hoefler, A. Lumsdaine and J. Dongarra**

## 2 - Background

MapReduce was created in the context of cloud analytics as a programming model for processing and generating large datasets. The main idea behind a MapReduce job is to split a certain input dataset into independent portions or *tasks*, which can then be processed in a completely parallel manner inside the **Map phase**. The output from this phase is transferred to the **Reduce phase**, where the data is aggregated to produce the result.  
  
Inside a MapReduce framework, users are responsible for the implementation of the **Map()** and **Reduce()** operations. These operations define the specific use-case or problem to solve in MapReduce. In particular, **Map()** is designed to split the input data into a collection of individual <key,value> pairs. Each tuple is then merged using the **Reduce()** function, producing an aggregation of all the <key,value> pairs with identical key.

The main important aspect to consider about MapReduce is that the design of the **Map()** and **Reduce()** operations is stateless. This is one of the main reason why this programming model can become highly parallel. Hence, the actions performed inside each of these functions do not depend on any particular execution order or process.

A visual representation of the MapReduce algorithm is illustrated in the following figure:



## 3 - Description of the project

For the final project of the course, you are requested to implement a MapReduce framework for HPC. Your MapReduce framework should be virtually split internally into two main components:

* **Back-end implementation.** Manages the communication between the processes using MPI, and calls the specific **Map()** and **Reduce()** functions, as necessary. This is the most complex part of the project.
* **Use-case implementation.** Defines the particular use-case and implements the **Map()** and **Reduce()** functions. This is the lightweight part of the project.

We have divided this section to provide you with guidelines on how to correctly implement the aforementioned components, how to store and transfer internally the key-values, and how to conduct your evaluations, among other relevant information.

Note that the chapter “Optional Features” provides you with suggestions for advanced features that can be integrated into your project (e.g., collective I/O) for a higher grade.

#### **3.1 - Back-end Implementation**

The back-end implementation of the MapReduce framework uses MPI to parallelize and distribute the data across the processes. The back-end is responsible for reading the dataset in blocks of 64MB, distribute these blocks to each process to perform Map (and generate key-value pairs), and aggregate all the intermediate key-value pairs to produce the result. **We consider a “task” each 64MB block of data to be manipulated**.

The responsibilities for each process are divided using a master-slave model, where rank 0 is designed to be the master process and the rest are the slave processes. We consider a Map phase, a Reduce phase, and an additional “Combine” phase that sends all the aggregated key-values from the Reduce phase to the master process to generate the result.

**The goal of the master process is to perform the following actions:**

1. **Map phase:**
   1. **Read the next (num\_procs-1)\*64MB blocks from the input dataset.** These are the inputs for each Map task, that will be assigned to the particular slave processes accordingly.
   2. **Send each 64MB block as the next task for each slave process.** For instance, if we have 16 processes in total, the master will read 15 blocks of 64MB and provide one per individual process.
   3. **Repeat from Step 1 until the whole input dataset is completely read.**
2. **Reduce phase:**
   1. The master will act as a any other slave process. See the description for slaves.
3. **Combine phase:**
   1. **Retrieve all the aggregated key-value pairs from each slave process.** The idea is for the master process to grab all the intermediate key-value pairs to produce the final result. For simplicity, we are not going to order the final output.
   2. **Store the result into a CSV file.** The master will store each <key,value> using a comma-separated entry. Two fields are expected per entry in the CSV file: the key and the value.

You can use individual MPI I/O operations to read the 64MB blocks from the input dataset. Assume that the length of the input dataset is multiple of the number of slave processes, or ignore the last part of the file otherwise. To distribute the data per “task”, you can use a collective **MPI\_Scatter()** operation. To generate the result in Combine, you can use a collective variable-length **MPI\_Gatherv()** operation, as each process may have different amount of key-value pairs. Finally, to store the result, see the MPI laboratory assignment of the course to replicate the data format and the individual I/O operations that you used.

