COMPSs

COMP Superscalar

PyCOMPSs Distributed Data Set
User Manual

November, 2018

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

This manual provides information only about the development of PyCOMPSs applications using Distributed Data Set Interface. Before starting programming with PyCOMPSs DDS, please make sure you are acquainted with COMP Superscalar already. For more information about COMPSs, its installation process, programming models, user manuals and extensive list of example applications, please refer to http://compss.bsc.es/.

PyCOMPSs Distributed Data Set

Distributed Data Set (DDS) is a lightweight library to ease development of PyCOMPSs applications. It provides an interface where programmers can load data from basic Python data structures, generators, or files, distribute the data on available nodes, and run some most common big data operations on it. By using DDS, number of code lines can be reduced, where performance improvement is not expected comparing with regular PyCOMPSs applications.

How It Works?

To take advantage of DDS, first of all, the user should load the data to a new instance of it. Once one of the 'load' functions is called, the data will be partitioned and sent to the available nodes, and after that the user can perform any of DDS operations to manipulate the data simply by calling methods of the instance. In DDS environment, the initial data is always distributed on arbitrary number of partitions, and passed from one task to another as 'Future Objects', until the programmer 'synchronizes' or 'collects' it.
Moreover, it is also possible to create a new DDS with a 'list' of 'Future Object's from user-defined functions, or send data from a DDS instance to other user defined functions as Future Objects without retrieving it on the master node. This flexibility gives the user an opportunity to use DDS methods anywhere in the code, mixing the data from those methods with his/her own functions without sticking to pre-defined data operations, as well as replace some methods with DDS ones on an existing project.

How To Use?

As a library, DDS comes along with PyCOMPSs, thus it is not required to install a new package. If PyCOMPSs is already installed on the system, the following single line of Python code is enough to import DDS:

```
>> from pycompss.dds import DDS
```

After that, we would have to create an instance of the DDS class and provide it with some data. In the following code snippet, we are filling our DDS instance with the numbers from 0 to 10, which basically means elements of the DDS will be those digits:

```
>> data = range(10)
>> dds = DDS().load(data)
```

Since the data set is ready to be used, we can simply call some methods of the DDS class. For example, let's assume we want to filter our numbers and keep only even numbers. Same as Python's built-in 'filter', all we need is a 'lambda' function- which will eliminate odd numbers, and send it as a parameter to the DDS's 'filter' method:

```
>> even_numbers = dds.filter ( lambda x : x % 2 == 0 ).collect()
```

As we have already mentioned, without calling the 'collect' method, the data is never transferred to the master node. Since in our example, we do not want to perform any other operation than *filtering*, we call it to retrieve the even numbers between 0 and 10 as a list:

```
>> print(even_numbers)
[0, 2, 4, 6 , 8]
```

This is a very simple example of the use of DDS and before listing all available methods, let us have a look at a more real-world case where we can take advantage of PyCOMPSs DDS. One of the most-known Big Data examples is Word Count. The required code to implement it with DDS would contain the following steps:

- reading data from a file
- splitting the lines into words (so that elements of DDS are not lines from the file, but words from each line)
- counting the amount of each element (word)

And all these three steps can be performed within a single line of code:

```
>> from pycompss.dds import DDS
>> results = DDS().load_text_file( 'book.txt' ).map_and_flatten( lambda x: x.split() ).count_by_value( True )
>> print ( results )
{'a' : 10, 'the' : 15, ...}
```

For an explicit explanation, we can add that 'load_text_file' reads 'book.txt' file line-by-line and loads it onto the DDS instance. At this point, elements of the DDS are 'string' lines, and each partition contains the same amount of them. Then, the 'map_and_flatten' method does the transformation from lines to words by parsing and spreading them inside the partitions. In other words, if a partition contained lines before 'map_and_flatten' method, afterwards it contains all the words from its lines as elements (see different mapping functions from 'Available Methods' section in order to have more clear idea). The last method called is 'count_by_value' which retrieves a dictionary where 'keys' are elements (words) of the DDS, and 'values' are times of occurrence. The argument for this function- 'True', represents whether we want to collect the results, or we prefer to have the final dictionary to be partitioned and distributed on nodes again. It would be useful to set it to 'False', if we wanted to perform more operations on our data set.

Available Methods

All the methods provided by DDS are listed below with their arguments list, and descriptions:

- **load –** has one obligatory (iterator) and one arbitrary (number of partitions) parameters. Iterator is any kind of 'iterable' object from Python, such as generators, lists, etc. Iterator represents the data that will be distributed, and result of each iteration will be an element on DDS. The number of partitions can be defined by user, and will be set to 3 by default. The return value of this method is a DDS with a partitioned data. When the number of partitions is set to '-1', DDS assumes that the 'iterator' is already a list of Future Objects and skips data partitioning (distributing) step.

- **load_file** – loads data from a file in chunks and creates one partition for each chunk. Since COMPSs gives us the opportunity to read the files either on the master or worker nodes, this option is enabled for this method as well (by default it will be read on the Master node and each partition will be sent to worker nodes one-by-one) . The chunk (partition) size is arbitrary and will be set to 1024 B if not defined by the user. The return value of this method is a DDS containing Python Strings as elements.

- **load_text_file –** basically, same as 'load_file' method. The only difference is the fact that reading a text file in bytes can cause incomplete words as elements in DDS. To avoid this situation, text files are read line-by-line, and the chunk size can define the size of partitions in 'amount of lines' or in bytes.

- **load_files_from_dir –** reads multiple files from a given directory and saves them onto DDS by creating (key, value) *tuples* where *keys* are file names, and *values* are the file contents stored as Strings. Partitions can contain more than one file, when it is not possible to distribute one file in more than one partition.

- **collect –** returns the data of a DDS. It is possible to synchronize the data and retrieve it inside a list. However, when the value of 'future_objects' parameter is 'True', there a synchronization point will not take place, and each partition will be retrieved as a Future Object. The programmer can apply more operations on those Future Objects without transferring them to the Master node.

```
>> DDS().load( range (10) ).collect()
[0, 1, 2, 3, 4, 5, 6, 7, 8 , 9]
```

- **map –** same as the Python's built-in *map* method, 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 the *map* method, with the difference that, the given function should return an 'iterable' object, and each element of that 'iterable' will be spread over the partition.

```
>> 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 the partitions of a DDS. It can be thought as a *map* function where the input is a partition of DDS instead of an element of a partition.

- **filter** – same as Python's built-in *filter* method, applies a given function to each element of the DDS; if the result of the function applied to the element is 'False', then the element is removed from the DDS.

- **distinct** – keeps only one of the repeating elements inside 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** – same as the Python's built-in *reduce* method, applies a given function to each pair of the DDS elements and returns a single value. Since reductions are done inside partitions locally and then merged in a tree structure, it is possible to define depth of the *reduction tree. The i*nitial value for the reduce can be set as well.

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

- **reduce_by_key** – similar to the regular *reduce*, with the only difference that the elements of the DDS considered to be (key, value) tuples at the beginning of the reduction. The results can be retrieved as a dictionary in the master node, or as *Future Objects* of ‘(key, value)’ pairs where *keys* are unique, and *values* are reduced results for each *key*.

- **foreach** – applies a given function to each element of the DDS without returning any value. Has a Barrier Point in order to make sure that all the tasks finish the execution.

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

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

- **min / max / sum / count** – some self-explanatory functions that walk through all elements of the DDS and return a single value.

```
>> DDS().load( range (100) ).count()
100
```