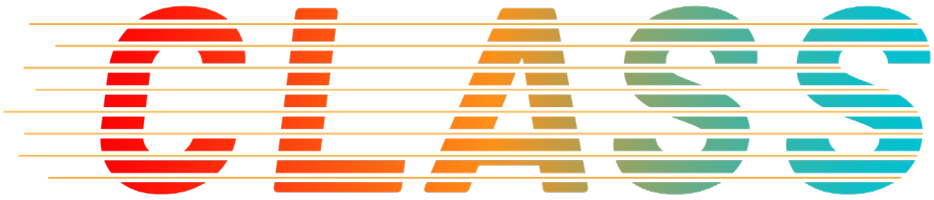


D2.5 – Final release of Spark and COMPSs integrated in CLASS architecture

Version 1.1

Document Information

Contract Number	780622
Project Website	https://class-project.eu/
Contractual Deadline	M29, May 2020 (Due to COVID situation this deliverable has been submitted on M31, July 2020)
Dissemination Level	PU
Nature	Demonstrator
Author(s)	Nihad Mammadli (BSC); Javier Álvarez (BSC)
Contributor(s)	Rosa M. Badia (BSC)
Reviewer(s)	Erez Hadad (IBM); Elli Kartsakli (BSC)
Keywords	integration, pyspark, pycompss, execution framework

Notices: *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No "780622".*

Change Log

Version	Author	Description of Change
0.1	Nihad Mammadli and Javier Álvarez (BSC)	Initial Draft
0.2	Rosa M Badia	Revision
0.3	Erez Hadad (IBM)	Revision
0.4	Nihad Mammadli	Review comments addressed
1.0	Elli Kartsakli (BSC)	Final Version, Ready to EC revision
1.1	BSC	Addressing EC requirements (28/10/2021)

Table of contents

1. Executive Summary

This deliverable presents the work carried out between months 7 and 29 in the context of Task 2.2 of WP2, defined as the "Integration of Spark and COMPSs programming models into a unified programming environment". The work presented in this deliverable contributes to MS3.

More precisely, this deliverable describes the final release of the Spark and COMPSs integration. We have achieved this integration by building the ***Distributed Dataset (DDS)*** structure on top of COMPSs. DDS is currently available for the Python programming language, and provides a very similar API to ***PySpark's Resilient Distributed Datasets (RDD)***. In this manner, many PySpark applications can be executed in PyCOMPSs by just replacing PySpark's RDD module for PyCOMPSs' DDS. In Python, this module can be changed at run time programmatically, which means that the execution framework can be defined as a configuration parameter. This enables the dynamic execution of applications both in PySpark and in PyCOMPSs without any changes in the source code.

This deliverable gives an overview of the implementation of PyCOMPSs DDS, and reports the changes and improvements with respect to the first integration release reported in D2.3 [1]. Furthermore, the currently available methods are listed, and detailed usage instructions are provided. This deliverable documents the successful accomplishment of the objectives set in MS3 regarding Task 2.2.

2. Design and implementation

We have implemented the DDS as an integral part of the PyCOMPSs package and included it in the regular PyCOMPSs installation. This means that DDS can be used in any PyCOMPSs application by just importing the 'pycompss.dds' module. In this way, DDS calls can be inserted at any point in the application workflow, mixed with calls to user-defined tasks.

PySpark's RDD provides two types of methods: transformations and actions. Respecting the ***lazy evaluation optimization technique,*** PySpark does not execute transformations immediately; instead, PySpark evaluates a sequence of transformation methods only when an action method is called. This avoids data transfers between nodes, as the whole sequence of transformations can be scheduled at once. With the latest version of DDS, we fully replicate this behavior by calling data loader methods and combined transformations within a single task right before an action method is called. Thus, compared with the previous version, the new DDS avoids intermediate data transfers amongst transformation tasks. Consequently, data transfers between nodes only occur when one of the action methods is called, or when the computed result is ready to be 'collected' in the main node. Additionally, since

