Check for updates

RESEARCH ARTICLE

REVISED DDS: integrating data analytics transformations in task-based workflows

[version 2; peer review: 3 approved]

Nihad Mammadli[1], Jorge Ejarque[1], Javier Alvarez[2], Rosa M. Badia[1]

[1]Workflows and Distributed Computing, Barcelona Supercomputing Center, Barcelona, Catalunya, 08034, Spain
[2]IOVLabs, Gibraltar, Gibraltar

v2 **First published:** 25 May 2022, **2**:66
https://doi.org/10.12688/openreseurope.14569.1
Latest published: 11 Apr 2023, **2**:66
https://doi.org/10.12688/openreseurope.14569.2

Abstract

High-performance data analytics (HPDA) is a current trend in e-science research that aims to integrate traditional HPC with recent data analytic frameworks. Most of the work done in this field has focused on improving data analytic frameworks by implementing their engines on top of HPC technologies such as Message Passing Interface. However, there is a lack of integration from an application development perspective. HPC workflows have their own parallel programming models, while data analytic (DA) algorithms are mainly implemented using data transformations and executed with frameworks like Spark. Task-based programming models (TBPMs) are a very efficient approach for implementing HPC workflows. Data analytic transformations can also be decomposed as a set of tasks and implemented with a task-based programming model.

In this paper, we present a methodology to develop HPDA applications on top of TBPMs that allow developers to combine HPC workflows and data analytic transformations seamlessly. A prototype of this approach has been implemented on top of the PyCOMPSs task-based programming model to validate two aspects: HPDA applications can be seamlessly developed and have better performance than Spark. We compare our results using different programs. Finally, we conclude with the idea of integrating DA into HPC applications and evaluation of our method against Spark.

Keywords

Big Data High Performance, Data Analytics, Parallel Computing, Task Based Programming Models

H2020 This article is included in the Horizon 2020 gateway.

Open Peer Review

Approval Status ✓ ✓ ✓

	1	2	3
version 2 (revision) 11 Apr 2023	✓ view		✓ view
version 1 25 May 2022	? view	✓ view	? view

1. **Peggy Lindner**, University of Houston, Houston, USA
2. **Florina M. Ciorba**, University of Basel, Basel, Switzerland
   Ahmed Eleliemy, University of Basel, Basel, Switzerland
3. **Rafael Ferreira da Silva**, Oak Ridge National Laboratory, Oak Ridge, USA

Any reports and responses or comments on the article can be found at the end of the article.



This article is included in the Data Science gateway.

Corresponding authors: Nihad Mammadli (nihad.mammadli@bsc.es), Jorge Ejarque (jorge.ejarque@bsc.es), Javier Alvarez (javier@iovlabs.org), Rosa M. Badia (rosa.m.badia@bsc.es)

Author roles: Mammadli N: Investigation, Software, Writing – Original Draft Preparation; **Ejarque J**: Methodology, Supervision, Writing – Review & Editing; **Alvarez J**: Methodology, Supervision; **Badia RM**: Funding Acquisition, Supervision, Writing – Review & Editing

Competing interests: No competing interests were disclosed.

Grant information: This research was financially supported by the European Union's Horizon 2020 research and innovation programme under the grant agreement No 780622; and the Spanish Government (PID2019-107255GB), Generalitat de Catalunya (2014-SGR-1051).

Copyright: © 2023 Mammadli N *et al*. This is an open access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

How to cite this article: Mammadli N, Ejarque J, Alvarez J and Badia RM. **DDS: integrating data analytics transformations in task-based workflows [version 2; peer review: 3 approved]** Open Research Europe 2023, **2**:66 https://doi.org/10.12688/openreseurope.14569.2

First published: 25 May 2022, **2**:66 https://doi.org/10.12688/openreseurope.14569.1

REVISED **Amendments from Version 1**

In this version, we have mentioned the main difference between our work and in-situ high performance workflows in the Related Work section. We have extended Implementation section with a pseudo-algorithm explained individual parts of the proposed method more in details. Moreover, trace file of the HPDA application is analyzed step-by-step to help the reader follow data transfers and the order of the tasks described in the DAG. Furthermore, in the Evaluation section, we have added missing data size for Word Count experiments. Finally we have included the current limitations of the proposed method regarding configuration constraints and execution on containers, and emphasized their potential implementation in future versions of DDS.

Any further responses from the reviewers can be found at the end of the article

I. Introduction

High-performance computing (HPC) provides computational resources, software environments and programming models to enable the execution of large-scale e-science applications, *i.e.*, with the objective of making predictions or simulations such as weather forecasting or protein interaction modelling. Recently, with the introduction of Big Data technologies, e-science applications have evolved to more complex workflows where traditional HPC simulations are combined with data analytic (DA) algorithms. However, implementing applications that combines both aspects requires a lot of engineering efforts in terms of deployment and of integration of the HPC and data analytic aspects. For HPC workflows, developers use parallel programming models, while DA algorithms are mainly implemented using DA transformations using frameworks such as Spark[1]. In addition, some glue code that coordinates the execution and exchanges of data between the application components needs to be implemented. At deployment and operation phase, the work doubles since both HPC and Big Data environments have to be installed, configured and run at the same time. Furthermore, Big Data frameworks have been designed to run in traditional data centres, and they do not get benefit from the specific HPC hardware, such as high-speed networks.

High-performance data analytics (HPDA) is a current trend in e-science research which aims at the integration of traditional HPC with the recent DA frameworks[2]. Most of the work done until now in this field has focused on improving the data analytic frameworks by implementing their engines on top of HPC technologies such as MPI to benefit from the HPC hardware and speed up the execution of DA algorithms[3,4]. However, as raised above, there is still a lack of integration from the programming interface point of view. In the literature of HPC programming models we find task-based programming models. These models provide a good abstraction for developers and they are an efficient approach for implementing parallel HPC workflows. On the other side, DA algorithms are mainly programmed by applying *transformations* and *actions* in a dataset (a set of data elements). Transformations are functions which are applied for each element of the dataset (without modifying the number of elements) and actions are functions which are applied to the whole dataset and can modify the number of elements. To efficiently parallelise these algorithms, the dataset is divided into partitions (a group of elements), and transformations and actions are applied to these partitions. Therefore, data analytic algorithms can be decomposed as a set of tasks on top of task-based programming models.

In this paper, we present a methodology to develop HPDA applications on top of a task-based programming model. We propose the Distributed Data Set (DDS), an implementation of the data analytic transformations and actions on top of TBPMs. It allows developers to combine HPC workflows and data analytic transformations in a seamless way, developed as a single application, without requiring the use and deployment of several frameworks and achieving good performance. A prototype of this approach has been implemented on top of the PyCOMPSs task-based programming model. The prototype has been validated from a functional point of view by implementing a complex workflow that combines different DA transformations with other computational tasks. Moreover, we have executed big data benchmarks on top of the prototype to compare its performance with PySpark, which outperforms. The rest of the paper is organized as follows: Section II presents the related work; Section III introduces the proposed methodology to seamless integrate HPC workflows and data analytic algorithms; and Section IV describes how it has been implemented on top of PyCOMPSs. Then, Section V presents the evaluation; Finally, Section VI draws the conclusions.