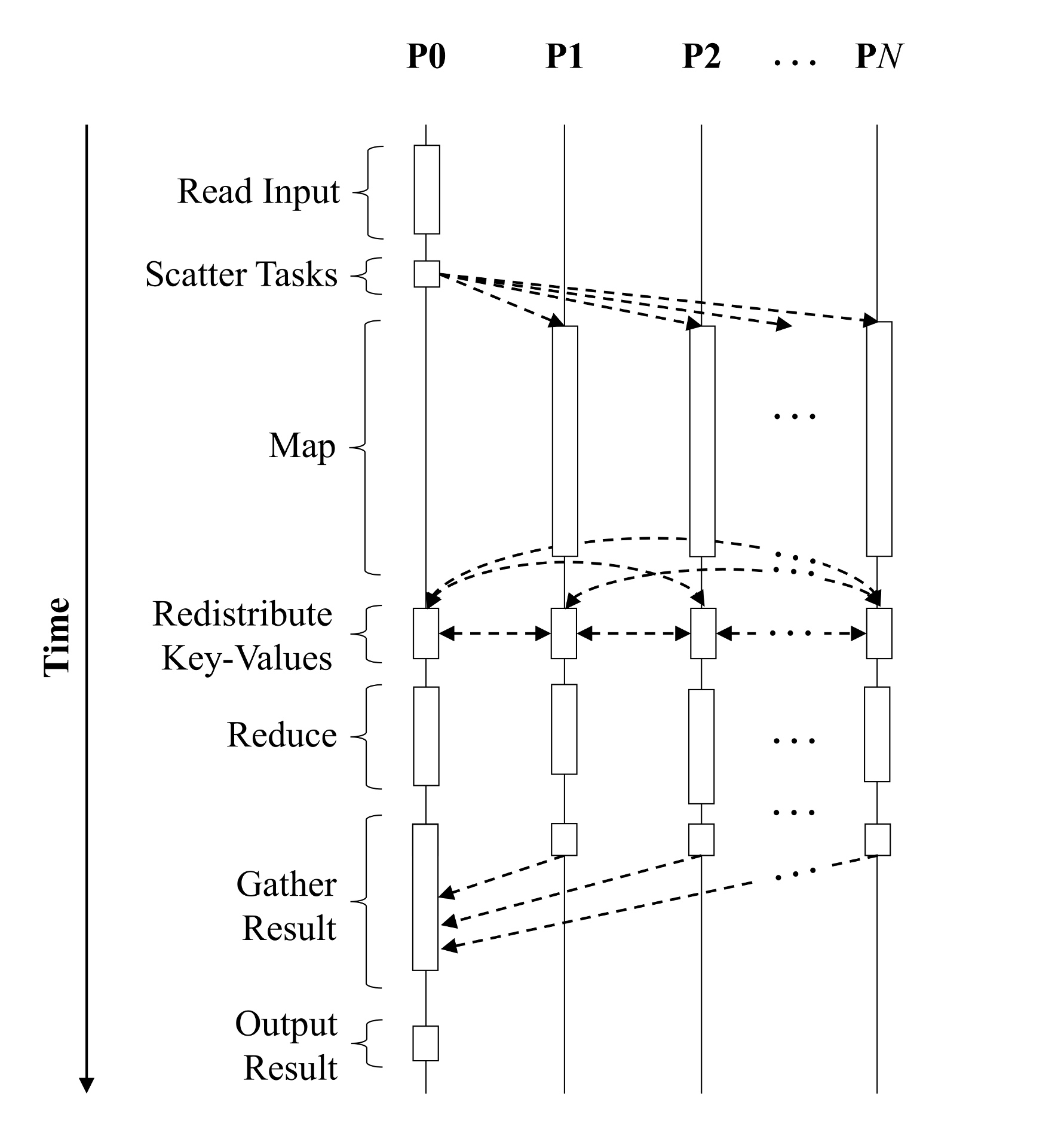
On the other hand, **the goal of the slave processes is to perform the following actions**:

1. **Map phase:**
   1. **Retrieve the next 64MB “task” to compute.** The master process is going to provide each slave with a 64MB task to compute, as previously described.
   2. **Until the input data of the 64MB task is consumed, repeat:**
      1. Call **Map()** with the current offset inside the 64MB block.
      2. Store locally the <key,value> output given by **Map()**, if any.
      3. Advance the offset for the next call to **Map()**. This fact implies that **Map()** should return a <key,value> tuple and also how much data was consumed from the input 64MB block.
   3. **Repeat from Step 1 until the whole input dataset is completely read.** You will need a mechanism from the master process to notify the end of the file.
2. **Reduce phase:**
   1. **Exchange the intermediate <key,value> pairs across all the processes.** Each process will receive groups of key-value tuples assigned to it, as described later in the document.
   2. **Aggregate locally all the <key,value> tuples**, calling **Reduce()** repeatedly until the values have been stored.
3. **Combine phase:**
   1. **Send the aggregated key-value pairs to the master process.** This will generate the result of the MapReduce execution.

To get the 64MB block from the master process, each slave must participate into the collective **MPI\_Scatter()** operation (i.e., with rank 0 as the root of the communication). You can have broadcast operation first from master to notify how much data is assigned per task. This is useful to notify the end of the file, which should be 0 bytes. Feel free to use other mechanisms, such as another scatter operation with individual task-size per process.

The intermediate exchange of <key,value> pairs can be accomplished with a variable-length all-to-all, named **MPI\_Alltoallv()**. The syntax is similar to a traditional all-to-all that we explained in the lectures, with the difference that now each process receives a different amount of data in the exchange. To understand how much data is expected per process, use first a traditional all-to-all operation or other mechanism that you prefer. Above all, **keep in mind that the idea behind this step is to allow the same key found by different processes to be aggregated into a final <key,value> tuple**. Otherwise, it might be that the result ends-up with duplicated key-value tuples with same key. More information on this topic will be provided afterwards.

To summarize the explanations, the following figure illustrates a timeline where a single group of tasks in the Map phase is executed and then the result is aggregated:

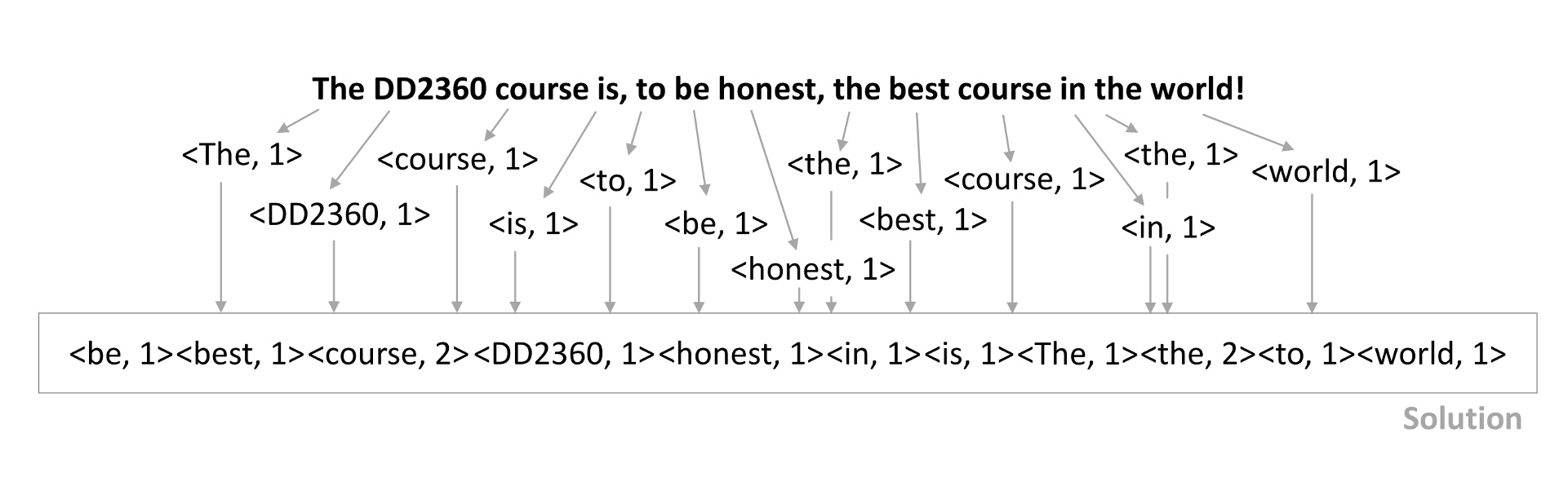


#### **3.2 - Use-case Implementation**

**We are going to evaluate the MapReduce framework using Word-Count**, a technique that has major relevance in Big Data analytics. The basic principle of Word-Count is to compute the occurrences of individual words over large collections of documents:

* **The Map phase process the input dataset to emit <word,1> key-value pairs**, where word represents the key and 1 the occurrence found. You are asked to aggregate locally the values during the Map phase as well, so that the exchange in the Reduce phase is performed using <word, count\_tmp> tuples instead.
* **The Reduce phase aggregates the occurrences for a given word**, as found by all the processes, **to generate its final <word, count>**. This is the value that will be sent to the master process during the Combine phase and represents the result for that word.

The following figure illustrates an example where an input phrase is split into individual key-value pairs, and then aggregated to generate the final result:



Note how the “course” word is aggregated and the count is increased to 2. Also, “The” and “the” are considered in the example as separate words. You are free to decide how to handle these distinctions, as long as the result remains consistent between executions.

For the project, **a word can be anything that contains characters from the English alphabet or numbers**. If you start reading “1”, then look for subsequent numbers to generate the word and stop as soon as you see any other character. This means that “1234.5678” is going to be split into <1234,1> and <5678,1>. If you start reading “h”, then proceed assuming a word of alphabet letters. Hence, this means that “hello there, world!” is going to be split into <hello,1>, <there,1> and <world,1>. Lastly, if you find a combination of the two, such as “hello, @user1234!”, the result would be <hello,1>, <user,1> and <1234,1>.

Following this description, **you must implement the Map() and Reduce() functions that are used by the back-end implementation as specified below**:

* **Map() function.** It receives a buffer and the length of the buffer as parameter. The function will analyze the buffer looking for a word and immediately stop after an occurrence is found. The function returns a <word,1> tuple and how many characters were read from the input buffer. This way, the next time the back-end calls **Map()**, it will provide a different pointer inside the buffer to retrieve the next word.
* **Reduce() function.** It receives two key-value tuples and produces a new one that corresponds to the aggregation of the two. For instance, if we provide <hello,5> and <hello,1>, the output from **Reduce()** would be <hello,6>. An option is to aggregate the value from the second key-value to the first one, avoiding to create a third tuple.