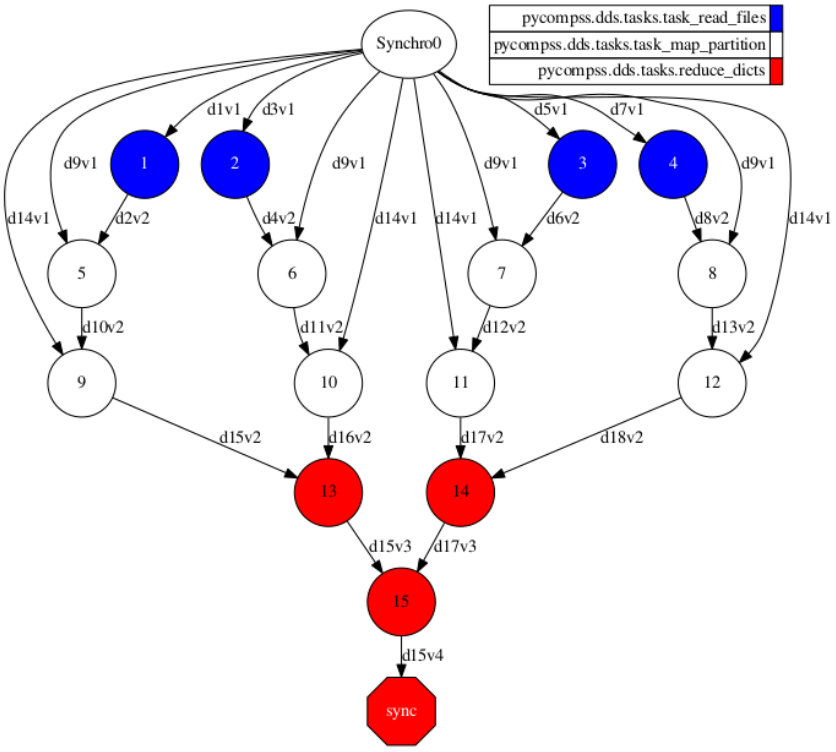
multiple tasks are combined into one, the runtime overhead of managing multiple small tasks has been reduced.

Similar to PySpark's RDD, we implemented most of DDS' functionality through the '`map_partitions`' method. This method runs one or more transformation functions on each data partition in parallel by creating a number of tasks equal to the number of partitions. The execution of these tasks is triggered when the 'collect' action is called. Using a single method like '`map_partitions`' for most parallel computations avoids code replication and eases the development of new methods.

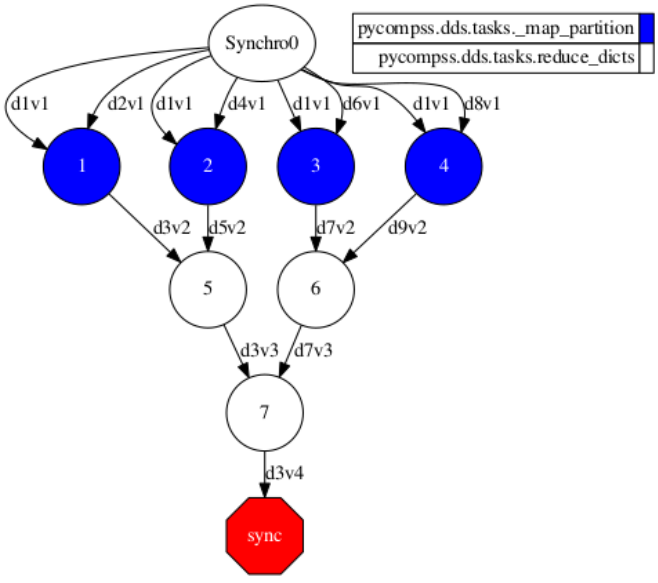
Moreover, the latest DDS library comes with a new interface named '`PartitionGenerator`'. This interface is the key point for the loading (reading) of the data within the 'map_partitions' tasks. When the user calls one of the data loader functions, the data is not processed nor read immediately. Instead, DDS creates '`PartitionGenerator`' objects per partition, which hold all the necessary information for the partition such as the data source, the number of the partition, etc. Initial partitioning is done by slicing the iterables based on the total number of partitions defined by the user. As in PySpark, partitioning can be considered partially dymanic, since the user can change the number of total partitioning when calling some DDS methods. Then, when one of the 'task' methods is called, these objects load only the data they are responsible for, and apply the DDS operations. Thanks to the '`DataLoader`' objects that implement the '`PartitionGenerator`' interface, PyCOMPSs tasks can read the data and immediately execute combined transformations in parallel, without using extra intermediate serializations.