II. Related work

HPC applications are implemented using parallel programming models, which allow developers to efficiently execute applications leveraging the supercomputer architecture. They are divided into two groups: shared memory models such as OpenMP[5] focus on the intranode parallelism; and distributed memory models that support applications whose parallelism involves multiple computing nodes, such as MPI[6] or Partitioned Global Address Space (PGAS). PGAS programming models assume a global memory address space logically partitioned and a portion of it is local to each process (e.g. UPC[7] or GASPI[8]). Another interesting approach for HPC is the task-based programming model which can be applied either in shared or in distributed environments and provides a good trade-off between abstraction and performance. Since version 3.0, OpenMP supports tasking parallelism. Other programming models which allow this model are OmpSs[9], StarPU[10], Legion[11] or COMPSs[12] for distributed environments.

Regarding Big Data applications, Spark[1] and MapReduce[13] are today's well-known frameworks. Spark provides in-memory DA operations for several programming languages. Its Resilient Distributed Dataset (RDD)[14] high-level abstraction has a rich set of methods for DA applications and eases the development for distributed data. On the other hand, MapReduce provides a programming model where developers define the computation with the map and reduce functions, and the parallelization is automatically done by the underlying runtime.

Most of the previous work on integrating Big Data and HPC are based on the implementation of Big data frameworks on top of the HPC technologies, especially on top of MPI. For instance, Alchemist[3], Spark-DIY[4] and Harp[15] allow users to call MPI-based libraries directly from Spark applications allowing users to substitute inefficient computations with calls to efficient MPI-based implementations. The main advantage of these approaches is that they keep Spark as a programming model, so users do not need to change their codes. Other approaches like Dask[16] offer a client API which can be used to implement big data applications together with other complex workflows. Some in situ high-performance workflows such as Decaf[17] provide a high-level Python API for the definition of workflow graphs, but require an additional implementation when used with other DA tools like Spark. It could be interesting for new applications but it requires a refactor for existing applications. Finally, Twister2[18] focuses on redesigning the whole big-data stack in order to get a balance between performance and usability which allows the inclusion of proven HPC existing technologies in a unified environment. In our approach, we have focused on the integration of HPC and Big Data technologies from a programming point of view, allowing developers to create applications which combine different types of algorithms in a single environment.

III. Methodology

This section presents the methodology proposed for programming HPDA applications on top of a task-based programming model. Figure 1 shows the overall approach of the methodology. On the left, we have the application code, composed of a part performing DA transformations and another part executing parallel code, implemented as task-based workflows. To allow users to seamlessly integrate data transformations in task-based workflows, we propose the Distributed Data Set (DDS). DDS is a library that provides an implementation of data transformations and actions (such as the ones defined in RDD's Spark's API) using task-based parallelism. Each transformation and action defined in the API will generate a task-dependency graph composed not only of other transformation graphs (such as in Spark) but also of graphs generated by the rest of the application. As depicted in the figure, the task-based runtime gets tasks from all application parts, generating a composed graph that includes the dependencies between the different parts. The generated Directed Acyclic Graph (DAG) of the whole application will be scheduled and executed in the available computing resources. We expect the following benefits:

- Developers do not need to perform significant changes to the DA algorithms since the API will almost be the same, and integrating them with other computations can be as easy as passing data between task invocations
- Since a DAG is generated for the whole application, there is no need for synchronization between DA parts and the rest of the parallel regions. The runtime will manage the data dependencies and data movements, avoiding unnecessary global synchronizations.
- Thanks to the composed DAG, the runtime will do better resource management, overlapping reduction regions whose resource usage is low with other independent parallel regions where the resource demand is higher.

A naïve approach for DDS can be proposed by creating a separate workflow of tasks for each transformation or action method that the user invokes. In this case, as described in Figure 2a, when the user calls a DDS method a set of tasks (one per partition) is added to the DAG for each transformation and a workflow of tasks per action. This approach has two main disadvantages: first, every new DDS method included in the library requires the implementation of a specific workflow. This requires a lot of effort for implementing and maintaining DDS. Second, DDS will create a new set of tasks for each method, which implies a high overhead to manage many tasks at runtime and to distribute them between the computing nodes.

To overcome the issues mentioned above, we have designed an optimized version of the DDS library. The main goal of the new approach is to minimize the number of generated tasks. First of all, we improved the data loader methods and implemented them in a way that there are no particular tasks only to load data. Partitions are loaded inside transformation or action tasks. Moreover, we also changed the behaviour of transformation tasks. Contrary to the naive implementation, we have incorporated the idea of combining consecutive transformation tasks and running them within a single task. In other words, multiple transformation methods are wrapped together and executed inside the same task that loads the partition. Furthermore, the invocation of that kind of transformation tasks is triggered when one of the action methods is called. Additionally, since multiple tasks are combined into one, the runtime overhead of managing multiple small tasks has been lowered. Figure 2b depicts the optimized version's behaviour, where we can observe the differences referred to earlier. Looking at the examples illustrated by Figure 2, in the optimized solution there is only one task per partition added to the DAG for the first three operations – data load and transformations 1 and 2 – whereas, in the naive solution, there would be three. Likewise, transformations 3 and 4 are combined and executed within a single task before the final synchronization. Finally, Figure 3 shows the code and the graph of a Word-Count application. In the code part, we compare the DDS and Spark implementations, where we can see the changes are minimal. In the generated graph, we can see the load and maps are executed in the first tasks (blue circles), and the *count by value* is implemented as a reduction in two phases (white circles).

Following pseudo-algorithm steps can be helpful to summarize the internal logic of the DDS:

1. initialize DDS
2. distribute the INPUT
3. while more transformation functions, wrap the functions
4. run combined transformation function(s) on each partition
5. run action function(s)