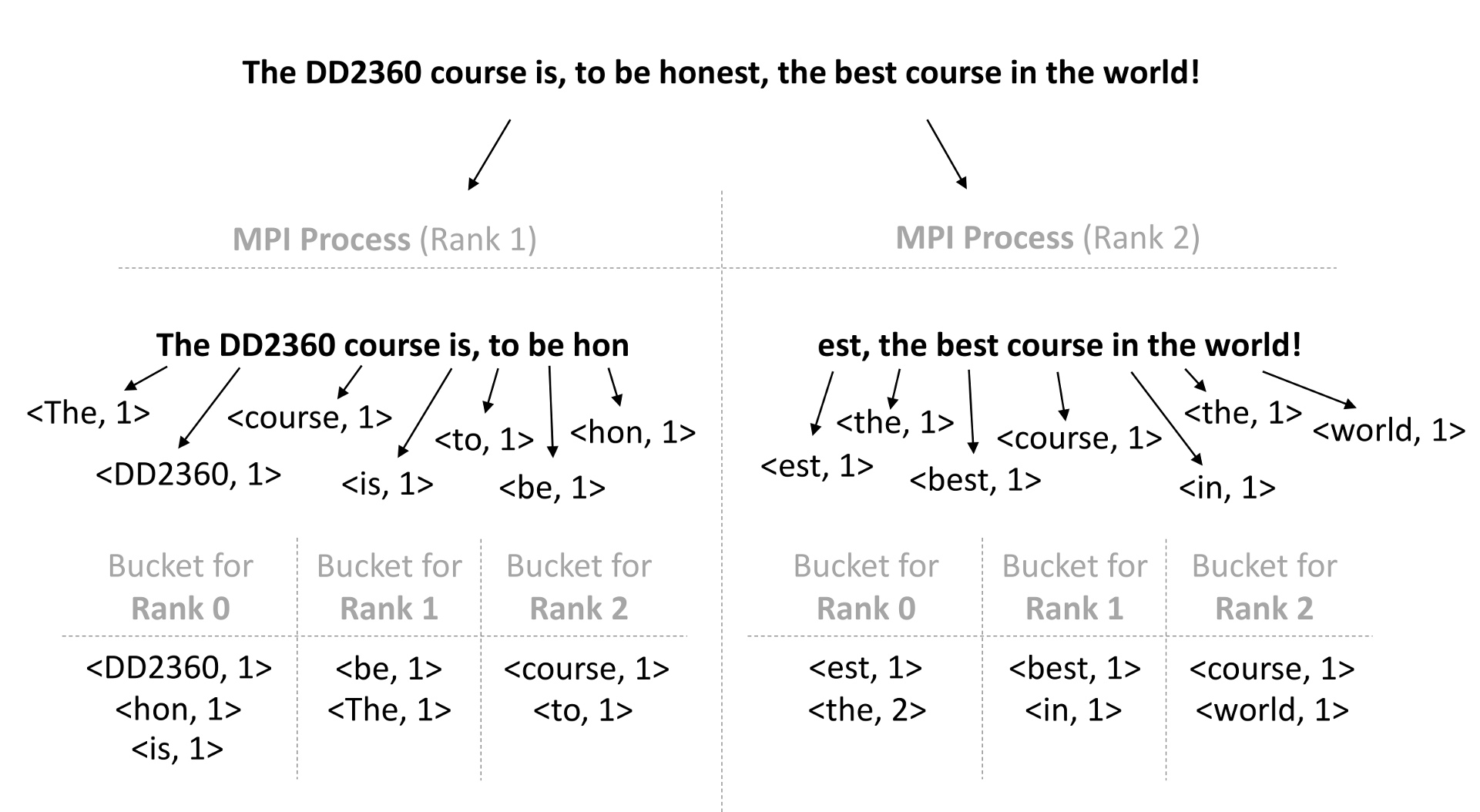
It is very important to note that we do not ask you to make your MapReduce framework generic. Consequently, **you can implement directly the functions assuming that the back-end is aware of the Word-Count application**. Also, you can use a fixed-length word of certain number of characters, otherwise you will need to support variable-length key-values. This fact implies that, if you find a word longer than the limit, you can split it into two or more words. If the length of the word is less than the limit, fill up the rest of the space with the null-character ‘\0’.

#### **3.3 - How to store and send the key-value pairs**

After a <key,value> tuple is found using **Map()**, you must store it locally and **assign it to a “process owner”**. Think of it as separating the key-values into different buckets or containers. Thereafter, we can provide these containers to the correspondent processes to aggregate the same key per <key,value> and produce the result. In our case, we separate the key-values per word, and aggregate the count afterwards.

The reason for this design consideration is to ease the exchange of the key-values per process, which will become rather trivial if you already have all the data per process separated into different buckets. Hence, the only work left would be to store this data into an array and send it to the correspondent process using the variable-length all-to-all operation of MPI.

The following figure illustrates visually the separation of the words per target bucket, assuming that rank 0 reads the input phrase and splits it into two tasks, assigned to rank 1 and rank 2:



These processes call **Map()** iteratively to produce the different <word,1> tuples. At the same time, the generated tuples are stored according to the “process owner” of the word. This means that, if the word already exists inside the target bucket, the **Reduce()** function must be used to minimize the amount of data to be transferred afterwards inside the Reduce phase. For instance, note how “the” is aggregated in rank 2 for target rank 0.