Figure 1 shows the task graphs of Word Count using the first and the second versions of DDS for a small data set, corresponding to plots (a) and (b), respectively. Word Count is a simple Map-Reduce program that contains several 'map' and 'reduce' phases. It is worth mentioning that similar to RDD's reduce, DDS' reduce method applies an accumulative and commutative function to a DDS to obtain a single result. Besides, in the case of DDS, the user can define the number of inputs for 'reduce' tasks, thus allowing to control the total number of reduce tasks as well as data granularity.

In the graphs of Figure 1, nodes represent tasks, and edges represent data dependencies between them. We see that the first version of DDS created one task to read the data (blue nodes), and two tasks for the transformation processes which parse and count the results locally (white nodes). In contrast to this, the second version of DDS optimizes the workflow by combining and executing the first three tasks in a single one (blue nodes). Considering the expensive serialization and deserialization of big python objects, this approach significantly increases the speed of the second version of DDS.



(a) Word Count application task graph with first release of DDS.



(b) Word Count application task graph with the second release of DDS.

Figure 1 – Task graphs of Word Count implemented with two different versions of DDS. We observe that, in the second version, reading from files and two transformation phases take place within a single 'map_partition' task.

3. List of available methods

In the following, we list and briefly describe the currently available DDS operators. This release of DDS has adapted and included the most common methods Spark RDD. However, some additional methods could be easily added in the future, if needed for specific data analytics functions.

- **`load()`** – equivalent of the `'paralellize()'` method of PySpark Context. Builds a DDS object from a given iterator (e.g., a list) and a given number of partitions. During the creation of the DDS object, the input object is divided in partitions, and data is distributed among workers. If the number of partitions is not provided, `'load()'` creates 10 partitions by default. If the number of partitions is -1, `'load()'` assumes that the iterator contains PyCOMPSs 'Future Objects', and skips the data distribution process. This helps to pass results from other user-defined PyCOMPSs tasks to DDS without synchronisation on the master node.
- **`load_file()`** – builds a DDS from an input file. The input file is read in chunks of specific size in bytes, and contents of the file are stored as Strings. The 'load_file' operator implements two reading modes: master-read and worker-read. In master-read, the file is opened and partitioned at the master node. In worker-read, the file is opened and loaded in tasks that run in worker nodes. In master-read mode, the file is opened only once, while in worker-read mode the partitioning is carried out in parallel. The most efficient mode depends on the use case.
- **`load_text_file()`**– equivalent of `'textFile'` method of PySpark Context. In DDS this method is the same as 'load_file', with only difference that the input file is partitioned in lines instead of bytes.
- **`load_files_from_dir()`** – reads multiple files from a given directory and creates a DDS of (key, value) tuples where keys are file names, and values are the file contents stored as a String. The number of partitions of the output DDS is defined by the user. Partitions can contain the contents of more than one file if the number of given partitions is lower than the number of files.
- **`load_pickle_files()`** – loads serialized partitions form 'pickle' files and automatically creates one partition per file.
- **`collect()`** – returns the contents of a DDS. The normal behavior of the 'collect' method is to synchronize and return the actual contents of the DDS. Nevertheless, 'collect' also might return a list of future objects if specified by the user. This can be useful to run user-defined tasks that take partitions as input parameters. Since the return value is a list containing all elements of the

DDS, should only be called when the resulting array expected to be small enough to fit in the driver's memory.

```
>> DDS().load( range(3) ).collect()
[0, 1, 2, 3]
```

- **`save_as_text_file()`** – saves string representations of the DDS elements by creating one file per partition.
- **`save_as_pickle()`** – serializes each partition by 'pickle' module and saves them in a given directory. Serialized partitions can be loaded onto another DDS object by using 'load_pickle_files' method later.
- **`map()`** –applies a given function to each element of the DDS, and replaces the old value with the result.

```
>> DDS().load( range(10) ).map( lambda x: x * 2).collect()
[0, 2, 4, 6, 8 ,10 ,12, 14, 16, 18]
```