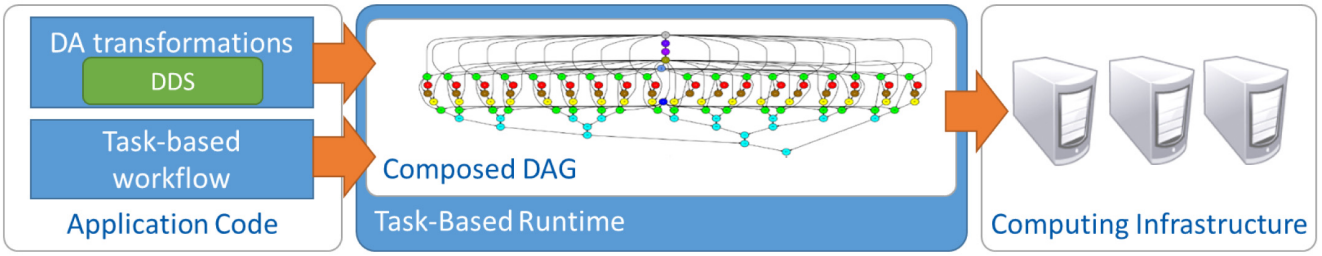
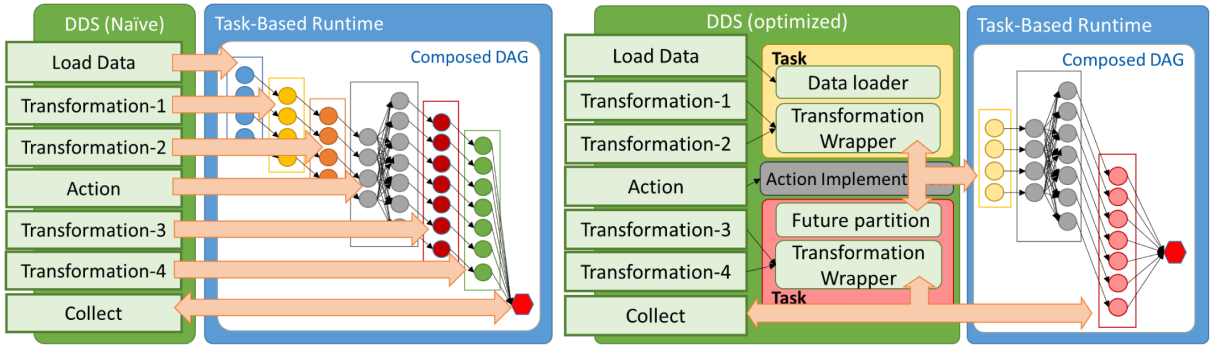


Figure 1. Overview of the proposed integration.



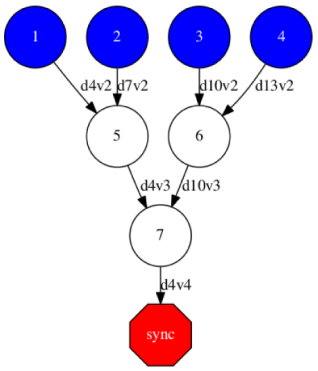
(a) Naive solution for DDS (b) Optimized solution for DDS

Figure 2. Distributed Data Set approaches.

```
# DDS implementation
from dds import DDS
wc_dds = DDS().load_files("<dir_path>")\
  .flat_map(lambda x: x[1].split()) \
  .map(lambda x: ''.join(e for e in x if e.isalnum())) \
  .count_by_value()

# RDD implementation
from pyspark import SparkContext
wc_rdd = SparkContext().wholeTextFiles("<dir_path>") \
  .flatMap(lambda pair: pair[1].split()) \
  .map(lambda x: ''.join(e for e in x if e.isalnum())) \
  .countByValue()
```

(a) Code comparison



(b) Task Graph

Figure 3. Word Count implemented with DDS.

6. repeat 3, 4, 5 if necessary

7. collect the results.

The most critical part to notice is that in the third step multiple transformation functions are combined and executed together on each partition. DDS handles it with the help of its Child object which is hidden from the user. Every time when the user calls a transformation function, the function is wrapped and passed to a Child DDS object. At the invocation of an action function, Child DDS object goes through the list of the transformation functions from its parents and applies them on each partition.

IV. Implementation

To validate the proposed methodology we have implemented a prototype of the DDS on top of the PyCOMPSs programming model. Next paragraphs provide an overview of PyCOMPSs and the details of the DDS prototype implementation.

PyCOMPSs[19] is the Python binding of the COMPSs framework[12] that facilitates the development of parallel computational workflows for distributed infrastructures. It offers a programming model based on sequential development – the application is a plain sequential Python script – where the user annotates the functions to be run as asynchronous parallel tasks. This

decorator also contains a description of the function parameters, such as type and direction, which are vital for building the dependency graph. In this graph, tasks are represented as nodes and data dependencies between tasks as edges. At execution time, asynchronous tasks are created for each decorated function and forwarded to the COMPSs Runtime which handles data dependency analysis, task scheduling and data transfers. The task creation is performed in an asynchronous way, and once the runtime has added a given task to the dependency graph, the execution of the main Python code continues, possibly generating new tasks. With this aim, PyCOMPSs manages future objects: a representant object is immediately returned to the main program when a task is invoked. A future object returned by a task can be involved in subsequent asynchronous task calls and PyCOMPSs will automatically find the corresponding data dependencies without requiring to wait for the actual result of the task. PyCOMPSs applications are deployed as master-worker applications, where the master executes the main code and invokes the runtime, and the workers execute the tasks.

DDS has been implemented on top of PyCOMPSs following the ideas presented in Section III. DA applications normally start with loading some data, then applying several transformation and action functions. Data loader and transformation operations are lazy operations in DDS. When the user calls a data loader function, DDS simply creates one DataLoader object per partition without retrieving the actual data. These DataLoader objects are later sent to the tasks to load the data at execution time. Then, when the user code 'maps' a transformation to the elements of the DDS, the DDS class creates a helper 'mapper' function as shown in Figure 4a. The helper 'mapper' is meant to take the user's transformation function as a parameter, and apply it to each element inside the partition. Thus, it is equivalent to the user's transformation operation with the only difference that the 'parameter' is the partition itself, instead of the elements. Moreover, every new transformation method creates a new ChildDDS object. A ChildDDS object is a DDS object that inherits its parent's transformation function and wraps it with the new-coming transformation. Following this structure, DDS can combine several transformations and operate them at once. The invocation of the combined transformations happens when a 'collect' or any other action method is called. As detailed in Figure 4b, DDS creates one 'map partition' task per partition, provided with a DataLoader object and wrapped transformations. At execution time, each 'map partition' task loads its own part of the initial data by calling DataLoader's 'retrieve data' method and then performs the combined transformations. Through the 'future objects' parameter of the 'collect' method, results of those tasks can be synchronized in the main program, or directly passed to other DDS or PyCOMPSs tasks.

V. Evaluation

We have evaluated our approach to demostrate that the prototype is valid for implementing integrated HPDA applications and does not underperform current Big Data framework such as PySpark. The results presented in this section have been obtained using the MareNostrum 4 (MN4) Supercomputer where each node has two Intel®Xeon Platinum 8160 (24 cores at 2.1 GHz each) and 96 GB of main memory[1]. Regarding the software, we have used COMPSs version 2.8[2] for the DDS executions, and for PySpark executions, we have used Spark version 3.0.0[3] in standalone mode on MareNostrum 4 with a basic configuration to fully exploit the cluster nodes. In all cases total number of threads per node were 48 and results are average of 3 repetitions per experiment.