**To determine the target process, a hash function has to be applied per given word**. The hash function will guarantee that, despite the word being found on different processes, each process will know which one is the target and correctly send the data correctly during the Reduce phase. If you look at the figure, we represented “course” in the target bucket for rank 2, both inside rank 1 and in rank 2. **We suggest you to use the following hash function**:

|  |
| --- |
| #define SEED\_LENGTH 65  const char key\_seed[SEED\_LENGTH] = "b4967483cf3fa84a3a233208c129471ebc49bdd3176c8fb7a2c50720eb349461"; const unsigned short \*key\_seed\_num = (unsigned short\*)key\_seed;  int calculateDestRank(char \*word, int length, int num\_ranks) {  uint64\_t hash = 0;    for (uint64\_t i = 0; i < length; i++)  {  uint64\_t num\_char = (uint64\_t)word[i];  uint64\_t seed = (uint64\_t)key\_seed\_num[(i % SEED\_LENGTH)];    hash += num\_char \* seed \* (i + 1);  }    return (int)(hash % (uint64\_t)num\_ranks); } |

**In order to store the key-value tuples locally, you need to use an efficient data structure.** The [Standard Template Library](https://en.wikipedia.org/wiki/Standard_Template_Library) (STL) of C++ contains many optimized implementations that can be very useful. For example, an std::map can be used to store each key-value per target process, while an std::vector can be used to index the std::map (i.e., bucket) per process. Feel free to choose how to store this information according to your needs and expertise.

Additionally, **we recommend you to create a “KeyValue” structure** that contains a fixed-length word and a 64-bit integer to represent the count. This way, after you exchange the key-values to produce the final aggregation, it is as simple as reading from the received array assuming this structure per tuple in the buffer. In addition, **if you create an MPI derived datatype to represent the “KeyValue”, you can use it to transfer data easily among the processes**. An example of **MPI\_Type\_create\_struct()** is illustrated in the MPI-3.1 standard:

<https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node91.htm>

#### **3.4 - How to evaluate your code**

To evaluate the MapReduce implementation, **we are going to use a large dataset from the Purdue MapReduce Benchmarks Suite (PUMA)[[1]](#footnote-0)**. This suite emerges as an on-going effort to provide rigorous benchmarks for MapReduce frameworks.

In particular, we are going to use the *Dataset3* from the PUMA-Wikipedia datasets, that contains approximately 300GB of data divided into multiple files. These files include articles, user discussions, and other metadata originally from Wikipedia.

We have pre-processed these files off-line to generate unified, large input datasets for concise results. Please, **follow the instructions provided in this subsection to understand how to retrieve the input files and evaluate your code for the final report**.

##### **3.4.1 - Evaluating your code locally**

Before running on Beskow, you must evaluate your code locally using two input test files, located in the following link:

<https://kth.box.com/v/DD2356-WordCount-TestFiles>

Download these files and place them whenever your code can have access. By default, we recommend you to use the “wikipedia\_test\_small.txt” file for tests and debugging purposes. This file contains a few hundred kilobytes of data and it should be relatively fast to process. Do not forget to reduce the task size (otherwise, only one process will handle the input). When you want to experiment with a more realistic dataset, then run your code locally with the file “wikipedia\_test\_large.txt”, that contains approximately 512MB of data.

**Ensure first that your code works correctly with the aforementioned files**, varying the number of processes and checking that the output remains the same. To compare two outputs, you can use the visual tool [**meld**](http://meldmerge.org/) or the **diff** command.

##### **3.4.2 - Evaluating your code on Beskow**

After your code is theoretically correct and produces the same output locally regardless the number of processes used, **we ask you to move to Beskow and start experimenting with larger process count and input files**. We provide you with the following input files, located inside the Klemming folder of one of the course instructors: (See next page)

|  |
| --- |
| /cfs/klemming/scratch/s/sergiorg/DD2356/input/wikipedia\_10GB.txt  /cfs/klemming/scratch/s/sergiorg/DD2356/input/wikipedia\_20GB.txt /cfs/klemming/scratch/s/sergiorg/DD2356/input/wikipedia\_40GB.txt /cfs/klemming/scratch/s/sergiorg/DD2356/input/wikipedia\_80GB.txt /cfs/klemming/scratch/s/sergiorg/DD2356/input/wikipedia\_160GB.txt  /cfs/klemming/scratch/s/sergiorg/DD2356/input/wikipedia\_320GB.txt  /cfs/klemming/scratch/s/sergiorg/DD2356/input/wikipedia\_640GB.txt |

**Do not copy, alter or move these files!** They are configured to get the best performance out of Lustre, with special stripe count settings. Moreover, all the students of the course are going to use these files, so be extremely careful.