- **`map_and_flatten()`** – similar to 'map', applies a function to each element of the DDS. However, this function needs to return an 'iterable' object. Then, each element of the output 'iterable' is converted to an element of the output DDS. This operation is equilvalent to PySpark's 'flatMap'.

```
>> DDS().load( ["First String", "Second String"] )\
        .map_and_flatten( lambda x: x.split() ).collect()
['First', 'String', 'Second', 'String']
```

- **`map_partitions()`** – applies a given function to each partition of the DDS.
- **`filter()`** –applies a given function to each element of the DDS, and removes the element from the DDS if the the applied function returns 'False'.
- **`distinct()`** – removes repeated elements in the DDS. The number of partitions is kept as initial and final elements are distributed proportionally.

```
>> DDS().load( ["First String", "Second String"] )\
        .map_and_flatten( lambda x: x.split() ).distinct().collect()
['First', 'String', 'Second']
```

- **`reduce()`** – applies a function to subsets of DDS elements until a single value remains. The 'reduce' operator first reduces elements in each partition independently, and then reduces the remaining values in a tree-like structure. The user can specify the arity of this structure, and an initial reduction value.

```
>> DDS().load( range(10) ).reduce( (lambda a, b: a + b), initial = 100)
145
```

- **`min()` / `max()` / `sum()` / `count()`** – some self-explanatory functions that walk through all elements of the DDS and return a single value.
- **`foreach()`** – applies a function to each element of the DDS without returning any value. The 'foreach' operator includes a barrier to make sure that all the tasks finish the execution.
- **`union()`** – combines the current DDS with an arbitrary amount of other DDS objects.

```
>> first = DDS().load([0, 1, 2, 3, 4], 2)
>> second = DDS().load([5, 6, 7, 8, 9], 3)
>> first.union(second).count()
10
```

- **`key_by()`** – creates (key, value) pairs from DDS data, where keys are generated by applying a given 'f' function to the elements, that is, key = f(element).

```
>> DDS().load( range (3) ).key_by( lambda x: str(x) ).collect()
[( '0', 0), ('1', 1), ('2', 2)]
```

- **`partition_by()`** – creates partitions based on a user-defined function. It gives the flexibility to the user to control the data granularity.
- **`map_values()`** – keeping the 'keys' as they are, it applies a given function to the 'values' of each element in the DDS where elements are considered to be (key, value) pairs.
- **`reduce_by_key()`** – similar to 'reduce', but elements of the DDS are considered to be (key, value) tuples.
- **`count_by_value()`** – returns the total count of keys per value in a DDS object where elements are in a (key, value) format.

```
>> first = DDS().load([0, 1, 2], 2)
>> second = DDS().load([2, 3, 4], 3)
>> first.union(second).count_by_value(as_dict=True)
{0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
```

- **`combine_by_key()`** – combines 'values' for each key based on a user-defined function, where each element of the DDS is represented as a '(key, value)' pair.
- **`flatten_by_key()`** – the reverse of `'combine_by_key'`. Given elements in (key, value) format where the 'value' is an iterable object, it creates multiple pairs for each element.

- **`sort_by_key()`** – sorts elements of the DDS by their key values where all the elements considered to be in a (key, value) format. The user can define his or her sorting function and the number of partitions to be created after sorting.

4. Performance

We have been testing the DDS performance since the first release, and there are some considerable improvements in the last version. All the experimental applications have been executed on MareNostrum 4 supercomputer of BSC with the various number of nodes/cores (each MareNostrum 4 node accounts for 48 cores). Two different applications have been tested, namely the WordCount and the Terasort algorithms.To evaluate the performance of the WordCount application, we have parsed and counted the words from multiple text files, generated through the *Lorem Ipsum* library with a total size of 213 GB. Table 1 presents the results of the first release of DDS, PySpark (with Spark version 2.3.2), and latest DDS version implementations of Word Count application.