[1]MareNostrum 4,https://www.bsc.es/marenostrum/, Accessed: Feb. 18, 2021

[2]COMPSs GitHub, https://github.com/bsc-wdc/compss, Accessed: Feb.18, 2021

[3]Spark GitHub,https://github.com/apache/spark, Accessed: Feb. 18, 2021

```python
class DDS:
 ...
 def map(self, fn, *args, **kwargs):
  # Apply func to each element
  def mapper(partition):
    results = list()
    for e in partition:
      results.append(fn(e,*args,**kwargs))
    return results
  return ChildDDS(self, mapper)

class ChildDDS(DDS):
 def __init__(self, parent, fn):
   ...
   self.partitions = parent.partitions
   par_fn = parent.fn
   def wrap_parent_fn(partition):
     return fn(par_fn(partition))
   self.fn = wrap_parent_fn
```

(a) DDS Transformation

```python
@task(returns=1)
def map_partition(func, part):
  if isinstance(part, DataLoader):
    part=part.retrieve_data()
  return func(part)

class DDS:
 ...
 def collect(self,future_objects=False):
   new_parts=list()
   #invoke tasks
   for p in self.partitions:
     new_parts.append(map_partition(self.fn,p))
   ...
   if future_objects:
     # without synchronization
     return new_parts
   else:
     return compss_wait_on(new_parts)
```

(b) DDS collect

Figure 4. Code snippets of the DDS implementation in PyCOMPSs.

A. HPDA Integrated application

To demonstrate that our approach can seamlessly integrate HPDA, we have implemented an application to detect similar documents. This application consists of different phases which combine multiple algorithms. Figure 5a shows a code snippet and the task dependency graph generated for the mentioned application (Figure 5b). It starts with a DA part generating a list containing all the words from the input files (vocabulary on 20). It can be easily retrieved using DDS methods such as 'load-files-from-dir', 'map-partition', and 'distinct'. For the second step, we pass this vocabulary to regular PyCOMPSs tasks to generate an appearance matrix for each file in the initial dataset. In these matrices, columns represent the words in alphabetical order, and values are their occurrences. Later on, in the third step, we run a distributed K-Means algorithm with the appearance matrices as input to detect clusters of files with a closer keyword affinity. Once we have the clusters, new tasks are called to compare each file with its cluster 'neighbours', and identify the most similar files. File comparisons are performed with the 'spaCy' Python module[4]. Finally, Figure 6 shows the execution trace automatically generated by PyCOMPSs. In this view, we can see how the PyCOMPSs runtime is scheduling and managing the execution of the DDS-generated tasks together with the rest of the application tasks. Following the same color coding as the DAG from Figure 5, the blue and the white lines represent tasks generated by the DDS. The next lines in the trace are of the "count_locally" function which has a "@task" decorator from PyCOMPSs. Different executions of those tasks are done in the worker node respecting the data locality from previous DDS tasks, thus avaoiding extra inter-node transfers. After that, pink, brown and green lines appear representing execution of the "K-Means" which is the last step of the HPDA application. This example demonstrates how DDS can seamlessly integrate DA algorithms together with other types of computations in the same application.

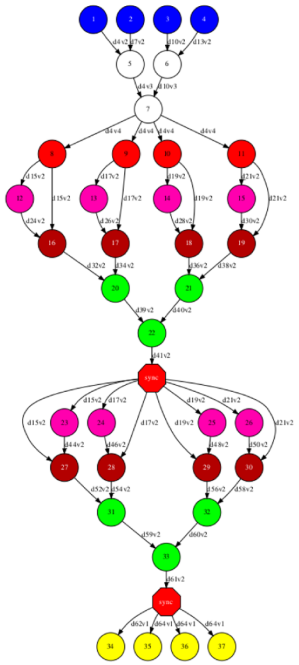
B. Performance Comparison Spark's RDD versus DDS

We have compared the performance of DDS with PySpark's RDD with different benchmark applications: the Word Count, Terasort and Transitive Closure. The first one is the Word Count program that consists of two phases; in the first step, files are read from the disk as partitions and words of each file

[4]spaCy, https://pypi.org/project/spacy/, Accessed: Feb. 18, 2021

```python
#Tasks definitions
@task(returns=list)
def count_locally(file_path,vocab):
  ...
@task(returns=dict,frag=COLLECTION_IN)
def partial_cluster(frag,mu,ind):
  ...
@task(returns=dict,frag=COLLECTION_IN)
def partial_sum(frag,clusters,ind):
  ...
@task(returns=dict)
def reduce_centers(a,b):
  ...
@task(returns=list)
def get_similar_files(fayl,cluster,threshold=0.90):
  ...
# Main code
# STEP 1: DDS get distinct words in all files
vocab=DDS().load_files(path).flat_map(lambda x:x[1].split())
  .map(lambda x:''.join(e for e in x if e.isalnum()))
  .count_by_value()
# STEP 2: Create apperance vectors for k-means
vects = list()
for fn in sorted(os.listdir(path)):
  f=os.path.join(path,fn)
  vects.append(count_locally(f,vocab))
# STEP 3: K-means
mu,n,frags=init values(vects, size)
while n < max_iter and not converged(mu,old_mu,epsilon):
  old_mu=mu
  clusters=[partial_cluster([vects[f]],mu,f*size) for f in range(frags)]
  p_results=[partial_sum([vects[f]],clusters[f],f*size) for f in range(frags)]
  mu=merge_reduce(reduce_centers,p_results)
  mu=compss_wait_on(mu)
  n += 1
clusters=compss_wait_on(clusters)
# STEP 4: Find similar files in clusters
for clus in clusters:
  for fayl in clus:
    sims_per_file[fayl]=get_similar_files(fayl,clus)
```

(a) Code snippet



(b) Task-dependency graph generated from the code

Figure 5. HPDA integrated application. Blue and white circles represent DDS tasks, red circles are count locally tasks, pink, red and green circles are the k-means algorithm tasks, and yellow circles correspond to get similar files tasks.

are counted locally. After that, local results are being merged within multiple reduce tasks. For these experiments, we run the Word Count with the classic books in English included in the Gutenberg Project[5] as input. Figure 7 shows the execution times for PySpark and DDS using a variable number of nodes. We can see that DDS performs better than PySpark but both of them are not scaling well due to the characteristics of the dataset (the speedup is computed using as baseline the execution time with PySpark and one worker node). The reduce part takes most of the time and it does not scale linearly with the number of resources. Figure 8 shows the same execution with the Lorem-Ipsum dataset. This dataset uses a reduced number of words and a better scalability is achieved in both cases, with slightly better results for DDS.

[5]Gutenberg Project, https://www.gutenberg.org/, Accessed: Feb. 18, 2021

The second benchmark we used for performance comparison is TeraSort. Even though the original algorithm widely used for benchmarks consists of two additional steps for data generation and validation, in our experiments, we only ran the sorting phase. TeraSort is tested with datasets containing key-value pairs where keys are 10 bytes of data to be sorted, and values are 90 bytes of data corresponding to each key. The algorithm's idea is to create multiple buckets for different key ranges with non-overlapping bounds and use a 'divide and conquer' strategy for the sorting process. First, each of the original data partitions assigns its elements to the corresponding buckets. When all the data has been distributed in the buckets, buckets are locally sorted. Considering that sorting the buckets by their bounds will also sort the whole dataset, no further computation is required after that step. Figure 9 shows the execution times and scalability results obtained when running the Terasort application with PySpark and DDS in a variable