As the name of the input files suggests, we have created single input files from 10GB, 20GB, …, up to 640GB of data. It is up to you to decide what is the best possible size for your program and the evaluations being conducted. At most, **each run should not take more than 5 minutes**. If it does, increase the process count or reduce the input size. An alternative is to manually limit the amount of tasks (e.g., use the “wikipedia\_10GB.txt” input, but take only 2GB of the 10GB available).

**For the performance analysis of the final report, you are expected to perform either a strong scaling or a weak scaling evaluation**:

* **Strong scaling.** Defines how the execution time varies with the number of processes for a fixed problem size. For instance, you can choose a 20GB input dataset and vary the process count from 8, 16, …, up to 256 processes. The figure should illustrate that the execution time is reduced in half per process count increase (ideally).
* **Weak scaling.** Defines how the execution time varies with the number of processes for a fixed problem size per processor. For instance, if we vary the process count from 8, 16, …, up to 256 processes, we can start with a 10GB dataset for 8 processes, then double the input dataset size with a 20GB dataset for 16 processes, and so on. The figure should illustrate that the execution time is approximately the same (ideally).

**You must create performance figures for your report according to one of these two metrics** (of your choice) and provide the details on how did you run each test. Including results for both metrics is not required, but it is considered a plus. Also, feel free to include any other relevant performance evaluations in your document, such as I/O performance, communication performance (e.g., variable-length all-to-all), and more. Above all, **keep in mind that you must use several nodes and hundreds of processes**. For instance, topping your scaling evaluation with 8 nodes (256 processes) can already show very interesting results.

#### **3.5 - Optional Features**

We propose you several features and optimizations. These are not mandatory, but will give you a higher grade in the project:

* **Integrate OpenMP by parallelizing each Map task assigned per process.** The idea is to take advantage of the multi-threading capabilities of each processor of Beskow and assign 2 threads per process (i.e., 32 MPI processes plus 2 threads per process, making 64 threads per node in total).
* **Introduce collective I/O for reading the input.** The master process will scatter the offset within the file and how much data to read, but the input is read collectively using MPI I/O operations.
* **Use non-blocking communication.** Try to use non-blocking communication to overlap computations and communication whenever possible. For instance, the master process can use a non-blocking scatter to distribute the tasks in parallel.
* **Create a custom memory management for buckets.** Each process can pre-allocate the buffers used for all-to-all and gather, and store the key-value pairs directly to avoid an additional copy before sending the data.

In addition, we also give you some hints for possible features that you can include in your framework. These will count for your final grade, but are less relevant for the course:

* **Make your framework generic and integrate C++ classes.** The purpose is to provide a multi-inheritance mechanism that simplifies the development of new use-cases.
* **Allow the parameters of MapReduce to be configurable from outside.** This way, you can set the task size (i.e., instead of a fixed 64MB), the input file to use, the path where to store the output, and other parameters.
* **Support for file lengths not multiple of number of slave processes.** For instance, your framework could assign a smaller task size for the last part of the file, if it is not enough for 64MB blocks.
* **Make the master process perform a small amount of work during Map.** This is to avoid having this particular process idle most of the time.
* **Store the output of the MapReduce execution in order.** After the master process receives all the key-values, you can order them before storing the result. A better solution is to perform a collective ordering, such as a parallel merge sort, in Combine.

If you decide to integrate advanced features, such as the ones described in this subsection, **it will also be considered a plus to compare performance with and without these features**.

1. <https://engineering.purdue.edu/~puma> [↑](#footnote-ref-0)