Table 1 – WordCount execution times

# of Worker Nodes / #cores	1/48	2/96	4/192	8/384	16/768
	Time Elapsed (sec)				
DDS 1		800.27	381.47	196.97	100
DDS 2	130.78	67.97	37.09	23.94	14.7
PySpark (v. 2.3.2)	328.48	186.28	129	88.02	65.79

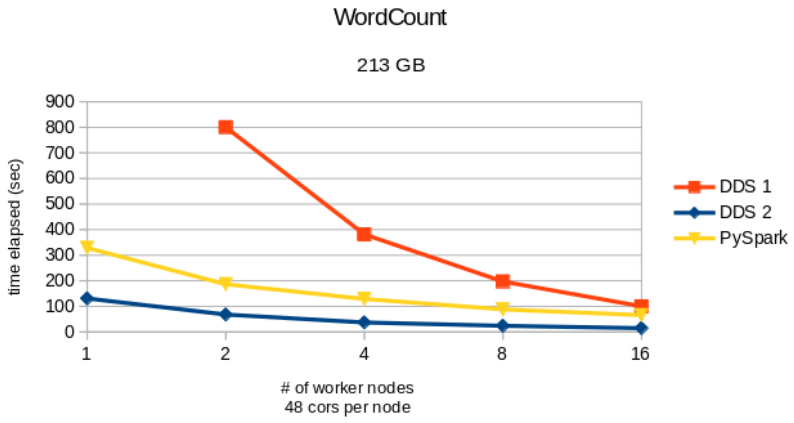


Figure 2 – Comparison of WordCount execution time with DDS 1, DDS 2, and PySpark

Figure 2 illustrates the results from Table 1. We observe that the latest version of DDS has the best performance, independently from the number of nodes, especially when

compared to the old version of DDS. The main reason for this improvement is the reduction of expensive intermediate tasks, as mentioned before.

As a second example, we have tested the TeraSort program, which serves in benchmarks to sort 1 Terabyte of data represented by key-value pairs. The DDS implementation of this program can be found in the last section, along with other examples. Table 2 shows the results from the executions of the program to sort 95 GB of data with multiple numbers of nodes.

Table 2 – TeraSort execution times

# of Worker Nodes / #cores	1/48	2/96	4/192	8/384
	Time Elapsed (sec)			
DDS 1	4364	1845	1033	2986
DDS 2	2127	989	434	262
PySpark (v 2.3.2)	1312	1320	521	331

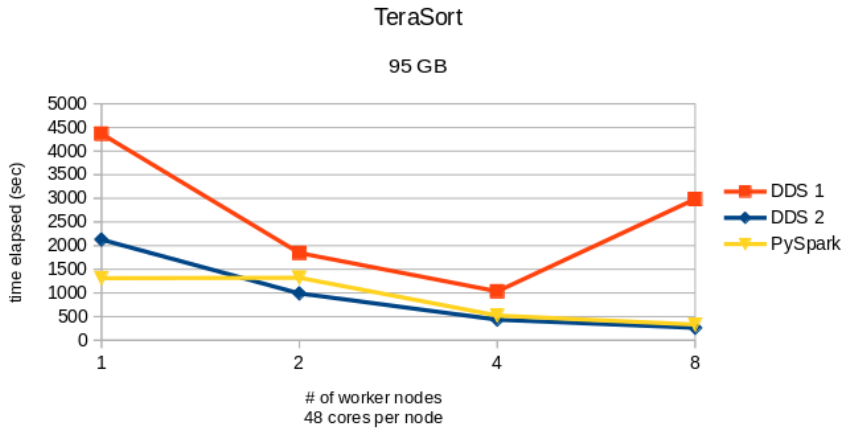


Figure 3 – Comparison of TeraSort execution time with DDS 1, DDS 2, and PySpark

Figure 3 visualizes the results from Table 2. It is clearly seen that the current version of DDS has reduced the execution time compared to the old version, and also has slightly better performance than PySpark, except for the single-node execution.

5. Usage instructions

As said before, DDS is distributed as part of PyCOMPSs (version 2.4 and above), and does not require the installation of additional packages. Users can test the DDS using PyCOMPSs Player package [1] available on PyPI.