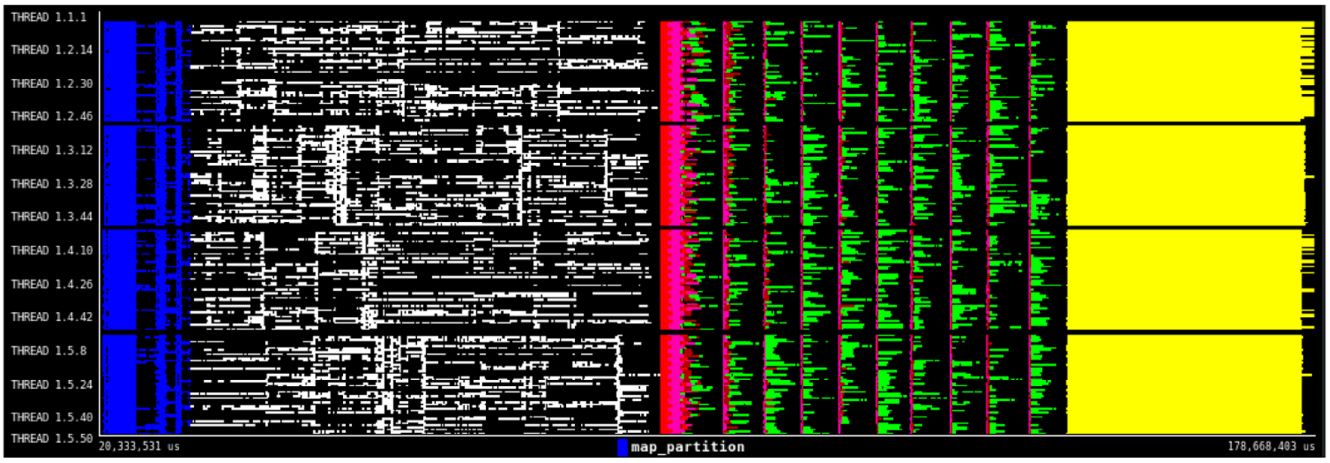


Figure 6. HPDA application execution trace. Y-axis shows the computing nodes threads and x-axis show the execution time. Each color line represents a task executed in a thread.

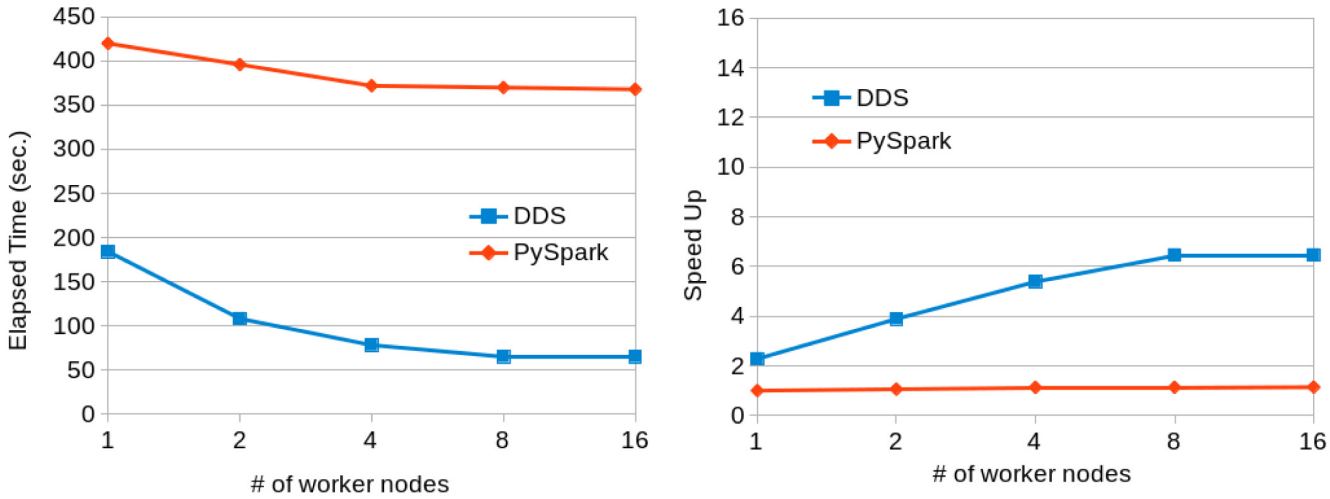


Figure 7. Word Count executions comparison for Gutenberg dataset (~ 80 GB).

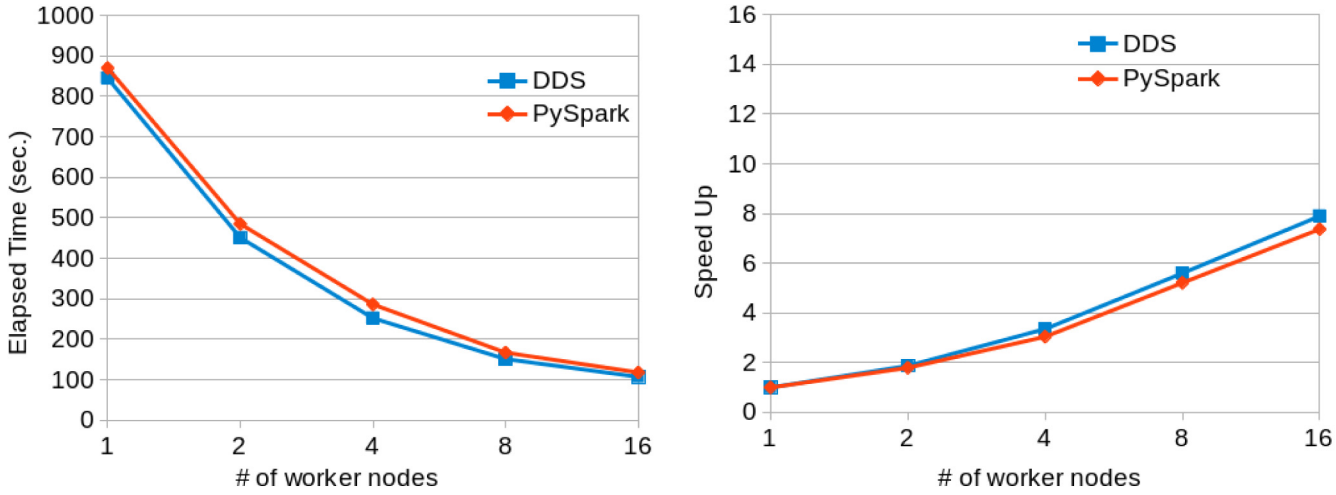


Figure 8. Word Count executions with Lorem Ipsum dataset (100GB).

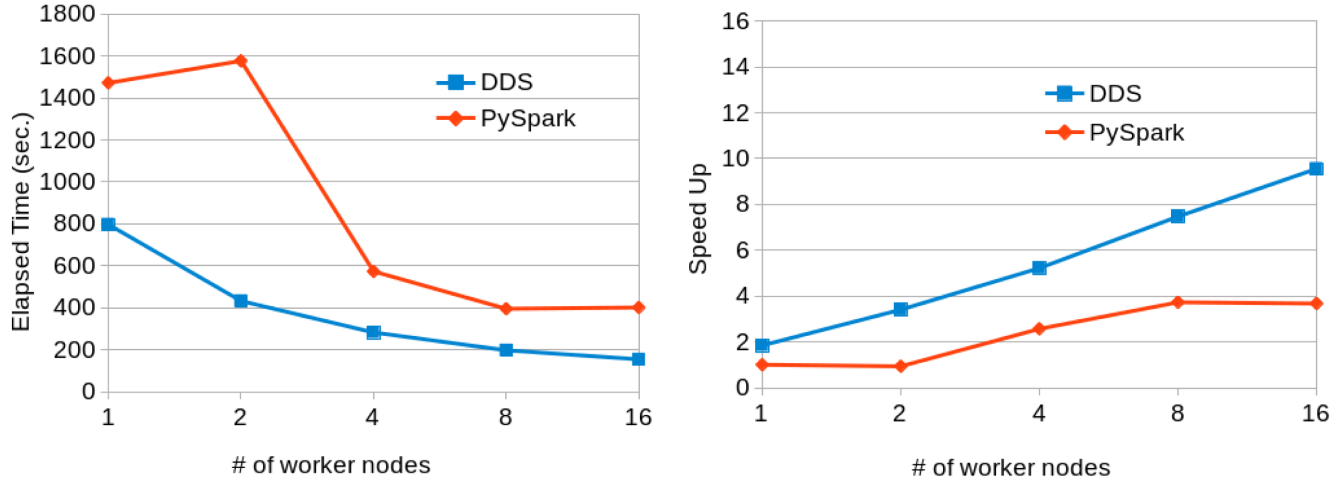


Figure 9. TeraSort executions comparison (200GB dataset).

number of MN4 nodes. Again, DDS has better performance than PySpark and also better scalability.

As the last example, we have implemented the Transitive Closure (TC) which is a simple reachability matrix within a given graph. The input data are ”source” and ”destination” nodes for each vertex, and the algorithm builds a final matrix where all possible connections are represented. In our implementation, we followed the PySpark's approach where in each iteration paths grow by one edge. For example, for edges (x,y) and (y,z), after the first round, (x,z) edge will be added to the discovered paths. The loop stops when the number of discovered edges does not change at the end of the iteration. Figure 10 shows the results obtained when running the program with 15-GB dataset with PySpark and DDS in a variable number of MN4 nodes. In this case, while DDS has better performance than PySpark, PySpark has slightly better scalability. However, not sufficient to perform better than DDS. The time achieved by Pyspark with 16 nodes can be achieved by DDS with just 4 nodes.

VI. Conclusion

This paper has presented a methodology to develop integrated HPDA applications where data analytics (DA) and HPC algorithms are combined on top of the task-based programming model. To achieve seamless integration of these two types of codes, we have presented the distributed data set (DDS) library, which implements the main used DA transformations and actions on top of a task-based programming model. From the different DA transformation and actions, the DDS produces a task graph whose input results can be seamlessly integrated with the rest of the task-based parallel codes, creating a composed task-dependency graph which is managed by the task-based runtime as a single application. A prototype of the DDS

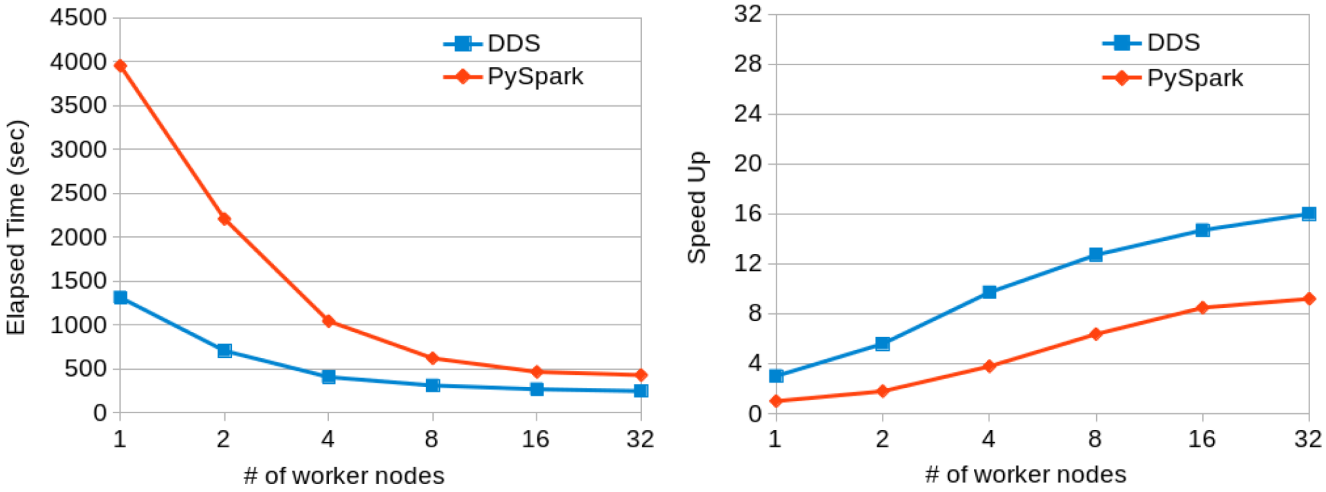


Figure 10. Transitive Closure executions comparison (15GB dataset).

has been implemented on top of the PyCOMPSs programming model and its validation has been focused on two aspects. On one hand, we have developed an HPDA application where a DA algorithm is combined with a K-means clustering algorithm and an algorithm to find text similarities. On the other hand, we have evaluated the performance of our DDS implementation comparing it with PySpark. Results from the evaluation demonstrate that DDS performs better than PySpark with similar scalability and DA codes can be seamlessly integrated with other task-based parallel codes allowing the users to create complex HPDA applications. Finally, although configuration constraints and execution on containers are supported in PyCOMPSs, current version of DDS does not provide these options. Nevertheless, they can be implemented in the future versions in order to fully benefit from PyCOMPSs' features.

VII. Data availability

Classical books from Gutenberg project that were used for Word Count experiments[21]. News articles that were used for HPDA application can be accessed on 20. Other datasets can be generated using generators on 22–24.