[1] https://pypi.org/project/pycompss-player/

The following steps show how to execute the WordCount application:

1. Install docker for python (if not already installed):

```
python3 -m pip install docker
```

2. Install pycompss-player using pip:

```
python3 -m pip install pycompss-player
```

3. Insert some text in a file named 'book.txt' using vim.
4. Create a 'wordcount.py' file with the following code:

```
from pycompss.dds import DDS
def main():
        results = DDS().load_text_file('book.txt') \
                .map_and_flatten(lambda x: x.split()) \
                .count_by_value(True)
        print(results)
if __name__ == "__main__":
        main()
```

5. Run the application using run command of pycompss:

```
pycompss run wordcount.py
```

6. Examples

The following code snippets show some example applications from public repository[2] that are implemented using DDS:

TeraSort:

```
from pycompss.dds import DDS
def files_to_pairs(element):
    """ helper function to parse files """
    tuples = list()
    lines = element[1].split("\n")
    for _l in lines:
        k_v = _l.split(",")
        tuples.append(tuple(k_v))
    return tuples


def terasort():
    dir_path = sys.argv[1]
    dest_path = sys.argv[2]
    start_time = time.time()
    dds = DDS().load_files_from_dir(dir_path)\
        .map_and_flatten(files_to_pairs)\
        .sort_by_key().save_as_text_file(dest_path)
```

Pi estimation:

```
from pycompss.dds import DDS
def inside(_):
    """ helper function to 'throw the dart' """
    import random
    x, y = random.random(), random.random()
    if (x * x) + (y * y) < 1:
        return True
def pi_estimation():
    print("Estimating Pi by 'throwing darts' algorithm.")
```

[2] https://github.com/bsc-wdc/compss/tree/stable/compss/programming_model/bindings/python/src/pycompss/dds

```
    tries = 100000
    count = DDS().load(range(0, tries), 10) \
        .filter(inside).count()
    print("Pi is roughly %f" % (4.0 * count / tries))
```

Inverted Indexing:

```
from pycompss.dds import DDS
def _invert_files(pair):
    """ helper function to parse files"""
    res = dict()
    for word in pair[1].split():
        res[word] = [pair[0]]
    return list(res.items())


def inverted_indexing():
    path = sys.argv[1]
    result = DDS().load_files_from_dir(path).map_and_flatten(_invert_files)\
        .reduce_by_key(lambda a, b: a + b).collect()
```

7. Final remark

Originally, CLASS intended to unify task-based and map-reduce models into a single parallel programming model, i.e., COMPSs, as presented in this deliverable. However, CLASS also initiated an exploration path into serverless paradigm.

In that regard, the CLASS project was one of the first EU projects to explore and promote the new serverless paradigm as a novel, elegant and efficient way to develop and execute software at scale in a heterogeneous distributed environment, such as the cloud-to-edge compute continuum. An open-source serverless platform of Apache OpenWhisk was chosen as the foundation for CLASS analytics, allowing developers to easily compose software that consists of multiple analytics frameworks that could collaborate over the serverless platform's protocol. From a project design perspective, CLASS is about event-driven programming, which is the natural computation model for serverless. Thus, serverless proved as a excellent fit for CLASS goals, with event response leveraged in the use-cases and edge analytics implementation via federation.

As a result, CLASS decided to replace Spark for the Lithops map-reduce engine (formerly known as PyWren), which actually executes the computation using serverless functions, exploiting additional valuable benefits of serverless. One is that computation is completely elastic for all data scales, allowing to leverage all platform resources with no development cost using the built-in auto-scaling. Another benefit is that multiple analytics computations can run concurrently and share the resources via the underlying global serverless scheduler. Lithops was used in the core components of Trajectory Prediction (TP) and Collision Detection (CD) and proved quite adequate, as reported in Deliverable D5.5, "Evaluation of CLASS Big-Data Analytics Layer". On the same note, we also learned of limitations of serverless platforms, such as statelessness, preset scheduler, and high on-demand overhead of initialization and finalization. The lessons have been published in the BDVA and HiPEAC panels, and greatly motivated the creation of the EXPRESS asset.

Finally, the consortium decided to maintain the activity of integrating Spark and COMPSs into a single programming framework as reported in this deliverable. The efforts required for use-case implementation where however shifted to serverless activities. This is why this deliverable only performs the evaluation on benchmarks not related to use-cases.

Acronyms and Abbreviations

- D – deliverable
- DDS – Distributed Dataset
- M – Month
- MS – Milestones
- RDD – Resilient Distributed Datasets

References

[1] CLASS, "D2.3 - First release of Spark and COMPSs integration," March 2019.