VIII. Software availability

Archived source code at time of publication on 25 Licence: Open Access

References

1. Zaharia M, Chowdhury M, Franklin MJ, *et al.*: **Spark: Cluster Computing with Working Sets**. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10.* (Berkeley, CA USA), USENIX Association, 2010.
Reference Source
2. Asch M, Moore T, Badia R, *et al.*: **Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry.** *Int J High Perform Comput Appl.* SAGE Publications Sage UK: London, England, 2018; **32**(4): 435–479.
Publisher Full Text
3. Gittens A, Rothauge K, Wang S, *et al.*: **Alchemist: An apache spark- mpi interface.** *Concurr Comput Pract Exp.* 2019; **31**(16): e5026.
Reference Source
4. Caíno-Lores S, Carretero J, Nicolae B, *et al.*: **Spark-diy: A framework for interoperable spark operations with high performance block-based data models**. In: *5th International Conference on Big Data Computing Applications and Technologies.* IEEE, 2018; 1–10.
Publisher Full Text
5. Dagum L, Menon R: **Openmp: an industry standard api for shared-memory programming.** *IEEE Comput Sci Eng.* 1998; **5**(1): 46–55.
Publisher Full Text
6. Gropp WD, Lusk E, Skjellum A: **Using MPI: portable parallel programming with the message-passing interface**. MIT press, 1999; **1**.
Reference Source
7. El-Ghazawi T, Carlson W, Sterling T, *et al.*: **UPC: distributed shared memory programming**. John Wiley & Sons, 2005; 40.
Reference Source
8. Grünewald D, Simmendinger C: **The gaspi api specification and its implementation gpi 2.0**. In: *7th International Conference on PGAS Programming Models.* 2013; 243.
Reference Source
9. Duran A, Perez JM, Ayguadé E, *et al.*: **Extending the OpenMP tasking model to allow dependent tasks.** *International Workshop on OpenMP.* Springer, 2008; **5004**: 111–122.
Publisher Full Text
10. Augonnet C, Thibault S, Namyst R, *et al.*: **StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures.** *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009.* 2011; **23**(2): 187–198.
Publisher Full Text
11. Bauer M, Treichler S, Slaughter E, *et al.*: **Legion: Expressing locality and independence with logical regions**. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.*

IEEE, 2012; 1–11.
Publisher Full Text

12. Badia RM, Conejero J, Diaz C, *et al.*: **COMP superscalar, an interoperable programming framework.** *SoftwareX.* 2015; **3–4**: 32–36.
Publisher Full Text

13. Dean J, Ghemawat S: **Mapreduce: simplified data processing on large clusters.** *Communications of the ACM.* 2008; **51**(1): 107–113.
Publisher Full Text

14. Zaharia M, Chowdhury M, Das T, *et al.*: **Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing**. In: *9th Symposium on Networked Systems Design and Implementation*. 2012; 15–28.
Reference Source

15. Zhang B, Peng B, Chen L, *et al.*: **Introduction to harp: when big data meets hpc**. Indiana University, 2017.
Reference Source

16. Rocklin M: **Dask: Parallel computation with blocked algorithms and task scheduling**. In: *Proceedings of the 14th python in science conference*. 2015; 126.
Publisher Full Text

17. Dreher M, Peterka T: **Decaf: Decoupled dataflows for in situ high-performance workflows. No. ANL/MCS-TM-371.** Argonne National Lab. (ANL), Argonne, IL (United States), 2017.
Reference Source

18. Kamburugamuve S, Govindarajan K, Wickramasinghe P, *et al.*: **Twister2: Design of a big data toolkit.** *Concurr Comput Pract Exp.* 2020; **32**(3): e5189.
Publisher Full Text

19. Tejedor E, Becerra Y, Alomar G, *et al.*: **PyCOMPSs: Parallel Computational Workflows in Python.** *Int J High Perform Comput Appl.* 2017; **31**(1): 66–82.
Publisher Full Text

20. **News Articles**. Accessed: May. 2, 2022.
http://www.doi.org/10.5281/zenodo.6420719

21. The Gutenberg Foundation: **Gutenberg Project**. Accessed: Feb. 18, 2021.

22. **Terasort dataset generator**. Accessed: May. 2, 2022.
http://www.doi.org/10.5281/zenodo.6424846

23. **Transitive Closure dataset generator**. Accessed: May. 2, 2022.
http://www.doi.org/10.5281/zenodo.6424848

24. **Lorem Ipsum dataset generator**. Accessed: May. 2, 2022.
http://www.doi.org/10.5281/zenodo.6424837

25. Barcelona Supercomputing Center: **COMPSs**. Accessed: May 2, 2022.
http://www.doi.org/10.5281/zenodo.6362651

Open Peer Review

Current Peer Review Status: ✔ ✔ ✔

Version 2

Reviewer Report 18 May 2023

https://doi.org/10.21956/openreseurope.17120.r31123

© **2023 Lindner P.** This is an open access peer review report distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

✔ **Peggy Lindner**
Department of Information & Logistics Technology, University of Houston, Houston, TX, USA

The authors have addressed the issues that I listed in my first review.

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: Workflows for Data Science applications

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Reviewer Report 17 April 2023

https://doi.org/10.21956/openreseurope.17120.r31122

© **2023 Ferreira da Silva R.** This is an open access peer review report distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

✔ **Rafael Ferreira da Silva**
Oak Ridge National Laboratory, Oak Ridge, TN, USA

The authors have successfully addressed all of the minor concerns I had, and as a result, the paper is now ready for indexing.

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: Distributed Computing, HPC, Scientific Workflows

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Version 1

Reviewer Report 12 July 2022

https://doi.org/10.21956/openreseurope.15731.r29377

© 2022 Ferreira da Silva R. This is an open access peer review report distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

?

Rafael Ferreira da Silva
[1] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN, USA

This paper provides a methodology for enabling the development of high-performance data analytics (HPDA) applications using task-based programming models (TBPMs). Specifically, authors propose a distributed data set (DDS) approach for implementing efficient data transformations and actions using task-based parallelism. The proposed methodology is evaluated using benchmark applications (word count, TeraSort, and transitive closure), and compared to the current widely used PySpark approach. Overall, DDS yields relatively significant better performance when handling data analytics applications.

General comments:

- The proposed methodology could be (to some extent) compared to in situ processing technologies. I would recommend authors to provide a brief discussion (potentially on the introduction and related work sections) about in situ data analytics and how it differs from the proposed methodology.

- Could authors provide a description of the proposed method using pseudo-algorithms?

- The experimental evaluation would benefit from some additional experiments/discussions to demonstrate how the DDS yields better resource management. Maybe authors could leverage some efficiency metrics?

- Experimental results are based on strong scaling, which demonstrates relevant improvement using DDS when compared to PySpark. The paper could also benefit from a weak scaling experiment in which results may lead to conclusions that could clearly provide insights about PySpark's low performance.

Minor comments:

- Consider replacing "What is more" by something like "Furthermore" or "Additionally".

- Typo in first paragraph section 3: "such as the defined" ==> "such as the ones defined".

- In Figs 7-10, maybe you can only show speed up, as the elapsed time plots provide relatively similar information.

- I would suggest authors to replace the term “master-worker” by “coordinator-worker” in an attempt to suppress oppressive language (see more at https://tools.ietf.org/id/draft-knodel-terminology-00.html#rfc.section.1.1).

Is the work clearly and accurately presented and does it cite the current literature?
Partly

Is the study design appropriate and does the work have academic merit?
Yes

Are sufficient details of methods and analysis provided to allow replication by others?
Partly

If applicable, is the statistical analysis and its interpretation appropriate?
Yes

Are all the source data underlying the results available to ensure full reproducibility?
Yes

Are the conclusions drawn adequately supported by the results?
Yes

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: Distributed Computing, HPC, Scientific Workflows

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard, however I have significant reservations, as outlined above.

Author Response 31 Mar 2023

nihad mammadli

First of all, thank you very much for your helpful comments, we have updated the document based on majority of your comments. However, even though we are aware that some more experiments would be useful to enhance the Evaluation section, unfortunately the project is already over and we do not have resources to run new experiments. We also believe that the evaluations that are already provided are sufficient to conclude the ideas.

Competing Interests: No competing interests were disclosed.

Reviewer Report 07 July 2022

https://doi.org/10.21956/openreseurope.15731.r29376

© 2022 Ciorba F et al. This is an open access peer review report distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Florina M. Ciorba
[1] Department of Mathematics and Computer Science, University of Basel, Basel, Switzerland
[2] Department of Mathematics and Computer Science, University of Basel, Basel, Switzerland
Ahmed Eleliemy
[1] University of Basel, Basel, Switzerland
[2] University of Basel, Basel, Switzerland

Summary
HPC Workflows use specific parallel programming models, while data analytic (DA) applications mainly exploit data transformations. Among those HPC parallel programing models, task-based programming models (TBPMs) are very efficient for HPC workflows. The authors proposed a methodology to develop HPDA applications on top of TBPMs. The authors proposed a library called Distributed Data Set (DSS) that implements data transformations and actions (such as the defined in RDD's Spark's API) using the PyCOMPs task-based library. The authors implemented multiple benchmarks, and the results showed that the proposed library outperforms the corresponding version implemented via PySpark.

Strengths
The authors close a gap in the state of practice for developing HPDA applications.

The proposed library (DDS) offers similar APIs to well-known data processing frameworks (Spark), i.e., applications will require minimal changes.

The manuscript is well written and structured.

Weaknesses
In the Evaluation section, the authors report the obtained results rather than discussing and giving insights on why these results, i.e., the Evaluation Section leaves the reader with more questions than insights about the work.

More details and suggestions for improvements
A detailed section about how the library internally works would be appreciated. What tasks are fusible and why?

Evaluation section: how many threads are configured per worker node? Do Spark and COMPs have the same number of threads per node?

More experiments would be beneficial that take into consideration the option of oversubscription

for the number of threads per node (in the case of Spark).

The authors attributed the performance superiority of their library to the ability to combine multiple tasks and have one data loader task. Please time individual parts within the code for both implementations, DDS and PySpark, to confirm your performance superiority claims.

Are the reported results based on one or several repetitions per experiment? Reporting this will increase the trustworthiness statistically.

Figures 7 and 8 show the performance of DSS and PySpark implementations of the Word Count benchmark. The obtained results require clarifications, i.e., both implementations have the same performance in the case of the Ipsum dataset, while DSS extremely outperforms PySpark in the case of Gutenberg.

Irrespective of the experiment, the performance of PySpark implementations is poor for a few nodes (1, 2, 4). But, it seems that PySpark implementations scale better than DSS. The performance of PySpark implementations improves beyond 4 nodes. Would it be better than DSS if we go beyond 32 nodes?

Why do specific experiments use 32 nodes while others use up to 16 nodes?

It is important to also include the limitations of the proposed DSS, at least those inherited from COMPs.

How does the DDS approach differ from recent approaches such as DAPHNE (https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf)?[1]

References

1. Damme P, Birkenbach M, Bitsakos C, Boehm M, et al.: DAPHNE: An Open and Extensible System Infrastructurefor Integrated Data Analysis Pipelines. *Conference on Innovative Data Systems Research*. 2022. Reference Source

Is the work clearly and accurately presented and does it cite the current literature?

Partly

Is the study design appropriate and does the work have academic merit?

Yes

Are sufficient details of methods and analysis provided to allow replication by others?

Partly

If applicable, is the statistical analysis and its interpretation appropriate?

Partly

Are all the source data underlying the results available to ensure full reproducibility?

Yes

Are the conclusions drawn adequately supported by the results?

Yes

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: High performance computing, high performance data analysis, scheduling and load balancing, tasking

We confirm that we have read this submission and believe that we have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Author Response 05 Apr 2023

nihad mammadli

For all experiments we have measured the execution times of Python code. It is true that timing individual parts would strengen our claims, it would be possible for us only to include different timers in our API to achieve this goal, we wouldn't be able to do the same with Spark's RDD. Thus, we only measured the execution times which can be done with both frameworks.

Competing Interests: No competing interests were disclosed.

Reviewer Report 27 June 2022

https://doi.org/10.21956/openreseurope.15731.r29379

© 2022 Lindner P. This is an open access peer review report distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

?

Peggy Lindner
[1] Department of Information & Logistics Technology, University of Houston, Houston, TX, USA
[2] Department of Information & Logistics Technology, University of Houston, Houston, TX, USA

The paper presents an approach to combining data analytical tasks with high-performance computing. The introduction provides a good, generalized background of the topic that quickly gives the reader an appreciation of the applicability of integrating a task-based programming model suitable for HPC environments with data analytics algorithms. The authors present the motivation and background for such an approach very well. The related work section is clear and sufficient.

The methodology section explains in an effective way how the distributed data set approach can be optimized for generated tasks. The implementation details are laid out detailed enough to follow the approach presented.

The evaluation section demonstrates a prototypical application implementation which is also used

for performance evaluation. It needs minor revisions and clarifications:

1. I would not call Part A) an "experiment" (see the first sentence in Part V. "*We have performed two experiments …".*) Please clarify the language used to introduce the evaluation section.

2. Please add some better explanations for the reader to describe the trace file (figure 6). While I appreciate the inclusion of the trace file, it would be good to refer to the color coding to tell readers which parts refer to "*see how the PyCOMPSs runtime is scheduling and managing the execution of the DDS-generated tasks together with the rest of the application tasks*."

3. Please add more details to the summary of Part A "*This example demonstrates how DDS can seamlessly integrate DA algorithms together with other types of computations in the same application.* "

4. The performance evaluation section should be enhanced with further details about the sample sizes used in the experiments (e.g., number of books/words etc.). Currently, it's hard to judge the experiments for their applicability to real-world scenarios (the selected algorithms are a good representation).

The conclusion section summarizes the presented work very well but is missing a discussion about limitations and/or future work.

Data Availability section: "News articles that were used for HPDA application can be accessed on[20]. " It is not very clear that this dataset is used in the prototype implementation example. Please mentioned the usage in the HDPA integrated application section.

I was not able to access the source code used for the evaluation. The source code link given under [12] points to https://github.com/ElsevierSoftwareX/SOFTX-D-15-00010 which is not accessible. That has to be fixed. The link for COMPSS 2.8.1 is fine.

Minor formatting issues: the first paragraph in the conclusion breaks up in the middle of a sentence. "*Results from the evaluation demonstrate that*
DDS performs better than PySpark with similar scalability …"

Is the work clearly and accurately presented and does it cite the current literature?
Yes

Is the study design appropriate and does the work have academic merit?
Yes

Are sufficient details of methods and analysis provided to allow replication by others?
Partly

If applicable, is the statistical analysis and its interpretation appropriate?
Not applicable

Are all the source data underlying the results available to ensure full reproducibility?

Yes

Are the conclusions drawn adequately supported by the results?
Yes

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: Workflows for Data Science applications

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard, however I have significant reservations, as outlined above.